Efficient Inter-Task Communication for Nested Loop Programs on a Multiprocessor System

Tjerk Bijlsma[1], Marco Bekooij[2], Gerard Smit[1] and Pierre Jansen[1]
[1]University of Twente, P.O. Box 217, 7500AE Enschede, The Netherlands
[2]NXP Semiconductors Research, 5656AE Eindhoven, The Netherlands
t.bijlsma@utwente.nl

Abstract— **In modern multiprocessor systems, processors can be stalled by inter-task communication when reading from a remote buffer. This paper presents a solution for the inter-task communication, that has a minimal impact on the performance of the system, hides the inter-task communication latency without requiring additional hardware. The solution applies to jobs, represented as task graphs, where the tasks are nested loop programs. Buffers are allocated in scratch-pad memories of the consuming tasks to provide low latency read access. For the nested loop programs, minimal buffer sizes can be determined to cover all possible communication patterns. The added computational complexity is low, as the solution adds only a few operations to the nested loop programs.**

Keywords— **Nested Loop Program, Scratch-Pad Memory, Circular Buffer.**

I. Introduction

Modern multimedia devices typically contain a multiprocessor system-on-chip (MPSoC). Jobs, represented by task graphs, are mapped on this MPSoC. Multimedia jobs typically process data streams and often have real-time requirements in the form of throughput and latency constraints.

In an MPSoC, the tasks are assigned to processors. The inter-task communication occurs via buffers. Inter-task communication has a minimal impact on the performance of the MPSoC if there is low latency access to the buffers, or the communication latency to the buffers is hidden. Both prevent processors from being stalled while buffers are accessed. Furthermore, the order in which data is read from and written in a buffer may differ. The buffer should be able to handle the different patterns without reordering the data, because this imposes additional software or hardware overhead on the system.

In this paper, a novel software solution is presented to hide the inter-task communication latency, without requiring additional hardware support. Buffers are used that, require only a small computational overhead of the tasks, can handle all possible inter-task communication patterns, and have a minimal size.

The jobs to which our solution can be applied, consist of a task graph in the form of a chain. Each task is an affine nested loop program (aNLP) [5], such that the pattern in which it accesses the memory is known at design time. A task that produces data sends it to the task that consumes it, thus preventing remote read requests. Reading from a memory that is local to the processor has a small communication latency, such that the stall time of the processor from the consumer is minimized. The producer writes the data in a circular buffer, allocated in the local scratch-pad memory (SPM) of the consumer. The circular buffer consists of a read and write part in which random access is allowed. It can be accessed by adding a few operations to the NLPs, thus no additional hardware is required.

The methods discussed in [4] and [10] copy complete shared data structures between various layers of their memory hierarchy. The most frequently accessed data structures are stored in an SPM and the sporadically accessed data in a background memory. In this method, the consuming task initiates the communication, by sending read requests for the data structure.

In contrast, our approach considers the communication of data items, rather than complete data structures and the producing task initiates the communication by sending the data items to the consuming process, preventing the communication latency because the data is read from a local SPM.

In [9], a method is proposed where a process network is derived from an aNLP. In a process network the communication between processes is explicit and performed through first-in-first-out (FIFO) buffers, note that there are no global variables. In this approach, data items are communicated instead of complete data structures, such that less buffer space is required. This advantage is also present in our approach. Another similarity with our approach is that the producer is initiating the communication, by writing the data in the buffer. If the inter-process communication is not in FIFO order, additional reorder buffers and reorder hardware or software is required. In contrast, in our approach a simple software solu-
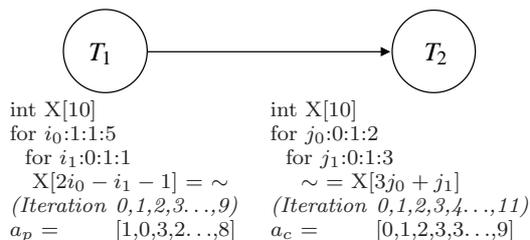
Fig. 1. A producing task T_1 and a consuming task T_2

Fig. 2. A target MPSoC, in which the communication to the buffer in the SPM of processor 2 is shown

tion consisting of a circular buffer, that allows random read and write access, is sufficient.

The organization of this paper is as follows. Section II discusses the code of the tasks in the job and the communication patterns between the tasks. Section III presents the architecture to which the job should be mapped. Based upon these sections, Section IV defines the problem statement, how to hide the inter-task communication latency. In Section V, a method for hiding the inter-task communication latency is presented. The conclusion and future work are presented in Section VI.

II. Consumption and production patterns

This section introduces aNLPs that express the pattern in which data is produced or consumed by tasks. The combination of both patterns determines the required size of the buffer, for deadlock freedom.

Figure 1 shows a producer-consumer pair from a task graph. The code of both tasks is in the form of an aNLP with single assignment code. Single assignment code means that for each execution of an aNLP, each variable is assigned a value only once. Furthermore, an aNLP consists of multiple nested for-loops. A for-loop is described with $for(i; l; s; u)$, where i is the iterator, l and u are the lower- and upper-bound and s is the stride. In this paper, we assume that the stride is one and the lower- and upper-bound are constants. The innermost loop contains operations that access shared data structures, the location to be accessed is given by an index expression. An index expression is affine and contains only iterators as variables. This means that the index expression consists of a summation of multiples of iterators, plus optionally a constant value. For aNLPs the dependency between when data is written and read in a shared data structure is explicit.

The bottom part of Figure 1, contains the *address lists* a_p and a_c for the producer T_1 and the consumer T_2, respectively. These lists contain the addresses of the shared data structure in the order that they are written or read. These lists can show two kinds of access patterns that cannot always be mapped on a FIFO buffer: out-of-order access and multiplicity.
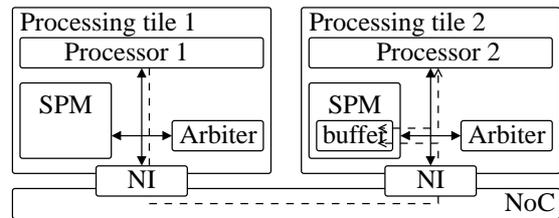
Out-of-order access oocurs when a producer or a consumer writes or reads the addresses of a data structure in a non-consecutive order. For example, the address list a_p of task T_1 in Figure 1 shows out-of-order writing, because address 1 is written in the first iteration ($a_p[0] = 1$), address 0 in the second iteration ($a_p[1] = 0$) and so on. Task T_2 shows in-order reading, since the addresses in the address list a_c show a consecutive order, starting at the first address of the shared data structure.

Another property of a pattern is *multiplicity*, for such a pattern one or more addresses are accessed multiple times during an execution of an aNLP. In our system we assume that there is no write multiplicity, since these are unnecessary operations for the producer. Read multiplicity can occur and is shown in Figure 1. The consumer T_2 is reading address 3 and 6 twice. When a consumer reads an address multiple times, the data item at that address should be buffered.

III. MPSoC architecture

The real-time requirements of the job to be mapped, require predictable behavior from an underlying architecture. In [1], a template is described for a predictable MPSoC. For our solution, it is sufficient if the target MPSoC corresponds to this template.

The architecture template in [1], consists of processing tiles that are interconnected via a network on chip (NoC), see Figure 2. The NoC provides guaranteed services, like uncorrupted, lossless and ordered data delivery, guaranteed throughput and bounded latency. A processing tile contains an SPM, a processor, an arbiter and a network interface (NI). The processor can read and write in the SPM of its own processing tile or write in the SPM of another processing tile. When the processor has to write in the SPM of another processing tile, it posts the write to the NI that autonomously handles this request, such that the processor can continue. The arbiter guarantees that the interference on the memory port, between accesses from the NI and accesses from the processor, is bounded.

IV. PROBLEM STATEMENT

The problem addressed in this paper is to determine an efficient method for inter-task communication, where inter-task communication latency has a minimal impact on the performance and no specialized hardware is required for the buffers. The method is applicable for task graphs that are chains, where the tasks are aNLPs. The challenges are to minimize the computational overhead for the tasks and the required buffer size. Furthermore, the buffers should enable out-of-order communication and multiplicity.

Splitting up the problem, the first subproblem identified is the location of the buffer. The second subproblem is the synchronization of the tasks and to guarantee mutually exclusive access to the data items in the shared memory. The third subproblem is how to determine the size of a buffer, such that out-of-order access and multiplicity are possible. The fourth subproblem is how to extend the tasks with operations for the synchronization and buffer accesses.

V. SOLUTION APPROACH

This section discusses how the inter-task communication between tasks on a target MPSoC can be realized. The example in Figure 1 is used to show how we solve the problem formulated in Section IV.

A. Communication grain

When the complete data structure X, from Figure 1, is communicated as one block between the tasks, it is called a *block data transfer* [3]. This kind of communication has the advantage that the communication overhead, for example the starting address of the data structure in the SPM, only has to be send for a single block. In our solution, communication is performed at a *word level granularity*, meaning that the data items of data structure X will be communicated individually. The advantage is that, often, a buffer smaller than the size of the data structure is sufficient and pipelining is possible by increasing the size of the buffer. To determine the buffer size the read and write pattern in the shared data structure has to be analyzed.

B. Buffer location

When the producer is writing in a different order than the consumer is reading, or when the consumer reads data multiple times, a buffer with more than one word is required. This buffer can be either located in the SPM of the producer or the consumer. When the buffer is located in the SPM of the producer, it is called *receiver initiated communication* [3]. In this case the consumer has to send a request to SPM in the processing tile of the producer. This causes the processor of the consumer to be stalled, from the moment it sends the read request until it receives the data. *Precommunication* [3] can be used to send the read request earlier, such that the data is in the SPM of the consumer at the moment it is required. Hardware solutions for precommunication detect at runtime which addresses from the memory to precommunicate. Software solutions typically insert precommunication operations in the code at compile time.

When the buffer is located in the SPM of the consumer, the producer sends its data to the SPM of the consumer. This is called *sender initiated communication* [3] and is considered as a special kind of precommunication that requires no additional hardware or software. In this case the consumer can read the data from the buffer in its local SPM with a low latency, minimizing the stall time of its processor.

Figure 2 shows an MPSoC according to our template. The task graph from Figure 1 can be mapped to this MPSoC, by assigning the producing task T_1 to processing tile 1 and the consuming task T_2 to processing tile 2. The buffer and its administration are located in the SPM of the consumer, to have the advantage of sender initiated precommunication. The dotted lines show the communication, from both the producer and the consumer, to the buffer.

C. Memory consistency

In our solution sender initiated communication is applied at a word level granularity. Therefore the producer and the consumer, communicate and synchronize via the shared address space of the SPM of the consumer. A memory consistency model is required for the synchronization between the tasks.

The memory consistency model we use is streaming memory consistency (StrC) [2]. According to this model, the address of a shared variable is acquired before it is accessed and released when it is not needed anymore, the acquire and release form a synchronization section. In comparison, release consistency (RC) [6] allows shared variables to be accessed outside synchronization sections. For synchronization sections from different buffers StrC has no strict ordering, but RC does. Both StrC and RC require no strict ordering of memory access operations within synchronization sections. Unlike RC, StrC even allows acquire operations of synchronization sections, from the same buffer, to overtake release operations.

The key advantage of streaming memory consistency is that it allows *posted writes*. A posted write is a write operation that allows the producer to continue executing, instead of stalling until the completion of

the write. A write operation is completed if the data is stored in the memory. Posted writes are allowed only if the synchronization variables are located in the same memory as the buffer and if the connection to the memory provides uncorrupted, lossless and ordered data delivery of reads and writes.

In our system, the producer initially performs an acquire operation for an address range in the buffer, which has to be confirmed. When the access to an address range is acquired, the writes are posted, as is the release of the address range, which can immediately be followed by an acquire. Because StrC allows an acquire to overtake releases, acquire operations can be pipelined, to hide the latency of the NoC, and when the first one is confirmed, the posting of write operations can be started, followed by release operations.

D. Buffer type

The buffer used for inter-task communication will be located in an SPM. The buffer should allow read multiplicity and out-of-order reading and writing in the buffer. This behavior is possible when using a *circular buffer* (CB). In order to arrange the buffer administration without additional hardware, a protocol such as C-HEAP [7] can be used.

In a CB, the producer can write all addresses between the write and the read pointer. Similarly, the consumer can read all addresses between the read and the write pointer. The producer makes data available to the consumer, by increasing the write pointer. When the consumer has read the address at the read pointer for the last time, it increments the read pointer, providing the address to be reused by the producer. Hence, both the producer and the consumer have exclusive access to their part of the buffer. The pointers are not allowed to overtake each other. Typically both pointers start at the same location and the first pointer to be increased is the write pointer. When a pointer reaches the end of the CB, it wraps around.

E. Read and write window

The producer and consumer task, from Figure 1, use a CB in combination with StrC. To use a circular buffer and StrC, the code of the tasks needs to be annotated with acquire and release operations. A task performs a release operation and increases its pointer in the CB, when it is finished with the value at its pointer address. Furthermore the address it accesses is acquired previously.

The operations added to the code are simple and limited in number. It is possible to acquire each address individually and release it when it is not needed
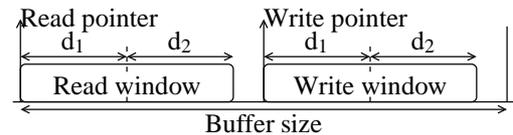

Fig. 3. A circular buffer, with a read and a write window

anymore, but this would impose to much control overhead, as shown in [8].

For our solution a *read window* and a *write window* are defined, for the consumer and the producer, respectively, as shown in Figure 3. Both windows are a sequence of acquired addresses, only the first and the last address of a sequence needs to be stored. When an acquire is performed, an address from the buffer is added at the head of the sequence. A release operation, removes the address at the tail of the sequence and increments the pointer in the CB, since the released address will not be accessed anymore.

To keep the control overhead in the code limited, every iteration performs one acquire, until all addresses of the data structure have been acquired. Possibly a number of initial acquires are required, to guarantee that in each iteration the address to be accessed is acquired, eg. due to out-of-order access. Furthermore after an initial number of iterations, every iteration performs a release. This pattern of acquires and releases make that the windows become *sliding windows*.

F. Window size

The size of a read or write window is influenced by the access pattern of the aNLP. Since every iteration of a task acquires at most a single address in the buffer, we have to guarantee for every iteration that at least the address that is accessed is acquired. This can be done by acquiring an initial number of addresses in the CB, before the loop-nest starts, called the *lead-in* (d_1). In the following text, the first address to be acquired in the CB is address 0.

Figure 4 shows a graphical method to determine the lead-in, for task T_1 from Figure 1. The top part of the figure shows the order in which the addresses are acquired by the write window, called addresses acquired. In the bottom part, the address list a_p is shown. To guarantee that an addresses is not written before it is acquired, the list with written addresses a_p is shifted right, such that every address is acquired before it is written. The first acquires, that are performed before
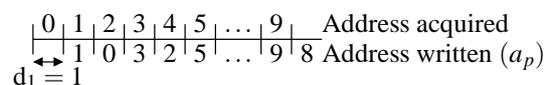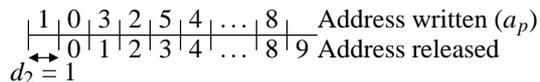

Fig. 4. Lead-in (d_1) of task T_1 from Figure 1

Fig. 5. Lead-out (d_2) of task T_1 from Figure 1

Fig. 6. Lead-out (d_2) of task T_2 from Figure 1

any write operation can be performed, are grouped as an initial number of acquires, called the lead-in. In Figure 4 a lead-in of an acquire of one word is found.

A formal expression to determine the lead-in can be given. Let a be the address list, i the iteration of the NLP and $\widehat{i}$ the total number of iterations:

Lemma 1: A lead-in of $d_1 = \max_i(a[i] - i)$, with $0 \leq i < \widehat{i}$, is minimal and ensures that every address is acquired before or when it is accessed.

Proof: In iteration j, address $a[j]$ is accessed, so at least $a[j]$ addresses should be acquired in the buffer. Each iteration performs one acquire, so initially $a[j] - j$ acquires should have been performed, if $a[j] \geq j$. To make sure that in each iteration of the loop-nest the accessed address is acquired, the minimal and sufficient number of initial acquires d_1 is found by $\max_i(a[i] - i)$, with $0 \leq i < \widehat{i}$. □

When the tail address from the window is written or read for the last time it can be released, such that the addresses in the buffer can be reused. Note for task T_1 from Figure 1 that it is not possible to start with releasing one address per iteration in the first iteration, because then after the first iteration address 0 is released, which should be written in the second iteration ($a_p[1] = 0$). Therefore an initial number of iterations in which no address is released has to be determined, called the *lead-out* (d_2).

In a similar way as for the lead-in, a lead-out is determined in Figure 5 for task T_1 from Figure 1. In this figure the top part shows the address list. In order to guarantee that an address is not released before it is written, the bottom part with the released addresses is shifted right. The figure shows that with a lead-out of one, all addresses are written before they are released.

As for the lead-in, a formal expression to determine the lead-out can be given. Let a be the address list, i the iteration of the NLP and $\widehat{i}$ the total number of iterations:

Lemma 2: A lead-out of $d_2 = \max_i(i - a[i])$, with $0 \leq i < \widehat{i}$, is the minimal number of iterations after which each accesses can be combined with a release.

Proof: In iteration j, at least the address $a[j]$ should still be acquired in the buffer. After an initial number of iterations each iteration releases 1 address. To make sure that address $a[j]$ is still acquired in iteration j, at least the first $j - a[j]$ iterations should release no
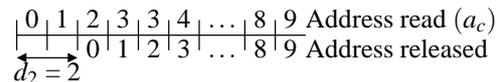
address, when $j \geq a[j]$. To make sure that in each iteration the accessed address is still acquired, minimally the first $\max_i(i - a[i])$ iterations, with $0 \leq i < \widehat{i}$, of the loop-nest should not release an address. □

A window is build up from a lead-in d_1 and a lead-out d_2 and a location for the current address, making the window size $w = d_1 + d_2 + 1$. We can define a read window for the consumer w_c and a write window for the producer w_p.

When the consumption pattern of the consumer shows multiplicity, it is possible that $d_1 + d_2 + 1$ is larger than the highest address in the address list a_c. A window containing the whole shared data structure is sufficient. Let $\widehat{a}$ be the highest address in the address list, which is also the maximum window size.

Theorem 1: In combination with a lead-in and a lead-out a window with size $w = \min(d_1 + d_2 + 1, \widehat{a})$ guarantees that the accessed address is in the window.

Proof: The Lemmas 1 and 2 state that the address to be accessed will be acquired. Lemma 1 shows that the lead-in is determined such that $a[i] \leq i + d_1$. Lemma 2 shows that the lead-out is determined such that $i - d_2 \leq a[i]$. Therefore the combination of both ensures that the accessed address is always in the window, $i - d_2 \leq a[i] \leq i + d_1$. Since at most all addresses of the data structure are acquired, the maximum window size is $\widehat{a}$. □

For the example from Figure 1, a write window w_p is $1 + 1 + 1 = 3$ is derived. The consumption pattern of task T_2 shows multiplicity. Figure 6 shows that a lead-out $d_2 = 2$ can be determined. Furthermore the figure shows that the addresses are accessed in order, requiring a lead-in $d_1 = 0$. The window required by the consumer is $w_c = 3$.

G. Buffer size

The CB in Figure 3, contains both a read and a write window. With the derived sizes of these windows, the minimum required size of the CB for this solution can be determined.

At the beginning of an iteration, a task acquires one address and at the end of the iteration one address is released, thus only during an iteration the whole window size is acquired. Before both the producer and the consumer start their iteration, thus before they performed the acquire operation in their inner-loop, there are $w_p - 1 + w_c - 1$ addresses acquired. To al-
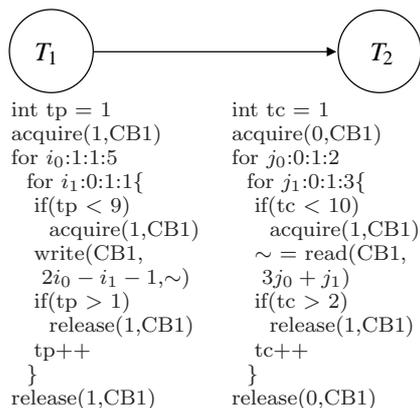
Fig. 7. Annotated producer consumer pair, from Figure 1

low either the consumer or the producer to perform an acquire and start an iteration, one additional address should be available in the buffer. Therefore the buffer size that allows the execution of the producer and consumer is $w_p + w_c - 1$.

A chain, in which every task has at most one input and one output buffer, is deadlock free if the buffer sizes are chosen as $w_p + w_c - 1$. Consider a task T_n, that shares its input buffer with task T_{n-1} and its output buffer with task T_{n+1}. If the input buffer contains data and there is space in the output buffer, T_n will execute. If there is no space in the output buffer, it is filled, so T_{n+1} will eventually execute and provide space. If there is no data in the input buffer, T_{n-1} will eventually execute and provide data. In all cases T_n can eventually execute, so the chain is deadlock free.

H. Annotating the NLP

The NLP of the tasks in the task graph needs to be extended to make use of the circular buffer and the sliding windows. Figure 7 shows how the NLPs from Figure 1 should be annotated. Initially both NLPs acquire d_1 addresses in the buffer. In the innermost loop of the loop-nest, before the operation is performed, an if-statement with an acquire is added. The if-statement guarantees that no more addresses are acquired than there are data items in the shared data structure. The access to the shared data structure is replaced with an access operation to the CB. At the end of an iteration an if-statement is inserted that starts releasing addresses from the window after d_2 iterations. At the end of the NLP the remaining addresses in the window are released.

VI. CONCLUSION

This paper presents a novel software solution for hiding inter-task communication latency, that requires no additional hardware. Locating the buffer for inter-task communication in the scratch-pad memory of the consuming task, provides low latency access to it. In combination with the streaming memory consistency protocol the producing task can post its writes, such that it is not stalling until writes complete. To be able to handle the different access patterns of the producer and the consumer to the buffer, a circular buffer is used with a write window for the producer and a read window for the consumer. The sizes of these windows can be determined to cover the different access patterns and for deriving the minimum required buffer size. In comparison, a single FIFO buffer cannot handle most different access patterns. Only a few operations need to be added to the nested loop programs of the tasks, such they can handle the circular buffers.

The next step is to apply this solution to a real-life application. Another step is to extend the solution to work for task graphs in the form of a directed acyclic graph. It is challenging to increase the expressivity of the nested loop programs that can be handled.

REFERENCES

[1] M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and composable multiprocessor system design: A constructive approach. In *Bits & Chips Embedded System symposium*, October 2007. Accepted for publication.

[2] J. v. d. Brand and M. Bekooij. Streaming memory consistency for efficient MPSoC design. In *Proc. Euromicro Symposium on Digital System Design*, 2007.

[3] D. Culler, A. Gupta, and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[4] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis. A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. In *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 2006.

[5] P. Feautrier. Dataflow analysis of array and scalar references. In *Int'l Journal of Parallel Programming*, 1991.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. Int'l Symposium on Computer Architecture*, 1990.

[7] A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. In *Proc. Design Automation Conference (DAC)*, 2002.

[8] A. Turjan, B. Kienhuis, and E. Deprettere. Realizations of the extended linearization model. In *Proc. Int'l Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2002.

[9] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *Proc. Int'l conference on Compilers, architecture, and synthesis for embedded systems*, 2004.

[10] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proc. Int'l Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2004.