IEEE Design and Test of Computers
Special issue on Embedded Processor Based Designs,
Ed.: Peter Marwedel, July-August 2002

Eclipse: A Heterogeneous Multiprocessor Architecture for Flexible Media Processing

Martijn J. Rutten, Jos T.J. van Eijndhoven, Evert-Jan D. Pol[†]
Egbert G.T. Jaspers, Pieter van der Wolf, Om Prakash Gangwal, Adwin Timmer
Philips Research Laboratories
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
{martijn.rutten, jos.van.eijndhoven}@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'. The multi-tasking coprocessors concurrently execute application tasks of one or more applications. To facilitate reuse, Eclipse separates coprocessor functionality from generic support, implemented by 'shells', attached to each coprocessor. The shells offer multi-tasking, inter-task synchronization, and data transport services to the coprocessors. This facilitates the design of coprocessors that require complex control to handle e.g. data-dependent I/O and saving/restoring task state. This paper presents the Eclipse architecture template as well as a first instantiation with coprocessors that support simultaneous MPEG-2 encoding and decoding.

1. Introduction

The advent of new media applications demands an increasing flexibility of consumer electronics products. These media applications include high-definition digital television, a set-top box with time-shift functionality, 3D games, video conferencing, or MPEG-4 like interactivity. Consumer electronics products are evolving into multi-functional devices that combine a set of such applications. The required set of applications and their format varies per product, per country, and over time as standards evolve. To accommodate these continuously changing requirements, complex consumer appliances are based on programmable architectures. As the development of a programmable architecture requires generic solutions for the complete application domain, the development cost of programmable architectures is significantly higher than the development cost for a function-specific architecture. However, once the flexible architecture with associated software-development environment is available, the time to market of a new software-based appliance can be very fast.

Managing complexity, design cost, and time-to-market of such programmable 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 [3][12]. Philips Electronics has been developing such a platform concept with instances such as the Viper [3]. Such System-on-Chip (SoC) solutions typically consist of a heterogeneous mix of fully programmable processors (*e.g.* MIPS, ARM, TriMedia VLIW) and coarse-grain application-specific subsystems (*e.g.* MPEG decoders, video filters). The application-specific subsystems are optimized for high performance with minimal power consumption and silicon area. Currently, these are dedicated for a single application and the hardware cannot be reused for other applications within the same application domain. Therefore,

[†] Evert-Jan Pol is currently with Philips Semiconductors, Eindhoven, The Netherlands.

even though the generic platform with its interconnect structure may be reused over various generations, a change in application requirements implies redesign of the application-specific subsystems. These subsystems take up a large part of the total design effort and silicon cost.

Eclipse is a template architecture for the design of versatile media-processing SoC subsystems. Instances of the Eclipse template are heterogeneous subsystems containing a mix of programmable and hardwired functions. Eclipse instances combine the performance density of function-specific hardwired modules, or *coprocessors*, with the flexibility of one or more programmable cores. These are linked into a network that resembles the application structure. The network configuration is programmable and set up in software.

Section 2 discusses the main architectural trade-offs that influence the design of Eclipse like media-processing architectures. This section serves as a rationale for the introduction of the Eclipse architecture template in Sections 3 through 5. In Section 6, we show a first instance of the Eclipse template that we used to validate the concepts outlined in this paper. Section 7 concludes with simulation results.

2. Design aspects of media-processing architectures

In the consumer-electronics domain, media-processing applications execute on resource-constrained systems. The performance, flexibility, and cost effectiveness of such resource-constrained systems are highly interrelated. The following subsections elaborate on the main architectural trade-offs that influence these parameters and provide the foundation for the design choices of the Eclipse architecture template.

2.1. Parallelism and flexibility

In the field of consumer electronics, design constraints, such as cost, power consumption, performance, and flexibility, are weighted differently than in the PC-market. Consumer media processors require an order of magnitude lower cost prize in combination with a significantly higher performance for the media-processing application domain. The key challenge is to design a media-processing architecture that meets the demand of high performance with a low power consumption and silicon cost.

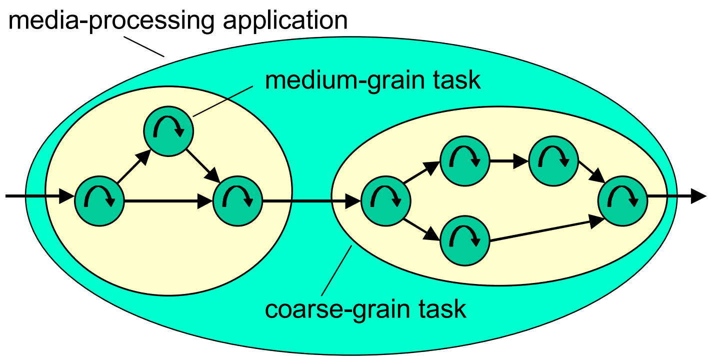


Figure 1. Granularity of parallelism in media-processing applications.

Media-processing applications typically exhibit parallelism at various levels of granularity (Figure 1). For instance, a time-shift recorder function consists of an encoding and decoding function which may be executed in parallel. Moreover, within the decoding function there are many medium-grain tasks that can execute in parallel, *e.g.* the discrete-cosine transform (DCT) and quantization tasks (Figure 2). Within such a DCT task, many operations can execute in parallel.

The parallelism is made explicit by specifying media-processing applications as a set of concurrently executing tasks that exchange information solely by unidirectional streams of data. A directed graph with a node for each task and an edge for each data stream represents the structure of the application. Kahn [8] introduced a formal model of such applications already in 1974, followed by an operational description by Kahn and MacQueen [9] in 1977. This formal model is now commonly referred to as the *Kahn Process Network* (KPN) model of computation, and defines the model of computation of the Eclipse architecture.

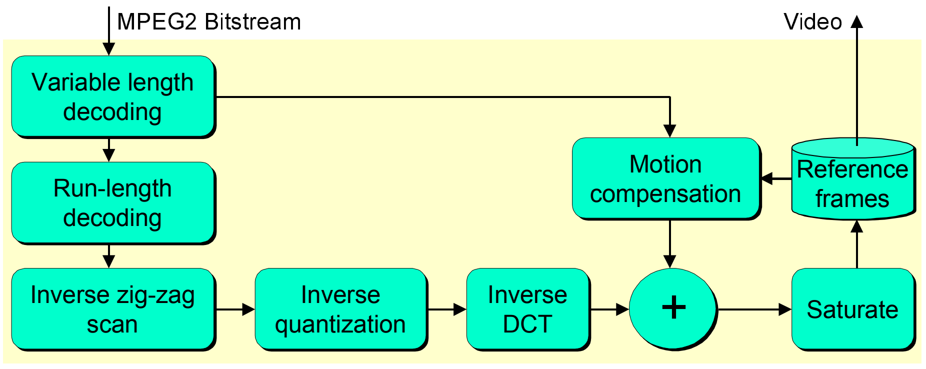


Figure 2. MPEG-2 decoder process network.

Figure 2 shows such a network for MPEG-2 video decoding. The data streams in the network are buffered. Each buffer is a FIFO, with precisely one producer and one or more consumers. Reading from a stream with insufficient data available causes a consuming task to stall. Kahn formally proved that such a system has a well-defined unique behavior. In particular, the functional behavior—observed as the sequence of data items that traverse the edges—is independent of the order in which the tasks are executed.

One of the nice features of the Kahn model is its inherent building-block nature. Once a set of basic functions has been defined as tasks, a multitude of applications can be configured by instantiating tasks and connecting them in a graph structure. Describing an application in these generic tasks is then followed by a mapping phase in which the system designer decides which modules in the architecture execute which tasks. For this mapping to make sense, the granularity of Kahn tasks must match the granularity of the architecture modules. The resulting solution is highly flexible if the building-block nature of the tasks is matched by a similar building-block nature of the modules, complemented by a generic infrastructure that allows run-time instantiation of task graphs.

For instance, complex media-processing SoCs exploit the performance density of sophisticated application-specific subsystems to implement critical parts of the targeted media applications, while endowing the system with a sufficient level of flexibility by embedding programmable cores. These subsystems execute concurrently to exploit task-level parallelism at a coarse granularity. Typical subsystem examples are a dedicated MPEG decoder, respectively a programmable embedded core. Such a heterogeneous mix of hardware and software is essential for a cost-effective yet flexible architecture. The subsystems implement the functions that the SoC needs to perform, while the SoC infrastructure takes care of the data communication between subsystems. Even though the subsystems may only be weakly programmable, the flexibility of the infrastructure to route data to and from different subsystems plays an important role in the flexibility of configuring different applications on the SoC.

For competitive consumer media-processing architectures, performance is maximized by exploiting parallelism wherever possible, while cost is minimized by introducing flexibility only where necessary. As noted in the introduction, state-of-the-art SoC subsystems have a coarse granularity, which renders them inflexible, *e.g.* an MEPG-2 decoder block cannot be reused for an MPEG-2 encoding application even though an MPEG-2 encoder embeds a large portion of an MPEG-2 decoder. In order to introduce flexibility of configuring applications on such SoC subsystems, the subsystem must support the mapping of medium-grain application tasks on internal function modules of corresponding granularity. Therefore, Eclipse introduces a programmable infrastructure that supports concurrent execution of medium-grain functions in a heterogeneous mix of software and reusable coprocessors.

2.2. Tightness of coupling

Hardware modules communicate data through communication buffers. The size of these buffers determines in how far the producer and consumer are coupled in the timing of their execution. The coupling of timing is determined by the regularity of processing in the application domain and the admissible amount of stalling behavior that sacrifices parallelism. For instance, regular tasks, such as in linear video filtering where worst-case communication requirements equal the average case, allow a tight coupling with minimal buffering [2][7]. Irregular tasks demand less tight coupling to allow individual progress of tasks, leading to larger buffer requirements. A typical irregular, data-dependent task is the variable-length decoder (VLD) in MPEG decoding where the quantity of input and output data can vary wildly per stream or even within a picture of a single stream. A less obvious example is the DCT function; the function itself is regular, but the number of DCT coded blocks to be processed varies per MPEG frame. Eclipse targets the application domain of video encoding and decoding, which exhibits a large amount of data-dependency, resulting in a high degree of irregularity. In practice, the ratio of worst-case versus average load can be as high as a factor of 10. Consequently, we have designed Eclipse to be a relatively loosely coupled system.

Communication requires both data transport and synchronization. *Data transport* refers to moving data into and out of communication buffers. Producers and consumers exchange information on the amount of produced or consumed data in the buffer that is available for consumption or production, respectively. We refer to this exchange of information as *synchronization*. Due to the FIFO buffering, the producer and consumer do not need to mutually synchronize individual read and write actions on the stream. Thus, synchronization granularity can be chosen independently. For instance, each individual data access can be synchronized, *i.e.* at the granularity of data transport, or synchronization can be closer related to the logical unit of input and output data on which a task operates, *e.g.* the granularity of a picture for an MPEG decoding task.

Decreasing the granularity of the application functions (*e.g.* from an MPEG-decoder function to a DCT function) to enhance parallelism and reuse introduces two issues. As the number of streams passing through the communication network increase, both communication buffering and bandwidth requirements increase. Eclipse reduces communication buffer requirements by changing the grain of synchronization to a finer level (*e.g.* from picture to macroblock level in MPEG). The resulting small communication buffers can be kept on-chip, allowing deployment of a dedicated communication network to cope with the high bandwidth requirements.

2.3. Scalability and separation of concerns

Reuse is crucial to keep the design cost of consumer devices at an acceptable level. Scalability is needed to implement reuse over generations of an architecture. Architecture templates are essential in supporting scalability by providing a set of parameterized rules for the composition of a (sub)system. Examples of template parameters are memory size, bus width, number and type of (co)processors, etc. Architecture templates have been widely deployed at SoC level [3][17][18]. However, subsystem-level templates are relatively unknown. Eclipse provides such a scalable architecture template at subsystem level (Section 3).

A key ingredient of such an architecture template is the infrastructure around the function modules, *e.g.* for routing data to and from communication buffers. The programmable infrastructure must be able to grow with the advance of IC technology, while reusing its function modules. Moreover, the infrastructure should be reusable over a varying number and type of function modules. Such scalability is generally accomplished by introducing a stable interface that separates the design of the function modules from the infrastructure and vice versa. Eclipse separates computation (CPU, coprocessors) from generic infrastructure through the task-level interface (Section 3.2).

A second aspect of scalability of an architecture is the autonomy of the function modules. Scalable architectures typically avoid complex centralized modules that control large parts of the architecture. For example, a coprocessor architecture where a single CPU synchronizes all coprocessors is not scalable as the interrupt rate will overload the CPU with an increasing number of coprocessors. In a more scalable solution, every coprocessor may control its own behavior without needing CPU support [4]. Thereto, all Eclipse coprocessors execute autonomously, without requiring CPU support for task scheduling or synchronizing access to the stream buffers. Thus, time-shared use of the coprocessors does not rely on run-time control by CPU software.

3. Eclipse architecture template

Eclipse exploits application-level parallelism by concurrently executing medium-grain functions in function-specific coprocessors and/or software executing on a media processor. Functions eligible for coprocessor implementation are those commonly encountered in media applications, such as the DCT transform used by decoders and encoders for *e.g.* JPEG, MPEG, and DV. Typically, the functions eligible for software implementation are specific for one application only—such as still-texture decoding in MPEG-4—or are likely to change as standards evolve. These medium-grain functions are linked at run-time into a Kahn-style application graph, using on-chip communication and data buffering. Figure 3 depicts how application tasks are mapped onto the coprocessors and/or software. Eclipse applications are specified as a set of tasks that communicate with each other through data *streams* with FIFO buffers allocated in shared on-chip memory. A stream connects the *output port* of a producing task and the *input port* of one or more consuming tasks. Contrary to fully hardwired SoC subsystems, the Eclipse infrastructure is programmable and provides the flexibility of configuring a given Eclipse instantiation for different applications graphs.

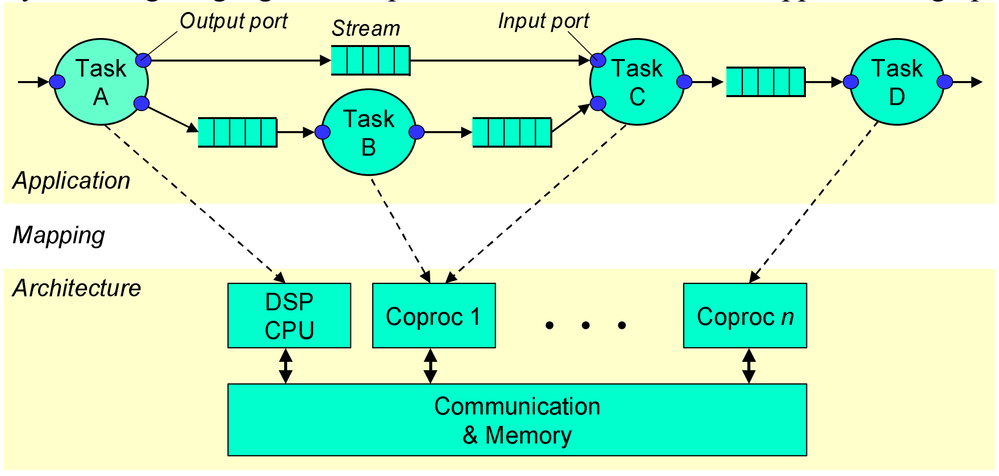


Figure 3. Application to architecture mapping example.

Eclipse coprocessors exploit the performance density of dedicated hardware function units and are only weakly programmable. All coprocessors run in parallel and execute their own thread of control. The coprocessors are multi-tasking to enhance the flexibility of configuring applications on Eclipse instances, *i.e.* each coprocessor can execute multiple Kahn tasks from a single Kahn network or from multiple and possibly different networks in a time-shared fashion. This way, application complexity is not restricted to the number of coprocessors in the architecture. Moreover, the programmable media processor can perform part of the application functionality whenever an application requires functionality beyond the implemented set of coprocessors.

The strong requirements on flexibility led us to design the Eclipse infrastructure with a centralized memory module where communication buffers can be allocated at run-time. For media processing, the streaming nature of the application functions gives a high spatial locality of reference. Eclipse instances exploit this to provide high data throughput (Gbytes per second) through deployment of a shared wide (*e.g.* 128 bits) bus in combination with communication buffers in a centralized, wide on-chip memory. The high synchronization rate is supported by a dedicated hardware implementation in the *coprocessor shell.*

3.1. Coprocessor shell

The communication network to transport and synchronize data between coprocessors may change over instances to match communication bandwidth requirements. Eclipse introduces the coprocessor shell to facilitate reuse of coprocessor designs over different Eclipse instances with different communication network characteristics [15]. 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.

The shells implement a uniform set of interface primitives towards the coprocessor with coprocessor-specific parameters, such as the width of the data path. While having such a customized interface towards the coprocessor, the shells have a uniform interface towards the communication hardware. This way, the shells allow reuse of coprocessor designs over different Eclipse instances with different communication network characteristics. Moreover, the architecture of the shell itself is designed as a parameterized template to facilitate reuse *within* an Eclipse instance. Shell instances with coprocessor-specific parameter settings are derived from this generic template. Examples of such parameters are the data width of the read and write interface between coprocessor and shell, or the size of data caches in the shell.

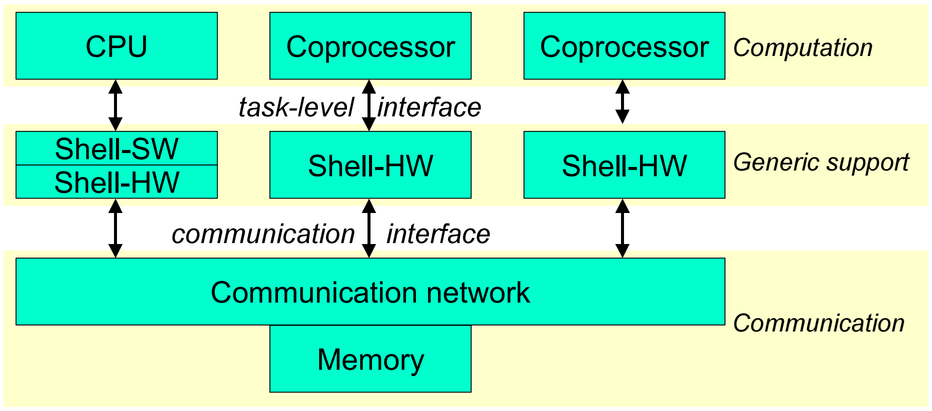


Figure 4. Coprocessor shell for system-level support.

Figure 4 depicts this hardware interface block that separates the computation hardware (coprocessors) from the communication hardware (buses, memory). The shells are distributed, such that each shell can be instantiated close to the coprocessor that it serves. While this article focuses on the coprocessor shell, the described concepts are directly applicable for the shell of the media processor. The media processor shell may implement parts of its functionality in software to increase flexibility and reduce hardware cost.

3.2. Task-level interface

Each coprocessor interacts with its shell through five generic interface primitives [14]. While specified in the form of software functions, Eclipse implements these primitives in hardware with an identical interface: a master-slave handshake with corresponding argument and result passing.

Figure 4 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 co-processor. The task_info value provides parameter values for the function the selected task should perform, *e.g.* one bit to select whether a forward or inverse DCT is to be performed.

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 )
```

for reading a number of bytes from the data stream connected to an input port, and

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

for writing a number of bytes to a data stream connected to an output port.

Designs that adopt the Kahn model typically synchronize on each read or write action. Instead, Eclipse advocates the separation of data I/O 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. The primitives for synchronizing access to data in the stream buffer are:

```
bool GetSpace(int task_id, int port_id, int n_bytes)
```

to *inquire* whether n_bytes valid bytes are available in the stream buffer for reading, or n_bytes of room are available for writing in the stream buffer. After reading or writing, the PutSpace primitive *commits* the number of consumed or produced bytes:

```
void PutSpace(int task_id, int port_id, int n_bytes).
```

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.

All task ports (Figure 3) map to one physical coprocessor-shell interface that handles Read, Write, GetSpace/PutSpace, and GetTask requests in parallel. The coprocessor is responsible for serializing simultaneous 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 or 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 about available data for reading, whereas a GetSpace on an output port inquires about available room for writing. Likewise, a PutSpace call on an input port commits empty room available for writing, while a PutSpace call 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 4, while Section 5 details the shell implementation. The 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. The coprocessor has the initiative for taking action; all five primitives are called by the coprocessor and implemented by the shell.

4. Eclipse computation architecture

As shown in Figure 4, the task-level interface separates computation architecture (coprocessors) from generic infrastructure (shells, buses, memory). This section describes the computation architecture design issues and shows how the task-level interface is used to build coprocessors. Subsequently, section 5 details the generic infrastructure to show how the task-level interface and associated services are implemented.

In the design of coprocessors, Kahn application models are gradually refined into task-level code that uses the task-level interface [15]. This code subsequently forms the starting point for the low-level coprocessor design. Eclipse coprocessors explicitly decide on the time instances at which they can switch the running task, thereby avoiding the hardware costs required for state saving at arbitrary points in time. 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 top-level 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
}
```

Before the task starts a read or write action, it first tests if there is sufficient data available for reading and sufficient room for writing via `GetSpace`. After a read or write action, the amount of generated data or room is committed to the shell via `PutSpace` calls. Section 4.1 details this process.

4.1. Synchronization of data access

From the view of a coprocessor task port, a data stream looks like an infinite tape of data, with a current ‘point of access’, as depicted in Figure 5. 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 at 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. The coprocessor is responsible for functionally correct behavior when using the interface primitives, *e.g.* the coprocessor must adhere to denied `GetSpace` requests, and not attempt to read or write data outside the window of granted space. Thus, when `GetSpace` fails, the coprocessor task cannot proceed and either the coprocessor can switch tasks or the task can keep on trying to proceed by repeatedly issuing `GetSpace` requests. Section 4.2 details these alternatives.

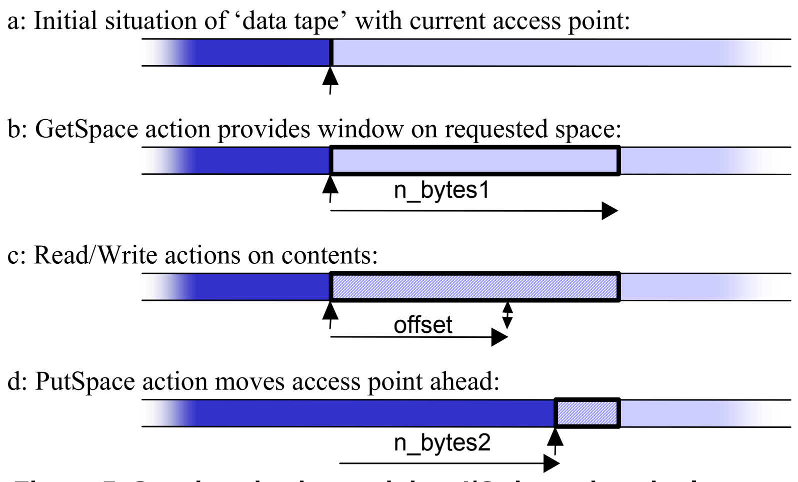

a: Initial situation of 'data tape' with current access point:
b: GetSpace action provides window on requested space:
n_bytes1
c: Read/Write actions on contents:
offset
d: PutSpace action moves access point ahead:
n_bytes2

Figure 5. 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 specified number of bytes ahead, in size constrained by the previously granted space.

4.2. Task switching

The Eclipse architecture supports multi-tasking, meaning that several application tasks may be mapped to a single coprocessor, as shown in Figure 3. New silicon technologies allow fast and efficient coprocessors that have sufficient computation speed for time-shared use. Support for multi-tasking is essential for achieving flexibility of the architecture towards configuring a range of applications and reapplying the same hardware coprocessors at different places in an application task graph. Multi-tasking on the Eclipse coprocessors is a shared responsibility between the coprocessor and the shell. The shell takes care of task scheduling, while the coprocessor is responsible for providing task switch points and saving and restoring the task state (if any) upon a task switch.

Eclipse coprocessors operate on a logical units 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, at the start of a processing step the coprocessors cannot always determine the required amount of space for completing the processing step. This is for instance the case when the coprocessor control has a data-dependent condition upon which it needs to 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.

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 previous task 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 postpones committing this read with `PutSpace(IN,…)` until a granted `GetSpace(IN2,…)` assures that the processing step can complete.

5. Eclipse generic infrastructure

We aim for a *generic* infrastructure that is used by all coprocessors. For a clean separation of generic infrastructure and computation, the infrastructure is deliberately does not interpret the data, whereas data is interpreted during processing. Communication hardware is a good candidate for being designed in such a uniform fashion [10], as it needs to be effective for all coprocessors. The shells maintain the application graph structure and implement a large part of this generic functionality, *i.e.* data transport, task scheduling, stream synchronization, and performance measurement support. Therefore, the shell is considered part of the generic infrastructure.

5.1. Stream synchronization

Communicating a stream of data requires a FIFO buffer (Figure 3), which in our case has a finite and constant size. It is pre-allocated in shared on-chip memory. The shell applies a cyclic addressing mechanism for proper FIFO behavior in the linear memory address range, using the `n_bytes` and `offset` arguments of the `Read/Write` calls in addition to the current access point and the buffer size. Figure 6 depicts the fixed size cyclic memory space used as FIFO.

The rotation arrow in the center of Figure 6 depicts the direction in which `GetSpace` calls confirm the granted window for `Read/Write`, which is the same direction in which `PutSpace` calls move the access points ahead. The small arrows denote the current access points of tasks *A* and *B*. In this example, *A* is a producer and hence leaves proper data behind, whereas *B* is a consumer and leaves empty space (*i.e.* already consumed data) behind. The shaded region ahead of each access point denotes the access window acquired through `GetSpace`.

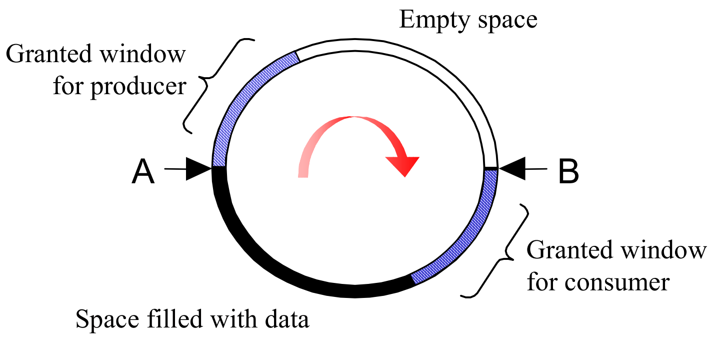


Figure 6. Basic stream mapped to a finite FIFO.

Each shell locally contains the configuration data for the streams that are incident with tasks mapped on its coprocessor and locally implements all the control logic to properly handle this configuration data. The shells implement a local *stream table* that contains a row of fields for each stream, or more precisely, for each access point. To handle the setup of Figure 6, the coprocessor shells of tasks *A* and *B* each contain one such row, holding the following fields:

- A *space* field containing a (maybe pessimistic) distance from its own point of access towards the other point of access in this buffer. The *space* value corresponds to the amount of available data for reading or the available room for writing;
- A *stream ID* denoting the remote shell with the task and port of the other point-of-access in this buffer.

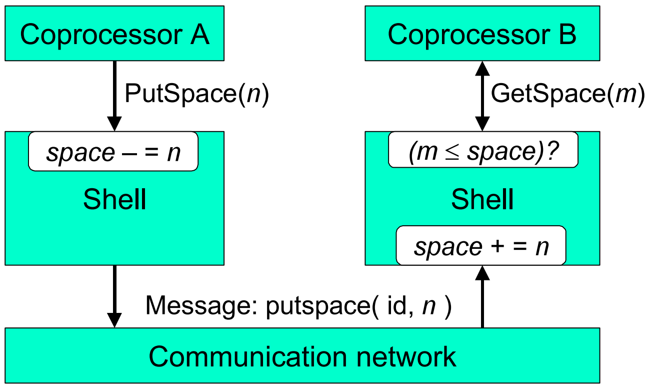


Figure 7. Updating local space values and sending putspace messages.

As shown in Figure 7, the shell of coprocessor *B* can answer a `GetSpace` request immediately by comparing the requested size *m* with the locally stored *space* value. When the shell of coprocessor *A* receives a `PutSpace` request, it locally decrements its space field with the indicated amount *n* and sends a ‘putspace’ message to the shell of coprocessor *B*. This remote shell holds the other point-of-access and increments its space field upon reception of such a ‘putspace’ message.

5.2. Data transport

Coprocessors transport all media data to and from their shell through the `Read` and `Write` primitives. The shells subsequently access corresponding locations in the shared stream buffers in on-chip memory. With the `Read` and `Write` primitives, the shell hides aspects such as the width of system data paths, data alignment in memory and cyclic buffer addressing, and data stream caching including coherency and prefetching control. Hereto, each stream entry in the stream table of Section 5.1 contains the current access point and the size of the stream buffer. Moreover, the shell incorporates separate read and write caches that play an important role in decoupling the coprocessor read and write ports from the global communication network.

The `GetSpace/PutSpace` synchronization mechanism explicitly controls cache coherency, fully transparent to the coprocessor. Using local `GetSpace` and `PutSpace` events for explicit cache coherency control results in a simple and efficient implementation in comparison with existing generic coherency mechanisms such as bus snooping. The cache-coherency mechanism builds on three key observations:

1. The access window, which is granted to a task port onto stream data, is guaranteed to be private. Thus, `Read/Write` operations in this area are safe and do not require intra-processor communication.
2. Local `GetSpace` requests extend the access window, obtaining new memory space in the cyclic buffer. Data in the cache that corresponds to this new memory space possibly needs invalidation. A subsequent `Read` action on such a cache location then results in a cache miss, upon which the cache loads fresh valid data from the cyclic buffer.
3. Local `PutSpace` requests reduce the access window, leaving new memory space to a successor in the cyclic buffer. Dirty data in the cache that corresponds to the memory space in the reduction interval needs to be flushed to the cyclic buffer to make the local data available for other processors. Sending the ‘putspace’ message to another coprocessor must be postponed until the cache flush is completed and safe ordering of memory operations can be guaranteed.

Apart from cache coherency, the shell also initiates stream prefetches upon local `GetSpace` and `Read` requests to reduce cache miss penalties.

5.3. Task scheduling

Clearly, multi-tasking implies the need for a task scheduler that decides which task any coprocessor must execute at which points in time to attain proper application progress. As Eclipse is targeted at irregular data-dependent stream processing and dynamic workloads, the scheduler must be effective for applications with dynamic workload such as to optimally utilize the Eclipse coprocessors. Therefore, task scheduling cannot be done off-line but is performed at run-time.

Eclipse targets relatively high performance, high data throughput applications with aggregate bandwidths in the order of GBytes per second. Due to the limited size for on-chip memory containing the stream FIFO buffers, high data synchronization and task switch rates are required. As the task switch rate is too high (10-100 kHz) for run-time scheduling in software, Eclipse implements task scheduling and synchronization in dedicated hardware as part of the coprocessor shell.

The parameterized shell template can be reused for each coprocessor shell. Therefore, the scheduling algorithm must be sufficiently simple to allow a cost-effective hardware implementation in each shell. On the other hand, the scheduling algorithm needs to be flexible enough to fit the needs of different coprocessors and applications. Autonomy of the coprocessors contributes to both scalability and cost-effectiveness. Therefore, task scheduling is distributed, where the task scheduler in each shell runs independent of task schedulers in other shells.

The target granularity for processing steps within the Eclipse architecture is in the range of 10—1000 clock cycles. Typically, the duration of a processing step is data dependent and can vary within this range. The number of processing steps needed to complete an application milestone (*e.g.* an MPEG frame), as well as the number of produced and consumed data items per processing step, may also be data dependent. The task scheduler must manage such highly data-dependent workloads in such a way that the coprocessor is used cost-effectively.
The task scheduler is based on round-robin style task selection as this can be efficiently implemented in hardware. The scheduler uses a weighted round-robin scheme, where the weights or *budgets* are configured as a guaranteed minimum number of cycles that a task may continuously execute, irrespective of the resource requirements of other tasks [13]. These budgets typically range from 1000 up to 10,000 clock cycles (10—100 processing steps). The tasks that are mapped onto the coprocessor are configured in the *task table* in the shell, which contains among others the resource budget per task.

The task scheduler cannot determine in advance whether a task can complete a processing step. Therefore, the scheduler performs a 'best guess' by considering both the available data and room in the stream buffers as well as previously denied data access (For details, see [13]). Section 5.1 shows that this information is locally available in the shell. Task scheduling with this 'best guess' strategy is effective by selecting the right tasks in the majority of the cases, and recover with a limited penalty otherwise.

5.4. Performance measurement support

Eclipse supports performance measurement (profiling) in hardware in the shells. Measurements include buffer filling, coprocessor utilization, data access latency, etc. These hardware measurements can be used for:

- Optimizing application behavior with given silicon when creating product applications;
- Run-time control for quality-of-service resource management [1] in the final product.

Measurement data is accumulated in the stream and task tables in the shell. This allows to collect performance measurement at application level, *i.e.* per task and per stream, instead of per coprocessor.

All shell tables are memory-mapped and accessible to the main CPU via a control bus (PI-bus). Thus, the main CPU can collect measurement data at regular time intervals, *e.g.* once per MPEG frame. However, accumulating a measurement every cycle for a complete MPEG frame requires a significant amount of memory in the shell. To reduce hardware costs of measurement support, a separate process in the shell takes measurement samples at regular intervals. With given memory size, the main CPU can balance the duration of these intervals with the duration of the total measurement.

6. Eclipse instance

Figure 8 depicts a first instantiation of the Eclipse template, to be deployed as an MPEG subsystem in SoC platforms aimed at high-definition television functionality, *e.g.* the Philips Nexperia line of chips for digital video [3]. This Eclipse instance targets decoding of two high-definition (HD) MPEG-2 streams simultaneously, or standard definition (SD) MPEG-2 encoding in parallel with decoding a number of SD MPEG-2 streams. Various combinations are possible, such as decoding one HD stream and decoding two SD streams in parallel, or transcoding for time-shift functionality. The CPU is responsible for configuring these applications at run-time by programming the stream and task tables in the shells through the PI-bus (not shown in Figure 8).

Figure 8 shows dedicated hardware units for MPEG processing. These coprocessors are multitasking and weakly programmable, *e.g.* the DCT coprocessor can time-share both the forward and inverse DCT functions of one or more MPEG encoding applications and the inverse DCT of one or more decoding applications. Equivalently, the RLSQ coprocessor performs the run-length decoding, inverse scan, and inverse quantization of the MPEG-2 decoding graph (Figure 2), as well as the encoding variant: quantization, zigzag scan and run-length encoding. The motion compensation/motion estimation (MC/ME) coprocessor has a dedicated connection to the system bus to access MPEG reference frames in off-chip memory. Similarly, the VLD coprocessor fetches the incoming compressed bit-streams from off-chip memory. Audio decoding, variable-length encoding, and de-multiplexing are executed in software on the media processor (DSP-CPU).

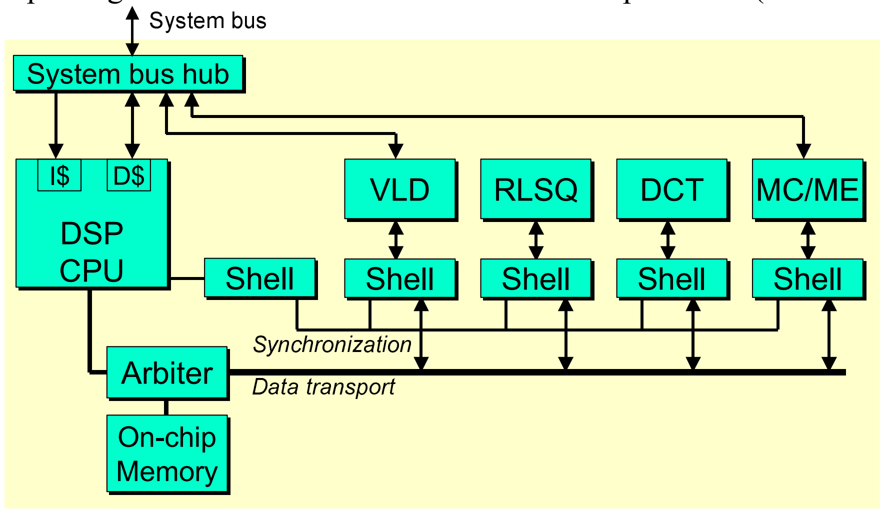


Figure 8. Eclipse instance for video coding.

The flexible connection of medium-grain functions requires a significant communication bandwidth from the system. For this instance, the targeted applications allow the use of a single on-chip memory (SRAM) for communication buffering with a wide data-path (128 bits) to provide the necessary bandwidth. For instances demanding a higher communication bandwidth, the architect must balance the flexibility of allocating buffers with configurable sizes in a centralized memory versus the scalability and performance of distributed memory implementations.

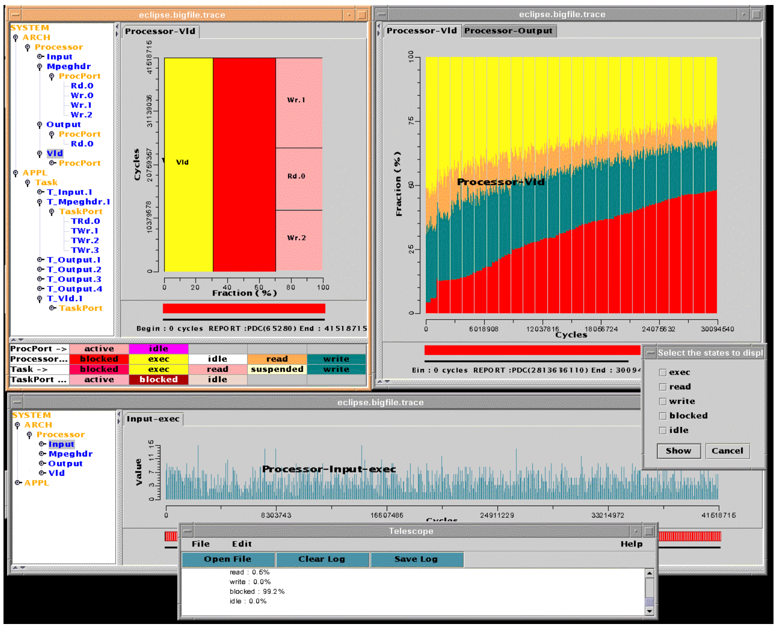
The computational performance for decoding two high-definition (HD) MPEG streams is roughly 36 Gops per second on mostly 16 bits data items. Our initial estimates indicate that the Eclipse instance of Figure 8 takes less than 7 mm^2 of silicon area in 0.18 micron CMOS technology. This includes 1.7 mm^2 for a 32 kB on-chip memory and 2.0 mm^2 for a programmable VLD coprocessor, but excludes the DSP-CPU. All coprocessors will be synthesized for operation at 150 MHz. The on-chip SRAM operates at 300 MHz needed to support separate read and write data buses, each running at 150 MHz. Total power consumption is estimated to be less than 240 mW for simultaneous decoding of two HD MPEG streams. Detailed analysis and design is currently in progress.

7. Eclipse simulation environment

At this moment in time, Eclipse exists only as a simulator model, supporting application execution and tuning for particular instances. The full architecture is modeled in a flexible cycle-accurate simulator, albeit at a high abstraction level. The simulation environment supports a design trajectory with gradual refinement of Kahn application models into cycle-accurate Eclipse coprocessor models. Thereto, the simulator supports mixed-level simulation at various levels of abstraction.

We developed the simulator as a *design* tool, similar to the approach of Hocevar et al. [6]. Functionally correct simulation models provide quantitative feedback and allow us to explore the design space of the Eclipse architecture before diving into gate-level design. To this end, the simulator is accompanied by graphical tools for configuring application graphs and visualizing the numerical results. Experiments include caching strategies in the shell (*e.g.* varying cache size, cache prefetching or not), bus latency and width, etc. Thereto, the simulator parses a setup file that contains these architectural parameters and collects measurement data such as the filling of communication buffers and the execution time of a coprocessor.

The most complex aspect of design-space exploration is to interpret the numerical results. Therefore, we developed new techniques for visualizing performance data of such multiprocessor architectures. Figure 9 gives an example output of an Eclipse simulation run. The viewer differentiates between architecture views (*e.g.* VLD coprocessor utilization) and application views (*e.g.* stream buffer filling, stall time of tasks). Note that the viewer is separated from the simulation environment, and can also be used to visualize the hardware measurements of Section 5.4.



```
Begin : 0 cycles  REPORT :PDC(6 5280) End : 41518715
Bin : 0 cycles  REPORT :PDC(28136 36110) End : 3009
```

ProcPort ->	active	idle			
Processor...	blocked	exec	idle	read	write
Task ->	blocked	exec	read	suspended	write
TaskPort ...	active	blocked	idle		

```
read : 0.5%
write : 0.0%
blocked : 99.2%
idle : 0.0%
```

Figure 9. Eclipse performance visualization example.

Figure 10 depicts the results of a simulation run early in the development of the first Eclipse instance. The figures show the available data in the stream buffers for the input of MPEG-decoding tasks running on the RLSQ, DCT, and MC coprocessors. The fluctuations in the diagrams clearly show the data-dependent behavior of MPEG-2 decoding. An MPEG sequence consists of *intra* frames (I-frames), *predicted* frames (P-frames), and *bi-directional* predicted frames (B-frames). The I-frames are coded using the data in the frame itself, whereas P-frames are coded using the data from the nearest previous I-frame or P-frame. To further exploit temporal redundancy, B-frames are coded using the data from the past as well as future I/P frames. Large variations in buffer filling correspond to the GOP sequence of MPEG-2 frames that served as input, *i.e.* the IPBBPBBP sequence of frames as shown in the top of the figure.

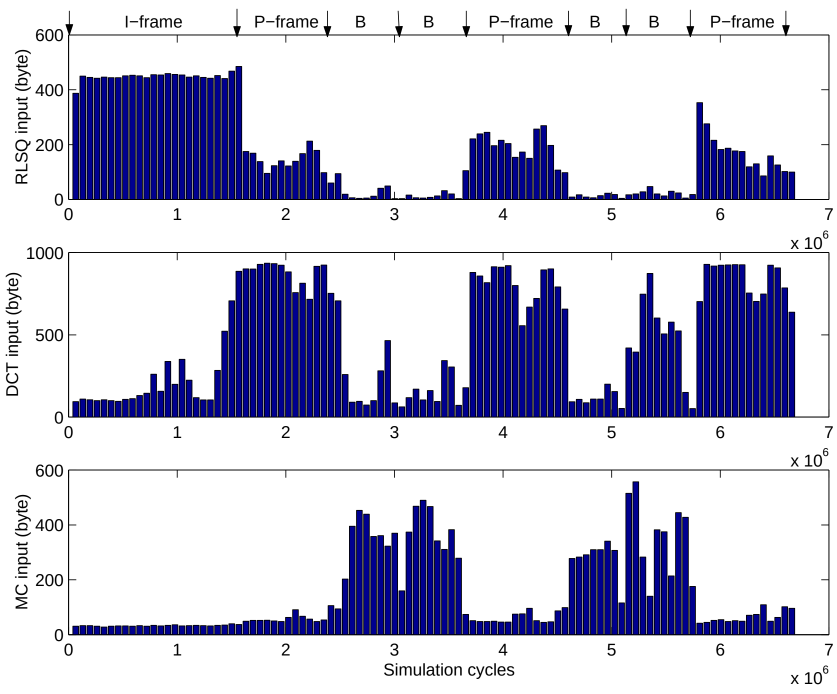


Figure 10. Available data for RLSQ, DCT, and MC input streams.

Furthermore, Figure 10 shows that for I-frames, the RLSQ task is the bottleneck in this application, as it cannot consume the available input data with the rate at which this data becomes available. Consequently, the DCT and MC tasks are waiting for input data from the RLSQ task. For P-frames, the bottleneck apparently shifts from the RLSQ task to the DCT task. For B-frames, the MC task needs to fetch macroblock data from both forward and backward prediction frames in external memory. This causes the bottleneck to shift to the MC task. Thus, the figure shows that the overall performance is constrained by a different task for each type of MPEG frame. Based on this feedback, we decided to increase performance by pipelining the DCT coprocessor [14] and improving the prefetching strategy of the data caches in the shell. The next step is to increase the average MC performance, *e.g.* by adding a cache to hide the latency of accessing prediction data in external memory.

8. Conclusion

Eclipse introduces a cost-effective and scalable template for SoC subsystems consisting of an adjustable mix of hardware and software modules. The targeted media applications combine real-time and dynamic behavior. The strict separation of application functionality from the generic interconnect structure was introduced at the start of design and adhered to in a rigorous fashion. The resulting uniform interface separates computation hardware, or *coprocessors*, from generic support for multi-tasking, synchronization, and data transport. This interface not only keeps coprocessor design simple, but also facilitates reuse of coprocessors over a set of media applications.

The interface copes with irregular and unpredictable application loads by separating the transport of streaming data from synchronizing the access to the data. The combined requirements of scalability and cost-effectiveness led us to a novel approach that features distributed scheduling and distributed synchronization with high task-switch and synchronization rates. These are supported by a generic hardware implementation, dedicated to each coprocessor. In general, a generic approach as followed in Eclipse generates additional overhead. However, according to our estimates, the resulting systems are still highly cost-effective. The design philosophy of separation of concerns proved to be essential to manage the otherwise daunting overall complexity.

Another aspect of our design philosophy was to guide our design trajectory by simulation results. To this end, a retargetable simulator of the Eclipse template was designed and implemented. The simulator provided quantitative feedback on the behavior of Eclipse early in the design phase. The Eclipse architecture has been explored in a first instantiation for simultaneous MPEG-2 encoding and decoding of multiple streams at various resolutions. Currently, we are studying extensions towards MPEG-4 functionality [16], such that a single Eclipse subsystem can support a programmable mix of MPEG-2 and MPEG-4 encoding and decoding applications.

References

[1] R.J. Bril et al., "Multimedia QoS in Consumer Terminals", *IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 332-344, Sept 2001, Antwerp, Belgium.

[2] V.M. Bove, Jr. and J.A. Watlington, "Cheops: A reconfigurable Data-flow System for Video Processing", *IEEE trans. On Circuits and Systems for Video Technology, vol. 5*, no. 2, pp. 140-149, April 1995.

[3] 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.

[4] 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.

[5] R.J. Grove, G.J. Hewlett, and D.B. Doherty, "The MVP: A Single-Chip Processor for Advanced Television Applications", *Proc. Int. Workshop on Signal Processing of HDTV,* vol. 6, pp. 479-487, Oct. 1994, Turin, Italy.

[6] D. Hocevar, S. Sriram, and C.Y. Hung, "A Performance Simulation Approach for MPEG Audio/Video Decoder Architectures", *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, vol. 6, pp. 203-206, June 1998, Monterey, CA, USA..

[7] E.G.T. Jaspers and P.H.N. de With, "Architecture of Embedded Video Processing in a Multimedia Chip-set", *Proc. of IEEE Int. Conf. on Image Proc,(ICIP 99)*, vol. 2, pp. 787-791, Oct. 1999, Kobe, Japan.

[8] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", *Proc. of Information Processing '74*, August 5-10, Stockholm, Sweden, North-Holland publ. Co., pp. 471-475, 1974.

[9] G. Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Programming", *Proc. of Information Processing '77*, North Holland publ., pp. 993-998, 1977.

[10] K. Keutzer et al., "System-Level Design: Orthogonalization of Concerns and Platform-Based Design", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523-1543, Dec 2000.

[11] E.A. de Kock et al., "YAPI: Application Modeling for Signal Processing Systems", Proc. 37th Design Automation Conf. (DAC), pp. 402-405, June 2000, Los Angeles, CA, USA.

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

[13] M.J. Rutten, J.T.J. van Eijndhoven, and E.J.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.

[14] M.J. Rutten, J.T.J. van Eijndhoven, and E.J.D. Pol, "Design of Multi-Tasking Coprocessor Control for Eclipse", *10th Int. Symp. On Hardware/Software Codesign (CODES)*, May 2002, Estes Park, CO, USA.

[15] M.J. Rutten, O.P. Gangwal, J. van Eijndhoven, E. Jaspers, and E.J. Pol, "Hardware/Software Codesign of Reusable MPEG Coprocessors for Eclipse", *Int. Conf. on Computer Design (ICCD)*, Sept 2002, Freiburg, Germany, *submitted for publication.*

[16] E.B. van der Tol and E.G.T. Jaspers, "Mapping of MPEG-4 decoding on a flexible architecture platform", *Media Processors 2002,* vol. 4674, Jan. 2002, San Jose, CA, USA.

[17] D. Wingard and A. Kurosawa, "Integration Architecture for System-on-a-Chip Design", *Proc. of the IEEE 1998 Custom Circuits Conf.*, pp. 85-88, May 1998.

[18] D.C. Wyland, "Media Processors Using a New Microsystem Architecture Designed for the Internet Era", *Proc. of the SPIE: Media Processors 2000*, vol. 3970, pp. 2-15, Jan. 2000.