A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems

Om Prakash Gangwal André Nieuwland Paul Lippens[†]

Embedded Systems Architectures on Silicon
Philips Research Laboratories, The Netherlands
o.p.gangwal@philips.com, andre.nieuwland@philips.com

ABSTRACT

This paper describes the implementation of a data-synchronization scheme that can be used in the functional description and hardware realization of algorithms for heterogeneous multi-processor architectures. In this scheme, synchronization primitives are chosen such that they can be implemented efficiently in both hardware and software on distributed shared memory architectures, without the need for atomic semaphore instructions. The proposed solution is flexible as the configuration of the data synchronization is programmable even after a hardware realization. It is also scalable since it can be implemented without the need for central resources. We show with experiments that distributed implementations are needed for scalable and high performance systems-on-a-chip.

1. INTRODUCTION

Signal-processing functions determine the performance requirements for many products based on standards like MPEG-x, DVB, DAB, and UMTS. This calls for efficient implementations of the signal processing components of these products. However, because of evolving standards and changing market requirements, the implementation requires flexibility and scalability as well. A macro-pipeline setup is a natural way to model these applications, since streams of data are processed; in this setup the functions (tasks) are the stages and there are buffers between the stages to form the pipeline. This is a way to exploit task-level parallelism (TLP), because all stages can operate in parallel.

In a Multiple Instruction Multiple Data (MIMD) composition each processor can have its own function on its own data. In this way it is possible to optimize processors to their function. This leads to more efficient solutions both in power and area. For example, for a demanding application, a RISC-CPU, a VLIW, a DSP, an ASIP, and dedicated hardware units can be composed into a single architecture as shown in Figure 1. To fully exploit the available processing power, we need an efficient yet scalable and flexible way of data communication.

[†]Paul Lippens is currently working with Magma design automation.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

In [1, 2, 3, 4], inter-task data transportation and synchronization are combined in a single action (e.g. *read*/*receive*/*write*/*send* operations). A survey on communication in HW-SW embedded systems can be found in [5]. Our proposal is significantly different from previous art, as we advocate the separation of data transportation and synchronization. This provides the flexibility to perform data transportation at a different granularity than synchronization. This separation is especially advantageous in shared memory architectures since only synchronization primitives are needed and no copying of data is required.

In homogeneous shared-memory multi-processor architectures, plenty of algorithms for scalable synchronization exist (see [9]). In this domain, one resource is shared by multiple tasks. This problem is solved either by *spin-lock* or *barrier algorithms* (see [9]), which require special instructions (e.g. `test_and_set`, `swap_with_memory`) or atomic read-modify-write operations.

In standard synchronization algorithms, the same resource may be claimed multiple times by a specific task before it is claimed by another task. With data communication, storage for data elements is claimed by the two tasks in an alternating fashion: First the storage will be claimed for writing by one task, and thereafter by another task for reading. Note that when buffering is available, one task can claim the next storage for data elements before the other task has read the data elements (stored recently). However, one task can not claim the same storage for data elements, which it has filled, until the other task has read that data. As we show in this paper, we can implement this form of synchronization without any special instructions such as atomic read-modify-write. Furthermore, we provide an efficient and flexible implementation of these primitives to support the mapping of signal processing applications onto heterogeneous multi-processor architectures.

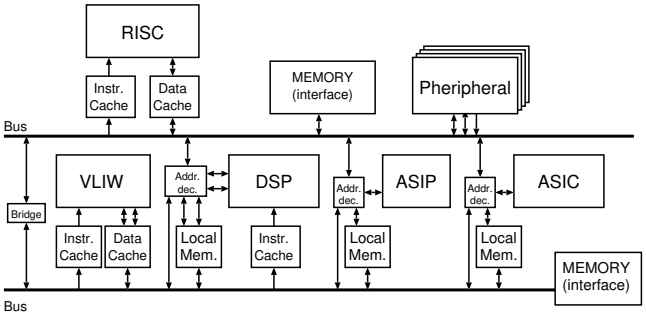


Figure 1: A MIMD architecture

The rest of this paper is organized as follows. In Section 2, we present our target architecture. In Section 3, we introduce the syn-

chronization protocol that is used to specify the signal processing functionality. Section 4 describes the implementation of the protocol on a distributed shared-memory architecture. In Section 5, we present experimental results, followed by conclusions in Section 6.

2. TARGET ARCHITECTURE

In an MIMD setup, the communication infrastructure is as important as the individual components for the overall performance of the system. Moreover, while the choice of processing components depends on the application, the same infrastructure can be used for several applications as long as this infrastructure is scalable.

A common approach to realize MIMD architectures with a macro-pipeline setup is to attach a number of (co-)processors to the main processor using some communication network. The main processor controls the co-processors, and the co-processors implement the functionality of the tasks. We call this kind of architecture a *co-processor* architecture. In order to control the co-processors, the main processor either polls (i.e. repeatedly reads) 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. Similarly, the time spent in polling should be low. This can be accomplished using (very) large grain synchronization. For example, in a video context, synchronizing on a field or a frame basis.

Large grain synchronization 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 subsided if 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 to decrease the on-chip buffer size. One way to accomplish this is to reduce the synchronization grain size. However, in a co-processor architecture, this would lead to unacceptable high interrupt rates for the main processor.

Therefore, we propose a different approach where the (co-)processors are autonomous with respect to synchronization and do not require service from a main processor. We call this kind of architecture a *multi-processor* architecture. With this approach, we can go to a smaller grain of synchronization, which allows smaller buffer sizes, and therefore, enables on-chip communication. When we use multi-processor architectures for the example discussed above, we can synchronize on a line or block basis instead of synchronizing on a frame or field basis.

Multi-processor architectures are better scalable than co-processor architectures. Synchronization in a multi-processor architecture is challenging since all tasks are autonomously scheduled. In this paper, we define and implement a data synchronization protocol for multi-processor architectures. However, the same protocol can be used in co-processor architectures.

3. SYNCHRONIZATION PROTOCOL

Our application specification is based on Kahn process networks [6]. In this model, the overall application is decomposed in a number of parallel processes communicating via *point-to-point unidirectional unbounded channels* with first-in-first-out (FIFO) behavior. In this paper, we refer to these processes as tasks. In this model, when a task wants to read from a channel, and there is no data available, the task will block. However, write actions are non-blocking.

In our model, *FIFOs* are *bounded* since we are addressing efficient implementation of applications and not only the specification. This means that a task will also block when it wants to write to a channel if the associated FIFO is full.

We clearly separate synchronization from data transportation since in a shared memory architecture no copying of data is required. In this context, data transportation is the set of activities required to transport the *data* from one task to another via a (buffered) channel. Synchronization is the set of activities required to *query/claim/release* some amount of the data on a channel. The synchronization takes place on a per *token* basis. While a token is the unit of synchronization, the amount of data associated with a token can vary. However, the size of a token for each channel is set at system configuration.

For efficiency reasons, all communication buffer memory is allocated at setup and is reused during operation. This implies the reuse of physical channel buffers 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.

At the data producing side, we want to claim empty token buffers (`claim_space`) and release full token buffers (`release_data`). At the consuming side we want to claim filled token buffers (`claim_data`) and release empty ones (`release_space`). Claiming a token buffer is blocking, i.e. when no buffer is available the task blocks. Releasing is non-blocking.

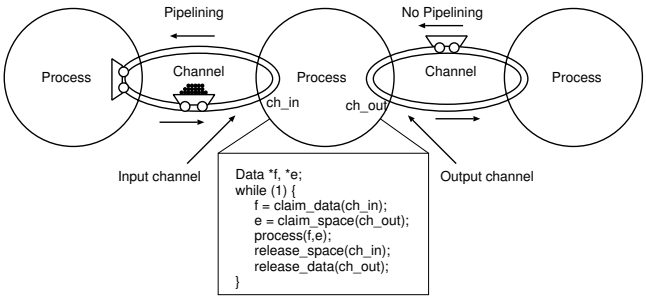


Figure 2: The four synchronization primitives

Figure 2 illustrates the use of the four primitives. The buffers are visualized as train wagons and 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 processing, the emptied wagon is pushed back on the input channel and the filled one on 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.

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 or equal to the number of consecutive `claim` calls, otherwise deadlock occurs.

A non-blocking version of `claim_data/space` primitives called `query_data/space` is also defined to query the availability of data/space. That information can be used by a task to perform some other work until the data/space becomes available, or in a multi-tasking environment where a task itself could indicate the readiness for task switching to an operating system or to a task scheduler.

This synchronization protocol can be extended to exchange tokens between more than two tasks as shown in [7].

4. IMPLEMENTATIONS OF THE PROTOCOL

The protocol is defined and implemented such 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.

We describe the implementation of channel FIFO buffers that are flexible enough to facilitate implementations of the protocol in hardware and software, in Section 4.1. In Section 4.2, a generic implementation of synchronization primitives is explained. In Section *4.2.1*, we explain optimizations for the software implementation. Optimizations for the hardware implementation are explained in Section *4.2.2*.

4.1 Implementation of channel FIFO buffers

We allocate space in shared memory and control that space in a FIFO manner to implement a channel buffer. This gives us the flexibility to tune the channel buffer and token sizes for an application even after we made a hardware realization. Furthermore, one can change the number of channel buffers and their logical interconnection structure in order to map different applications on the same hardware. In order to provide FIFO behavior of a buffer, some administrative information has to be maintained. The administrative information contains some static and dynamic values. The static values are those values that are written only once when the system is configured. The dynamic values are those values that are modified while the system is running. 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*/*etokens*.
2. *writec* and *etokens*/*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. For example, when the initial value of *ftokens* is 4, a producing task increments *ftokens*' value from 4 to 5 (after producing data) and simultaneously the consuming task connected to the same channel decrements the same *ftokens*' value from 4 to 3 (after consuming data), which results in a wrong value of *ftokens* (i.e. either 5 or 3) where the correct value after these operations should be 4. Since two concurrent tasks modify the same variable, access to that variable needs to be protected 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* is not yet incremented to make the value of *writec* correct (i.e. 17). Therefore *readc* and *ftokens* have to be updated by a consumer task 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 producer (consumer) task never sees more empty (full) tokens than available at any time. The *ftokens*/*etokens* field can always be derived from the counter values. The counters are initialized to zero and count modulo the *maxtokens* value. To solve the ambiguity problem (whether 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 initialized 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 the sliding window protocol for data communication [8]. We choose option 3 since with this option no consistency problem arises

4.2 Implementations of the 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 calculated 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.

To facilitate consecutive `claim_data/space` calls, a separate reservation counter (i.e. *readrsvc* corresponding to *readc* and *writersvc* corresponding to *writec*) is implemented. The reservation counters *readrsvc* and *writersvc* behave in the same manner as the *readc* or *writec*. However, they move ahead of them. The reservation counter value is private for each task. With the addition of the reservation scheme, a consuming (producing) task would perform comparisons of the *readrsvc (writersvc)* and *writec(readc)* on a `claim_data` (`claim_space`) call. When the call becomes successful, the token pointer corresponding to the reservation counter *readrsvc (writersvc)* is calculated and the *readrsvc* (*writersvc*) is incremented. However, the reservation scheme does not change the actions required for `release_space` (`release_data`) call. Figure 3 shows a snapshot of a FIFO where two tokens are reserved for reading (writing) while four tokens are available for reading (writing). The figure also shows values of *readc*, *readrsvc*, *writec* and *writersvc* with associated wrap flag. Random accesses are permitted within the reserved tokens.

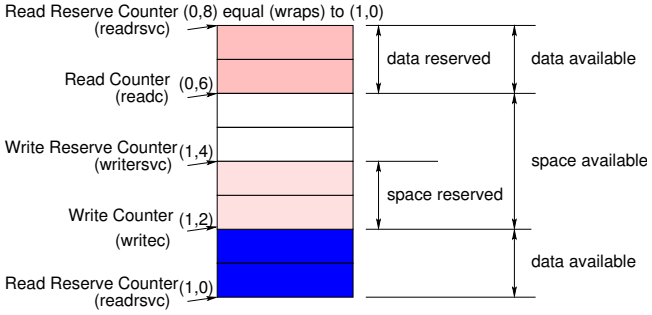


Figure 3: A snapshot of a FIFO with the reservation scheme

In principle, a blocked task polls on the counter value until its synchronization call becomes successful. This *polling-based synchronization* scheme is useful, if the counter value is updated around the same time when a task starts polling. However, polling is not always efficient since it increases 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 synchronization*. One should choose between polling-based and interrupt-based synchronization depending on the expected wait time, with respect to the time required to serve the wake-up signal.

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

4.2.1 Optimizations in the software implementation

With an *interrupt-based synchronization*, a task sends a signal at every *release* synchronization call. Whenever an interrupt signal is received in a generic processor, an interrupt service routine (ISR) is called. However, a task does not need these signals (hence no need to execute ISR) if it is *not* blocked. We reduced the number of interrupts seen by the processor by introducing a signal controller (see Figure 4). The signal controller has a mask register to mask the signal register (i.e. used to store signals written by tasks) during normal operation and enables the signal register only when the task is blocked. The signal controller is memory-mapped and instantiated for each generic processor in the architecture.

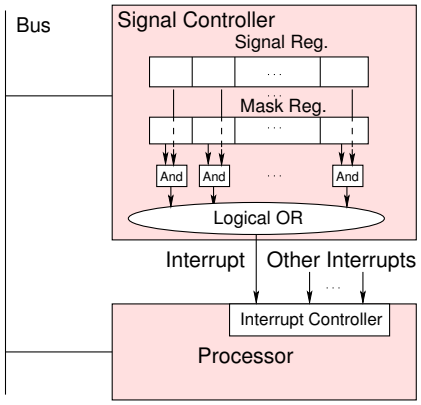


Figure 4: A Signal Controller for a processor

4.2.2 Optimizations in the hardware implementation

The hardware implementation of the channel synchronization protocol is called a *channel controller*. Channel controllers are instantiated per channel in a synchronization shell that is attached to an application specific component (see Figure 5). The channel controller is implemented with a generic bus interface to facilitate reuse with any particular bus by adding a *bus adapter*. The synchronization shell has a *signal register* that is used for receiving signals in the interrupt-based synchronization scheme. This signal register is instantiated only once per device. All registers in the synchronization shell are memory-mapped.

Only one channel controller is implemented in hardware, the mode (i.e. input or output) of a channel controller is set at the system configuration. This controller is instantiated for each channel.

In order to reduce the number of system bus accesses, some administrative information values are copied in the channel controller.

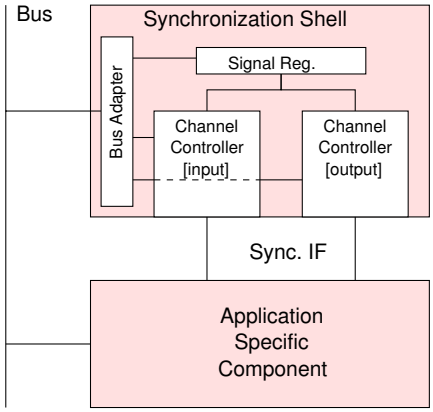


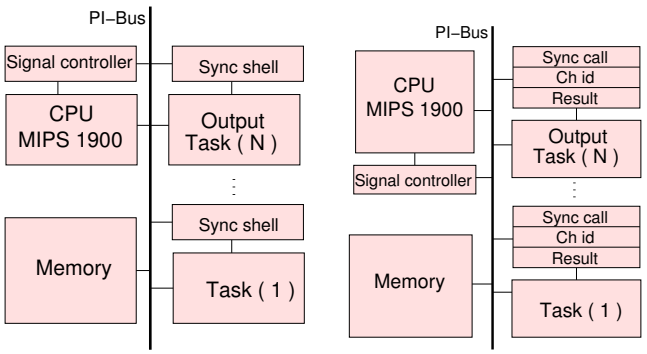
Figure 5: A Synchronization Shell for an Application Specific Component

The copied information also helps to reduce the delay in servicing synchronization calls.

In our implementation, when a coprocessor needs a token address (i.e. by a `claim_data/space` call), the token address can be made available in next clock cycle of the synchronization request. If more tokens are available then they can also be delivered every clock cycle, one by one. In this implementation, `release_space/data` calls also take 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 is implemented in 0.09 mm^2 for 0.18 micron CMOS technology.

5. RESULTS

We have extended our simulation models [10] to include all previously described implementations of the synchronization primitives. The effect of software and hardware optimizations are presented as well.



(a) Multi-processor (b) Co-processor

Figure 6: Architecture used for experiments

We built a multi-processor architecture (see Figure 6 a), in which all tasks are mapped onto hardware modules and synchronization is performed by the synchronization shells attached to the hardware modules. On encountering a synchronization call, the task reads the value of a token pointer if a token is available, otherwise it waits. A CPU (i.e. a low-power, low-cost MIPS running pSOS) is used

to configure the communication channels between the tasks. After configuration, the CPU goes idle.

We used the same set-up to simulate the classical co-processor architecture. For this purpose, we modified the synchronization shells such that the synchronization can be performed by the CPU (see Figure 6 b). Therefore, we added some (memory mapped) registers to the shells to hold the cause for a service request (*sync-call*), the channel-id for which the request was issued (*ch-id*), and a result register (*result*) to store the value returned by the CPU after completing the service request (*sync-call*).

On encountering a synchronization call, a task fills desired synchronization action, channel-index and a zero value in the *sync-call*, *ch-id* and *result* registers respectively. Subsequently, it sends an interrupt to the CPU. The interrupt can be sent directly to the CPU using traditional methods, e.g. daisy-chain, (i.e. *co-processor non-optimized case*) or by writing the task index of the task to the signal register in the signal controller, which is attached to the CPU (i.e. *co-processor optimized case*). The task polls the *result* register for a non-zero value. The task uses this value as a pointer to the requested token when the value of the *result* register is non-zero. On receiving an interrupt, the CPU invokes an ISR which identifies the source of the interrupt. Thereafter, the ISR reads the values of the *sync-call* and *ch-id* registers from the synchronization shell requesting service. Based on these values, the proper synchronization call is executed for that task. If the synchronization call does not pass, due to the fullness or emptiness of the FIFO buffer, the ISR saves the request until the FIFO buffer is updated by the connected task (producer or consumer). As soon as the FIFO buffer is updated, the saved synchronization call is invoked and the result is written in the result register of the pending task.

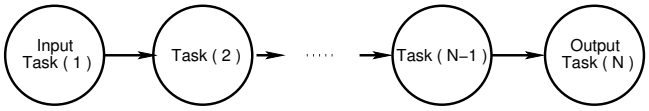


Figure 7: Application used in experiments

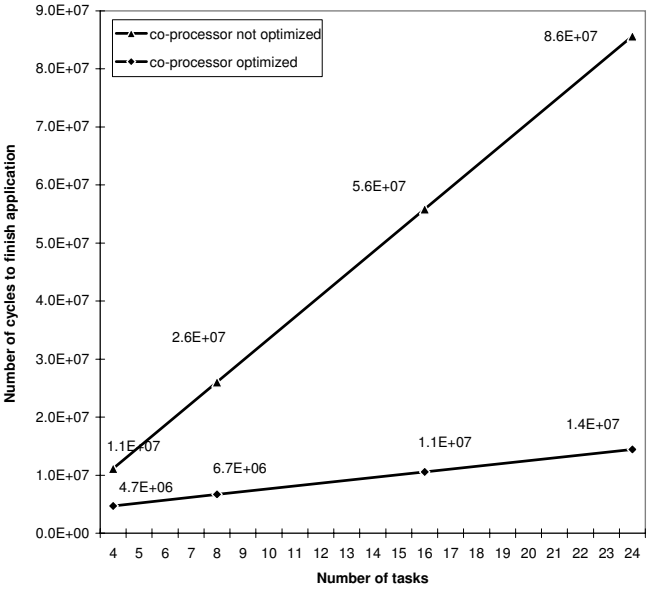


Figure 8: Total time to finish application (using 1000 tokens) in co-processor architectures

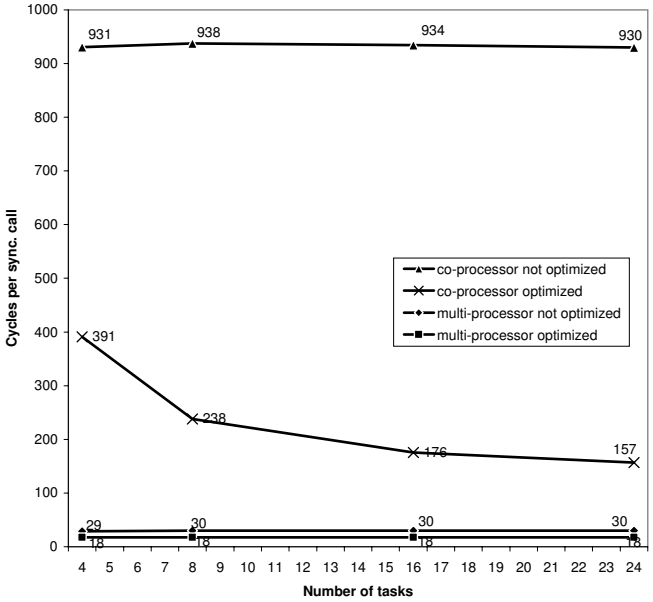


Figure 9: Time per synchronization call

In our experiments, all tasks are only performing synchronization to facilitate the analysis of synchronization performance. No buffer data is accessed. All tasks are connected in a chain in which every task is passing tokens to the next task (see Figure 7). A fixed number of tokens, i.e. 1000, are introduced in the system and the simulation is performed until they reach the last task in the chain. Each task is mapped onto a separate hardware module, which posts a new synchronization call one clock cycle after completion of the previous call. We varied the number of tasks from 4 to 24 in steps of 4, for both the classical co-processor communication scheme as for the proposed multi-processor architecture scheme.

The experimental results for the total execution time for the application is shown in Figures 8, 10 and the average time per synchronization call is shown in Figure 9.

Figure 8 presents the total execution time on the co-processor architecture for this application, as a function of the number of (hardware) tasks. As can be concluded from this figure, the optimized co-processor architecture (using our signal controller) requires significantly less cycles (upto a factor of 6 less) than the non-optimized (traditional interrupt scheme) one. In all variants of the co-processor architecture, the CPU is fully loaded by serving interrupts generated by all tasks since all tasks are performing synchronization at high rates.

As can be seen in figure 9, the average time to execute a synchronization call for the traditional interrupt scheme is nearly constant over the number of tasks. This time can be reduced significantly when our signal controller is used (optimized co-processor curve). Because now, synchronization requests of multiple tasks are serviced within a single ISR execution. Hence, the ISR overhead is divided among all synchronization calls being serviced.

Due to the large synchronization delay for the co-processor, fine grain synchronization, which is needed to reduce on-chip buffering, is hardly possible.

As can be seen from the same figure (Figure 9), the non-optimized multi-processor architecture is a factor of 5 to 13 faster than the optimized co-processor architecture with respect to the number of cycles per synchronization call. The optimized multi-processor architecture is 8 to 21 times faster than the optimized co-processor

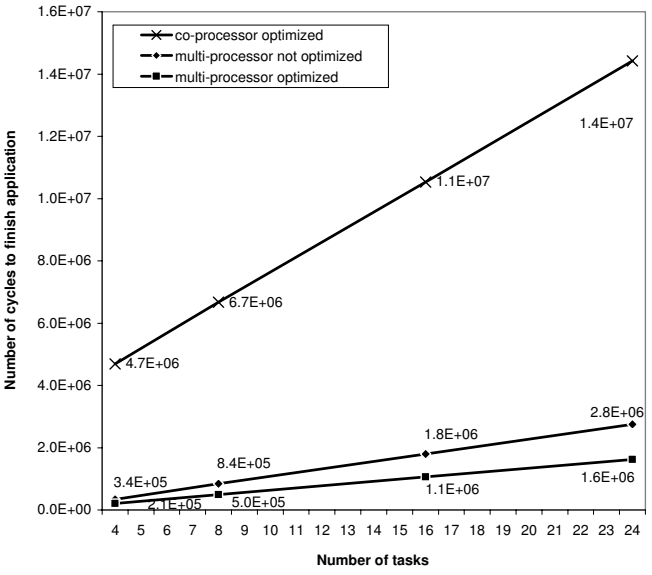


Figure 10: Total time to finish application (using 1000 tokens) in the multi- and optimized co-processor architectures

architecture. Due to the lower synchronization delay, less memory needs to be allocated for communication buffers.

To support the flexible part of an application, tasks could be mapped as software running on the CPU. With the co-processor architecture, these software tasks can not be executed until all interrupts are served. This means that the software tasks are only executed when hardware tasks (which may be communicating to software tasks) are silent, e.g. after filling the FIFO buffer. As a result, the system performance is reduced since all tasks do not execute really in parallel, that is, the software tasks are not running in parallel with the hardware modules because the CPU is (fully) loaded with the synchronization of the hardware modules.

A solution to improve the performance of this co-processor system is to map the centralized synchronizer onto a separate programmable processor or on dedicated hardware. However, the worst case dynamic delay for a synchronization call, which is the waiting time to get its turn on the centralized synchronizer, can be as high as $((number\_of\_tasks - 1) * time\_per\_sync\_call)$.

Hence, the system performance can be affected by these delays for a single synchronization call. Moreover, these larger synchronization times result in larger grain-size communication, hence increased buffer space, which is just what we try to minimize.

In all variants of the multi-processor architecture, the CPU is free to execute (signal processing) tasks since the synchronization is executed in a distributed and autonomous manner. In this case the software tasks synchronize (through software routines) only with the hardware (or software) tasks it is actually communicating with. The hardware tasks use the same shells for communicating with the software as they would use for communicating with each-other. Neither software nor hardware tasks need intervention of a 'third party', nor are they disturbed by synchronization of others. Therefore, the synchronization delays are quite low.

Figure 10 shows the run-time of the synchronization application, for the optimized co-processor architecture and the (non-) optimized multi-processor architectures. From this figure, we can conclude that high performance systems need to be implemented using multi-processor architectures.

6. CONCLUSIONS

In this paper, we have presented an efficient, scalable, and flexible data synchronization protocol for mapping signal processing functions onto heterogeneous distributed shared memory multiprocessor architectures. We assume that the functionality is expressed as a set of parallel tasks communicating through FIFO channels. Both hardware and software implementations of the protocol are explained. The protocol is transparent to the tasks, i.e. a task does not know about the protocol implementation and the implementation of the other tasks. This allows us to easily migrate functionality from hardware to software or from one processor to another. By experiments we have shown that the distributed implementation of this protocol is 8 to 21 times faster than an optimized centralized implementation. Hence it enables the high synchronization rates required to reduce on-chip buffering. Moreover, the area overhead of the distributed implementation is limited (i.e. 0.09 mm^2 per channel controller for 0.18 micron CMOS technology).

7. ACKNOWLEDGMENTS

We would like to thank Albert van der Werf for encouraging this work. We are grateful to Pieter van der Wolf, Wido Kruijtzer and Erwin de Kock for providing critical and stimulating feedback to improve this article.

8. REFERENCES

[1] S. Vercauteren, B. Lin, et al. "Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications," in *Proc. of DAC*, 1996.

[2] D. Verkest, K. Van Rompaey, et al. "CoWare - A Design Environment for Heterogeneous Hardware/Software Systems", Design Automations for Embedded Systems, 1(4), 357-386, 1996.

[3] E.A. de Kock, G. Essink, et al. "YAPI: Application modeling for signal processing systems" in *Proc. of DAC*, 2000, pp. 402–405.

[4] R. Ernst, et al. "Hardware-Software Cosynthesis for Microcontrollers", in *Proc. of IEE ED&RC*, December 1993.

[5] Mattias O'Nils, "Communication within HW/SW Embedded Systems", *ESDLab, Department of Electronics, Royal Institute of Technology, Sweden*, report no. TRITA-ESD-1997-08, ESDlab, KTH-Electrum, Electrum 229, S-16440 Kista, Sweden, 1997.

[6] G. Kahn, "The semantics of a simple language for parallel programming" in *Information Processing*, J.L. Rosenfeld, 1974

[7] J. Kang, A. van der Werf, P. Lippens, "Mapping Array communication on to the FIFO Communication-Towards an Implementation" in *Proc.of International Symposium on System Synthesis*, 2000, pp. 207–213.

[8] Tanenbaum, A. S., *Computer Networks*, Prentice/Hall International, Inc., The Netherlands, 1981.

[9] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," in *ACM Trans. on Computers Systems*, vol. 9, Feb. 1991, pp. 21-65.

[10] A.K. Nieuwland, P.E.R. Lippens "A heterogeneous HW-SW architecture for hand-held multi-media terminals", in *Proc. of IEEE Workshop on Signal Processing Systems*, 1998, pp. 113–122.