Analysing the impact of a communication assist in a multiprocessor system-on-chip

Arno Moonen[1], Marco Bekooij[2], Rene van den Berg[2], Jef van Meerbergen[1,3]

[1]University of Technology, Eindhoven, The Netherlands
[2]NXP Semiconductors, The Netherlands
[3]Philips Research, Eindhoven, The Netherlands

ES Reports
ISSN 1574-9517

ESR-2007-05
21 June 2007

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Analysing the impact of a communication assist in a multiprocessor system-on-chip

Arno Moonen[1], Marco Bekooij[2], René van den Berg[2], and Jef van Meerbergen[1,3]
[1] University of Technology, Eindhoven, The Netherlands
[2] NXP Semiconductors, The Netherlands
[3] Philips Research, Eindhoven, The Netherlands
a.j.m.moonen@tue.nl

Abstract. *In an embedded multiprocessor system the minimum throughput and maximum latency of real-time applications are usually derived given the worst-case execution time of the software tasks. Derivation of the worst-case execution time becomes easier if it is independent of the available communication bandwidth. In this paper we show that the worst-case execution time of tasks does not depend on communication bandwidth if a Communication Assist (CA) is applied, despite that memory ports are shared. Furthermore we show that adding a CA increases the processor utilization and reduces the required communication bandwidth. Finally we show that the difference between the measured and computed worst-case processor utilization is less than 6%, for our MP3 playback application.*

1 Introduction

Embedded multiprocessor systems are widely used for multimedia applications that process data streams. Examples of these applications are channel demodulation and audio processing. These applications often have firm real-time requirements, because they suffer from steep quality degradation if the throughput and latency requirements are not met.

Applications that process data streams can be represented by a task graph. The tasks are for performance and power-efficiency reasons, typically distributed over a number of processors. In order to meet the throughput and latency requirements, the system designer must show at design-time that a schedule exists that satisfies these constraints. This schedule is constructed given the worst-case execution times of the tasks [6, 15]. The worst-case execution time is the maximum time between start and finish of one execution of a task. The worst-case execution time does not include the time a task has to wait for input data and output space. It is desirable that the multiprocessor architecture enables the derivation of a *tight* worst-case execution time, since a too conservative worst-case execution time can result in a significantly over-dimensioned system.

We focus on a multiprocessor system with a distributed memory such that each processor has low access latency to its local memory. A processor can write data via a communication infrastructure in the local memory of another processor. The processor is stalled until the communication infrastructure accepts the data that needs to be transferred. Predicting the number of processor stall cycles can be difficult because this depends on the traffic pattern generated by the processor (which is often input data dependent) and the availability of the communication infrastructure (which depends on traffic generated by other processors). Therefore, predicting a tight bound on the worst-case execution time is also difficult. The derivation of a worst-case execution time should be easy and generically applicable. Derivation becomes easier if the execution time of a task does not depend on the speed at which the communication infrastructure absorbs and transfers data.

In this paper an architecture is described in which the shared data is first stored in a local memory and then transferred to the communication infrastructure by an autonomous DMA controller. This autonomous DMA controller is called a Communication Assist (CA) [3]. Its purpose is to offload the processor from pushing data into the communication infrastructure. Instead of the processor the CA is stalled until the communication infrastructure accepts the data.

Adding a CA adds costs. (i) End-to-end latency, because data is first stored locally before it is transferred. Therefore, it is suitable for applications that can tolerate additional latency, as is often the case for applications that process data streams. (ii) Memory, we need two buffers instead of one. (iii) Power consumption, because data is first stored locally before it is transferred.

The architecture with a CA has some important advantages. (i) The worst-case execution time of a task is decoupled from the communication. It is decoupled because the input and output buffers are located in the local memory of the processor and the task only starts its execution if sufficient input data and output space is available. Therefore is the worst-case execution time of a task independent of the traffic generated by other processors. (ii) A CA can decrease the worst-case execution time (instead of the processor the CA is stalled until the communication infrastructure accepts the data). (iii) The communication infrastructure can be designed for the average communication bandwidth requirements because the CA can send the data in small messages at a regular interval, whereas in the architecture without a CA the communication infrastructure is designed to absorb the communication bursts as fast as possible. The more relaxed communication bandwidth require-

ments can lead to a lower clock frequency of the communication infrastructure, potentially compensating the increase of power consumption caused by first storing data locally before transmitting.

The aim of this paper is to show that despite sharing the local memory port the upper bound on the number of processor stall cycles is independent of the communication pattern and the absorption and transfer rate of the communication infrastructure if a CA is applied. We show this by means of analytical expressions for the upper bound on the number of processor stall cycles. We see this as a key contribution of this paper.

The outline of this paper is as follows. We first describe the related work in Section 2. Section 3 explains how throughput and end-to-end latency can be derived. The reference architecture without a CA is described in Section 4 and the architecture with a CA is described in Section 5. Expressions for the upper bound on the number of processor stall cycles are derived in Section 6. With these upper bounds the impact of the CA is investigated for an MP3 application, in Section 7. Finally, the conclusions are stated in Section 8.

2 Related work

The worst-case execution time of a task is an input to system level analysis. System level analysis is necessary to verify that end-to-end performance requirements are met. The worst-case execution time of a task is determined by analysis of the program flow [7]. During the static program analysis a fixed delay is accounted for accessing the memory, but this delay can vary due to arbitration at the memory port and interconnect. In [9, 8] all effects that have an influence on the transaction are taken into account in deriving the worst-case execution time of a task. This analysis is an iterative process due to the cyclic dependency between the worst-case execution times of tasks and interference in the communication infrastructure. They claim that convergence of the iterative process is ensured but do not provide a proof in their paper. This paper follows a different approach. We decouple the computation and communication by using a CA, making the worst-case execution time analysis easier because we only need an upper bound on the number of local memory accesses and don't need any knowledge about the access pattern.

The worst-case execution time of a task can be decoupled from the communication if the input and output buffers are located local at the processor and the task only starts its execution if sufficient input data and output space is available. In the Æthereal [2] and SonicsMX [11] network-on-chip the decoupling buffers are implemented in hardware and have a fixed size. There are three important reasons why the system designer wants to have these buffers implemented in memory. (i) The bursts of data produced by the processor can exceed the capacity of a hardware buffer. (ii) Random access within an element of the buffer can be required. (iii) It is desirable that the buffer capacity can be changed by adapting the software, because the required FIFO capacity is application dependent.

When the buffers are implemented in the memory local to the processor, a CA is required to transfer the data from the buffer to the communication infrastructure. In [3, 10, 1, 12] such a CA has already been introduced and in [14] an implementation of a CA is described for a CA that supports four communication streams. Although the goal of the CA is similar (to offload the processor with communication tasks), we are not aware of a paper in which the effect of the CA on the worst-case execution time is quantified. In this paper we present such a quantitative analysis. Furthermore, we show that the worst-case execution time of a task is independent of the communication if a CA is applied.

3 Throughput analysis

In this section we describe the analysis to derive the end-to-end performance of an application that processes a data stream. First, the application is described as a task graph. Secondly, a schedule is constructed for this task graph. Finally, from this schedule the minimum throughput and maximum latency are derived.

An application is represented by a task graph $G = (T, C)$ with T a finite set of tasks and C a finite set of channels. A task $t_i \in T$ has a finite set of input ports I_i and a finite set of output ports O_i, with $I_i \cap O_i = \varnothing$. Tasks are repetitively executed. A task cannot start before sufficient data is available at its inputs and sufficient space is available at its outputs. The fact that a task waits until sufficient space is available at its outputs leads to an efficient mechanism to prevent buffer overflow. The upper bound on the execution time of task t_i is represented by $\tau(t_i) \in \mathbb{N}$ number of clock cycles. The upper bound on the number of processor stall cycles during one execution of task t_i is represented by $\sigma(t_i) \in \mathbb{N}$ number of clock cycles. The worst-case execution time is defined as $\tau(t_i) + \sigma(t_i)$. The upper bound on the number of accesses to the local memory made by the processor during one execution of task t_i is represented by $\alpha(t_i) \in \mathbb{N}$.

A channel connects an output port of a task to an input port of a task. The channel c_j that is connected to port p is denoted by $c_j(p) \in C$. The synchronization granularity of a channel is a token. A token is a container in which a predefined amount of data can be stored. The number of data words that can be stored in a token is denoted by $\eta(c_j) \in \mathbb{N}$. During one execution of a task $\lambda(p)$ tokens are consumed or produced from port p, with $\lambda(p) \in \mathbb{N}$. We call $\lambda(p)$ the quantum of port p.

We use a generic (producer-consumer) application to show that the computation and communication are decoupled with a CA, but the technique is applicable for any arbitrary application graph. The application is represented by the task graph in Fig. 1. The tasks t_1 and t_2 are represented by the nodes and the communication channel c_1 is repre-

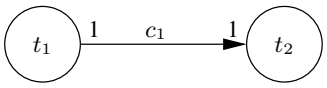

Figure 1. Task graph of a streaming application.

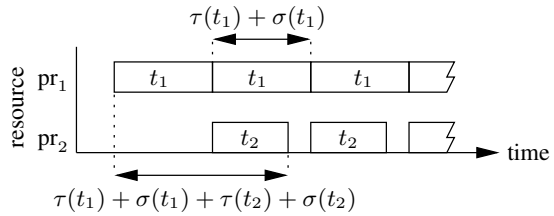

Figure 2. Constructed schedule for the application.

sented by the edge. The quantum of the output port $r \in O_1$ of task t_1 is $\lambda(r) = 1$. The quantum of the input port $s \in I_2$ of task t_2 is $\lambda(s) = 1$.

The task graph in Fig. 1 along with the worst-case execution times allows us to determine a schedule from which the throughput and end-to-end latency are derived. A computed schedule is depicted in Fig. 2. Task t_1 is executed on processor pr_1 and task t_2 is executed on processor pr_2. The k-th execution of task t_2 can start its execution after the k-th execution of task t_1 is finished. The schedule in Fig. 2 requires a channel capacity of at least 2 tokens such that task t_1 can produce a token while task t_2 consumes the previously produced token. According to the schedule in Fig. 2, task t_1 can execute immediately after it finished its previous execution. Therefore the minimum throughput on channel c_1 is one token per $\tau(t_1) + \sigma(t_1)$ clock cycles. A tighter bound on the worst-case execution time of task t_1 will result in a higher guaranteed throughput of the application. The maximum end-to-end latency is the sum of the worst-case execution time of task t_1 and t_2, i.e. $\tau(t_1) + \sigma(t_1) + \tau(t_2) + \sigma(t_2)$.

4 Reference architecture without a CA

In this section we describe an architecture template where the processor pushes the data into the communication infrastructure. The architecture template is based on the sea-of-dsp architecture presented in [13].

The architecture template consists of tiles and a communication infrastructure. A tile consists of a processor (pr), a memory (mem) and an arbiter (ar). The arbiter grants the processor or the communication infrastructure access to the memory. The application in Fig. 1 is mapped onto the multiprocessor instance in Fig. 3. Task t_1 is executed on processor pr_1, task t_2 is executed on processor pr_2 and the communication channel c_1 is implemented with a FIFO buffer. This FIFO buffer is located in the memory of tile 2 and it is implemented as a circular buffer [4], in such a way that memory consistency is guaranteed. Processor pr_1 will generate remote write accesses during the execution of task t_1. The remote write accesses are posted, which means that the processor does not have to wait for an acknowledgement that data has arrived in the remote memory. Therefore, it

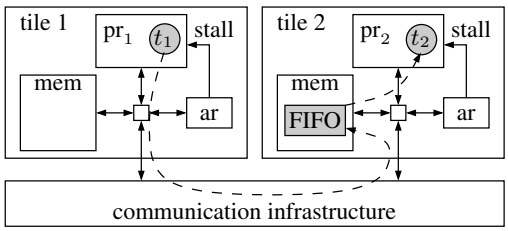

Figure 3. Multiprocessor architecture without a CA.

can continue doing useful work while the communication infrastructure is transferring the data. Task t_2 reads its input data from the FIFO, which is located in its own local memory.

Processor pr_1 and pr_2 can be stalled while executing task t_1 and t_2. There are two reasons for these processor stalls. (i) The processor is stalled if it performs a remote write access while the communication infrastructure is occupied. This depends on the pattern of remote write calls and how fast the communication infrastructure accepts data (which depends on the allocated bandwidth to the remote memory). (ii) The processor can be stalled if it performs a local memory access while the communication infrastructure also wants to access the memory. The number of processor stalls depends on the arbitration between the processor and the communication infrastructure.

The architecture puts high pressure to the communication infrastructure, because a higher allocated bandwidth reduces the number of processor stall cycles. In the next section we will show that the architecture with a CA will enable the allocation of a lower communication bandwidth.

5 Architecture with a CA

In this section we introduce an architecture where data is first stored locally and then transferred to the remote memory by a CA.

The application in Fig. 1 is mapped onto the architecture with a CA, as depicted in Fig. 4. Task t_1 is executed on processor pr_1 and task t_2 is executed on processor pr_2. The communication channel c_1 contains two FIFOs, one FIFO in tile 1 and one FIFO in tile 2. The output data of task t_1 is first stored in the FIFO which is located in the memory of tile 1. The CA transfers the data from the FIFO in tile 1 to the FIFO in tile 2 via the communication infrastructure. Finally, task t_2 reads the input data from its local FIFO which

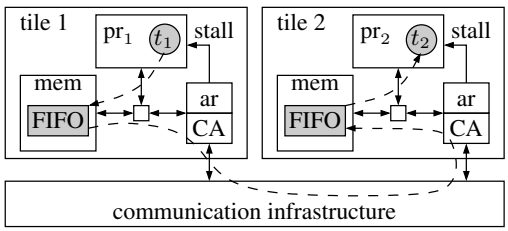

Figure 4. Multiprocessor architecture with a CA.

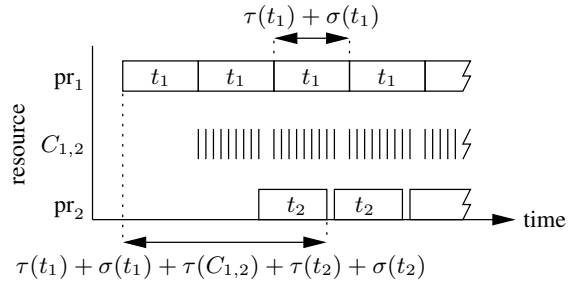

Figure 5. Schedule of the implementation with a CA.

is located in the memory of tile 2.

The task of the CA is to offload the processor from pushing data into the communication infrastructure. The processor only accesses its shared local memory. The processor can be stalled if it performs a local memory access while the CA also wants to access the memory. The number of processor stalls depends on the arbitration between the processor and the CA.

The schedule of task t_1, task t_2 and the communication between tile 1 and tile 2 is shown in Fig. 5. This schedule requires a capacity of two tokens for both FIFOs. From this schedule we conclude that the end-to-end latency in the architecture with a CA is larger then in the architecture without a CA. The end-to-end latency is the sum of worst-case execution time of task t_1 and t_2 and the time to transfer the data between the FIFO in tile 1 and the FIFO in tile 2 ($C_{1,2}$).

The number of stalls of pr_1 and pr_2 is low if the allocate bandwidth from the processor to its local memory is large for both arbiters. The allocated bandwidth for the CA to access the memory should be large enough to transport the data according to the specified throughput. Therefore, the tasks t_1 and t_2 can write to the FIFO in bursts and the CA can be forced to spread the remote write accesses over time, as depicted in Fig. 5. An additional advantage is that the traffic pattern in the communication infrastructure becomes more regular, which allows for a lower bandwidth allocated in the communication infrastructure.

6 Upper bound on the number of processor stall cycles

6.1 Remote write accesses

In the architecture without a CA, a processor generates remote write accesses to communicate between processors. If the processor issues a remote write access and the communication infrastructure does not immediate accepts the data then the processor is stalled.

Nevertheless an upper bound on the number of stall cycles can be given if the communication infrastructure and the arbiter guarantee a minimum throughput. In this paper the maximum time before the communication infrastructure accepts one word is M clock cycles, with $M \in \mathbb{N}$. Note that M is dependent of the allocated bandwidth in the communication infrastructure (which depends on the occupation of the communication infrastructure) and the allocated bandwidth to the remote memory (which depends on the arbitration at the memory port). When the processor accesses its local memory then it would take only one clock cycle. This duration of a memory access of one clock cycle is already taken into account in the execution time $\tau(t_i)$. Therefore, one remote write access results in at most $(M-1)$ stall cycles. An upper bound on the number of processor stall cycles during one execution of task t_i is given by:

$$\sigma(t_i) \leq (M-1) \cdot \sum_{p \in O_i} (\lambda(p) \cdot \eta(c(p))) \qquad (1)$$

With $\sum_{p \in O_i} (\lambda(p) \cdot \eta(c(p)))$ the total number of remote write accesses of task t_i. From (1) it follows that the worst-case number of processor stall cycles depends on the amount of data communicated. Therefore, the upper bound on the number of stalls is dependent on the ratio between communication and computation. We define the communication computation ratio ρ as the number of accesses to write output data divided by the upper bound on the execution time in clock cycles. In our architecture the communication computation ratio of a task t_i is given by:

$$\rho(t_i) = \frac{\sum_{p \in O_i} \lambda(p) \cdot \eta(c_j(p))}{\tau(t_i)} \;,\; 0 \leq \rho(t_i) \leq 1 \qquad (2)$$

The value of $\rho(t_i)$ is zero if every cycle on the processor is spent on computation and $\rho(t_i)$ is one if every cycle on the processor is spent on communication. Equation (2) can be substituted in (1). Therefore, in a multiprocessor architecture without a CA, the upper bound on the number of processor stalls due to remote write accesses is:

$$\sigma(t_i) \leq (M-1) \cdot \tau(t_i) \cdot \rho(t_i) \qquad (3)$$

6.2 Local memory sharing

In both architectures (with and without a CA), the local memory of a processor is shared. Therefore, the processor can be stalled when accessing its local memory.

During one execution of a task the worst-case number of memory accesses from the communication infrastructure to the local memory can be large, due to three reasons. (i) The actual execution time of a producing task can be smaller then the worst-case execution time. In this case the producing task can execute a number of times during one execution of the consuming task, i.e. if there is sufficient space in the FIFO. (ii) The quantum of the output port of the producing task can be large compared to the quantum of the input port of the consuming task. (iii) A number of tasks can be mapped onto one processor. In this case, a large token of one task can arrive during the execution of another small task.

The maximum number of processor stall cycles during the execution of a task can be limited by selecting an appropriate arbitration scheme. The arbitration scheme of the

arbiter in the tile must have three characteristics. (i) A low latency for the local memory accesses of the processor. (ii) A guaranteed throughput for the communication infrastructure to access the memory. (iii) It must be simple and cost-efficient. Hosseine-Khayat and Bovopoulos [5] proposed a bus arbitration scheme that is conform to these three requirements. The arbitration has a period, which is called the *service cycle time*. Each service cycle is divided into a fixed number of *time slots*. A portion of the time slots is reserved for communication. This ensures that memory bandwidth for communication is guaranteed. In this paper the reserved time for communication is one time slot. One time slot is equal to one clock cycle and the service cycle time is N clock cycles, with $N \in \mathbb{N}$. If the service cycle time $N = 5$ then it is guaranteed that the communication infrastructure or CA can access the memory at least once every five cycles. In which slot it can access the memory depends on the access requests of the processor.

Given this arbitration scheme, the processor can access the memory $(N - 1)$ times within the service cycle N. Therefore, the upper bound on the number of stalls during one execution of task t_i is given by:

$$\sigma(t_i) \leq \left\lceil \frac{1}{N-1} \cdot \alpha(t_i) \right\rceil \tag{4}$$

We will normalize the number of local memory accesses ($\alpha(t_i)$) to the execution time. This gives us a metric on how much the memory bandwidth is occupied by the processor. We define $a(t_i)$ as the normalized number of memory accesses when executing task t_i:

$$a(t_i) = \frac{\alpha(t_i)}{\tau(t_i)}, 0 \leq a(t_i) \leq 1 \tag{5}$$

The value of $a(t_i)$ is zero if the processor does not access the memory and $a(t_i)$ is one if the processor generates a memory access every clock cycle. Equation (5) can be substituted into (4). Therefore, the upper bound on the number of stalls for task t_i is:

$$\sigma(t_i) \leq \left\lceil \frac{1}{N-1} \cdot \tau(t_i) \cdot a(t_i) \right\rceil \tag{6}$$

7 Case study

In this section we describe a case study for which we compute the lower bound on the processor utilization for the architecture with and without a CA. Furthermore, the tightness of the derived bound for the architecture with a CA is verified by means of cycle true simulation.

The case study is an MP3 decoder application that consists of four tasks. The first task is a block reader that reads the input data from a compact disc and transfers the data in large chucks to the tile where the MP3 decoder is executed. The capacity of the input FIFO of the MP3 decoder is large. The MP3 task decodes the compressed audio stream and outputs 1152 stereo samples per execution. The decoded audio stream has a sample frequency of 48KHz. The SRC task, which is executed on a separate tile, converts this audio stream to a sample frequency of 44.1KHz and outputs one stereo sample per execution. The sample rate conversion is necessary because the DA converter, which is the final task, is designed for a sample frequency of 44.1KHz. The MP3 and SRC tasks are executed on two different Digital Signal Processors (DSP).

t	MP3	SRC
$\tau(t)$	467899	791
$\alpha(t)$	112898	431
$a(t)$	0.24	0.54
$\sum_{p \in O \cup I} \lambda(p) \cdot \eta(c(p))$	2305	3
$\rho(t)$	0.0049	0.0038
without a CA		
M	10	10
N	10	10
$\sigma(t)$ (Eq. (3) + Eq. (6))	33290	75
$\tau(t) + \sigma(t)$	501189	866
$u(t)$ (Eq. (7))	0.93	0.91
with a CA		
N	10	10
$\sigma(t)$ (Eq. (6))	12545	48
$\tau(t) + \sigma(t)$	480444	839
$u(t)$ (Eq. (7))	0.97	0.94

Table 1. Computation of the WCET ($\tau(t)+\sigma(t)$) and lower bound on the processor utilization ($u(t)$) for the MP3 and SRC tasks.

We define the processor utilization to compare the architecture with and without a CA. The processor utilization is a suitable metric for comparing the architectures because it makes the number of processor stall cycles relative to the execution time. The processor utilization is defined as the utilization of the processor when executing a specific task. The processor utilization $u(t_i)$, when executing task t_i, is defined as:

$$u(t_i) = \frac{\tau(t_i)}{\tau(t_i) + \sigma(t_i)}, \ 0 < u(t_i) \leq 1 \tag{7}$$

The lower bound on the processor utilization is computed for the DSPs where the MP3 and SRC tasks are executed. These bounds are computed for the multiprocessor architecture with and without a CA, as shown in Table 1. The execution time of the MP3 task (τ(MP3)) is much larger than the execution time of the SRC task (τ(SRC)), but the MP3 task processes 1152 samples (which are 2304 data words plus one synchronization word) while the SRC task processes only one sample (which are two data words plus one synchronization word). Therefore, the communication computation ratios ($\rho(t)$) are similar (0.0049 for the MP3 task and 0.0038 for the SRC task). In the multiprocessor architecture without a CA, the processor can be stalled when writing data to the communication infrastructure and when accessing its local memory. The upper bound on the number of processor stall cycles is computed with (3) and (6). In the multiprocessor architecture with a CA, the processor can only be stalled when accessing its local memory. There-

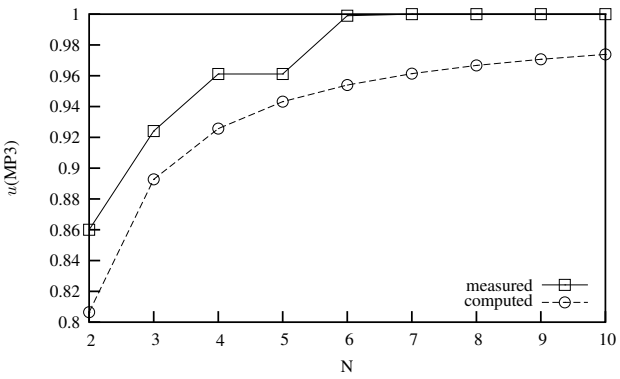

Figure 6. Processor utilization u(MP3) as a function of the service cycle N.

fore the upper bound on the number of processor stall cycles is computed with (6). The difference in the number of processor stall cycles increases if the communication computation ratio increases (i.e. if $\rho(t_i)$ increases then $\sigma(t_i)$ in (3) increases). Therefore, the impact of the CA on the guaranteed processor utilization increases if the communication computation ratio increases. For this paper, M and N are assumed to be 10. The processor utilization when executing the MP3 task is at least 0.93 on the architecture without a CA and at least 0.97 on the architecture with a CA. The processor utilization when executing the SRC task is at least 0.91 on the architecture without a CA and at least 0.94 on the architecture with a CA.

In the architecture without a CA the communication infrastructure can accept a new data word every 10 processor cycles. Assuming a processor frequency of 125MHz and a 24bit data bus, the allocated bandwidth in the communication infrastructure is 37.5MByte/sec. However, in the architecture with a CA we allocate for the average communication bandwidth, which is approximately 0.288MByte/sec (one stereo sample of 6byte at a sample frequency of 48KHz) between the MP3 and SRC task.

We measured the processor utilization u(MP3) for different configurations of the service cycle N. Both the measured processor utilization as well as the with (6) and (7) computed lower bound are shown in Fig. 6. An indication for the accuracy of the computed lower bound is the difference between the computed lower bound and the measured processor utilization. From Fig. 6 we conclude that this difference is less than 6%. Furthermore, the measured processor utilization is already 100% given a service cycle N of seven clock cycles. Therefore, it seems that when processing this particular stream the bursts from the DSP to the memory are at most six clock cycles. We have seen a similar maximum burst size for other audio applications. Therefore, typically the CA has sufficient available bandwidth for accessing the memory. For example, the normalized number of memory accesses ($a(t)$) is 0.24 and 0.54 for the MP3 and SRC tasks, respectively, as shown in Table 1.

8 Conclusions

In this paper we evaluated a multiprocessor architecture with a CA and compared it with an architecture without a CA. We conclude based on analytical expressions that the bound on the number of processor stall cycles is independent of the absorption and transfer rate of the communication infrastructure if a CA is applied. We have shown that the impact of the CA on the guaranteed processor utilization increases if the communication computation ratio increases. In our case study the communication computation ratio is very low (0.5%), therefore, the impact of the CA on the guaranteed processor utilization is only 4%. But, in the architecture with a CA, the communication bandwidth requirements of the communication infrastructure are much less than in the case without a CA (0.288MByte/sec compared to 37.5MByte/sec). It is shown by means of cycle true simulation that the computed lower bound on the processor utilization in the architecture with a CA has an accuracy of at least 6% for our case study. The end-to-end latency in the architecture with a CA is increased, but the experience is that many multimedia applications that process data streams can tolerate this additional latency.

References

[1] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. v. Meerbergen. Predictable embedded multiprocessor system design. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

[2] M. Coenen, S. Murali, A. Ruadulescu, K. Goossens, and G. De Micheli. A buffer-sizing algorithm for networks on chip using tdma and credit-based end-to-end flow control. In *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006.

[3] D. Culler, J. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann Publishers, Inc., 1999.

[4] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Int'l Symposium on System Synthesis (ISSS)*, 2001.

[5] S. Hosseine-Khayat and A. Bovopoulos. A simple and efficient bus management scheme that supports continuous streams. *ACM Transactions on Computer Systems*, 1995.

[6] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[7] Y.-T. S. Li and S. Malik. *Performance analysis of real-time embedded software*. ISBN 0-7923-8382-6, Kluwer academic publishers, 1999.

[8] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2006.

[9] S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in mpsocs. In *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006.

[10] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proc. Design Automation Conference (DAC)*, 2001.

[11] Sonics. *Datasheet SonicsMX SMART Interconnect*. http://www.sonicsinc.com.

[12] S. Stuijk, T. Basten, B. Mesman, and M. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2005.

[13] R. van den Berg and H. Bhullar. Next generation philips digital car radios, based on a sea-of-dsp concept. In *Proc. Int'l Conf. on Global Signal Processing (GSPx)*, 2004.

[14] J. Wickstrom. Design and implementation of a communication assist in a real-time multiprocessor system. Master's thesis, Chalmers University of Technology, 2005.

[15] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *Proc. Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, 2007.