Euromicro Conference on Real-Time Systems
Vienna, Austria, 19th—21th June, 2002.
http://www.idt.mdh.se/ecrts02/

Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors

Martijn J. Rutten
Philips Research Laboratories

Jos T.J. van Eijndhoven
Philips Research Laboratories

Evert-Jan D. Pol
Philips Semiconductors

Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
{martijn.rutten, jos.van.eijndhoven, evert-jan.pol}@philips.com

Abstract

Eclipse defines a heterogeneous multiprocessor architecture for high-performance streaming media as a subsystem of a system-on-silicon platform for the consumer electronics market. The scalable architecture template supports multiple function-specific coprocessors that operate in parallel and independently. Each coprocessor is multi-tasking, allowing multiple applications to proceed concurrently. Eclipse instances combine application configuration flexibility with the efficiency of function-specific hardware.

The Eclipse template introduces novel hardware units, called 'shells', dedicated to each coprocessor. The combination of limited available buffer memory and high data-bandwidth causes high task-switch rates and synchronization rates, necessitating full support by the shell. Thereto, each shell implements a task scheduler and a transport synchronization unit. The task scheduler is designed for a dynamic workload environment with guarantees for minimum resource budgets, and achieves online task selection within 10 clock cycles.

1. Introduction

Consumer audio/video appliances become increasingly open and flexible to accommodate a variety of applications. These media applications include high-definition digital television with time shift functionality (*e.g.* a set-top box with pause button), 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 resource-constrained appliances requires a generic and scalable media processing platform that can be deployed in a wide range of products. Such a platform must support simultaneous execution of very diverse tasks, such as high-throughput stream-oriented data processing and highly data-dependent irregular processing with complex control flows.

Currently, many vendors enter the market with platforms that address these issues to some extent (*e.g.* OMAP [1], PrimeXSys [2]). Philips Electronics has been developing such a platform concept for many years with instances such as the Viper system on silicon [3]. Such complex media processing chips contain sophisticated hardwired function modules that implement critical parts of the targeted media applications. Up to this point in time, these function modules could not be reused over different applications. The same complex media processing chips also contain multiple programmable cores, so as to endow the system with a high level of flexibility. However, these highly reusable programmable cores have a significantly lower performance and correspondingly higher power consumption as compared to the non-reusable hardwired coprocessors.

We developed the Eclipse architecture [4] as a template for designing flexible yet cost-effective function modules. Eclipse instances combine application configuration flexibility with the efficiency of function-specific hardwired modules. Cost-effectiveness is achieved by introducing high levels of parallelism and multi-tasking, while scalability is achieved by avoiding centralized control in the system.

Section 2 describes the Eclipse architecture template and the system level support provided by the Eclipse 'coprocessor shells'. Section 3 introduces distributed multi-tasking along with robustness considerations for time-sharing compute resources with a limited capacity. This paves the way for the implementation of scheduling streaming media tasks in Section 4.

2. Eclipse architecture

The model of computation of Eclipse is based on *Kahn Process Networks* [5][6], in which media processing applications are specified 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. The data streams in the network are buffered. Each buffer is a FIFO, with precisely one producer and one or more consumers. Due to this buffering, the producer and consumers do not need to mutually synchronize individual read and write actions on the channel. Reading from a channel with insufficient data avail-

able causes the consuming task to stall. Kahn [5] 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.

Eclipse supports the implementation of medium grain functions in function specific coprocessors or CPU software executing on a media processor (*e.g.* the TriMedia VLIW [3]). This allows a trade-off in the architecture for software flexibility versus the efficiency (power, area) of function-specific hardware. Functions eligible for coprocessor implementation are those commonly encountered in media applications, such as the discrete cosine transform (DCT) used by encoders and decoders 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. 1 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. Exploiting this thread-level parallelism in complex applications is required to obtain sufficient application throughput, even in conjunction with instruction-level or SIMD data-level parallelism inside the coprocessor.

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.

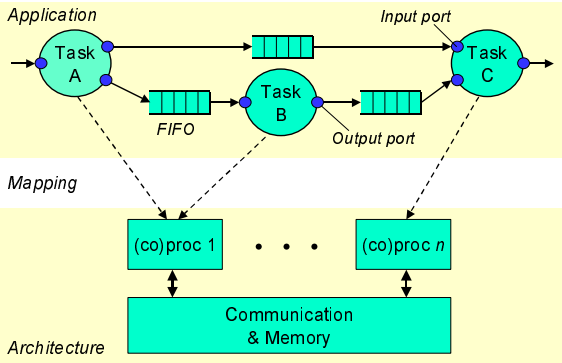

Fig. 1. Application to architecture mapping.

In this paper, we discuss the distributed multi-tasking support to execute multiple Kahn tasks on the Eclipse coprocessors. Although we focus on the scheduling of Eclipse coprocessors, the presented concepts are directly applicable to scheduling Kahn tasks that are mapped on the media processor.

2.1. Shell services

To effectively separate communication hardware (buses, memory) and computation hardware (coprocessors), Eclipse introduces the coprocessor *shell*. Fig. 2 depicts this interface block between coprocessors and the communication hardware. The shells have a uniform interface towards the communication hardware, while having a customized interface towards the coprocessor. 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.

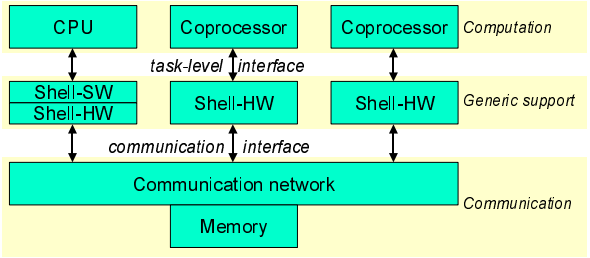

Fig. 2. Coprocessor shell interfacing computation and communication.

Basically, the shell offers operating system like services to each Eclipse coprocessor. The shell provides the capability for stream synchronization, data transport, and task switching through five interface primitives. In all cases the coprocessor has the initiative for taking action; all these primitives are called by the coprocessor and implemented by the shell. The five interface primitives are:

- `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 );`

for (media) data communication, and

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

for data synchronization, and

- `int GetTask(...);`

for task switching. The following section details the use of the communication and synchronization primitives, while the `GetTask` primitive is presented in Section 4.2.

2.2. Data transport and 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, and available room for writing data to an output stream respectively. 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 processing (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. 3 depicts this process.

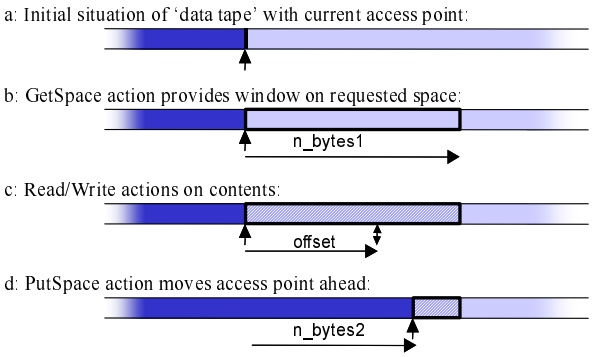

Fig. 3. Synchronization and data I/O through a single port.

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. 4 depicts the fixed size cyclic memory space used as FIFO.

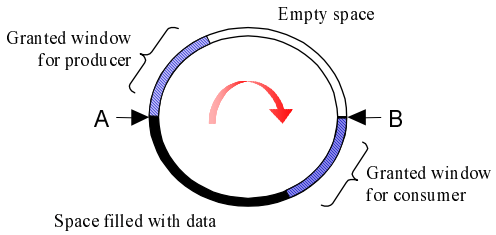

Fig. 4. Basic stream mapped to a finite FIFO.

The rotation arrow in the center of Fig. 4 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 is *A* 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 periods in time while other tasks execute on the multi-tasking coprocessors. The shells provide the coprocessors on which *A* and *B* run with information to ensure that the access points of *A* and *B* maintain their respective ordering, or more strictly, 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.

The Eclipse shells are distributed, such that each shell can be instantiated close to the coprocessor that it serves. 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 data. This means that each shell contains 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. 4, 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.
- A *stream ID* denoting the remote shell with the task and port of the other point-of-access in this buffer.

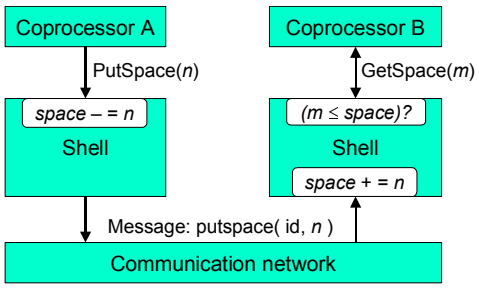

Fig. 5. Local space values and putspace messages.

As depicted in Fig. 5, the shell of coprocessor *B* can answer a `GetSpace` request immediately and locally by comparing the requested size *m* with the locally stored *space* value. Upon a `PutSpace` call, the local shell of coprocessor *A* 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* value there.

If `GetSpace` returns `false`, the coprocessor is free to decide on how to continue. 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.

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 [7].

3. Distributed multi-tasking

The Eclipse architecture supports multi-tasking in each coprocessor, *i.e.* several application tasks may be mapped to a single coprocessor, as shown previously in Fig. 1. This section defines the architectural concepts that are essential for the definition of such multi-tasking coprocessors.

3.1. Robust resource-sharing

Support for multi-tasking is essential in 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. We believe that new silicon technologies allow fast and efficient coprocessors that have sufficient computation speed for timeshared use.

The requirements for Eclipse are derived from the characteristics of the application domain of high-end media processing. This implies many applications running concurrently, each with a high degree of irregularity. In practice, the ratio of worst-case versus average-case load can be as high as a factor of 10. Under these circumstances, architecting the system for worst case scenarios is not competitive.

The direct consequence of running more applications than that the system can handle in worst case situations is that the system must be able to handle temporary overload situations in a robust way. Eclipse applications are created by instantiating appropriate tasks on multi-tasking coprocessors. Robustness is implemented by assigning each task a guaranteed minimum of compute resources, which we term as a *budget contract* [8]. The budget is set at some level between average and worst-case task resource requirements, which means that usually the task has sufficient resources available and sometimes not. The contract guarantees that each task can use the resources that are assigned to it by its contract, independent of the possible excess resource requirements of other tasks. Thus, system robustness is implemented by endowing the system a separation of concerns with respect to resource assignment per task. This separation of concerns is termed *justice*.

Usually, temporary overload occurs only for one or few applications concurrently, while others remain at or below average. Per task budgets are assigned somewhere between average and worst case, which means that in the average case not all hardware resources are used. If, in spite of the fact that some tasks have excess resource needs, the total load on the system still falls within the available resources, then reallocating surplus resources to tasks in need raises the cost-effectiveness of the system. Note that this reallocation is done only if the resources would remain otherwise unused.

3.2. Unit of execution

As shown in Section 2.2, 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. This setup can be regarded as non-preemptive scheduling with switch points provided by the coprocessor. This is often termed ‘cooperative multi-tasking’; the scheduler cannot interrupt the coprocessor but waits for the coprocessor to finish a processing step and request a new task.

At such moments, the coprocessor asks the shell for which task it should perform work next. This inquiry is done through the `GetTask` primitive, as detailed in Section 4.2. 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.

Preventing starvation is a shared responsibility between the coprocessor and the shell. In order to guarantee non-starvation, the length of the processing steps of each task mapped on the coprocessor must be bounded. For instance, as the coprocessor may perform a busy wait on a `GetSpace` call, the coprocessor must implement a time-out period after which the coprocessor performs a state save and ends the processing step [7].

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) may also be data dependent.

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

3.3. Budget assignment

Assignment of task computation budgets is done at system level, based on resource requirements and relative importance of each application with respect to the current system load [8][9]. Every application undergoes an acceptance test to verify that the cumulative application resource requirements do not exceed the system capacity. After acceptance, the system assigns budgets to the application. These system-level application budgets are translated into budgets per task for use in the operating system defined by each Eclipse shell.

The shell must support a *policing* strategy to ensure budget protection, such that a higher authority can implement justice. This responsibility of the Eclipse shells correspond directly to the US police adage 'to serve and protect'. Thereto, task budgets are specified in units of processing time. Assignment of such task budgets has two opposing aspects that need to be balanced:

1. In the acceptance test, the system must consider the additional resource requirements of non-preemptive task scheduling in the Eclipse shells; whenever a task depletes its budget, the task overruns this budget by the remaining duration of its processing step. The relative overhead of this budget overrun can be minimized by assigning a high budget value in relation to the (worst-case) duration of a processing step.
2. The absolute budgets of tasks in a coprocessor determine the running time of these tasks, and therefore the task switch rate of the coprocessor. In turn, the task switch rate of the coprocessor relates to the buffer sizes for all its streams. A lower task switch rate means a longer sleep time for tasks, leading to larger buffer requirements and latency.

Eclipse task switch rates are fairly high, in the order of 10-100 kHz. Task budgets typically range from 1000 up to 10,000 clock cycles (10—100 processing steps). Assigning budget values that are relatively high with respect to processing step duration makes the system behave more like a preemptive scheduled system.

4. Task scheduling for streaming media

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 load the Eclipse coprocessors. Therefore, task scheduling cannot be done off-line but must be performed on-line. This section describes the distributed implementation of task scheduling and shows how it handles the robustness requirements of the previous sections and the data-dependency of streaming media tasks.

4.1. Distributed task scheduling in hardware

The Eclipse architecture allows any set of coprocessors to be included in some targeted instance. More specifically, the number of coprocessors and their associated shells may vary over instances. Therefore, Eclipse must be a scalable template. One significant aspect of such a template is whether scheduling is organized in a centralized or distributed fashion. Centralized scheduling is expected to become a bottleneck in terms of scheduling performance and wiring for larger instances of the architecture. Therefore, Eclipse employs distributed task schedulers.

The Eclipse architecture supports relatively high performance, high data throughput applications (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 task switch rates are too high for on-line scheduling in software, Eclipse implements task scheduling in dedicated hardware. Each coprocessor shell incorporates its own task scheduler.

The scheduling algorithm must be sufficiently simple to allow a cost-effective hardware implementation in each shell. On the other hand, the scheduler 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, the task scheduler in each shell runs independent of task schedulers in other shells.

4.2. GetTask primitive

The task scheduler implements the `GetTask` functionality. The coprocessor calls `GetTask` before each processing step. The actual function prototype as shown in Section 2.1 has several arguments for signaling of errors and application progress, however these extra arguments do not interact with the task scheduler, and are beyond the scope of this paper. The return value is a task ID, a small nonnegative number that identifies the task context. Thus, upon request of the coprocessor, the scheduler provides the next best suitable task to the coprocessor.

4.3. 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. The scheduler uses time slices as the unit of measurement, *i.e.* a predetermined fixed number of cycles, typically in the order of the length of a processing step. The task budget is expressed as a number of time slices.

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. 6 depicts the task selection mechanism, executed on each `GetTask` request from the coprocessor as a two-stage selection process:

1. *Active task*. The active task may continue if there is some work to do for the active task, *i.e.* the task is 'runnable', and if it still has sufficient budget.
2. *Next task*. If the active task cannot continue, the scheduler selects the next runnable task from the list of tasks that are mapped onto the coprocessor.

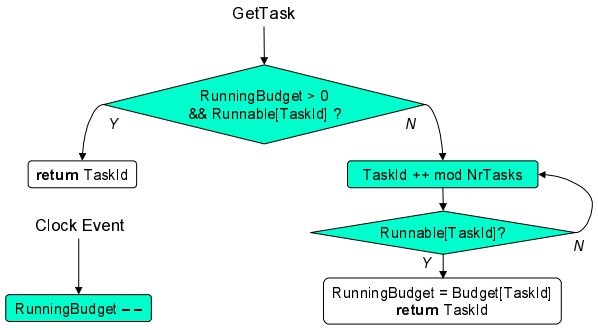

Fig. 6. Task scheduling algorithm.

When the task scheduler selects a new task, it assigns the configured budget from the task table entry of the selected task to a 'running budget' field. The running budget holds the remaining budget of the active task and is decremented by the scheduler after every time slice of task execution. This way, the budget is independent of the length of a processing step, and the scheduler restricts the active task to the number of time slices given by its budget.

The running budget is discarded when coprocessor decides to terminate the active task when it blocks on communication, *i.e.* a denied `GetSpace` request. The next task starts immediately when the terminated task returns to the scheduler. This way, tasks with sufficient workload can use the excess computation time by spending their budget more often. Shifting the time window of other tasks to an earlier point in time when a task blocks and discards its budget allows a maximal utilization of the coprocessor, while maintaining the budget contracts in case some task requires more resources than assigned by its budget.

4.4. Dynamic workload and task runnability

Eclipse streaming media tasks have a dynamic workload. They can be data dependent in execution time, stream selection, and/or packet size. This data dependency influences the design of the scheduler, as it cannot determine in advance whether a task can make progress or not. We propose a scheduler that performs a 'best guess'. This type of scheduler can be effective by selecting the right task in the majority of the cases, and recover with limited penalty otherwise. The aim of the scheduler is to improve the utilization of coprocessors, and schedule such that tasks can make as much progress as possible. Due to the data dependent operation of the tasks, it cannot guarantee that a selected task can complete a processing step.

Data dependent packet size

Task runnability is based on the available workload for the task. All streams associated with a task should have sufficient input data or output room to allow the completion of at least one processing step. The shell, including the 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. Therefore, the scheduler does not have sufficient information to guarantee success on `GetSpace` actions since it has no notion of how much space the task is going to request on which stream.

The scheduler issues a 'best guess' by selecting tasks with at least some available workload for all associated streams, regardless of how much space is available or required for task execution, *i.e.* *Space* > 0, with the *Space* parameter holding the available data or room in the stream, updated at run-time via the `PutSpace` primitive. Checking if there is some data or room available in the buffer—regardless of the amount—suffices for the completion of a single processing step in the cases that:

- The consuming task synchronizes at an equal or lower grain size than the producing task. Therefore, if data is available, this is at least the amount of data that is necessary for the execution of one processing step.
- The consuming and producing tasks work on the same logical unit of operation, *i.e.* the same granularity of processing steps. For instance, if there is some but insufficient data in the buffer, this indicates that the producing task is currently active and that the missing data will arrive fast enough to allow the consuming task to wait instead of performing a task switch.

In current practice, including the MPEG-2 decoding and encoding applications, all coprocessor communication is covered by the above two cases.

Data dependent stream selection
A task is runnable if there is at least some available space in all streams of the task. The selection of input or output streams however can depend on the data being processed. This means that even if *Space* = 0 for some of the streams associated with a task, the task may still be runnable if it does not access these streams in the upcoming processing step. In the task runnability test, the scheduler simply does not check the space value for streams for which it is unclear whether or not the task is going to access the data.

Blocking on communication
To assess task runnability, each stream is associated with a 'runnable' flag. Apart from the non-deterministic streams of the previous paragraph, this flag is set if the stream has sufficient available workload (*Space > 0*). Task runnability now simply resorts to the AND operation over the runnable flags of the streams associated with the task.

The runnable flags also help the task scheduler to select tasks that can progress in the case that coprocessor stream I/O selection or packet size is data dependent and cannot be predicted by the scheduler. If a task cannot make progress due to insufficient space, the `GetSpace` inquiry on one of its streams must have returned `false`. In this case, the stream is 'blocked', and the shell sets the runnable flag to `false`, thereby making sure this task is not selected in a next scheduling round until this stream again has sufficient data. Note that after a failing `GetSpace` request, the active task can also issue a second `GetSpace` inquiry for a smaller number of bytes, and thereby set the runnable flag `true`.

Thus, on a `GetSpace` inquiry, the shell sets the runnable flag of the indicated stream to the return value of the `GetSpace` call. Additionally, the shell sets the runnable flag `true` when an external 'putspace' increases the space for the blocked stream. This runnable flag therefore improves the scheduling accuracy, in the sense that it avoids repeated activation of a blocked task.

5. Results

The Eclipse architecture template is implemented in a retargetable cycle-accurate simulator. This includes simulation models of a set of four multi-tasking coprocessors for MPEG-2 decoding in a first instance of the Eclipse template. These can be configured at run-time to decode maximally two high-definition MPEG-2 streams (1920x1080 pixels interlaced), or up to six standard definition MPEG-2 streams (720x576 pixels interlaced) in a time-shared fashion. Initial simulation results, as well as estimated performance, silicon area, and power consumption, are presented in [4]. Currently, detailed analysis and design of this first Eclipse instance is in progress.

6. Related work

Existing heterogeneous architectures for streaming media with multi-tasking processors either operate on a course function grain with communication through large buffers on off-chip memory [3][10], or focus toward fine grain processing elements with very regular workload and an extensive on-chip communication network [11][12]. In the latter architectures, a central CPU takes care of scheduling the tasks on all processing elements. Catering to the requirements on high-performance, scalability, and the irregular behavior of streaming media, Eclipse introduces distributed multi-tasking in dedicated hardware support. Eclipse complements the existing palette of function modules, consisting of programmable cores and hardwired coprocessors, as these may be combined in a single course-grain system. For example, such a combination may consist of an Eclipse subsystem for flexible media processing and a statically scheduled subsystem for regular video display processing. In order to address the highly challenging requirements of flexibility and cost-effectiveness, the Eclipse designers deployed a number of known concepts in an entirely new setting.

Budget contracts are described by Bril *et al* [8], similar to the processor capacity reservation mechanism of Mercer, Savage, and Tokuda [9]. The 'firewall' concept for rate-controlled scheduling of Yau and Lam [13] resembles our concept of justice. Moreover, the reservation system as described by Mercer *et al.* [9] forms the foundation for the implementation of justice at system level.

Weighted round-robin scheduling techniques are well known in network routing [14]. Shreedhar and Varghese [15] describe in detail a similar round robin scheduling scheme for packet queues in network routing, while keeping track of budgets for each queue. However, their style of saving surplus budgets does not fit our requirements on real-time behavior. Furthermore, we present several novel optimizations to the basic algorithms that are specific for the media processing domain.

The context in which the mentioned known concepts are applied contains new elements not previously described in literature. This context is characterized by our solution of non-preemptive distributed scheduling of medium-grain tasks with dynamic behavior.

Eclipse instances can be managed by the quality of service resource management framework of Bril *et al* [8]. Such a resource-management framework can control Eclipse scheduling by configuring the low-level budgets as used in the task scheduler in the shell.

7. Conclusion

Eclipse offers a cost-effective and scalable solution for re-using 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.

The task scheduler in each shell observes available workload and recognizes data dependent behavior, while guaranteeing each task a minimum computation budget and a maximum sleep time. High task switch rates are supported with a hardware implementation of the shells.

8. Further research

Eclipse multi-tasking is distributed. The tasks of each coprocessor are scheduled independently by their respective shells. This means that the Eclipse coprocessors are loosely coupled, implying that within the time-scale that the buffer can bridge, scheduling of tasks on one coprocessor is independent of the instantaneous scheduling of tasks on other coprocessors. However, on a time scale larger than the buffer can bridge, the scheduling of tasks on different coprocessors is coupled due to synchronization on data streams in shared buffers. Utilizing distributed, unsynchronized scheduling shifts the problem of controlling overall system behavior to the system level. The precise effects of multiple, unsynchronized schedulers on the overall system behavior—especially in relation to FIFO sizes—is subject of further investigation. As we expect that many concepts as developed within the real-time community are applicable in this setting, we are open to proposals for collaboration in this field.

9. Acknowledgment

We are indebted to Sjir van Loo and Liesbeth Steffens for their contribution to the Eclipse scheduling concepts as presented in this paper. We thank Lui Sha for coining the term ‘justice’ and triggering the discussion on budget policing in Eclipse.

10. References

[1] Texas Instruments, *OMAP™ Platform: Overview*, http://www.ti.com/sc/omap

[2] ARM, *PrimeXsys™ Platforms, Extendible Platform Architecture*, http://www.arm.com/armtech/PrimeXsys

[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] M.J. Rutten, et al., “Eclipse: A Heterogeneous Multiprocessor Architecture for Flexible Media Processing”, *IEEE Design and Test of Computers: Embedded Processor Based* Designs, 2002.

[5] G. Kahn, *The Semantics of a Simple Language for Parallel Programming*, proc. of the IFIP congress 74, August 5-10, Stockholm, Sweden, North-Holland publ. Co., pp. 471-475, 1974.

[6] G. Kahn and D.B. MacQueen, *Coroutines and Networks of Parallel Programming*, Information Processing 77, B. Gilchhrist (Ed.), North Holland publ., pp. 993-998, 1977.

[7] 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.

[8] R.J. Bril, et al, “Multimedia QoS in consumer terminals”, *IEEE Workshop on Signal Processing System (SIPS)*, pp. 332-344, Sept. 2001, Antwerp, Belgium.

[9] C.W. Mercer, S. Savage, and H. Tokuda, “Processor Capability Reserves: Operating System Support for Multimedia Applications”, *Int. Conf. on Multimedia Computing and Systems (ICMCS)*, pp. 90-99, May 1994.

[10] W. Lee and C. Bosaglu, “MPEG-2 Decoder Implementation on MAP-CA Mediaprocessor using the C Language”, *SPIE Proc. on Media Processors 2000*, vol. 3970, 2000.

[11] 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.

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

[13] D.K.Y. Yau and S.S. Lam, “Adaptive Rate-Controlled Scheduling for Multimedia Applications”, *ACM Multimedia 96*, pp. 129-140, Nov. 1996.

[14] M. Katevanis., S. Sidiropoulos, and C. Courcoubetis, “Weighted Round Robin Cell Multiplexing in a General Purpose ATM Switch Chip”, *IEEE Journal on Selected Areas in Communication*, vol. 9, no. 8, pp. 1265-1279, 1991.

[15] M. Shreedhar and G. Varghese, "Efficient Fair Queueing using Deficit Round Robin", *SIGCOMM 95*, pp. 231-242, and *ACM/IEEE Trans Networking*, vol. 4, no.3, pp. 375-385, Oct. 1995.