Workshop on Parallel and Distributed Computing
in Image Processing, Video Processing, and Multimedia (PDVIM'02)
April 15, 2002, Fort Lauderdale, Florida, USA

Eclipse: 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 Eindhoven (PRLE)
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
martijn.rutten@philips.com

Abstract

Eclipse is a heterogeneous multiprocessor architecture for high-performance media processing, including high-definition MPEG encoding/decoding. The scalable architecture framework concurrently executes media processing kernels in function-specific multi-tasking coprocessors and a media processor, communicating via on-chip memory. Eclipse instances combine application configuration flexibility with the efficiency of function-specific hardware.

1. Introduction

Consumer audio/video systems are becoming increasingly flexible to accommodate a variety of applications in a wide range of products. Targeted media applications include high-definition digital television with time shift (*e.g.* a set-top box with pause button) functionality, 3D games, video conferencing, or MPEG-4 like interactivity. The required set of applications and their format varies per product, per country, and over time as standards evolve.

Managing complexity, design cost, and time-to-market of such systems requires a generic and scalable media processing platform that enables simultaneous execution of very diverse tasks, such as high-throughput stream-oriented data processing and highly data-dependent irregular processing with complex control flows. In this paper, we propose the Eclipse architecture template as a flexible and cost-effective, high-performance media processing subsystem of such a platform. The key innovation of Eclipse is that it allows an adjustable mix of programmable and hardwired functions within a single subsystem.

Section 2 describes the targeted application domain and introduces the model of computation of the Eclipse architecture. Section 3 subsequently describes the main architectural trade-offs that influence the design of a media processing architecture. This section serves as a rationale for the description of the Eclipse architecture template in Section 4. In Section 5, we show a first instance of the Eclipse template that we used to validate the concepts outlined in this paper.

2. Media processing applications

Eclipse targets high-performance, data-dependent media processing, such as high-definition MPEG-2 decoding and MPEG-4 decoding for digital TV broadcast, as well as standard definition MPEG-2 encoding and transcoding for time-shift applications. The characteristics of this application domain influence the architectural design choices. This section sketches the characteristics of the Eclipse application domain as a basis for the architecture considerations of Section 3.

We specify 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 [1] introduced a formal model of such applications already in 1974, followed by an operational description by Kahn and MacQueen [2] in 1977. This formal model is now commonly referred to as a *Kahn Process Network* (KPN), and defines the model of computation of the Eclipse architecture. Fig. 1 shows such a network for MPEG-2 decoding.

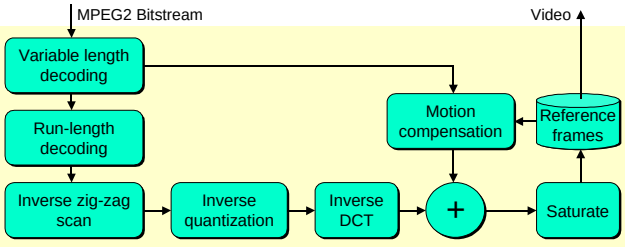

Fig. 1. MPEG-2 decoder process network.

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 the 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 scheduling of the tasks.

One of the most important strengths of this model is the inherent building block nature. Once a set of basic functions is defined as tasks, a multitude of applications can be configured by instantiating tasks and connecting them in a graph structure. Kahn process networks are used in a variety of other environments. Kienhuis [3] describes a systematic approach to create process networks from a sequential specification. Ptolemy [4] provides an integrated system for specification and analysis of, among others, such process networks.

3. 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 [5]-[7]. The following subsections elaborate on the main architectural trade-offs that influence these parameters as the foundation for the design choices of the Eclipse architecture template. The actual design decisions of Eclipse are subject of Section 4.

3.1. Performance and granularity of parallelism

Media processing applications exhibit different forms of parallelism that can be exploited to increase the computational performance of programmable digital signal processors [8]. The system architect has a choice how to exploit the implicitly present parallelism in the applications. The exploitation techniques are not mutually exclusive but can co-exist in a single architecture. Techniques range from exploiting instruction and data level parallelism in superscalar and VLIW architectures to the task level parallelism of multi-processor and coprocessor architectures.

For instance, a time-shift recorder function consists of an encoding and decoding function which may be executed in parallel. Providing separate encoder and decoder hardware exploits this parallelism at a course granularity. Within the decoder (Fig. 1) there are many medium-grain tasks that in principle can proceed in parallel, *e.g.* the discrete cosine transform (DCT) and quantization tasks. This parallelism can be exploited by a number of parallel units that execute these tasks simultaneously. Within such a DCT task, many operations can execute in parallel. This parallelism is called instruction-level parallelism (ILP), and can be addressed for example in a VLIW architecture [9]-[12].

3.2. Performance and granularity of synchronization

Parallel Kahn tasks communicate data through communication buffers (FIFOs). Buffer requirements relate to the regularity of processing in the application domain versus the coupling of timing of the functions. Regular tasks, such as in linear video filtering where the worst-case communication requirements equal the average case, allow a tight coupling with minimal buffering [13][14]. Irregular tasks demand less tight coupling to allow individual progress, 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; the function itself is regular, but the number of DCT coded blocks to be processed varies per MPEG frame.

Communication requires both data transport and synchronization. Producers and consumers exchange information on the amount of produced or consumed data in the buffer that is available for consumption respectively for production. 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 channel. 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 to enhance parallelism and reuse increases the number of streams passing through the communication network. Thereby, both communication buffering and bandwidth requirements increase. Finer grain synchronization of data access to communication buffers (*e.g.* macroblock level in MPEG) reduces these communication buffer requirements per stream, at the expense of a higher synchronization rate. This aids in coping with the increased bandwidth requirements by allowing to keep data streams on-chip. On-chip communication buffers allow a dedicated communication network optimized for the type of data access in the application domain. For media processing, the streaming nature of the application functions gives a high spatial locality of reference, allowing deployment of a shared wide (*e.g.* 128 bits) bus in combination with communication buffers in a centralized, wide on-chip memory.

3.3. Flexibility and programmability

Complex chip designs may implement their functionality fully in hardware, fully in software, or in a mixture of hardware and software. An example of a complex hardwired chip is the Melzonic IC [15], implementing picture improvement functionality. Members of the Intel Pentium line of processors are prime examples of a fully programmable design. In the field of consumer electronics, design constraints, such as cost (including power consumption),

performance, and flexibility, are weighted differently. Consumer media processors approach the flexibility of the processors in the PC market, but require an order of magnitude lower cost prize in combination with a significantly higher performance for the media-processing application domain. For this reason, complex consumer electronics chips target a mixture of hardware and software, leading to the realm of Systems-on-a-Chip (SoC) [16]-[18]. These SoCs consist of interconnected subsystems, each with their own specific purpose. Currently, subsystems are either hardwired or fully programmable. Typical subsystem examples are a dedicated MPEG decoder, respectively a programmable embedded core. In contrast, Eclipse introduces an adjustable mix of programmable and hardwired functions in a single subsystem (Section 4).

3.4. Flexibility and reuse

Infrastructure flexibility is a trade-off of performance density (*i.e.* the application throughput per unit area and per unit power) versus development cost. A hardwired MPEG-2 decoder will be more efficient in performance, area, and power consumption than an equivalent MPEG decoder implemented as a network of coprocessors connected via a bus to shared memory. Moreover, 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 single function architecture. However, once the flexible architecture with associated software development environment is available, the time-to-market of a new software-based application can be very fast.

If the coprocessors in the above example are sufficiently generic, the flexible infrastructure will allow reuse of the coprocessors for other applications in the same domain, such as MPEG-2 encoding. This is clearly exploited by the DSPs in the MVP architecture [19]. This type of flexibility provides both the reuse of an existing hardware platform for newly defined functions or evolving standards, as well as the reuse of hardware by time-sharing a single coprocessor over a set of similar functions.

The design of dedicated coprocessors that can nevertheless be reused over a set of applications heavily relies on the presence of common application functions in the targeted application domain that allow such a dedicated hardware implementation. For instance, MPEG-2 decoding and encoding applications can reuse a single inverse DCT implementation. Moreover, the forward and inverse DCT functions of an MPEG-2 encoding application are sufficiently similar to execute on a single dedicated DCT coprocessor through time-sharing. Thus, a time-shift recording application can reuse a single DCT coprocessor three times.

3.5. Design cost and scalability

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 support 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 [16]-[18]. However, subsystem-level templates are relatively unknown. Eclipse provides such a template at subsystem level (Section 4).

The programmable infrastructure must be able to grow with the advance of IC technology, while reusing its elementary functions. Moreover, the infrastructure should be reusable over a varying number and type of elementary functions. Such scalability is generally accomplished by separating the design of the elementary functions from the infrastructure and vice versa through a stable interface.

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 [20]. In a more scalable solution, every coprocessor may control its own behavior without needing CPU support.

4. Eclipse architecture template

The Eclipse architecture supports the implementation of medium-grain functions in function-specific coprocessors and/or CPU software executing on a media processor (*e.g.* the TriMedia VLIW [9][10]). 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. These medium-grain functions are linked at run-time into a Kahn-style application, using on-chip communication and data buffering. Fig. 2 indicates such a mapping of Kahn tasks onto coprocessors and/or CPU software.

Eclipse coprocessors are dedicated hardware function units which are only weakly programmable. All coprocessors run in parallel and execute their own thread of control. The coprocessors allow multi-tasking, *i.e.* each coprocessor concurrently supports multiple Kahn tasks from a single Kahn network or from multiple and possibly different networks. Time-shared use of the coprocessors does not rely on run-time control by CPU software. CPU software however, is responsible for configuring the application networks.

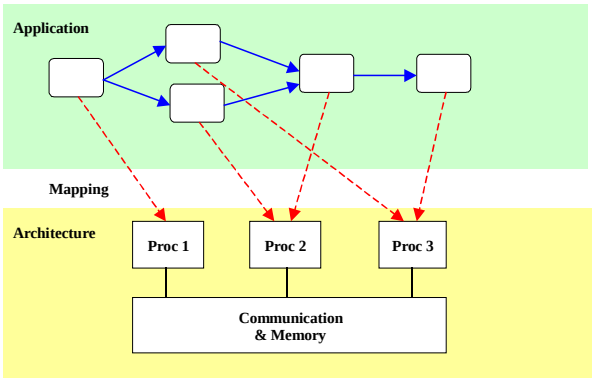

Fig. 2. Application to architecture mapping.

4.1. Coprocessor shell

Eclipse is an architecture template at subsystem level, supporting a mix of hardware and software. The scalable Eclipse architecture allows any set of coprocessors to be included in each targeted instance. More specifically, the number of coprocessors may vary over instances. The template can scale from run-time configurable dedicated hardware that only needs configuration support from a control processor somewhere else in the system, up to instances that support mainly software functions with limited hardware acceleration.

To effectively separate communication hardware (buses, memories) and computation hardware (coprocessors), Eclipse introduces the coprocessor *shell*. Fig. 3 depicts this interface block between coprocessors and the communication hardware. 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 real application functionality.

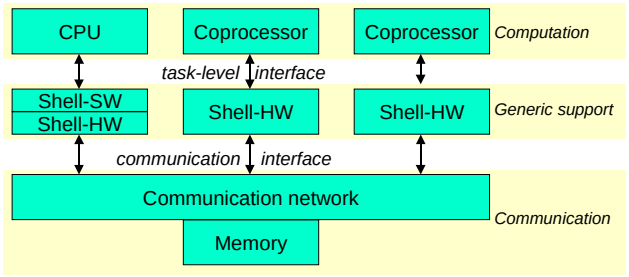

Fig. 3. Coprocessor shell interfacing computation and communication.

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 interface primitives:

`int` **`GetTask`**`(…)` for multi-tasking; the following primitives for data communication:

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

and the primitives for data synchronization:

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

Although these primitives are generic and simplify coprocessor design, the five interface primitives allow complex coprocessor control to cope with for instance data dependent I/O, variable packet sizes, error recovery, and pipelined processing. In all cases the coprocessor has the initiative for taking action; all primitives are called by the coprocessor and implemented by the shell.

While having a customized (data width) 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.

4.2. Multi-tasking

The Eclipse architecture supports multi-tasking, meaning that several application tasks may be mapped to a single coprocessor, as shown previously in Fig. 2. 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.

Clearly, multi-tasking implies the need for a task scheduler that decides which task any coprocessor must execute at which points in time to obtain 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 explained in Section 3.2. 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.

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 within the generic shell template. 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.

Unit of execution. The coprocessor explicitly decides on the time instances during task execution at which it can switch the running task. This way, the hardware architecture does not need provisions for saving context at arbitrary points in time. The coprocessor can continue processing up to a point where it has little or no state. These are the moments at which the coprocessor can perform a task switch most easily.

At such moments, the coprocessor asks the shell for which task it should perform work next. This inquiry is done through the `GetTask` primitive. The return value is a task ID, a small non-negative number that identifies the new task context. The intervals between such inquiries are by definition denoted as *processing steps*. Generally, a processing step involves reading in one or more packets of data, performing some operations on the acquired data, and writing out one or more packets of data.

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 data-dependent workload in such a way that the coprocessor is used cost-effectively.

Budget-based round-robin. 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 [21]. 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. Fig. 4 depicts how the task scheduler selects a new task from the task table upon a `GetTask` request from the coprocessor:

1. *Active task*. The active task may continue if there is more work to do for the active task, *i.e.* the task is 'runnable', and if it still has sufficient budget (as identified by the 'running budget' variable);
2. *Next task*. If the active task cannot continue, the scheduler selects the next runnable task from the task table in round-robin order.

The running budget is discarded when the active task blocks on communication. The next task starts immediately when the blocking task returns to the scheduler. This way, tasks with sufficient workload can use the excess computation time by spending their budget more often.

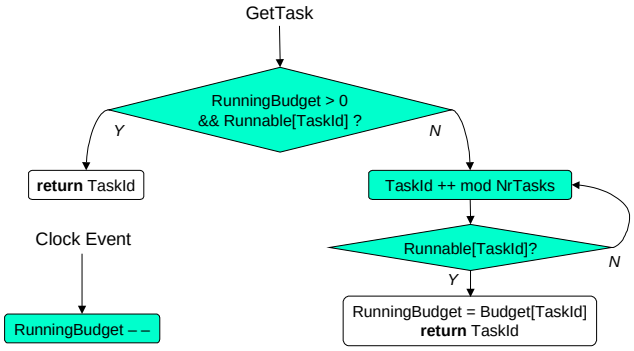

Fig. 4. Task scheduling algorithm.

The shell, including its task scheduler, does not interpret the media data and has no notion of data packets. Data packet sizes may vary per task and packet size can be data dependent. Thus, the task scheduler cannot determine in advance whether a task can complete a processing step. Therefore, the runnable test of Fig. 4 provides a 'best guess' by considering both the available data and room in the stream buffers as well as on denied data access (For details, see [21]). Section 4.3 shows that this information is locally available in the shell. Task scheduling with this runnable test can be effective by selecting the right tasks in the majority of the cases, and recover with a limited penalty otherwise.

4.3. Stream synchronization

From the coprocessor point of view, 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 to access a certain data space ahead of this current point of access. Here, data space signifies available data for reading from an input data stream, respectively available room for writing data to an output stream. If the shell grants permission, the coprocessor can perform `Read` or `Write` actions inside its requested space, with variable-length data (through the `n_bytes` argument), and on random access positions (through the `offset` argument). If the shell does not grant permission, the `GetSpace` call returns `false`. After one or more `GetSpace` calls—and optionally several `Read`/`Write` actions—the coprocessor can decide it is finished with proc-

essing (some part of) the data space 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. Fig. 5 depicts this process.

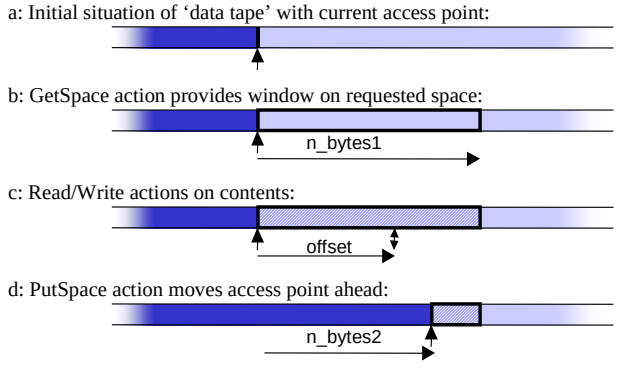

Fig. 5. Synchronization and data I/O through a single port.

Communicating a stream of data requires a FIFO buffer, 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. Fig. 6 depicts the fixed size cyclic memory space used as FIFO.

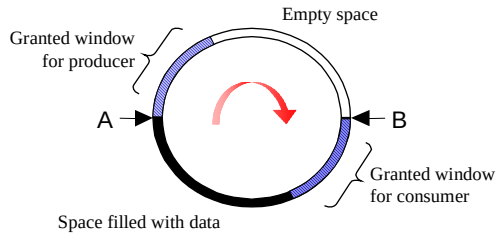

Fig. 6. Basic stream mapped to a finite FIFO.

The rotation arrow in the center of Fig. 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`.

Tasks *A* and *B* may proceed at different speeds, and/or may not be serviced for some period in time while other tasks execute on the multi-tasking coprocessors. The shells provide sufficient information to maintain the respective ordering of the access points *A* and *B*, or more strictly, to ensure that the granted access windows never overlap. It is the responsibility of the coprocessors to adhere to the information provided by the shell, thereby maintaining overall functional correctness. For example, the shell may sometimes answer `false` on `GetSpace` requests from the coprocessor, due to insufficient available space in the buffer. The coprocessor should then honor the denied request for access, and refrain from issuing `Read/Write` calls on the requested space. If `GetSpace` returns `false`, the coprocessor is free to decide on how to react. Possibilities are:

- Try a new `GetSpace` with a smaller `n_bytes` argument.
- Wait for a moment and then try again.
- Quit the current task and allow another task on this coprocessor to proceed by calling `GetTask`.

This allows the decision for task switching to depend upon the expected arrival time of more data and the amount of internally accumulated state with associated state saving cost. For non-programmable dedicated hardware coprocessors, this decision is part of their architectural design process.

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 Fig. 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 corresponds to the amount of available data for reading or the available room in the buffer for writing;
- An ID denoting the remote shell with the task and port of the other point-of-access in this buffer.

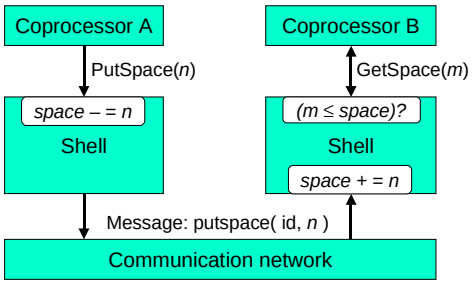

Fig. 7. Updating local space values and sending putspace messages.

As shown in Fig. 7, the shell can answer a `GetSpace` request immediately and locally 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.

4.4. Data transport

The coprocessors transport all media data to and from their shells through the `Read` and `Write` primitives. The shells subsequently access data 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 and cyclic buffer addressing, and data stream caching including coherency and prefetching control. Hereto, each stream entry in the stream table of Section 4.3 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 uncoupling 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 penalty. The memory space difference between the `n_bytes` arguments of subsequent `GetSpace` calls—as outlined in observation 2—is sufficient for functional correct invalidation and prefetching behavior. However, the valid memory space in the stream buffer is typically larger than the number of bytes asked for by `GetSpace` requests. This valid memory space is also available locally in the shells through the `Space` field in the stream table. The Eclipse shells use this field to further reduce the number of invalidates and extend the region from which valid data can be prefetched.

5. Eclipse instance

Fig. 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 [16]. 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 four 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.

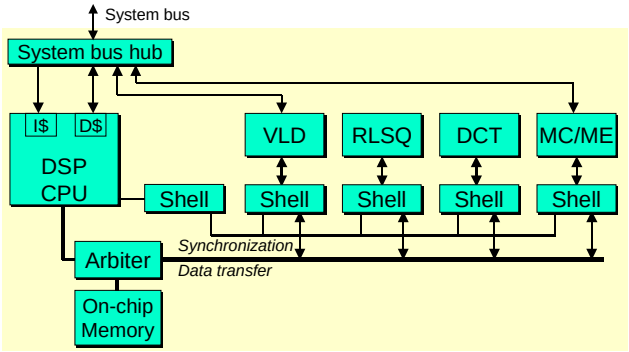

Fig. 8. Eclipse instance for video coding.

The figure shows dedicated hardware units for MPEG processing. These coprocessors are multi-tasking 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, 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 CPU.

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 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 full architecture is simulated in a flexible cycle-accurate simulator, accompanied by graphical tools for configuring application graphs and analyzing system behavior.

6. Conclusion

Eclipse introduces a cost-effective and scalable template for subsystems consisting of an adjustable mix of hardware and software modules. It allows reuse of computation hardware over a set of media applications that combine real-time and dynamic behavior. These characteristics are achieved by a novel approach which combines distributed multi-tasking and distributed synchronization. A scalable hardware implementation supports high task switch and synchronization rates.

These concepts have been explored in a first instantiation of the Eclipse template for simultaneous MPEG-2 encoding and decoding of multiple streams at various resolutions. Currently, we are studying extensions towards MPEG-4 and 3D graphics functionality [22], such that a single Eclipse subsystem can support a programmable mix of MPEG-2, MPEG-4 and 3D graphics applications in a system-on-silicon platform.

References

[1] 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.

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

[3] B. Kienhuis, E. Rijpkema, and E.F. Deprettere, "Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures", *8th Int. Workshop on Hardware/Software Codesign (CODES)*, May 3-5, 2000, San Diego, CA, USA.

[4] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development", vol. 4, pp. 155-182, April 1994.

[5] S. Panchanatan, O. Fatemi, and T.M. Le, "Survey of Architectures for Media Processing", *SPIE Conf. on Input/Output and Imaging Technologies*, vol. 3422, pp.199-209, July 1998, Taipei, Taiwan.

[6] P. Pirsch and H.J. Stolberg, "Implementation of Media Processors", *IEEE Signal Processing Magazine*, vol. 14, no. 4, pp. 48-51, July 1997.

[7] I. Kuroda and T. Nishitani, "Multimedia Processors", *Proc. of the IEEE*, vol. 86, no. 6, pp. 1203-1221, June 1998.

[8] H. Liao and A. Wolfe, "Available Parallelism in Video Applications," *30th Annual Int. Symposium on Microarchitecture*, Dec. 1997.

[9] S. Rathnam and G. Slavenburg, "Processing the New World of Interactive Media", *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 108-117, March 1998.

[10] J.T.J. van Eijndhoven, *et al*., "TriMedia CPU64 Architecture", *Proc. Int. Conf. on Computer Design (ICCD '99)*, Austin, Texas, pp. 586-592, October 10-13, 1999.

[11] 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.

[12] P. Kalapathy, "Hardware/Software Interactions on Mpact", *IEEE Micro*, vol. 17, no. 2, pp. 20-26, March/April 1997.

[13] 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.

[14] 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.

[15] G. de-Haan, J. Kettenis, A. Loning, and B. De-Loore, "IC for motion-compensated 100 Hz TV with natural-motion movie-mode", *IEEE Trans. on Consumer Electronics*, vol. 42, no. 2, pp. 165-174, May 1996.

[16] 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, September-Oct. 2001.

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

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

[20] 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, Montréal, Canada, Oct. 2001*.

[21] M.J. Rutten, J.T.J. van Eijndhoven, E.J.D. Pol, "Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors", *Euromicro Conf. on Real-Time Systems*, Vienna, Austria, submitted for publication.

[22] 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.