C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems

André Nieuwland (andre.nieuwland@philips.com), Jeffrey Kang (jeffrey.kang@philips.com), Om Prakash Gangwal (o.p.gangwal@philips.com), Ramanathan Sethuraman (ramanathan.sethuraman@philips.com), Natalino Busá (natalino.busa@philips.com), Kees Goossens (kees.goossens@philips.com), Rafael Peset Llopis (rafael.peset.llopis@philips.com) and Paul Lippens†
(lippens@magma-da.com)
Philips Research Laboratories,
Prof. Holstlaan 4,
5656 AA Eindhoven,
The Netherlands

Abstract. The key issue in the design of Systems-on-a-Chip (SoC) is to trade-off efficiency against flexibility, and time to market versus cost. Current deep sub-micron processing technologies enable integration of multiple software programmable processors (e.g. CPUs, DSPs) and dedicated hardware components into a single cost-efficient IC. Our top-down design methodology with various abstraction levels helps designing these ICs in a reasonable amount of time. This methodology starts with a high-level executable specification, and converges towards a silicon implementation. A major task in the design process is to ensure that all components (hardware and software) communicate with each other correctly. In this article, we tackle this problem in the context of the signal processing domain in two ways: we propose a modular, flexible, and scalable heterogeneous multi-processor architecture template based on distributed shared memory, and we present an efficient and transparent protocol for communication and (re)configuration. The protocol implementations have been incorporated in libraries, which allows quick traversal of the various abstraction levels, so enabling incremental design. The design decisions to be taken at each abstraction level are evaluated by means of (co-)simulation. Prototyping is used too, to verify the system's functional correctness. The effectiveness of our approach is illustrated by a design case of a multi-standard video and image codec.

Keywords: Embedded systems, heterogeneous multi-processor architectures, distributed shared memory, Kahn process networks, design methodology, communication, synchronisation protocol, signal processing applications.

† Paul Lippens currently works with Magma Design Automation

1. Introduction

Embedded systems can be classified into two domains: reactive/control and signal processing. A key aspect of the control domain is that the time to react is critical for proper behaviour of the system. In this type of systems, there is limited availability of task level parallelism (TLP), and performance improvement is mainly realised by using newer technologies which allows for higher clock rates, the use of more deeply pipelined circuits, and the exploitation of instruction level parallelism (ILP) as in VLIW (Very Long Instruction Word) and super-scalar processors. Good examples of products in this domain are automotive and hard-disk controllers.

In this article we are more concerned with the second domain: signal processing. The main issue here is the often enormous processing demand and data bandwidth. Timing constraints are mainly related to the prevention of buffer over-/under-flows, and in a lesser extent to the required response times. The performance requirements for many products in this area are related to standards like MPEG, DVB, DAB, and UMTS. Commercialisation calls for cost-efficient implementation of the signal processing part of these products. However, because of evolving standards and changing market requirements, the implementation also requires flexibility and scalability. Dedicated hardware implementation provide the performance required by signal processing applications but they are not flexible. On the other hand, today's technology permits more programmable (flexible) solutions due to the increase in processor performance. This allows for a new balance between efficiency, flexibility and design time (time to market).

Using a single processor is not an option because the processing performance required to implement signal processing applications (e.g. GSM) is often beyond its capabilities [39]. Furthermore, the ILP available in such applications is limited and can be exploited to a certain extent only. Further performance increase can only be achieved by exploiting the TLP. TLP is abundant in such applications because signal processing applications often comprise different stages of operations, which are performed on an endless stream of data in a pipelined fashion. These stages have little dependencies with the others and can be modelled as separate tasks. A single processor is unable to exploit this property. Waiting for the next generation of CPUs or DSPs is not an option (see Figure 1), since the demand for signal processing performance (in operations per second (OPS)) will also increase in time. The growing gap between available generic processing performance (in instructions per second (IPS)) and the requested signal processing performance can only be bridged by adding dedicated hardware to the processing system for the computationally intensive parts.

Power consumption is another reason why signal processing is not likely to be implemented completely in software on a generic processor. Due to the required flexibility of CPUs and DSPs, the power consumed in performing a specific signal processing function is two to three orders of magnitude higher than when the same function is performed by dedicated hardware.

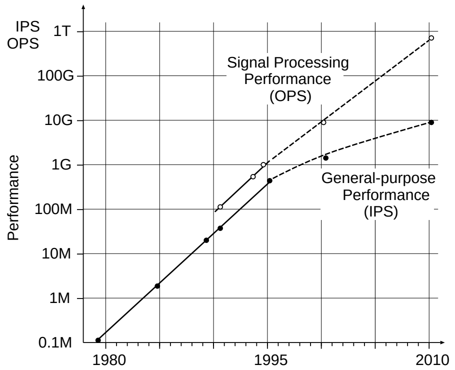

Figure 1. Required media processing performance and available (generic) processing performance [39].

By having multiple processors in a single system, the available TLP can be effectively exploited to achieve high performance. In order to combine this with flexibility in a cost-effective way, we need to have a heterogeneous mix of different processing modules so that each can be tuned to its specific function. Therefore, we believe that signal processing systems should be built as heterogeneous multi-processor systems: a combination of programmable devices such as CPUs, DSPs and ASIPs (Application-Specific Instruction-set Processors) for the more flexible part of an application, and dedicated hardware for the heavier signal processing kernels.

These heterogeneous systems are generally difficult to design. Since there are multiple cores of different natures, the design space of such systems is very large. Coordination of various different types of devices is error prone and not trivial and, if it is poorly performed, the whole system's performance will deteriorate. An additional difficulty comes from the complexity of the applications. We believe that three solutions are needed to effectively overcome these problems: 1) a top-down design methodology with various abstraction levels, which allows the designer to focus on the right problems at each level, 2) a design template to limit the design and modelling efforts through the reuse of pre-defined modules, and 3) a protocol for cooperation and communication between autonomous tasks, which should be transparent to the designer to ease the design process. These are the main topics addressed in this article. The central entity in our approach is a CPU which configures all the processing modules and their communication at system start-up time and also performs some reconfiguration at run-time, but the communication thereafter is independent of the CPU. Therefore, the combination of our template and protocol is called C-HEAP, which stands for **C**PU-controlled **HE**terogeneous **A**rchitectures for signal **P**rocessing.

This article is organised as follows. We describe related work in Section 2. Section 3 presents our design methodology. The target architecture template is introduced in Section 4. The C-HEAP primitives are divided into two categories: communication primitves and (re)configuration primitives. We discuss the C-HEAP communication primitves and its implementations in detail in Section 5. We do the same for the (re)configuration primitives in Section 6. We validate our design methodology and evaluate the efficiency of our protocol and architecture template through a design case: a multi-standard video and image codec. This design case is presented in Section 7. The article ends with conclusions and future work in Section 8.

2. Related work

The first step in building a system is getting the specifications right. Typically, specifications are textual documents with some (vague) requirements, which may be unclear and prone to misinterpretation by different designers. Rather, executable specifications are more concise and can be used for validation. Hence, specifications are increasingly often written using programming languages (e.g. C) which usually express sequential behaviour of an application. Concurrency in an application can be made explicit when TLP can be expressed, which is inherent in signal processing applications. A model of computation often used for this purpose is Kahn Process Networks (KPN) [25]. In this model, the application is decomposed into a number of parallel processes communicating via point-to-point uni-directional unbounded channels with first-in-first-out (FIFO) behaviour. It is a deterministic, composable and hierarchical model of computation. It separates communication and computation parts of an application. This orthogonalisation is a basic foundation for template (platform)-based design [27, 20, 40, 11, 45].

A special case of KPN called Dataflow Process Networks [30] has also been used for modelling signal processing applications [29, 7, 13]. Dataflow Process Networks is more suitable for regular signal processing applications. However, for irregular signal processing applications (e.g. MPEG-x) KPN is preferred since it avoids explicit state saves [18]. Because of the generality of KPN, which allows embedded use of Dataflow Process Networks, we choose KPN as our model of computation.

It has been reasoned in the literature that template-based design is necessary for designing embedded systems to handle the complexity of applications [27], to achieve the required performance with efficient implementations [31, 5], to generate product families [43], to enable reuse [11, 8], and to reduce time-to-market [20, 38]. A template provides a standard way of communication among the vast variety of computational elements (e.g. processors, DSPs, ASIPs, dedicated hardware) [45, 31, 5]. A template with a method for connecting processing elements through some interconnection network (e.g. CoreConnect [3], AMBA [1]) is not enough for multi-processor systems. Coordination of processing devices in a systematic manner through the use of a protocol which can take care of communication, (re)configuration etc., is becoming necessary [22].

Communication between processing elements is based on either message passing or shared memory. Both communication principles need a form of signalling to indicate the presence of data. With message passing communication, this signalling is often a side effect of sending data from one task to another. This means that inter-task data transportation and synchronisation are combined in a single action. The disadvantage of message passing protocols is that copies of data have to be created, which requires extra time and memory. In shared memory architectures, by separating synchronisation and data transportation, copies of the FIFO buffer data need not be created. The data can stay at the place where the FIFO is mapped in the memory and a task can access that data after performing a synchronisation action (claiming the data). Data transport in the simplest form can be load/store instructions for a processor. However, synchronisation requires some attention. In homogeneous shared-memory multi-processor architectures, plenty of algorithms for scalable synchronisation exist (see [33]). In this domain, one resource is shared by multiple tasks. This problem is solved either by *spin-lock* or *barrier algorithms*, which require special instructions (e.g. `test_and_set, swap_with_memory`) or atomic read-modify-write operations. It has the disadvantage that semaphores or atomic operations are needed. This limits the system's scalability and retargetability. Therefore, solutions with atomic operations must be avoided.

We describe closely related work of Prophid [31], COSY [11, 12, 9, 10], CoWare [44, 45, 8] and TIMA laboratory [5, 32]. We will restrict our discussion to the work in the signal-processing domain.

Prophid is a heterogeneous multi-processor architecture template. This template distinguishes between control-oriented tasks and high performance media processing tasks. A CPU connected to a central bus is used for control-oriented tasks and possibly low to medium performance signal-processing tasks. A number of application-specific processors implement high performance media processing tasks. These processors are connected to a reconfigurable high-throughput communication network to meet the high communication bandwidth requirements. Hardware FIFO buffers are placed between the communication network and the inputs and outputs of the application-specific processors to efficiently implement stream-based communication.

The COSY methodology provides a gradual path for communication refinement in a top-down fashion for a given platform and a communication protocol. The main goal of the COSY methodology is to perform design space exploration of a system at a high abstraction level. In order to achieve this, the methodology provides a mechanism for modelling communication interfaces at a high level of abstraction (including the behaviour of selected protocol) with various parameters, e.g. delay of execution of the protocol itself. The COSY methodology uses a message passing communication protocol with *read* and *write* primitives.

CoWare [44] reports a methodology along with an architecture template (called Symphony [45, 8]). In the CoWare methodology, signal-processing kernels are specified as functions, which are called from the main software task using remote procedure calls. At the time of implementation, these kernels can be bound to hardware accelerators. The interfaces between the (main) program and the signal processing kernels are automatically generated. In Symphony, the physical architecture is abstracted as an interconnection of "Processor Component Units" with point-to-point communication channels. Communication on the channels is realised with the rendez-vous protocol (using *send/receive* primitives), in which both sender and receiver must be ready (i.e. implicite synchronisation) before data transfer can take place. Each component has ports through which data communication can be performed. FIFO buffers of channels can be mapped onto a hardware FIFO component between input and output ports. It allows shared memory among different processing units with multi-port memories.

The work done at the TIMA laboratory resulted in a generic architecture model [5] intended for the design of application-specific multi-processor Systems-on-a-Chip. This model is modular, flexible, scalable and generic enough to be able to tackle a broad range of application domains. It also allows for systematic design of multi-processor systems. Communication among the computation units is handled through communication co-processors. An architecture instance can be generated by customising the various parameters such as the number of CPUs and the communication protocol for the communication co-processors. In [32], an automatic architecture generation flow is presented based on this model, starting with a macro-architecture level specification (that is, a view in which modules with wrappers are connected to each other via channels). At this level, allocation of processors and mapping of behaviour and communication (e.g. protocol) have already been done.

COSY and Prophid use the Kahn Process Network as the model of computation, CoWare uses a similar data model with processes and channels. However, the work of TIMA laboratory leaves this choice open. As explained above, we also make this choice clear so as to be able to use KPN as the model of computation. The Prophid architecture template uses a Time-Space-Time (TST) switch matrix network. A TST network offers very high communication bandwidth and performance. It suits regular data processing applications, but it is less flexible. It is not suitable for our applications as they perform irregular data processing. CoWare has a point-to-point automatically generated physical interconnection network. It offers guaranteed network access but the interconnection network may explode as the number of ports increases. The COSY methodology and the work of the TIMA laboratory leave open the choice of the interconnect network. This gives the system designer the freedom to choose one for its application. In addition to these ideas, we define some rules to facilitate the definition of a protocol for communication and (re)configuration. Note that the definition of protocols and the selection of an architecture template will be usually interwined [22].

The other important part of the template is the mapping of FIFO buffers. Prophid and CoWare implement these FIFO buffers as hardware components, which are not flexible. A FIFO buffer mapped onto a memory is flexible since the element size and the length of the FIFO can be changed even after final silicon realisation. CoWare allows a shared memory in the architecture template, but it does not define how shared memory can be used to map channel FIFO buffers. COSY provides this mapping. As we target flexible systems, we advocate the mapping of the FIFO buffers onto memory.

CoWare and COSY use communication protocols based on message passing which has the drawback of having to copy data. Prophid, which also uses the message passing protocol, circumvents this drawback by using a very high bandwidth interconnection network (TST network) instead of a shared memory with communication buffers. The architecture developed by the TIMA laboratory is very generic and does not specify any communication protocols. We propose a synchronisation method which supports true TLP for an unspecified number of tasks without the need for semaphores or atomic instructions. Furthermore, the proposed synchronisation is transparent so that hardware and software communicate in an identical way. Similar to CoWare, we also provide library-based interface synthesis only for synchronisation; existing approaches for data transport (load/store of data) can be used.

3. Design methodology

Due to the ever increasing complexity of SoC designs, there is a need for a structured approach in the design process. Hierarchy in this process helps to guide the designer in taking the design decisions required to arrive at an implementation. In this way, few but larger decisions can be taken at an early stage in the design process and the more detailed decisions (which are as important) at a later stage. A design methodology should enable a designer to make all the necessary decisions at each stage in the SoC design process. All these decisions have to be based on quantitative data obtained in an analysis of the application. Note that the number of decisions to be taken by a designer can be reduced by using an architecture template.

Our design methodology corresponds to this top-down approach, i.e. the design process starts with an abstract description of the application/algorithm, and gradually arrives at a detailed implementation [19]. This design methodology is shown in Figure 2. The designer starts with a selected algorithm/application and analyses it. The next step is the partitioning of the application, and the mapping onto an instance of the architecture template. The performance and functional correctness of this set-up are assessed by means of simulation. If the performance requirements are not met then different choices can be made (sometimes) for the algorithms, the partitioning, the architecture template instance or the mapping. Then the performance is evaluated again. This design space exploration step is iterative because these steps may have to be repeated a few times before the results are satisfactory.

If the combination of partitioning, architecture and mapping yields results that meet the design requirements, the system can be implemented. This step entails compiling the software parts of the application on the selected processor(s), and designing and implementing the hardware parts. We propose to break this large step into smaller steps, in which the critical parts of the application, e.g. the communication network, are implemented in detail first, to reduce design effort and

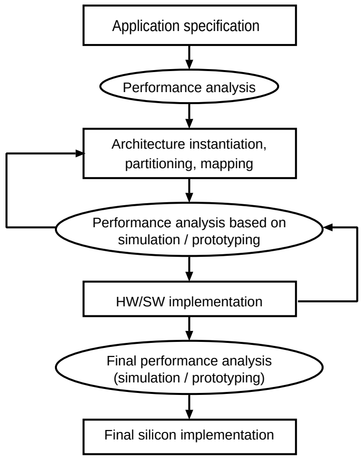

Figure 2. Design methodology

simulation time. This is again an iterative step which results in a bit-true and cycle-true design description of the system. This final model behaves in exactly the same way as the final silicon realisation.

From this level on, prototyping can be used as an alternative to simulation. Prototyping allows the use of the targeted embedded processors instead of instruction set simulators, and Field Programmable Gate Array (FPGA) based implementations for hardware blocks instead of VHDL/Verilog simulators. Therefore, prototyping speeds up the execution of the modelled system. It offers an application (embedded) software development environment and the possibility to build demonstrators for customer surveys. However, note that the communication performance can not be easily evaluated because the communication infrastructure of the prototyping platform (e.g. PCI bus) may differ from the communication infrastructure of the targeted system (e.g. AHB bus).

We can ease the decision making in the design process by defining several abstraction levels. In this way, the designer can focus on the relevant problems at each level, without being confused by unnecessary details. For example, if the communication performance of a functional partition has to be evaluated then a designer may not want to spend time on implementation details such as the exact implementation of interrupt service routines on the targeted processor, or the implementation of a function as a state machine in hardware.

We identify the following levels of abstraction:

a) algorithmic specification level (single-threaded)

b) partitioned algorithmic specification level (concurrent tasks, multi-threaded)

c) cycle-true communication level; bit- and cycle-true models for the communication and abstract functional models for the processing tasks

d) cycle-true embedded software level; bit- and cycle-true models for the communication, bit- and cycle-true functional models for the processing tasks which are implemented in software, behavioural models for the processing tasks which will be implemented in hardware

e) partially implemented cycle-true hardware level; bit- and cycle-true models for the communication, bit- and cycle-true functional models for the processing tasks which will be implemented in software or hardware, combined with behavioural models for tasks still to be implemented in hardware

f) bit-true and cycle-true level; bit- and cycle-true models for the communication, bit- and cycle-true functional models for the processing tasks to be implemented in software or hardware

These abstraction levels are depicted in Figure 3. The starting point of the design is an executable functional specification of an application (Figure 3(a)), which describes the behaviour in the C language. This specification can be simulated to check the functional correctness. At this level the simulation speed is high since no timing and architecture details are considered. Profiling techniques can be used to estimate the computational load of different functions. This information can be used for functional partitioning of the application, and provides input to the hardware-software mapping step. After partitioning, the application is represented as a Kahn process network (see Figure 3(b)), with tasks that perform the processing functions and channels through which data are communicated. Simulation at this level is multi-threaded (running on a multi-threading environment e.g. PAMELA [4]), and is used to check the functional correctness of the application after partitioning.

The communication bandwidth is often a major bottleneck for media processing applications mapped onto multi-processor systems [40, 31]. To check the feasibility of the current partition, we introduce an abstraction level in which communication is simulated in a bit- and cycle-true manner while the functions may be still untimed (see Figure 3(c)). Note that the functional C models used at this level are the same as those used in the previous step, only now they are executing as UNIX processes. This has the advantage that the simulations are performed with realistic data and include all data dependencies. Computation delays can be inserted in the functional models to mimic their real (partial) timed behaviour. In this step, template-based design helps

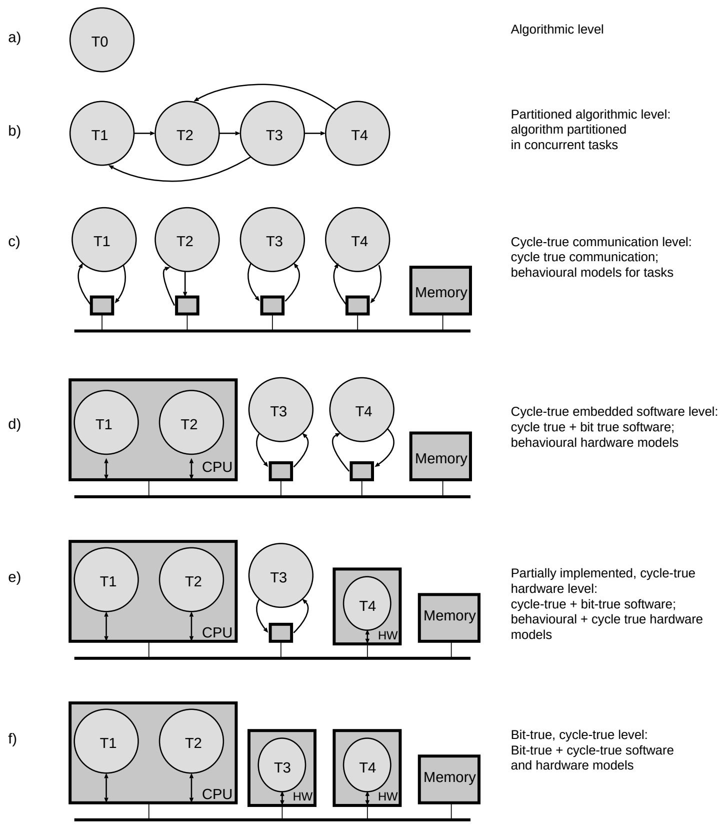

Figure 3. Levels of abstraction for incremental performance evaluation

since models of communication network and/or memory are available. Simulations are performed with these models to obtain an initial estimate of the performance, bandwidth utilisation and latency of the critical parts. Prototyping at this level of abstraction, as an alternative to simulation, is not yet useful as only little speed-up can be gained for functional parts of applications. Furthermore, the communication network of the prototype should match the communication network of the template, which is often not the case.

The mapping of functions on hardware or software is the next step in the design methodology. For the software tasks, the same C code as used in the levels above is compiled for the target processors. They can then run on cycle-true simulation models of the processors. For the tasks that are to be implemented in hardware, we still keep them as abstract functional models. This has the advantage that with little effort (cross-compilation), the more unpredictable (e.g. due to cache behaviour) software tasks which may have a major impact on the overall system performance, can be evaluated before any effort is put into the hardware design. This set-up is shown in Figure 3(d). At this level prototyping becomes an interesting alternative since software can be executed on the targeted processors rather than on instruction-set simulators.

Once this type of performance analysis has resulted in sufficient confidence that the system can be implemented within the specification limits, further refinements have to be performed. All the hardware accelerator units can be converted incrementally into bit-true and cycle-true models (C or VHDL) (see Figure 3(e)). These models can be generated from the C based description by architecture level synthesis tools, e.g. A|RT [2]. The system's performance and functional correctness can be evaluated through multi-level co-simulations or by prototyping. Prototyping becomes more important since simulating the detailed hardware models in VHDL/Verilog is rather slow in comparison with executing the function on an FPGA board. The latencies of the implemented hardware accelerators could be used to tune the delay annotation in the abstract models to further refine the simulation accuracy at higher abstraction levels. The end result of all these steps is a cycle-true model for the entire system (see Figure 3(f)).

Essential for this design methodology are smooth and seamless transitions between all the abstraction levels. For this reason, our design flow is completely C based. Furthermore, it is necessary to have simple-to-use, well-defined communication primitives [20, 18]. That reduces the amount of work required for repartitioning. By using the same primitives on all levels of abstraction, a uniform view on communication is guaranteed, and no modifications are required in the processes at transitions to a different abstraction level. To enable such smooth transitions, the primitives should be implemented in libraries for various types of components (e.g. MIPS, ARM, dedicated hardware, ASIPs). When a task moves to a different abstraction level only the different library has to be linked with that task. Note that keeping part of the implementation of primitives fixed and common makes seamless interaction at all abstraction levels and between all types of devices possible. This allows easy shifting of tasks between hardware and software, or between timed and untimed implementations.

4. Architecture template

One of the solutions for dealing with the complexity of embedded system design is the use of architecture templates. The use of architecture templates can drastically speed up the design process and reduce the development costs for a number of reasons:

- By fixing a number of aspects in the target architecture (e.g. the memory architecture), the designer can concentrate on the complexity of the application itself, and hence spend his/her time more effectively.
- Reuse of IP blocks is natural, and can be achieved by sharing components that match the template, both among different products and across different product generations [20]. Therefore, the design of product families is facilitated.
- Various models, libraries and tools can be supported for a template.

This section presents the architecture template that we use for the design of embedded signal processing systems. The template consists of multiple processing devices, and is therefore a multiple instruction multiple data (MIMD) type of architecture. In such a multi-processor system, each task can run on a different processor, hence effectively exploiting the TLP available in the application. To save silicon area, multiple tasks may be mapped on one processing device (multi-tasking). As the processing devices are of different types (e.g. CPU/DSP/ASIP/ASIC), we use the term heterogeneous multi-processor architecture. The heterogeneity of the architecture allows us to achieve an optimum balance between performance, power consumption, flexibility and efficiency.

For high performance data processing, the devices need to have a high bandwidth to memory. If they have to access this memory via a shared interconnection network, then the network quickly becomes a bottleneck. Therefore, we choose to distribute memory over the different processing devices. This provides higher bandwidth with lower latency, which results in a higher performance at a lower power consumption [41, 16, 15, 34, 35]. By making the memories part of a *global memory map*, they become accessible to other processing devices as well (i.e. *Distributed Shared Memory*). This makes these memories suitable for mapping communication buffers to. Communication buffers are needed to decouple the different tasks to achieve a high degree of parallelism, especially for irregular data processing. Otherwise, due to the differences between the tasks (e.g. latency, data access patterns), processing capacity is wasted in the time spent waiting for data (idle time). By mapping buffers onto shared memory, a producer of data can simply write into a communication buffer, and a consumer can read from the same buffer some time later. Because of the distributed nature of the memory, the access time of a data element depends on its physical location in memory and is not uniform. Therefore, this architecture is called a *Non Uniform Memory Access* architecture (NUMA) [23]. Note that pieces of memory may also be kept private to certain processing devices (e.g. scratch pad purposes), which are not visible in the global memory map.

The C-HEAP architecture template is depicted in Figure 4. The different processing devices are connected by an interconnection network. The interconnection network (e.g. busses, switch matrix) can be freely chosen by the designer. This network must be address aware, and the data transactions must be completed in order (for the synchronisation signalling required in our protocol, see Section 5). Low latency is a desired feature of the network, in order to reduce the protocol execution delay. We want to have a scalable architecture to be able to cope with the increasing complexity of applications. This means that it must be possible to extend the network easily, so that more processing and memory modules can be plugged into the architecture to enhance the system's functionality and performance, without the risk of insufficient communication bandwidth. By selecting and configuring the appropriate processing cores and interconnect, an *instance* of the architecture template is created.

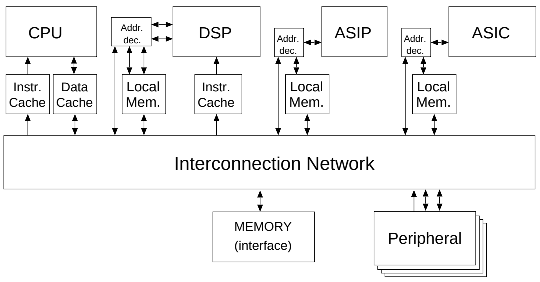

Figure 4. C-HEAP architecture template

In general, multi-processor systems are much harder to program than uni-processor systems. However, it is not difficult for the signal processing application domain because these applications can be expressed in a simple programming model like KPN. Section 5). One of the difficulties here is cache coherence: a processor might have stale data in its cache when it wants to read data recently written by a task on another processor. There are two ways of dealing with this issue: 1) by explicitly invalidating the cache lines or reading from uncached address space in software, and 2) by adding hardware cache coherence support. Explicitly invalidating the caches is in the general case awkward for the programmer, since he/she has to keep track of which variables are shared and might be invalid. On the other hand, hardware support may be complex to implement for larger interconnection networks. For signal processing applications, however, all shared data are explicitly communicated, which makes invalidating the proper cache lines rather simple.

5. The C-HEAP communication protocol

Our architecture template consists of several processing devices of different natures. Since they all operate in the same application, they need to communicate with each other. If each block communicates in a unique (i.e. hardware specific) way, it is very difficult and time-consuming to build a full application. Once a functionally working implementation has been made, it is almost impossible to tune the implementation to meet the requirements, due to all the details of the communication. By standardising the communication in a transparent way (i.e. independent of the processing device), it is much easier to construct a complex system and to tune the implementation to meet the requirements.

To support the different abstraction levels defined in our design methodology (see Section 3), an implementation of the communication protocol is needed at each level. Because the protocol has been standardised, library implementations can be made for each abstraction level in our design flow. In this way, we can easily iterate on different implementations during the design, at all levels of abstraction. This also facilitates multi-level co-simulation, necessary to assess the viability of an implementation in a reasonable time. For ease of use, a well-defined set of primitives is required to hide the implementation of this protocol.

Section 5.1 describes the C-HEAP communication protocol. In Section 5.2, the different implementations of this protocol will be presented.

5.1. Communication protocol

One approach for controlling the data-flow in MIMD architectures, such as the template presented in Section 4, is to have a main processor controlling the other processing devices. In this section we call this set-up a *co-processor* architecture. Here the co-processors do not operate autonomously, but are controlled as slaves by the main processor. In order to control the co-processors, the main processor either polls the status of the co-processors or the co-processors themselves notify (interrupt) the main processor when their task is finished. In order not to overload the main processor, the interrupt rate should be low. This can be accomplished by using (very) large grain synchronisation, for example, in a video context, synchronising on a field or a frame basis.

Large-grain synchronisation requires large buffers to store the data to be communicated, which due to their size have to reside in off-chip memory. Off-chip memory bandwidth is expensive and power consuming, and is already a major bottleneck in systems. This bottleneck can be alleviated by ensuring that the data remains on chip. Since the amount of on-chip memory is limited and quite often already dominant in cost (area), we should look for ways of decreasing the on-chip buffer size. One way of accomplishing this is to reduce the synchronisation grain size. However, in a co-processor architecture, this would lead to unacceptably high interrupt rates for the main processor.

Therefore, we propose a different approach in which the processing devices are autonomous with respect to synchronisation and do not require service from a main processor [36]. This implies that each processing device should be able to initiate communication with any other device, hence the communication protocol must have a distributed implementation. We call this kind of architecture a *multi-processor* architecture. With this approach, we can go to a smaller grain of synchronisation, which allows smaller buffer sizes, and hence on-chip communication. For example, in a video context, we can synchronise on a line or block basis.

Our application specification is based on Kahn process networks [25] (see Figure 5). In this article, we refer to these processes as tasks. In this model, when a task wants to read from a channel and no data is available, the task will block. However, write actions are non-blocking. In our model, *FIFOs* are *bounded* since we are addressing efficient *implementation* of the function and not only specification. This means that a task will also block when it wants to write to a channel if the associated FIFO is full. In the remainder of this article we will refer to this model as a process network or task graph. The synchronisation takes place on a per-token basis. While a token is the unit of synchronisation, the amount of data associated with a token can vary. However, the size of a token is fixed per channel, statically.

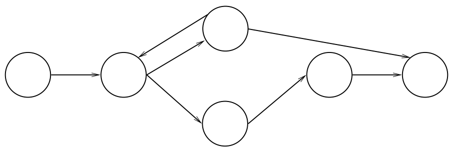

Figure 5. Example of a Kahn process network. The spheres represent the processes and the arrows represent the communication channels

The communication protocol we describe involves the realisation of the FIFO-based communication between the tasks, and does not refer to low-level protocols, e.g. bus protocols. Communication in our case is divided into 1) synchronisation, and 2) data transportation. This is done because in a shared memory architecture no copying of data is required and only synchronisation primitives are needed. For efficiency reasons, all communication buffer memory is allocated at set-up and is reused during operation. Therefore, we need primitives to *claim* and *release* buffer memory space. Since we communicate tokens, these primitives operate on a per-token basis. On the data producing side we want to claim empty token buffers and release full buffers. On the consuming side we want to claim filled token buffers and release empty ones. Claiming a token buffer is blocking, i.e. when no buffer is available the task blocks. Releasing is non-blocking. The synchronisation primitives are listed in Table I.

Table I. Synchronisation primitives

Primitive	Description
`claim_space`	Claims an empty token buffer (blocking)
`release_data`	Releases a full token buffer (non-blocking)
`claim_data`	Claims a full token buffer (blocking)
`release_space`	Releases an empty token buffer (non-blocking)

Figure 6 illustrates the use of the four primitives. The buffers are visualised as train wagons and the channels as rail roads. The middle task first has to acquire a full wagon at its input channel and an empty wagon at its output channel. After the input data have been processed, the emptied wagon is pushed back along the input channel and the filled one along the output channel. The initial number of wagons on the railroad determines how strongly the tasks are coupled. Only one wagon means that the tasks have to be executed alternately, whereas more than one wagon allows pipelining (parallel execution) of the tasks.

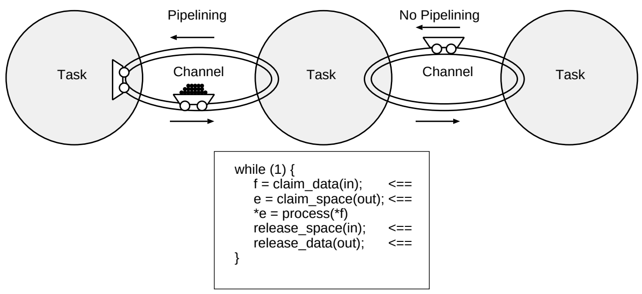

Figure 6. Example of the use of the four synchronisation primitives

In our implementation, it is allowed to claim (reserve) a number of tokens and process them out of order before releasing them in order. This means that multiple `claim` primitives can be called before the corresponding tokens are released by `release` calls. Note that in that case the number of buffers on the channel should be greater than or equal to the number of consecutive `claim` calls, otherwise deadlock will occur. The protocol is defined and implemented so that it is transparent to the tasks, i.e. a task does not have to know whether the task it is communicating with is implemented in hardware or software.

Certain channels may have a single producer and multiple consumers (we call such channels which consist of multiple *branches*, *multi-cast* channels). In this case, the producer sees only one channel and performs synchronisation actions only on this channel. It will be transparent to the producer whether it is communicating data through a uni-cast or a multi-cast channel; this is taken care of by the protocol. At any given time, the fullness of each branch may be completely different, and a `claim_space` call by the producer will block if any of the branches is full.

5.2. Implementation of the communication protocol

Flexible implementation of the communication channel buffers is needed to facilitate a protocol implementation that is transparent to hardware and software. This transparency enables a task to communicate with another task, irrespective of the implementation of that task. We will explain how this can be achieved in Section 5.2.1. Section 5.2.2 describes a general implementation of the synchronisation primitives. Different implementations for simulation and prototyping purposes will be described in Sections 5.2.3 and 5.2.4. Some experimental results demonstrating the efficiency of the communication protocol and its implementation will be presented in Section 5.3.

5.2.1. *Implementation of channel FIFO buffers*

The reconfiguration manager (Section 6.1.1) is responsible for allocating space in shared memory during the configuration phase (start-up time). The communication primitives used by the tasks control the allocated space in a FIFO manner to implement a channel FIFO buffer. This gives us the flexibility to tune the FIFO and token sizes for an application even after a system's silicon realisation. Furthermore, the number of FIFOs and their interconnection structure can be changed in order to map different applications on the same hardware. In order to provide FIFO behaviour of a buffer, some administrative information has to be maintained. The administrative information contains some static and dynamic values. Examples of static values are the size of a token, a base address of the allocated memory, the maximum number of tokens in a FIFO (*maxtokens*), etc. The dynamic values are a read counter (*readc*), a write counter (*writec*) and the number of filled or empty tokens (*ftokens*/*etokens*). Two of the dynamic values are necessary and sufficient to control the allocated space in a FIFO manner. Thus, we have the following minimum pairs:

1. *readc* and *ftokens* or *etokens*.
2. *writec* and *etokens* or *ftokens*.
3. *readc* and *writec*.

Options 1 and 2 introduce a consistency problem. After producing data, the producing task would increment *ftokens*. The data consuming task is supposed to decrement the same variable after consuming data. Since two concurrent tasks modify the same variable, access to that variable needs to be atomic either by means of an atomic read-modify-write, or by guarding through semaphores.

In addition to the consistency problem with the token field in options 1 and 2, there is another consistency problem related to the calculation of the third dynamic value (*readc*, or *writec*). For example, in option 1, when *readc* is 13 and *ftokens* is 4, the derived value of *writec* is 17 (i.e. 13 + 4). After consuming one token the consumer task decrements *ftokens*' value atomically to 3. At this moment, the derived value of *writec* (16, i.e. 13 + 3) is wrong since the value of *readc* has not yet been incremented to make the value of *writec* correct (i.e. 17). Therefore *readc* and *ftokens* have to be updated as an atomic unit to derive the correct value of *writec*.

In option 3, a task increments only its own counter, i.e. a producing task increments *writec* and a consuming task *readc*. Because *etokens/ftokens* are derived from these values, a task never sees more tokens than are available at any time. The counters are initialised to zero and count modulo the *maxtokens* value. To solve the ambiguity problem (whether the FIFO is full or empty) that arises whenever *writec* and *readc* are equal, we extend both values with an extra bit (*wrap flag*). The wrap flag is initialised to zero, and toggled when the corresponding counter reaches the *maxtokens* value. If the counters are equal and the wrap flags are different, then the FIFO is full, otherwise (with identical wrap flags) the FIFO is empty. The wrap flag increases the range of the counters to twice the size of the maximum number of tokens in the FIFO, akin to the sliding window protocol for data communication [42]. We choose option 3, since with this option no consistency problem arises.

For a multi-cast channel, a copy of the administrative information of the FIFO channel is made for each branch. The copied information remains in memory to facilitate reconfiguration and to ensure that it remains compatible with the single branch channels. In the copied information, the static values are the same for all branches, and initially the same holds for the dynamic *readc* and *writec* counters. Although some memory overhead is caused by duplicating channel administrative information, the advantage is that synchronisation primitives can be implemented in a more generic way. Note that in practice there will be only few multi-cast channels, so the overhead of the copies will be small.

5.2.2. *General implementation of the synchronisation primitives*

When a `claim_data` call is executed and a full token is available in the FIFO, a pointer to the token is returned. The token pointer is derived from the *readc* value and some static values. Similar actions are performed for a `claim_space` call. When the `release_space` (`release_data`) primitive is called the *readc* (*writec*) counter is incremented. In the case of multiple consumers, multiple values of *readc* have to be read during a `claim_space` call, and the incremented *writec* counter has to be written to each branch during `release_data`.

In principle, a blocked task polls on (i.e. repeatedly reads) the counter value until it gets through. This *polling-based synchronisation* scheme is useful if the counter value is updated around the same time that a task starts polling. However, polling is not always efficient since it increases the bus load and power consumption. Alternatively, a blocked task can be notified (e.g. by sending a signal) that the status of the FIFO has been changed. We call this scheme *interrupt-based synchronisation.* One should choose between polling-based and interrupt-based synchronisation depending on the expected wait time with respect to the time required to serve the wake-up signal.

The signalling scheme should be scalable because the whole system should be scalable. We implemented the signalling in a memory mapped fashion since it is scalable, and eliminates the need for a dedicated interrupt network. To prevent the activated task from reading old data the signal should not arrive before the data has reached its destination. As long as no re-ordering is done in the interconnection network, the memory mapped signals will follow the data and will not arrive too early.

5.2.3. *Simulated implementations*

These implementations are intended for the simulation steps in the design trajectory (Section 3). Depending on the abstraction level, these implementations may be untimed, partially cycle-true or completely cycle-true. The cycle-true model should behave the in same way as the final silicon realisation. The protocol has been implemented for PAMELA, UNIX processes and as embedded software for DSP, MIPS and ARM (running on an instruction-set simulator). In all implementations, the protocol uses (shared) memory reads and writes. The main differences are in the implementation of the blocking and signalling mechanism.

At the functional simulation level, the PAMELA [4] run-time library is used. Here the blocking and signalling mechanisms are realised using PAMELA semaphores. In order to simulate cycle-true communication for untimed functions, a *process interface block* is used to connect an untimed UNIX (possibly annotated with delay statements) process to the communication network in the simulation environment. This block translates the memory request calls from the UNIX processes into the communication network accesses, which are simulated cycle-accurately. Blocking is done by using UNIX socket communication mechanisms. The process interface block also has at least one memory mapped register to which other processes can write to implement signalling. On the cycle-true level, a software and a hardware implementation of the protocol are distinguished.

Software implementation & optimisation

The software implementation of the protocol is the same as the generic implementation. We provide some optimisations to reduce the interrupt overhead. With interrupt-based synchronisation, a task sends a signal at every `release` synchronisation call. Whenever an interrupt signal is received in a CPU or DSP, an interrupt service routine is called. However, a task does not need these signals if it is *not* blocked, so the overhead of executing interrupt service routines can be omitted. We reduced the number of interrupts seen by the physical processor by introducing a signal controller (see Figure 7). The signal controller has a mask register to mask the signal register during normal operation and enables the signal register only when the task is blocked. The signal controller is memory mapped and instantiated for each CPU and DSP in the architecture.

The ASIPs used in our architecture can be generated by an architectural synthesis tool (e.g. [2]). ASIPs provide high performance for application-specific functions but the primitives are executed in a similar manner as for general-purpose processors. Part of the ASIP memory should be visible in the global memory map to allow signalling, i.e. to write the synchronisation signals. On blocking these synchronisation signals are polled locally by the ASIP.

Hardware implementation & optimisation

The hardware implementation of the protocol for a channel is called a *channel controller*. Channel controllers are instantiated per channel in

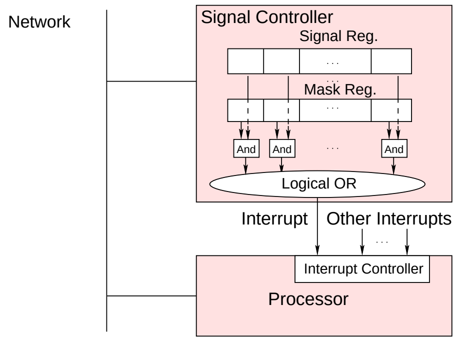

Figure 7. A signal controller for a processor

a synchronisation shell (see Figure 8), which is attached to a hardware processing device. The channel controller is implemented with a generic network interface to facilitate reuse with any particular network by adding a *network adapter*. The synchronisation shell has a *signal register* that is used for signalling in the interrupt-based synchronisation scheme. This signal register is instantiated per device. All registers in the synchronisation shell are memory mapped. A channel controller is implemented in hardware for an input or output channel, and is instantiated for each channel. The mode (i.e. input or output) of a channel controller is set at the system configuration time. We introduced an optimisation to reduce the number of system bus accesses by copying some administrative information values from memory into the channel controller at configuration/start-up time. This helps to reduce the delay in servicing synchronisation calls too. When a hardware device needs a token address (i.e. on a `claim_data/space` call), the token address can be made available in succeeding clock cycles and if more tokens are available then they can also be delivered every clock cycle, one by one. In this implementation, a `release_space/data` call takes just one clock cycle for the task. The `release_space/data` primitives are carried out by the channel controller. A channel controller running at 100 MHz has been made within 0.09 mm^2 in 0.18 μm CMOS technology.

5.2.4. *Prototyped implementations*

Prototyping becomes an interesting alternative to simulation at the (partial) clock-cycle true levels (see Section 3). Our prototyping environment is a PCI-based platform residing in a host PC [14]. This is a convenient platform because a large number of FPGA and embedded processor boards as well as development and debugging tools are available for PCI bus-based systems. Embedded processor boards are used to run the developed software, while the FPGA boards are used to map the hardware on. In the prototyping environment, a common

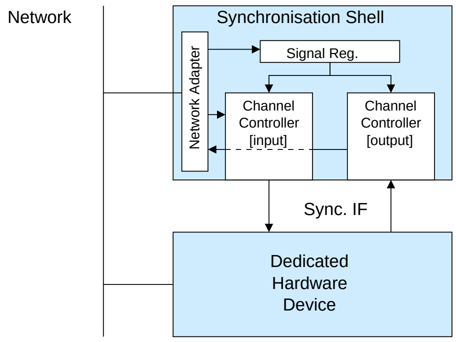

Figure 8. A synchronisation shell

shared memory space should be visible to all the components (embedded processors and FPGA boards) executing C-HEAP tasks. This shared memory is implemented using a PCI memory board.

Because the PCI architecture is plug & play, a hard coded address can not be used to identify a device. The host processor (e.g. Pentium) is responsible for the first PCI architecture initialisation. This step is needed because the PCI addresses must be dynamically obtained (plug & play) before the C-HEAP memory map can be determined. During the PCI configuration phase, the PC's host processor passes the PCI memory base addresses of the boards to the components of the prototyping system. From this moment on, the host processor and processor boards are able to communicate directly with one another by means of the PCI bus. We are currently extending this direct communication to the FPGA boards too. After the PCI configuration phase, the host processor starts all the C-HEAP tasks. For this purpose and for signalling, each board maintains a table (similar to the synchronisation shell) in the (local on board) shared memory space, which is polled. The task command tables used for signalling are also used for configuration. C-HEAP tasks are started for the first time by writing a start command (by specifying the corresponding command opcode) to this table.

5.3. Efficiency of the communication protocol

We have built a simulation environment to include all previously described implementations of the synchronisation primitives. We performed a set of experiments to compare the *co-processor architecture* (i.e. centralised implementation of the protocol) with the *multi-processor architecture* (i.e. distributed implementation of the protocol), and to evaluate the impact of the proposed optimisations on the performance [21]. For this purpose, we implemented a multi-processor architecture with the optimisations (whith static data stored locally) and without, and a co-processor architecture is implemented with optimisation (whith a signal controller attached to the central processor) and without.

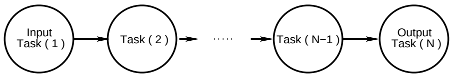

Figure 9. Application used in our experiments

In our experiments, all tasks were performing synchronisation *only* to facilitate the analysis of synchronisation performance. *No buffer data was accessed*. All tasks were connected in a chain in which every task passed tokens to the next task (see Figure 9). A fixed number of tokens (e.g. 1000) were introduced in the system and the simulation was performed until they reached the last task in the chain. Each task was mapped onto a separate hardware module, which posted a new synchronisation call one clock cycle after completion of the previous call. All modules were connected to a central bus. We varied the number of tasks from 4 to 24 in increments of 4, for both the classical co-processor communication scheme and for the proposed multi-processor architecture scheme. The experimental results obtained for average time per synchronisation call are shown in Figure 10.

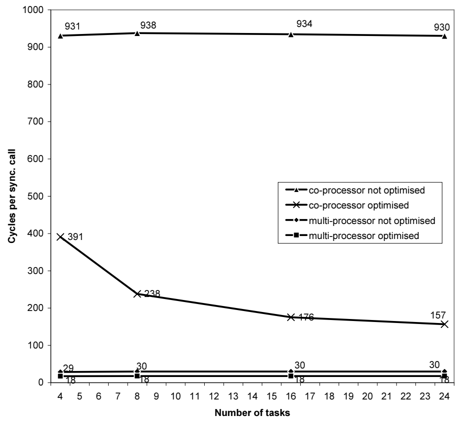

Figure 10. Time per synchronisation call

As can be seen in figure 10, the average time to execute a synchronisation call for the traditional interrupt scheme (co-processor not optimised) is almost constant over the number of tasks. This time can be reduced significantly when our signal controller is used (optimised co-processor curve). This is because with the signal controller, synchronisation requests of multiple tasks are serviced within a single ISR execution. Hence, the ISR overhead is divided among all synchronisation calls being serviced. Due to the long synchronisation delay in the co-processor architecture, fine-grain synchronisation, which is needed to reduce on-chip buffering, is hardly possible.

The optimised multi-processor architecture is 1.5 times faster than the non-optimised multi-processor architecture, and 8 to 21 times faster than the optimised co-processor architecture[1]. The execution delay (excluding bus contention) of a synchronisation primitive in the distributed implementation is only 4 cycles. The observed delay is much larger (18 to 30 cycles) beacause the bus load is 100%. The high bus load is due to the fact that all tasks are only performing synchronisation. Due to the lower synchronisation delay, less memory needs to be allocated for communication buffers. This allows on-chip buffering.

To support the flexible part of an application, tasks could be mapped as software running on the CPU. With the co-processor architecture the software tasks are not running in parallel with the hardware modules because the CPU is (fully) loaded with the synchronisation of the hardware modules. In all variants of the multi-processor architecture the CPU is free to execute (signal processing) tasks since the synchronisation is executed in a distributed and autonomous manner. In this case the software tasks synchronise (through software routines) only with the hardware (or software) tasks they are actually communicating with. Neither software nor hardware tasks need intervention of a 'third party', nor are they interrupted by synchronisation of others. Therefore, the synchronisation delays are quite low.

The time per synchronisation call is very important in comparing different protocols, but what matters for an application is the total execution time. In these experiments the total execution time was determined only by the synchronisation delay. This means that the total execution time in the non-optimised co-processor implementation is about 50 times larger than in the optimised multi-processor implementation [21]. When data are processed along with the synchronisation, the relative gain will be less but the absolute gain will be still the same.

6. The C-HEAP (re)configuration protocol

Section 5 discussed the protocol used to communicate between the tasks and processing devices in our architecture. Another issue is the system's configuration. For simple systems, the configuration can be done at start-up time, whereafter the different modules start to perform

[1] In these experiments all tasks were mapped on hardware devices to enable us to observe the extremes of the synchronisation delays. The delays in the software implementation using the polling scheme in a multi-processor architecture are given in Table IV in Section 7.2.

their tasks. However, embedded systems have to offer more and more functionality in order to be attractive for consumers. This leads to different function modes, which need to be supported by the system. Therefore, apart from the initial configuration and start-up of the system, a certain degree of *reconfigurability* is needed to be able to switch between different function modes at run-time. For instance, in the case of a mobile phone, it may be required to be able to switch between a normal telephone conversation and listening to the radio.

Section 6.1 describes the C-HEAP reconfiguration protocol. Just like the communication protocol, this protocol has been standardised and is transparent to the processing devices, and implementations of this protocol are available in libraries at all abstraction levels. Section 6.2 describes these implementations.

6.1. (Re)configuration protocol

The protocol for communicating between the tasks in a task graph has been extensively described. However, there should also be a protocol and a corresponding set of primitives for setting up and configuring the task graph. This includes creating, configuring and starting the tasks, as well as configuring the channels (e.g. number of buffers) and channel tokens (e.g. token size). If the system has been designed to handle a single application which is realised by a single fixed task graph (for example a video encoder), then these operations, which are basically executed once at system boot time, will be sufficient, and after this phase the tasks will be able to execute independently.

In a system that supports several function modes, multiple task graphs exist, each implementing a certain function mode. During operation, the user may want to switch to a different function mode at run-time, which implies dynamically switching from one task graph to another. Another use for different task graphs is for Quality-of-Service, in which for instance a picture improvement filter task may be dynamically inserted into or removed from the task graph depending on the selected quality level and/or available resources. Or, a new stream can be added to the graph, for instance to support Picture-in-Picture display. All this implies that it must be possible to dynamically reconfigure the task graph. An introduction to issues relating to this topic will follow in Section 6.1.1. The primitives and protocol for configuring and reconfiguring the task graph will be presented in Sections 6.1.2 and 6.1.3.

6.1.1. *Reconfiguration*

Tasks, channels and tokens are the entities that make up a task graph. An entity is either *active* (the entity has a thread of control) or *passive* (the entity consists of data without a thread of control). Thus tasks are active, and channels and tokens are passive. Whereas all entities can be created and destroyed, active entities can also be started, stopped, suspended and resumed. Note that the difference between stopping and suspending an entity is that the entity loses its state when it is stopped, while it retains its state when it is suspended and can continue where it left off when it is resumed. In the following paragraphs we shall consider in turn 1) the parties involved in changing a configuration, 2) when a reconfiguration can take place and what it will entail for the participants, 3) the effects of a reconfiguration, and 4) the impact of reconfiguration on the system's performance and efficiency.

If reconfiguration is performed by a single task (henceforth called the *reconfiguration manager*) it is centralised, otherwise it is *distributed* (and tasks modify themselves). If reconfiguration occurs relatively infrequently and involves only a limited number of entities it can be decided upon and performed by a central reconfiguration manager. This will simplify the reconfiguration protocol because the number of possible interactions will be smaller. Reconfiguration of active entities can be imposed (e.g. a task can be killed), requested (e.g. a task suspends itself after being asked), or autonomous (e.g. a task initiates a change without prompting).

During a task's operation, there may be different points at which it can be reconfigured. There are several options for defining such *reconfiguration points*. One option is to explicitly code them in the control flow of a task (e.g. a task checks whether reconfiguration has been requested via shared data or a control channel). Another option is to insert special reconfiguration tokens in the data stream. When reconfiguration is allowed to take place will depend on the task's functionality and on the granularity of communication. A frame-based image improvement algorithm can perhaps be changed without artefacts only between frames. Whether this will be allowed or not will depend on the application as a whole. Although the granularity of reconfiguration is independent of the granularity of communication (they can after all be transmitted via different channels), in practice, the granularity of communication will be smaller than (or equal to) that of reconfiguration. Figure 11 contains an example with various possible reconfiguration points. They are: 1) at the beginning (or possibly the end) of each iteration loop, 2) when trying to obtain a full buffer on an input channel or an empty buffer on an output channel, and 3) within the processing loop. Reconfiguration point (1) is the most coarse-grained one, e.g. it may correspond to a

```
        while (1) {
(1)       for (i = 0; i < M; i++) {
(2)         data_in* a = claim_data(in);
            data_out* b = claim_space(out);
            for (j = 0; j < N; j++) {
(3)           b[j] = process(a[j]);
            }
            release_space(in);
            release_data(out);
          }
        }
```

Figure 11. Typical task loop with possible reconfiguration points

video frame period. Such a location for the reconfiguration point is a logical choice for e.g. a display task, because if it reconfigures at frame boundaries no artefacts will be introduced. The disadvantage is that it may take the task a long time to react to a reconfiguration command, i.e. the latency will be quite high. At the other extreme, reconfiguration point (3) is located within the fine-grain processing loop (for example over pixels within a frame), and allows fast reconfiguration. However, we may end up with partially processed tokens at the time of reconfiguration because the task is allowed to reconfigure at a higher frequency than the synchronisation. Furthermore, the overhead of checking for reconfiguration will be high. Reconfiguration point (2) coincides with a synchronisation primitive. This implies that reconfiguration may take place at the same data grain as the communicated tokens. In this case there will be no partially processed tokens left when the task has been reconfigured. In conclusion, a protocol supporting reconfiguration should support all of the above because the optimum reconfiguration points are very task- and application-dependent.

Entities can be created and destroyed during a reconfiguration. However, modification of entities is more interesting because it entails suspending the use of the entity, modifying it and resuming operation. The state of the entity (e.g. the token's data) can be reinitialised or preserved. For example, in the case of passive entities changing the capacity of a channel reinitialises the channel's state, but moving a channel (i.e. disconnecting and reconnecting it) retains its state. The state of active entities is reinitialised by stopping and (re)starting, while suspending and resuming leaves the state unchanged. Before a task can be started for the first time, it must be created. Similarly it must be stopped before its destruction.

Although there is some latency involved in reconfiguration, a small delay is tolerable since reconfiguration occurs infrequently in relation to normal signal processing (say once every few seconds or even minutes). The performance overhead of reconfiguration support during regular processing consists of extra checks at reconfiguration point locations, but this overhead is negligible unless reconfiguration points with a small granularity are chosen (see above). In addition, the control flow of the tasks becomes slightly more complex, which results in a larger area for the hardware and a bigger code size for the software. However, we have a more flexible system which can support more functionality. Furthermore, dynamic reconfiguration can be considerably faster for realising mode switches than having to first destroy the entire task graph and then create another one, and it is possible to achieve seamless transitions.

6.1.2. *(Re)configuration primitives*

In C-HEAP, (one of) the CPU(s) in the architecture is responsible for the configuration and reconfiguration of the task graph. It executes a reconfiguration manager which is responsible for (re)configuration of the task graph (centralised reconfiguration). Table II lists the primitives relating to configuration and reconfiguration that are called by the reconfiguration manager. Table III shows the primitives called by the individual tasks.

Table II. (Re)configuration primitives used by the reconfiguration manager

Primitive	Description
`task_create/destroy`	Creates and destroys a task
`task_start/stop/restart`	Starts, stops and restarts a task
`task_suspend/resume`	Suspends and resumes a task
`channel_create/destroy`	Creates and destroys a channel
`channel_add_branch`	Adds a branch to a channel
`channel_reconfigure`	Reconfigures a channel
`set_buffers`	Assigns the location of the allocated buffer memory to a channel

6.1.3. *Procedure*

During the system configuration time (i.e. during set-up of the initial task graph), the reconfiguration manager performs the following actions:

1. Initialisation.

2. Creation of the tasks (`task_create`). This involves allocating memory for the task data structures and initialising it with information about task identifiers, mapping on processing devices, etc.

Table III. Primitives called by the other tasks

Primitive	Description
`get_task`	Obtains task information
`get_channel`	Obtains channel information
`task_check_reconfigure`	Checks whether a stop or suspend command has been issued by the reconfiguration manager
`wait_restart`	Exits processing loop and waits to be restarted
`wait_resume`	Suspends itself and waits to be resumed

3. Creation of the channels (`channel_create`). This includes creating the channel data structures and specifying the channel identifier, the producer and consumer tasks, number of buffers and some operation mode flags.

4. Allocating the memory for the communication buffers and assigning it to the channels (`set_buffers`).

5. Starting the tasks (`task_start`).

Each task initially waits to be started and to obtain its task and channel information (by doing `get_task` and `get_channel` calls). After that, it typically enters its processing loop.

When a certain task graph is active and executing in steady state, it can be dynamically reconfigured by the reconfiguration manager. This manager reconfigures other tasks by issuing reconfiguration commands (such as start, stop, restart, suspend, resume and destroy). In our model, tasks and channels are tightly coupled, and channels cannot be reconfigured unless their corresponding tasks have been reconfigured first. A key issue is how to halt the tasks' steady-state operation (active entities). Since the reconfiguration manager does not exactly know the progress of each task (communicating this information back and forth would incur too much overhead), this is performed with a *handshake* protocol: the reconfiguration manager requests that a certain task stops or suspends its operation, and the corresponding task acknowledges it when it has done so. Tasks can check whether a reconfiguration command has been issued by using the `task_check_reconfigure` primitive. This primitive is called in the task loop, its location in the loop corresponds to a reconfiguration point discussed in Section 6.1.1. The tasks then acknowledge the configuration command (using the `wait_restart` or `wait_resume` primitives), after performing some state cleanup actions (if necessary).

C-HEAP allows the following reconfiguration actions on a channel: destroy a channel, reconnect a channel, change the capacity, change the size of the tokens, change operation mode flags and move the buffers to a different memory or memory location. In the case of multi-cast channels, individual branches can be added or removed at run-time.

6.2. Implementation of (re)configuration protocol

This section describes the implementations of the configuration and reconfiguration protocol for the different processing devices and abstraction levels. We will focus on configuration and reconfiguration of the tasks only, since the other entities (channels, tokens) are passive and (re)configuring them is trivial and the same for all implementations (basically just updating their data structures in memory).

6.2.1. *Simulated implementations*

Within this category we can also distinguish between a software and a hardware implementation. The main difference between all the software implementations concerns how to implement a task. In our case, a task may be implemented as a (UNIX) process or thread. Threads may come in different flavours, depending on the run-time environment or operating system used (e.g. PAMELA/pSOS/Linux). Multiple tasks may be mapped onto the same processor. In this case the operating system takes care of the scheduling of these tasks, and the designer has control over the priority difference between these tasks. The reconfiguration manager is a task of its own, therefore the other reconfiguration commands (stop, restart, suspend, resume and destroy) and their corresponding acknowledgements are communicated to and from the other tasks by using inter-task communication mechanisms. The PAMELA implementation uses semaphores for this purpose, and the other implementations (pSOS and LINUX) use message queues.

In the case of the ASIPs in our architecture, the reconfiguration commands and acknowledgements are written into the globally visible piece of memory inside the ASIP, in a manner similar to that in the signalling mechanism for synchronisation. In the case of DSPs, some memory locations allocated in shared memory are used for communicating reconfiguration commands and acknowledgements. Our ASIPs and DSPs are single-tasking so no operating system support is required.

In the case of hardware devices (ASICs) designed, implemented and simulated using VHDL, a synchronisation shell is available (see Figure 8) to synchronise the data communication. This shell also contains two registers dedicated for reconfiguration. One is a command register in which the reconfiguration manager writes to reconfigure the hardware task. The other register is used by the co-processor to acknowledge the reconfiguration commands. The reconfiguration manager may receive this acknowledgement in two different ways: by polling this register or by using an interrupt. As in the case of synchronisation signalling, both approaches have their pros and cons, depending on the expected acknowledgement latency. A set of control lines from the shell to the co-processor are used to control its execution.

7. Case study: a multi-standard video and image codec

We will now present a design case of a video and image codec [37] based on the C-HEAP design methodology and architecture template. The codec supports both the MPEG-4 (Simple Profile) and H.263-based video coding standards and the JPEG-based image coding standard. These coding standards use coding methods [6] based on Discrete Cosine Transform (DCT). Figures 12 and 13 depict the typical video codec and JPEG image codec, respectively. FoIn video coding, each MB is coded using a combination of motion-compensated temporal prediction and transform coding. That is, an MB is first predicted on the basis of a matching MB in a previously coded reference frame. The location of the best matching MB is estimated by the motion estimator (ME). We used the 3DRS algorithm [17] for motion estimation. The displacement between the current MB and the matching MB is represented by the motion vector (MV). On the basis of the MV, the current MB is predicted (P) and motion-compensated (MC), resulting in the prediction error for the current MB. The prediction error of the current MB is then transformed using DCT and the resulting DCT coefficients are quantised (Q), zig-zag (ZZ) scanned and entropy coded (run-length encoding (RLE) and variable length encoded (VLE)). In addition, the quantised coefficients are inverse quantised (IQ), inverse transformed (IDCT) and inverse motion-compensated (IMC) and stored in the loop memory as the next reference frame. In JPEG image coding, each MB is transformed using DCT and the resulting DCT coefficients are quantised, zig-zag scanned and entropy coded.

We will first explain the followed design trajectory in Section 7.1. The design and architecture of the codec will be presented in Section 7.2, followed by the evaluation and benchmarking of the design in Section 7.3.

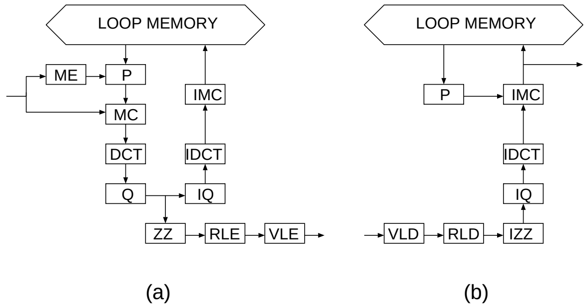

Figure 12. MPEG-4 (SP) and H.263 (a) video encoder and (b) video decoder

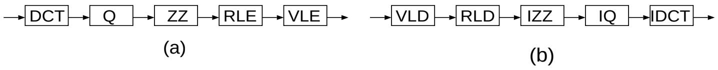

Figure 13. JPEG (a) image encoder and (b) image decoder

7.1. Design trajectory

Figure 14 summarises the design and verification flow of the video/image codec. The arrows indicate the subsequent steps taken in the top-down flow. After each step, the design was verified against the previous abstraction level. During the design, steps were taken backwards too in an iterative way (see Figure 2); these arrows have been omitted from figure.

The functional requirements of the video/image codec were first specified in a textual format. Then a functional description was constructed in C for the codec. In order to obtain an estimate of the computational load, the C code was profiled on an ARM processor. This yielded the clock frequency required for a software-only solution and a breakdown of the computational load for different functional modules of the codec. The latter was used as a starting point for determining the hardware-software partitioning. Our main strategy was to implement the computationally intensive parts that were common to all standards in hardware, while keeping the encoding parts that were different over the standards and the control-related parts in software. This provides the flexibility to adapt the system to different standards.

Thereafter, the selected hardware parts were clustered into processors, resulting in four processors for the video/image codec: 1) pixel processor, 2) motion estimator processor, 3) texture processor and 4) stream processor. The objective of this clustering was twofold. First, to reduce the synchronisation overhead between hardware and software. Secondly, to prevent all local communication inside the processors from generating global bus activity. The latter is crucial for low power. The communication between the processors is based on the previously de-

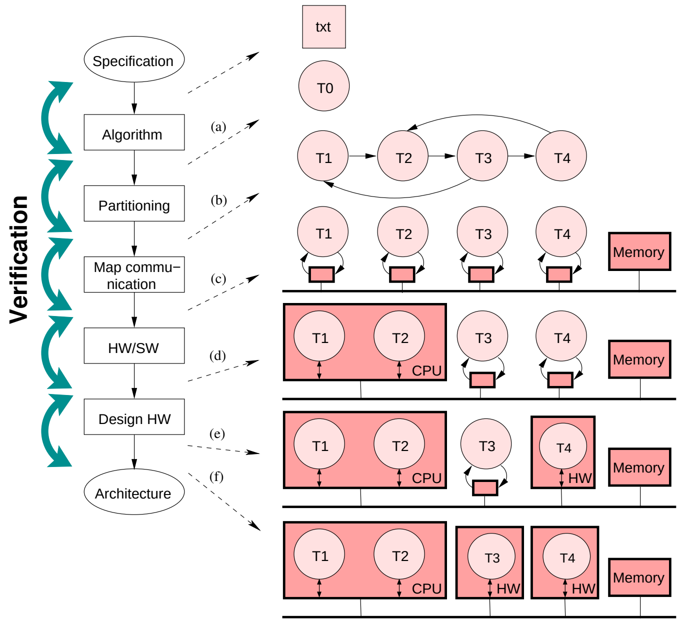

Figure 14. Top-down design and verification flow of the video/image codec

scribed C-HEAP protocol. The partitioned codec application (shown in Figure 15) was simulated extensively, and the (de)coded images were verified by comparing them with the output images of the original code. Since the codecs should behave identical, the output images of both implementations should be the same. This simulation set-up corresponds to verification step (b) in Figure 14.

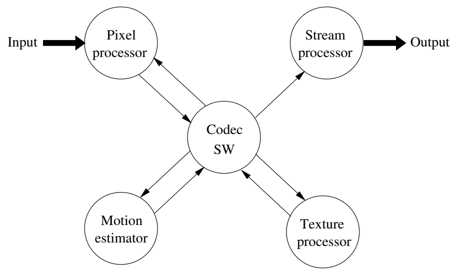

Figure 15. Partitioned codec application

The input parameters to the hardware processors consisted of two parts: general settings (e.g. number of pixels in the horizontal and vertical directions of a picture) and run-time parameters (e.g. the coordinates of the macroblock (MB) currently encoded). In the functional simulations, pointers to all the parameters were passed to the processors and were then accessed by the processors. Once the communication had been mapped to a hardware interconnect, the processors had to read the general parameters for each macroblock from a shared memory. However, it is much more efficient to store the general parameters locally during initialisation. Therefore, an initialisation mode was added to each processor, during which the parameters were read and stored locally. Only the run-time parameters which vary across macroblocks are communicated. This reduces the bandwidth on the bus considerably.

System simulations were carried out using a bit- and cycle-true version of the C-HEAP protocol for the communication based on the ARM bus. The models employed for the processing parts were abstract processes (UNIX), while the communication models were cycle-true (verification step (c) in Figure 14). These simulations were used to obtain bus utilisation figures, and to optimise the communication structure. This resulted in the use of two busses, one for the control data and the other for the pixel data.

The penultimate step focused on improving the performance of the implementation. The result of the previous step was a system without concurrency. A software task starts a hardware task and awaits the completion of that hardware task (and vice versa), thus using the resources inefficiently. Concurrency can be implemented by pipelining the software and hardware tasks. However, some dependencies between software and hardware needed to be broken, e.g. through modification of algorithmic feedback loops. Since the breaking of the dependencies led to a different behaviour, extensive simulations were carried out in order to verify that the compression ratio and SNR (signal-to-noise ratio) were comparable with that of the original reference C description. The C-HEAP communication and synchronisation primitives combined with bit- and cycle-true models for the ARM processor and busses were used to verify performance. These simulations showed that the number of clock cycles required for the software tasks was within the available cycle budget. This step corresponds to verification step (d) in Figure 14.

The last step related to the design of hardware processors. Each hardware processor consists of a customised VLIW core (ASIP) generated by an architectural synthesis tool [2]. This core consists of an ALU, one or more arithemetic control units (ACUs), a small instruction memory, a bus, a controller and some application specific units ASUs. The ASUs are responsible for accelerating the computationally intensive operations like (I)DCT, (I)Q, (I)MC, etc. were designed in RTL-C. The VLIW core, programmed in the C language, is responsible for scheduling the ASUs.

The resulting architecture was simulated by using the VHDL version for the C-HEAP communication primitives based hardware processors and the bit- and cycle-true C-HEAP communication primitives based software task and communication busses. This simulation was performed via C-VHDL co-simulation, and was used to verify the hardware processors. This corresponds to verification steps (e) and (f) in Figure 14.

7.2. Implementation details of video and image codec

The architecture of the video and image codec is shown in Figure 16. It is a dual-bus architecture with separate control and data busses. The software tasks are executed on the ARM processor and the hardware tasks are executed on the customised VLIW processors (namely the pixel processor, motion estimator processor, texture processor and stream processor). An example of the architecture of such a customised VLIW is shown in Figure 17. The pixel processor (PP) communicates with the pixel domain (image sensor or display) and performs line-to-stripe conversion and vice versa for a video/image encode and decode operation, respectively. The motion estimator processor (MEP) evaluates a set of candidate vectors received from software and selects the best vector for pixel refinements. The output of the MEP consists of motion vectors, sum-of-absolute-difference (SAD) values, and intra metrics. This information is used in software (running on ARM) to determine the encoding approach for the current MB. The MEP is used only for video-encoding operations.

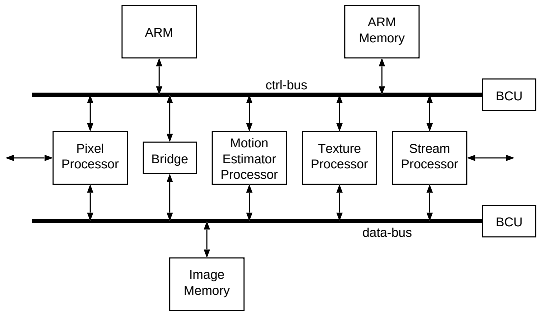

Figure 16. Architecture instance of the multi-standard video and image codec

The texture processor (TP) encodes and decodes MBs and stores the decoded MBs in the loop memory for the video encode operation. The output of the TP consists of VLE codes for the DCT coefficients of the current MB. The TP also does the core functionality for video decode and JPEG encode/decode. The stream processor (SP) packs the VLE codes generated by the TP and software for the video/image encode operation. The SP also unpacks the VLE codes for the video/image decode operation. The SP communicates with the compressed domain (storage or communication channel) for the encode and decode operations.

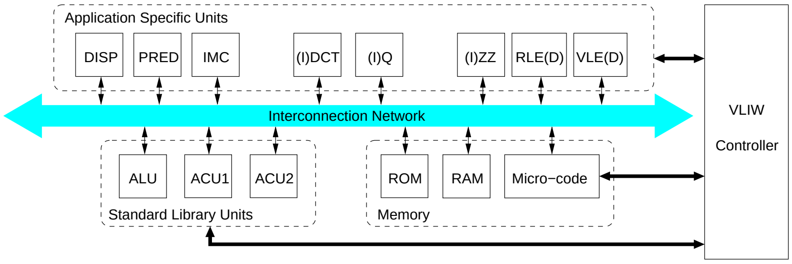

Figure 17. VLIW architecture of the texture processor

Figure 18 shows the pipelining and concurrency of various HW/SW tasks. The pixel processor is one stripe ahead of the motion estimator processor, which in turn is two macroblocks ahead of the texture processor. The bit rate control running in software comes one macroblock after the texture processor. Finally, the stream processor lags one stripe behind.

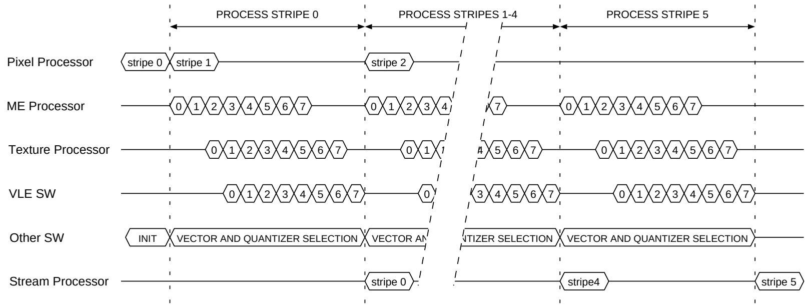

Figure 18. Pipelining of hardware and software tasks for the video encode function

The software tasks performed on the ARM processor (for the video/image codec) consist of:

– Initialisation and control-dependent actions.
– Candidate vector generation for motion estimation.
– Compression mode selection for video (inter/intra).
– VLE and VLD for the headers and motion vectors.
– Bit rate control (for video and JPEG encode) and handshake with hardware processors.

The embedded memory in our implementation consists of two parts, namely the ARM memory and the image memory. The ARM memory contains the operating system, the application software and the data structures. The image memory consists of the buffer space for the pixel processor and the loop memory. The size of the loop memory can be reduced by using an embedded compression technique described in [28]. This allows us to keep it on-chip for fast access and low power.

Table IV presents the average synchronisation latencies in our codec design, given in the number of clock cycles. The numbers are about the same for both the ARM and the customised VLIWs (the synchronisation protocol is essentially implemented in software). The numbers given for the `claim_data` and `claim_space` primitives hold for the case that data/space is available so the task will not be blocked. Complementary to the results given in Section 5.3 and [21] for C-HEAP synchronisation latencies using hardware shells (18 cycles with a completely congested bus), these figures again prove the efficiency of the C-HEAP protocol.

Table IV. Codec synchronisation latencies in clock cycles

Synchronisation primitives	Latency
`claim_data/space`	40
`release_data/space`	36

7.3. Evaluation and benchmarking

We evaluated the efficiency of our design against a number of existing video and image codecs. The results are presented in Table V. As can be seen in the table, our design is more efficient in terms of area, power and performance than the rest. The Hitachi design is only capable of encoding and is a complete software solution, and results in a large area and power consumption. The other extreme is the Fujitsu codec, which is completely implemented in hardware. Compared to our design, the area is larger and it lacks the flexibility of a partial software solution (only supports one standard). The area of the TI codec is unknown, but it is expected to be larger than ours, considering the area of the TMS320C5X DSP and the extra hardware assistance. Of all the designs, only the design of Motorola and our own design use a multi-processor solution, and the processors in both systems synchronise with each other on a macroblock basis. Very significant is also the small loop memory needed by our design because of the embedded compression.

8. Conclusions and future work

In this article we have addressed the design of complex signal processing embedded systems. We have applied a top-down design methodology,

Table V. Comparison of different existing codecs

	Supported standard(s)	Frame size*	Frame rate (fps)	Area** (mm^2)	Energy (μJ/MB)	Sync. gran.	HW-SW partitioning	Loop memory
Hitachi***	MPEG-4 (SP@L1) encoder	QCIF	15	43	115	n.a.	Fused RISC/ DSP CPU (SW solution)	1.0 Mbits on-chip SRAM
Motorola***	MPEG-4 (SP@L1), JPEG codec	QCIF	15	40	67	MB	CPU + texture proc., ME proc.	2.5 Mbits on-chip SRAM
Fujitsu	MPEG-4 (SP@L3) codec	CIF	15	28	4.9****	n.a.	ASIC (HW solution)	Off-chip SDRAM
Texas Instruments	MPEG-4 (SP@L1) codec	QCIF	30	?	17	n.a.	TMS320C5X (HW assist for video kernels)	On-chip SRAM
Our design	MPEG-4 (SP@L1), H.263, JPEG codec	CIF	30	17	5.5	MB	CPU + pixel proc., texture proc., ME proc., stream proc.	0.5 Mbits on-chip SRAM

*QCIF: 176×144 pixels, CIF: 352×288 pixels
**In 0.18 μm technology
***Published in ISSCC 2002
****Does not include off-chip SDRAM power

starting with a functional description and proceeding to a silicon implementation in an incremental way. Different abstraction levels that are traversed throughout the design process have been identified. Design decisions taken at each level can be evaluated by means of (multi-level) simulation.

The important aspects of our methodology are the definition of a scalable, flexible and modular architecture template along with a standardised protocol for communication and (re)configuration called C-HEAP. The use of a template helps shorten the design time and reduce the design costs by facilitating reuse of components and providing the models and tools associated with it. We propose an architecture template based on Distributed Shared Memory that allows for the use of a variety of processing devices such as CPUs, DSPs, ASIPs and dedicated hardware to achieve a balance between performance, flexibility, and efficiency. The architecture is scalable by allowing the addition of extra processing devices and extension of the intercommunication network.

The C-HEAP protocol, based on Kahn Process Networks, has been defined to handle the communication between different processing devices. This protocol is implemented in a distributed manner, so that it is scalable and enables fast synchronisation, thereby making effective multi-processing possible. Furthermore, the protocol has been extended to allow the application to configure the system during start-up, and to reconfigure it at run-time. The C-HEAP protocol comes with well-defined primitives and a set of libraries containing the implementations for different abstraction levels and processing devices for simulation. For prototyping, we have similar implementations (for the relevant abstraction levels) of the C-HEAP primitives. This allows designers to quickly iterate between different abstraction levels and evaluate different hardware-software partitioning options, and hence to cover a much broader range of the design space in a limited amount of time.

We have illustrated the use of C-HEAP with the design case of a multi-standard video and image codec. The effectiveness of our combination of design methodology, architecture template and protocol was proven as our design turned out to be more efficient than other codecs found in literature. During the design phase, many iterations were performed across different abstraction levels to evaluate different hardware-software partitioning and mapping options. Due to the well-defined primitives and complete library implementations at all levels of the design flow the iteration time was minimised. Another very important aspect is the completely C based design flow. Fitting the architectural synthesis tool (for generating the customised VLIW cores) into our flow made it possible to use (with some re-writing) the C code from the higher abstraction levels to generate the hardware processors without writing any VHDL code by hand. This simplified testing by reusing the test method designed at the top level, shortened the design iteration time and made the design much easier to maintain and extend. The reconfiguration protocol was not used in this design as the supported function modes were sufficiently similar, which meant that switching between them could be realised centrally in the software task, instead of having to do it by reconfiguring the task graph. However, we foresee that future applications and systems will need this kind of reconfiguration possibilities.

C-HEAP has proven to be efficient for designing medium-grain signal processing systems such as encoders and decoders. Complete systems or applications (e.g. digital video recorders) could comprise of a combination of such sub-systems. Although the C-HEAP protocol is also capable of handling the communication between such sub-systems, larger systems will involve other issues, for instance memory usage. C-HEAP allows for quick synchronisation and therefore small token buffer sizes, but the tokens are locally allocated and managed per channel. This has a number of disadvantages. First, data can not be passed from one channel to another without being copyied because full and empty tokens circulate in FIFO-fashion on one channel. Filters, (de)multiplexers and shufflers are then more expensive to implement [26]. Secondly, buffer usage cannot be optimised globally (over channels). For example, the number of tokens in a system as a whole may be considerably smaller than the sum of the worst-case token usage of all the channels combined. Thirdly, run-time choices of token size and location (memory) are precluded. Global buffer management does not suffer from these drawbacks. Arachne [24] is an example of a protocol based on global buffer management which offers more flexibility and dynamism, and allows more efficient memory management. It uses a central token manager and is therefore less scalable and has longer synchronisation latencies than C-HEAP. We feel its advantages make it suitable for large systems, in which higher latency is acceptable. Integration of the two techniques in which C-HEAP is used to design efficient sub-systems which are then combined into one large system using Arachne could result in an optimum overall system. This investigation is a topic for future work.

References

1. 'AMBA specification overview'. ARM, http://www.arm.com/Pro+Peripherals/AMBA.
2. 'A|RT Designer'. Adelante Technologies, http://www.adelantetechnologies.com.
3. 'The CoreConnect Bus Architecture'. IBM, 1999, http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB/crcon_wp.pdf.
4. 'PAMELA - A performance modeling language'. http://ce-serv.et.tudelft.nl.
5. Baghdadi, A., D. Lyonnard, N. Zergainoh, and A. A. Jerraya: 2001, 'An efficient architecture model for systematic design of application-specific multiprocessor SoC'. In: *Proceedings of the Design, Automation and Test in Europe (DATE) Conference and Exhibition*. pp. 55–62.
6. Bhaskaran, V. and K. Konstandinitis: 1996, 'Image and video compression standards; algorithms and architectures'. In: *Kluwer Academic Publishers*.
7. Bilsen, G., M. Engels, R. Lauwereins, and J. Peperstraete: 1994, 'Static scheduling of multi-rate and cyclo-static DSP applications'. In: *Workshop on VLSI Signal Processing*. pp. 137–146.

8. Bolsens, I., H. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest: 1997, 'Hardware/software co-design of digital telecommunication systems'. In: *Proceedings of the IEEE*. pp. 391–418.
9. Brunel, J.-Y.: 1999, 'COSY tutorial: IP-based system design'. In: *Proceedings of the VLSI Conference*.
10. Brunel, J.-Y., E. A. de Kock, W. M. Kruijtzer, H. J. H. N. Kenter, and W. J. M. Smits: 1999, 'Communication refinement in video systems on chip'. In: *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES)*. pp. 142–146.
11. Brunel, J.-Y., W. Kruijtzer, H. Kenter, F. Ptrot, L. Pasquier, E. de Kock, and W. Smits: 2000, 'COSY communication IP's'. In: *Proceedings 37th Design Automation Conference*. pp. 406–410.
12. Brunel, J.-Y., A. Sangiovanni-Vincentelli, R. Kress, and W. Kruytzer: 1998, 'COSY: a methodology for system design based on reusable hardware & software IP's'. In: *Proceedings of the European Multimedia, Microprocessor Systems and Electronic Commerce Conference*. pp. 709–716.
13. Buck, J.: 1994, 'Static scheduling and code generation from dynamic dataflow graphs with integer valued control signals'. In: *Asilomar Conference Signals Systems and Computers, Pacific Grove, California*. pp. 508–513.
14. Busa, N., G. Alkadi, M. Verberne, R. Peset Llopis, and S. Ramanathan: 2002, 'RAPIDO: A modular, multi-board, heterogeneous multi-processor, PCI bus based prototyping framework for the validation of SoC VLSI designs'. accepted for the 13th IEEE Workshop on Rapid System Prototyping.
15. Catthoor, F., F. Franssen, S. Wuytack, L. Nachtergaele, and H. de Man: 1994, 'Global communication and memory optimizing transformations for low-power signal processing systems'. In: *Proceedings of the IEEE Workshop on Signal Processing, La Jolla, CA*.
16. Dasygenis, M., N. Kroupis, A. Argyriou, K. Tatas, D. Soudris, and N. Zervas: 2001, 'A memory management approach for efficient implementation of multimedia kernels on programmable architectures'. In: *Proceedings of the IEEE Computer Society Annual Workshop on VLSI*.
17. de Haan, G. and P. W. A. C. Biezen: 1995, 'Sub-pixel motion estimation with 3-D recursive search block-matching'. In: *Signal Processing: Image Communications 6*. pp. 485–498.
18. de Kock, E. A., G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers: 2000, 'YAPI: Application modelling for signal processing systems'. In: *Proceedings of the Design Automation Conference*. pp. 402–405.
19. Ernst, R.: 1998, 'Codesign of embedded systems: Status and trends'. In: *IEEE Design & Test of Computers*. pp. 45–54.
20. Ferrari, A. and A. Sangiovanni-Vincentelli: 1999, 'System design: traditional concepts and new paradigms'. In: *Proceedings of the International Conference on Computer Design*. pp. 2–12.
21. Gangwal, O. P., A. K. Nieuwland, and P. E. R. Lippens: 2001, 'A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems'. In: *Proceedings of the International Symposium on System Synthesis*. pp. 1–6.
22. Goossens, K. G. W. and O. P. Gangwal: 2002, 'The Cost of Communication Protocols and Coordination Languages in Embedded Systems'. In: *Proceedings of the $5^t h$ international conference on Coordination languages and models, COORDINATION 2002, York (UK)*. pp. 174–190.

23. Hennessy, J. L. and D. A. Patterson: 1995, *Computer Architecture: A Quantitative Approach, Second Edition*. San Mateo, CA: Morgan Kaufmann.
24. K. G. W. Goossens: 2001, 'A protocol and memory manager for on-chip communication'. In: *Proceedings of the International Symposium on Circuits and Systems*, Vol. II. Sydney, pp. 225–228.
25. Kahn, G.: 1974, 'The semantics of a simple language for parallel programming'. In: *Information Processing*. J.L. Rosenfeld, Ed. North-Holland Publishing Co.
26. Kang, J., A. van der Werf, and P. E. R. Lippens: 2000, 'Mapping array communication onto FIFO communication – Towards an implementation'. In: *Proceedings of the International Symposium on System Synthesis*. pp. 207–213.
27. Keutzer, K., S. Malik, R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli: 2000, 'System-level design: orthogonalization of concerns and platform-based design'. In: *IEEE transactions on computer-aided design of integrated circuits and systems*, Vol. 19. pp. 1523–1543.
28. Kleihorst, R. P. and R. J. van der Vleuten: 2000, 'DCT-domain embedded memory compression for hybrid video coders'. In: *Journal of VLSI Signal Processing Systems*, Vol. 24. pp. 31–41.
29. Lee, E. A. and D. G. Messerschmidt: 1987, 'Static Scheduling of Synchronous Data Flow Graphs for Digital Signal Processors'. In: *Proceedings of the IEEE*, Vol. 75. pp. 1235–1245.
30. Lee, E. A. and T. M. Parks: 1995, 'Dataflow process networks'. In: *Proceedings of the IEEE*, Vol. 83, no. 5. pp. 773–799.
31. Leijten, J. A. J., J. L. van Meerbergen, A. H. Timmer, and J. A. G. Jess: 2000, 'Prophid: A platform-based design method'. In: *Journal of Design Automation for Embedded Systems*, Vol. 6, no. 1. pp. 5–37.
32. Lyonnard, D., S. Yoo, A. Baghdadi, and A. A. Jerraya: 2001, 'Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip'. In: *Proceedings of the Design Automation Conference*. pp. 518–523.
33. Mellor-Crummey, J. M. and M. L. Scott: 1991, 'Algorithms for scalable synchronization on shared-memory multiprocessors'. *ACM Transactions on Computers Systems* **9**, 21–65.
34. Nachtergaele, L., F. Catthoor, F. Balasa, F. Franssen, E. de Greef, H. Samsom, and H. de Man: 1995, 'Optimization of memory organization and hierarchy for decreased size and power in video and image processing systems'. In: *Proceedings of the International Workshop on Memory Technology, San Jose, CA*.
35. Nachtergaele, L., D. Molenaar, B. Vanhoof, F. Catthoor, and H. de Man: 1998, 'System-level power optimization of video codecs on embedded cores: A systematic approach'. In: *Journal of VLSI Signal Processing*, Vol. 18. pp. 89–109.
36. Nieuwland, A. K. and P. E. R. Lippens: 1998, 'A heterogeneous HW-SW architecture for hand-held multi-media terminals'. In: *IEEE workshop on Signal Processing Systems*. pp. 113–122.
37. Peset Llopis, R., M. Oosterhuis, S. Ramanathan, P. E. R. Lippens, A. van der Werf, S. Maul, and J. Lin: 2001, 'HW-SW co-design and verification of a multi-standard video and image codec'. In: *Proceedings of the IEEE International Symposium on Quality Electronic Design*. pp. 393–398.
38. Sangiovanni-Vincentelli, A. and G. Martin: 2001, 'Platform-based design and software design methodology for embedded systems'. In: *IEEE Design & Test*. pp. 23–33.

39. Sasaki, H.: 1996, 'Multimedia complex on a chip'. In: *Proceedings of the International Solid State Circuits Conference*. pp. 16–19.
40. Sgroi, M., M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli: 2001, 'Addressing the system-on-a-chip interconnect woes through communication-based design'. In: *Proceedings of Design Automation Conference*. pp. 667–672.
41. Soudris, D., N. Zervas, A. Argyriou, M. Dasygenis, K. Tatas, C. Goutis, and A. Thanailakis: 2000, 'Data-reuse and parallel embedded architecures for low-power, real-time multimedia applications'. In: *Proceedings of the IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation*. pp. 243–254.
42. Tanenbaum, A. S.: 1981, *Computer networks*. The Netherlands: Prentice Hall International Inc.
43. van Ommering, R., F. van der Linden, J. Kramer, and J. Magee: 2000, 'The Koala Component Model for Consumer Electronics Software'. In: *IEEE Computer*, Vol. 33. pp. 78–85.
44. van Rompaey, K., D. Verkest, I. Bolsens, and H. de Man: 1996, 'CoWare - A design environment for heterogeneous Hardware/Software systems'. In: *Proceedings of the Design Automation for Embedded Systems Conference*. pp. 357–386.
45. Vercauteren, S., B. Lin, and H. D. Man: 1996, 'Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications'. In: *Proceedings of the Design Automation Conference*.