Omphale: Streamlining the Communication for Jobs in a Multi Processor System on Chip

T. Bijlsma, M.J.G Bekooij, G.J.M. Smit and P.G. Jansen

July 5, 2007

Abstract

Our Multi Processor System on Chip (MPSoC) template provides processing tiles that are connected via a network on chip. A processing tile contains a processing unit and a Scratch Pad Memory (SPM). This paper presents the Omphale tool that performs the first step in mapping a job, represented by a task graph, to such an MPSoC, given the SPM sizes as constraints. Furthermore a memory tile is introduced. The result of Omphale is a Cyclo Static DataFlow (CSDF) model and a task graph where tasks communicate via sliding windows that are located in circular buffers. The CSDF model is used to determine the size of the buffers and the communication pattern of the data. A buffer must fit in the SPM of the processing unit that is reading from it, such that low latency access is realized with a minimized number of stall cycles. If a task and its buffer exceed the size of the SPM, the task is examined for additional parallelism or the circular buffer is partly located in a memory tile. This results in an extended task graph that satisfies the SPM size constraints.

1 Introduction

The shrinking transistor size makes it possible for embedded systems to have multiple processors on a single chip, such a chip is called a Multi Processor System on Chip (MPSoC). A job is represented by a task graph and can for example be specified by a C-program. Typically multimedia jobs process streams of data, this requires the computational power that an MPSoC can provide.

In order to run streaming multimedia jobs on an MPSoC, throughput and latency requirements have to be satisfied, in order to deliver a certain quality of service (QOS). The QOS can only be guaranteed when the MPSoC behaves in a *predictable* manner. In general an MPSoC is not used for running a single job, a user can run multiple jobs, starting or stopping the jobs at an arbitrary moment. The integration effort of multiple jobs on a single MPSoC can be reduced by making sure that the jobs do not influence each others temporal behavior, such a system is called *composable* [1].

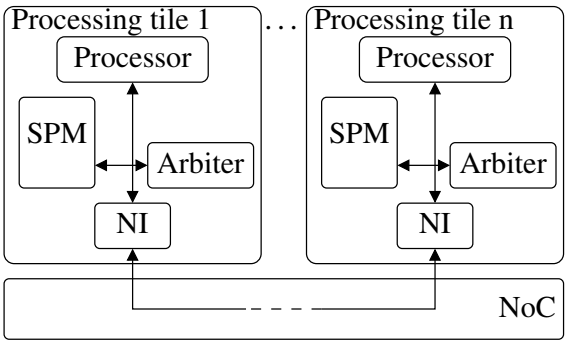

Figure 1: MPSoC template, with n processing tiles

The template defined in [1] describes a predictable and composable architecture for an MPSoC. The architecture consists of processing tiles that are connected with each other via a Network on Chip (NoC), as shown in figure 1. A processing tile contains a processor, a local Scratch Pad Memory (SPM), a network interface (NI) and an arbiter. The processor can write and read from the local SPM and can write in the SPM of another processing tile. Jobs interact by writing in each others SPM, where arbiters makes sure that jobs do not influence each others temporal behavior. Although the focus is on communication via a SPM it might be possible to use a cache, however using caches

may introduce predictability issues.

To enable communication between multiple processing tiles, a tile is connected to the NoC via its network interface. The NoC should provide guaranteed services, like uncorrupted lossless ordered data delivery, guaranteed throughput and bounded latency. The Æthereal [9] network provides such a guaranteed service.

The C-program of a job that serves as input should contain task level parallelism. This means that the job should be composed out of multiple tasks that can be performed concurrently. The C-program either explicitly defines the tasks, or they need to be manually identified. Tasks communicate via channels. In the task graph of the job, these channels correspond to the edges between tasks. The analysis and simulation flow accepts a Dynamic Data Flow (DDF) graph derived from the task graph, where the tasks are mapped to actors and they are still connected via edges. Analysis techniques are used to determine the temporal behavior of a job and the resources it requires, for example the buffer sizes. At run-time the last step in the mapping process can be performed, by assigning the tasks to processors in the MPSoC, given the determined resource requirements.

A task graph of a multimedia job communicates parts of shared data structures over its edges. With modern multimedia jobs, data structures tend to get larger due to higher quality requirements, for example the large frames used for High-Definition Television. Since the cost of off-chip DRAM is at least a factor 10 lower than on-chip SRAM and the memory required differs per task, additional memory should be provided for some tasks. This can be done in the form of a memory tile where jobs can store their data. Such a memory tile should be connected to the NoC, making it a shared resource in the system. In order to keep the system predictable and composable, the memory tile should also be predictable and composable.

It may be necessary to derive additional task level parallelism from the task graph, when a task requires more memory than available on a processing tile. Trying to derive task level parallelism from sequential C-code can be difficult, due to unclear dependencies [2]. An alternative is to restrict the syntax used in C-code, allowing only Nested Loop Programs (NLPs) with affine expressions for which all dependencies can be derived [8]. The Compaan tool flow [11] shows that a sequential NLP can automatically be converted to a Kahn Process Network (KPN), that contains task level parallelism.

This paper describes the Omphale tool flow that performs the first step in mapping a job to a target MPSoC, given the memory constraints of the MPSoC. The code of the tasks in the job should be in the form of NLPs. A software solution, that partly can be replaced by hardware, is applied where instructions are inserted in the code of the tasks, to organize the communication via circular buffers [13]. When a task combined with its buffer requires more memory than available in the SPM of the processing tile, two solutions are available. The first solution is to extract additional parallelism from the NLP of the task, thereby reducing the amount of memory required by the job. The second solution is to store a part of the circular buffer in a memory tile. The result of Omphale is a task graph where the tasks communicate via circular buffers and a corresponding Cyclo Static Data Flow (CSDF) model, which shows the communication patterns.

2 Related work

Many research has been performed in the area of mapping applications to a system on chip. In this section the focus is on related methods, that extract additional task level parallelism and organize the inter-task communication in order to perform the mapping. First the tool flows are described that organize the inter-task communication at the granularity of a data structure. The last tool flow organizes the inter-task communication at a word level granularity, as the Omphale tool flow will do.

Two sub categories of task level parallelism can be identified, *data level* parallelism and *function level* parallelism. Data level parallelism is extracted when a single task is divided in multiple similar tasks, not necessarily identical, where each task processes another part of the data structure, reducing the part of the data structure that has to be buffered. Data level parallelism can be exploited to balance the load of a computational intensive task over multiple processing tiles.

Function parallelism means that the set of operations from a job can be divided in non overlapping subsets, where each subset represents a task that can be performed concurrent with the other

tasks. When function parallelism is exploited, each task performs a specific operation. This allows the exploitation of heterogeneous processors, such that tasks can be performed on specialized cores. It also keeps the code size small, since every task has it own group of operations. The dependencies between the tasks lead to a natural pipeline.

Data needs to be communicated between the tasks of a job. When the data is stored in a background memory communication latency is introduced. Communication latency can be hidden by making sure that the data, or a part of it, is available at the processing tile before it is actually being used. When it is locally available it can be accessed at a very low latency, which reduces the number of the stall cycles of the processor and with it the required processing time, thus increasing its effectiveness. Culler et al. [6] describes the precommunication method, which generates the communication before data is actually needed by the job. Precommunication at the receiver, known as *receiver initiated communication*, can be either performed in hardware or in software. Hardware can be used to detect at run-time which address from the memory to precommunicate. Software can insert precommunication operations in the code at compile time. A special kind of precommunication distinguished is *sender initiated communication*. Here the sender produces data and writes it in the local memory of the receiver. In this case the receiver can read the data with a low latency from its local memory.

Different granularities can be used for the data communicated.It is possible to keep the inter-task communication simple by communicating the complete data structure. Communicating multiple words together is called a *block data transfer* by Culler et al. [6] and has the advantage that there is only the communication overhead for a single block. A large drawback is that the whole data structure needs to be stored in the local memory of the receiver and the block data transfers cause communication peaks on the NoC. An alternative is performing the receiver or sender initiated communication at a *word level granularity*. This requires a thorough analysis of the read patterns and write patterns of the shared data structure. In this case the communication overhead per word is compensated by a more balanced load of the NoC, possibly requiring less communication bandwidth. Furthermore memory space can be saved when pipelining is based on words instead of larger data blocks.

A common hardware solution to reduce the communication latency when reading from an off-tile memory is a cache. A cache is build up from cache lines, where a cache line contains the data of a number of consecutive addresses in the memory. When an address is read the data is stored in the corresponding location in the cache line, and the data for the consecutive addresses in the cache line is precommunicated. Since the cache is smaller than the off-tile memory, multiple addresses from the off-tile memory are mapped to one cache line, in case an associative cache is used an address can be mapped to a number of cache lines. The functioning of a cache is based on the locality of reference, assuming that when an address is read it is probable that the next read will be in a consecutive address. Care has to be taken when a data structure, used by a task, is stored over multiple addresses that are mapped to the same cache line. In case the data structure is not read in a consecutive way from its first address till the last, it is possible that cache lines are overwritten before they are actually used, causing additional communication latency. In [15] a method is proposed to store the data structures and scalars, such that there are as few as possible conflicts in the cache, reducing the communication latency. Even though there is always communication latency when a cache line is not yet precommunicated and communicating cache lines may causes a communication peak on the NoC, depending on the number of words in the cache line.

The Decoupled SoftWare Pipelinig (DSWP) approach [14] extracts function parallelism from C-code. This approach creates a dependence graph from the input program and groups the strongly connected components from the graph according to a load balancing heuristic. This approach is focused on the outer loop of loop-nests in which multiple operations are performed. The groups distinguished in the dependence graph can be run in parallel and the dependences are such that the groups form a pipeline. The communication of data between groups is performed via a synchronization array [16], to make sure that data is produced before it is consumed. The synchronization array is a list in which the production of data is registered. Data can be read and written using a PRODUCE and WRITE command.

The DSWP approach is targeted at an architec-

ture with multiple processors and a central shared memory. It exploits function parallelism. The approach does not implement a method for hiding the communication latency when a group is reading from the central memory. They propose the use of caches for partial hiding the communication latency. Predictability and composability are not considered in this approach.

The Phideo tool flow [12] is a high-level hardware synthesis design methodology, targeted at the design of high performance real-time systems. The considered target jobs contain many loop-nests. Phideo accepts a job represented as a signal flow graph described in the programming language SIL. In the signal flow graph the nodes are one or more operations and the arcs are the dependencies. Based on the signal flow graph the producer and the consumers of data can be identified, they are grouped into a data stream. According to the dependencies, so when the data is produced and consumed, this data stream can be scheduled and made into a data path. Scheduling means that for each consumer the starting time and the period is determined, such that when the consumer is running the data required has been already produced by the producer of the stream. An Integer Linear Problem (ILP) formulation is used to solve the scheduling problem [20], respecting dependencies and optimizing the memory usage. Each data path is connected to one or more memories, to consume and produce data. The addresses at which the data produced by a data path needs to be written in a memory is controlled by an address generator. A central controller is used to synchronize the address generator and the data path.

The Phideo tool flow performs synthesis on a job considering real-time constraints, thus creating a predictable system. By creating data paths for the node in the signal flow graph, it reveals function level parallelism in the jobs. Phideo has no reason to hide communication latency, since each data path is connected to its own memory. The system is focused at the synthesis of one job into application specific hardware, thus it has no need to consider composability.

A reduction of the synchronization, as forced by the central controller, in a Phideo system is proposed by [10]. It proposes to replace the synchronized address generation, by sending the address of the available data via a FIFO buffer. In this situation the consumer of data can only start when an address is available in the FIFO buffer. The producer writes data in the memory, till no more addresses can be stored in the FIFO buffer that communicates the addresses to the consumer.

The M4 design flow developed by IMEC, composed out of multiple tools, accepts C-code as input and delivers a software mapping on a target platform or a hardware description. In the first phase of the design flow, the MHLA tool [7] processes C-code. The target of the tool is to prefetch data structures as much as possible in a local memory using a DMA, while increasing performance and reducing energy consumption. MHLA uses a heuristic to determine the best execution order of the data transfers in the memory hierarchy of the considered target architecture. The result is C-code that is extended with instructions to program the DMA to copy the data structures between the various memory layers. The target architecture is a multi processor system with shared on chip and off chip memories, connected via a bus. The method to determine an order for the prefetching is similar to the method earlier proposed by Vermat et al. [21], where an exact solution in the form of an ILP is used to find the optimal order to copy the data structures between the memory layers. The target architecture considered by Vermat et al. is an MPSoC that may include caches and includes SPMs.

In the second phase of the M4 design flow the Sprint tool [5] identifies function parallelism in the annotated C-code. Sprint tries to optimize the communication between the tasks to reduce the power consumption, increase the performance and possibly use a heterogeneous architecture. The parallelization requires user directives to point out the boundaries of the tasks in the C-code. The tasks can communicate via a FIFO buffer or via a shared data channel. The FIFO buffer is only used for simple communication patterns, when data is read multiple times or out-of-order reading occurs a shared data channel is used. The result of the tool is a SystemC model, that has to be verified for correctness by simulation.

The M4 design flow is optimizing the throughput for the whole job, not considering predictability or composability. They apply receiver initiated precommunication at a data structure granularity. The possible target architecture is a multi proces-

sor system with shared on chip and off chip memories, connected via a bus. They consider a system with distributed shared memories with non-uniform communication latencies, since the access latency can differ per memory. The communication latency can only be hidden when all data structures fit completely in the local memory.

The Compaan tool chain as described by [18] automatically derives a Process Network (PN) from an NLP. The considered NLP has affine indices for the used arrays and static control. An affine expression is defined as a constant value that is added to (or subtracted from) a sum of variables, where the variables can be multiplied with a constant value. Static control means that the bounds of the loop-nest and the conditions of if-statements in the loop-body contain affine expressions of loop iterators and parameters, their values do not change during the execution of the loop-nest. In [17] this is relaxed, by allowing if-statements to contain arbitrary expressions in their condition.

Compaan translates an NLP to a Single Assignment Program (SAP), which can be translated in a Polyhedral Reduced Dependency Graph (PRDG). The PRDG show the precedence relations between the operations in the SAP. This PRDG is converted to a PN. The constructed PN is such that if an array is written by multiple processes, the array is split such that each process writes its own array. The same is done if an array is read by multiple processes, the array is split such that each process receives its own array.

In a PN each array is communicated through a FIFO buffer. When data will be read multiple times, this is called multiplicity, a buffer is used instead of a FIFO. The data is stored in the buffer till it is read for the last time. In case the process that is reading data from the FIFO reads it in a different order as it is written, the data needs to be reordered. This is solved by adding a reordering memory and a controller at the receiving process. Multiple techniques are proposed for addressing the data in the reordering memory, ranging from a complex pseudo-polynomial expression used for the address in combination with a general Random Access Memory (RAM) to an exotic Content Addressable Memory (CAM) memory where data is retrieved with special tokens.

The constructed PN can be translated to Java or C++, such that the code can be mapped on a heterogeneous multi processor platform or directly to VHDL. The considered heterogeneous multiprocessor platform contains processing units and shared memories, that connect via a NoC.

In the Compaan tool chain functional parallelism is exploited. The processes write their results in FIFO buffers, realizing sender initiated communication at a word level granularity. This is different from the functional parallelism extracted by the M4 tool flow, where receiver initiated communication is performed at a data structure level granularity.

The Compaan solution partitions a job and applies sender initiated communication. It does not consider predictability or composability. The resource constraints of a possible target MPSoC are not considered while constructing the PN, since the resulting PN can be used for mapping the job to an MPSoC or for translation to a hardware description. If data is read out-of-order in the PN a controller and reordering memory are required, increasing the complexity of the receiver dramatically.

3 The Omphale tool flow

The Omphale tool flow performs the first step in mapping a job to an architecture instance that is according to the template described in [1]. The main goal of the tool flow is to reshape a job for a target architecture, satisfying the architectural, composability and predictability constraints.

Initially Omphale tries to locate the buffers, represented by the edges in the task graph of the job, in the SPM of the consuming task. This results in a solution with sender initiated communication. It is possible that a task combined with its buffer requires more memory in the SPM than available, in this case two solutions are available. The first solution is to extract task level parallelism by splitting the task in smaller parts. If task level parallelism can be extracted, the task is replaced by n tasks that communicate shared data structures via circular buffers. Deriving task level parallelism can result in tasks that have less private data to be stored. The second solution is to locate a part of the buffer in a memory tile. When the buffer is partly located in a memory tile, the communication latency between the memory tile and the consumer should be hidden. In the memory tile a Communication Assist (CA) is located, which precommunicates the

information from the buffer in the memory tile to the buffer in SPM of the consumer.

The result of Omphale should satisfy memory size constraints, while the costs of the solution should be minimized. The cost of *computational overhead* in the tasks, caused by restructuring the communication, is kept as small as possible. Thus the *addressing schemes* used for memory access should be relatively simple, meaning it should match with the addressing used in a typical SPM. Furthermore the *amount of data transfered* should be minimized, to reduce the load on the NoC. The *additional hardware* on the memory tile, called the Communication Assist (CA), should be kept as simple as possible.

This section starts with defining a so called sliding window used in a circular buffer, that is used for inter-task communication. In subsection 3.2 the inter-task communication patterns are discussed. This explains the complex read or write patterns in circular buffers that can be distinguished. Next in subsection 3.3 the memory tile is introduced for our template. Circular buffers can be partly stored in a memory tile. Subsection 3.4 describes the tool flow that tries to fit a job to its target architecture. Circular buffers are inserted for the inter-task communication in the job and if necessary a part of a circular buffer is located in a memory tile or additional parallelism is extracted from a task, such that the memory constraints of the target architecture are satisfied.

3.1 Sliding window

In our template *Streaming Consistency* [4] is supported for inter-task communication. This consistency model enforces that a task *acquires* a block of data in the memory, before it accesses it. Furthermore, the accesses to the block of data by the task should be finished before the block of data is *released*. When a task has acquired a block of data in the memory, other tasks are not allowed to access it. A task may only access the memory using synchronization sections, i.e. only acquired blocks in the memory may be accessed. To other tasks, the acquires and releases of a task on a buffer appear in the order as they are performed.

Streaming consistency is applied because it allows *posted writes*, only if the synchronization variables are located in the same memory as the buffer. A posted write is a write operation that allows a writing task to continue, without waiting for a confirmation of the completion of the write. A write has completed when the data has been stored in the memory. The communication latency is hidden for a sending task when it uses posted writes, because its processing is not stalled while waiting for the confirmation. Note that more relaxed consistency protocols do not allow posted writes and thus require stalling the processor till the confirmation of a write is received.

Inter-task communication is realized via *circular buffers* placed in a memory. The C-HEAP protocol [13] is used for the buffer administration. No additional hardware or mutexes are required when using this protocol. In this protocol the buffer administration, consisting of a read and a write pointer, are stored with the buffer in the memory. A task, called the *producer*, can write at all addresses between the write and the read pointer. Similarly a reading task, called the *consumer*, can read all addresses between the read and the write pointer. To make the data at the address of the write pointer available to the consumer, the producer increases its write pointer. Only after the write pointer has been updated, the consumer can read the added data. Similar things can be done when the consumer does not need the address at the read pointer anymore. 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 circular buffer, it wraps around.

Omphale combines circular buffers with streaming memory consistency. In addition to a read and a write pointer in a circular buffer, the producer has to acquire the block of addresses starting at the write pointer, till at most the read pointer, that it will write. The consumer has to acquire the block of addresses starting from the read pointer, till at most the write pointer, that it will read. The block of addresses acquired by the producer will be called the *write window* and the block of addresses acquired by the consumer the *read window*. The producer or consumer can perform an acquire that extends their window with a consecutive address. When they perform a release, the address at their pointer is released and the pointer is increased. Re-

leasing and acquiring an address makes that the write and read window become sliding windows.

When the producer performs a release, the released address contains data for the consumer. When the consumer releases an address, the consumer can use it to store new data.

The *window size* of a read or write window is defined as the maximum number of addresses that will be acquired at a moment in time by the consumer or the producer, respectively. When a consumer performs an acquire but the address is still acquired by the producer the consumer is blocked, untill the address becomes available. The same occurs when the producer wants to acquire an address that is still acquired by the consumer. This results in the so-called back pressure in the task graph.

It is possible to generalize a circular buffer, such that multiple consumers can read from the circular buffer at the same time without interfering. This is possible because a read is not destructive, an address can be read multiple times. This allows multiple overlapping read windows. The read windows should be located between the read pointers and the write pointer, where one of the read pointers is increased when its corresponding consumer releases an address.

3.2 Inter-task communication

A circular buffer, which is a generalization of a FIFO buffer, is used for the buffering required by an edge in the task graph. The pattern in which the addresses will be accessed can be found in the NLPs of the two tasks connected by one edge. The NLPs contain loop-nests, the body of the loop-nest reads from the shared data structure or writes into the circular buffers.

A loop-nest consists of several loops that are nested. An example of a loop-nest is given in figure 2. A loop k has an *iterator* j_k, which is a variable that is increased by the loop. Furthermore loop k has two *bound expressions* B_k^l and B_k^u, expressing the minimum and the maximum value of j_k, respectively. The difference between two consecutive iterator values of a loop k is called the *stride* s_k, with $s_k \in \mathbb{Z}$. A loop k has the same subscript k assigned to all its variables and expressions, for example the outermost loop has subscript 0 for its iterator j_0, bounds B_0^l, B_0^u and stride s_0. The subscript is assigned consecutive, starting at k = 0 for the outermost loop, increasing the k, with one, for every loop in this loop. In the body of the loop-nest a data structure is written or read, using an affine index expression E to determine the address in the data structure.

$$\begin{array}{l} \text{for } j_0 \ : \ B_0^l \ : \ s_0 \ : \ B_0^u \\ \quad \text{for } j_1 \ : \ B_1^l \ : \ s_1 \ : \ B_1^u \\ \quad\quad \text{for } j_2 \ : \ B_2^l \ : \ s_2 \ : \ B_2^u \\ \quad\quad\quad \mathrm{Y} = \mathrm{X}[E] \end{array}$$

Figure 2: Example of a nested loop

The simplest communication pattern is when both the producing and consuming loop-nests start at the first address in the shared data structure, that is for example an array, and they consecutively proceed till the last address. In this case they write to and read from the array in the consecutive order as it is stored in the memory and use an address only once. Note that in this case the information could even be communicated through a FIFO. However, in general different read and write patterns will occur. For communication patterns two special types of behavior are distinguished that should be supported in the applied buffer solution, namely out-of-order communication and multiplicity.

Out-of-order reading or writing is defined as not consecutively reading all addresses from the shared data structure, starting from the first address. An example for out-of-order reading is shown in figure 3, where the loop-nest starts with reading address 1, followed by address 0 and so on.

$$\begin{array}{l} \text{for } j_0 \ : \ 1 \ : \ 1 \ : \ 4 \\ \quad \text{for } j_1 \ : \ 0 \ : \ 1 \ : \ 1 \\ \quad\quad \sim \ = \ X[2{*}\mathrm{j}_0 - j_1 - 1] \end{array}$$

Figure 3: Out-of-order reading with a nested loop

Read multiplicity is defined as a single address from the shared data structure that is read more than once. It is possible that between the multiple reads of an address other addresses are read. Write multiplicity cannot occur, since we require the loop-nest to be a Single Assignment Program (SAP), which constrains a variable to be written at most once.

The bounds and index expressions in the loop-nest do not have to be static. When they are static the complete communication pattern between two

loop-nests can be determined at compile-time. In case a parameter is used in the bounds and the index expressions, where the value of the parameter is not yet known at compile time but is known when the loop-nest is started, it might be possible to derive a parametrized communication pattern from the loop-nests. It is also possible that the bound and index expressions depend upon values in the shared data structure, these are called input data dependent expressions.

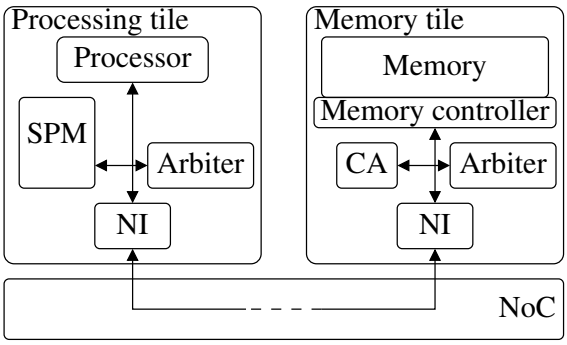

Figure 4: MPSoC template, with a processing and a memory tile

3.3 Memory tile

To allow the storage of large data structures, a memory tile is introduced in our template. The memory tile will contain circular buffers and should support streaming consistency. In order to hide the communication latency from the memory tile to the processing tile of the consumer, the memory tile should perform sender-initiated communication in combination with posted writes. When the producing processing tile performs posted writes, the memory tile must store the synchronization variables for the circular buffer.

The memory tile, shown in figure 4, contains a Communication Assist (CA), an arbiter, a network interface and a memory. The memory is for example a fast on-chip SRAM or a DDR memory. Data combined with its address can be written directly in the memory from the network interface, allowing a producer to perform posted writes. The CA is a small controller, that can read from the memory and post the data on the NoC, such that it is sent to an SPM of a consuming processing tile. As in the processing tile, the arbiter regulates the memory accesses from the CA and the NI.

The CA hides the communication latency for the processor tiles that require data from a circular buffer in the memory with sender initiated communication. The CA is located near the memory, such that it communicates directly with the memory and not via the NoC. The CA reads the data from the memory, so for the circular buffer in the memory it acts as a consumer. In order to keep the hardware of the CA simple, it will read the data written by the producer in the circular buffer in the memory, consecutively. When the producer slides its write window, the CA slides its read window. The CA has a copy of the read and write pointer from the circular buffer in the SPM of the consumer. When the write pointer is smaller than the read pointer, the CA acquires a location in the circular buffer in the SPM at the consumer. This is followed by precommunicating the data to the circular buffer, using a posted write. If the CA cannot immediately acquire a location in the circular buffer, it has to wait till the read pointer is increased. As proposed by [13] checking the read pointer can be implemented by the CA polling the SPM for it or by the consumer sending an interrupt to the CA when the read pointer is changed.

3.4 The tool flow

The Omphale tool flow is shown in figure 5. The core part of the tool flow derives a CSDF model that shows the communication pattern between the tasks in the task graph, determines the location of the buffers in the resulting task graph and examines if additional parallelism is required given the memory sizes of the target architecture. Techniques as used in for example the Compaan [11] and PN [19] tool can be applied to extract additional task level parallelism. Four operations can be distinguished that the Omphale tool flow has to perform: derive a CSDF graph from a task-graph, derive additional task level parallelism and add it to the task-graph, insert tasks that represent the copying action of a CA in the task-graph and expand the tasks with acquires and releases to use the circular buffers.

The input of Omphale is the task-graph of a job. The code of the tasks should be in the form of a NLP, such that the communication pattern that is required for sender initiated communication can be automatically recognized. The resource constraints

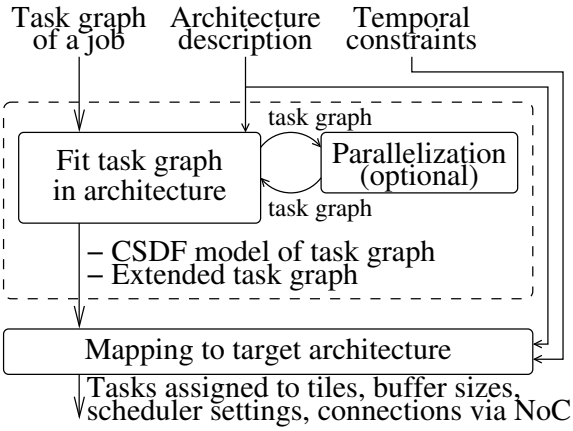

Figure 5: The Omphale tool flow, delivering a task graph and a corresponding CSDF model, satisfying the available memory sizes

required by Omphale are the sizes of the different memories available in the MPSoC.

Omphale delivers a task-graph and a corresponding Cyclo Static Data Flow (CSDF) model [3]. The resulting task graph may contain additional tasks, due to the inclusion of memory tiles or additional tasks found by the extraction of task level parallelism. The code of the tasks is extended with acquires and releases enabling sender initiated communication via circular buffers. The CSDF graph is a model of the task-graph. The phases of the actors show the communication pattern of each actor. These phases show how many memory addresses the task, corresponding to the actor, requires and can be used to derive buffer capacities.

3.4.1 Additional task level parallelism

In some cases for a task in a job additional task level parallelism can be extracted. The task level parallelism can be derived from an NLP, since all data dependencies in an NLP can be derived at design time. Omphale could for example use techniques similar to the ones in the Compaan and PN tools.

The Compaan [11] and PN [19] tools extract a PRDG from an NLP. This is done by first extracting a Single Assignment Program (SAP) from the NLP with their MathParser tool. In a SAP, a variable is assigned only once. From this SAP a Polyhedral Reduced Dependency Graph (PRDG) is derived with their DgParser tool. The nodes in this PRDG have input port domains and output port domains, these are polyhedral spaces that are shaped according to the nested-loop in the code of the node. The edges connect the output port domains to input port domains, their combinations show the communication patterns for the shared data structures.

The nodes in the PRDG represent loop-nests in the SAP, so they show task level parallelism. The edges show the communication of data between the ports of the nodes. Therefore a task graph can be derived from the PRDG, with probably more tasks than the input task graph of a job.

3.4.2 Fitting the task graph to the target architecture

Omphale should fit the task graph of a job to a target architecture, such that tasks and their buffers do not require more memory than available. To realize this the communication patterns of data between the tasks needs to be analyzed and reorganized at compile time. If possible a single circular buffer in the SPM of the consumer is used. When a single buffer would exceed the capacity of the SPM of the consumer, a combination of two buffers can be used, one in the SPM of the consumer and one in a memory tile.

Every edge in the task graph represents a circular buffer, with a read and a write window. In order to realize a simple acquire/release scheme in the tasks, at most a single address is acquired at the beginning of an iteration of the loop-nest and released at the end. Thus the window is sliding through the circular buffer at the pace of the statements in the inner loop of the loop-nest. The read and write window start at the first address of the shared data structure.

In order to have the window sliding through the circular buffer at the pace of the statements in the inner loop of the loop-nest, it should be verified that the address that is read or written is in the window. For example, if a consumer is reading consecutive addresses starting at address 0 of the shared data structure, a window of size 1 would suffice. Every iteration this consumer performs an acquire, reads the data and releases the address. A different example is shown in figure 3, where the consumer starts with reading address 1 and consecutively reads address 0. In this example an

acquire should be performed before the loop-nest starts. Now in the first iteration of the loop-nest address 1 is acquired, the address that also has to be read. Furthermore the data at the read pointer cannot be released in the first iteration of the loop, because address 0 is read in the second iteration. This means the releasing should be delayed till the second iteration of the loop-nest.

The number of acquires that have to be performed before the loop-nest is started is called the *lead-in*. The number of iterations after which releases can be performed, is called the *lead-out*. The window size can be determined as the sum of the lead-in, the lead-out plus one. The Omphale tool determines the size of the windows such that out-of order reading and writing and read multiplicity are covered, for every NLP that has affine loop bounds and an affine index expression. Note that parametric loop-bounds and parametric index expression can make it difficult to determine the window size. An even broader and more challenging extensions would be to allow an input data dependent index expression and input data dependent bound expressions.

With the lead-in and lead-out, the CSDF model can be derived from the task graph. The actors in a CSDF model operate in a periodic fashion, where the period consists of a number of phases. The edges of an actor are annotated with the number of tokens produced or consumed for each phase. The number of produced and consumed tokens for an actor can differ per phase. During the first phase the number of tokens for the lead-in are consumed and in the last phase the last tokens for the lead-out are released.

When the read and the write windows are smaller than the shared data structure it is not necessary to have a circular buffer with the size of the shared data structure. It is possible to use the addresses of the data structure modulo the size of the circular buffer. When the circular buffer is one word smaller than the read and the write window, it is known that the windows can slide through the buffer.

The Omphale tool flow tries to locate the circular buffer in the SPM of the consuming processing tile, such that it can perform reads with a low communication latency. In case it is not possible to fit a circular buffer with the read and the write window in the SPM, the read and write window are located in two separate circular buffers. A circular buffer with the write window is located in a memory tile and a circular buffer with the read window is located in the SPM of the consumer. Now the CA from the memory tile is reading the in-order data from its circular buffer and posting this data to the circular buffer in the SPM of the consumer. Because the producer releases the data in-order, the CA can always read the data in-order from its circular buffer and writes it in-order in the SPM, thus its read window in the memory tile and its write window in the SPM have a size of 1. This keeps the hardware of the CA simple.

When a CA is inserted between a producer and a consumer this should be shown in the task graph and the CSDF graph. Therefore Omphale replaces the original edge with an actor that represents the CA in the CSDF graph that has an edge to both the producing and the consuming actor. In a similar way the edge in the task graph is replaced by a task representing the CA that is connected to the producer and the consumer.

Using the architecture constraints Omphale examines if the memory requirements of the tasks do not exceed the available SPM in the processing tiles. If a task exceeds the amount of available memory, Omphale uses a Compaan like tool flow (subsection 3.4.1) to examine if additional task level parallelism can be extracted. If it can be extracted, the single task in the original task graph is replaced by multiple tasks that are connected with edges. In the CSDF graph the corresponding actor is replaced by multiple actors. When task level parallelism is extracted the size of the private data per task or the size of the shared data structures over an edge are reduced. For the new edges in the tasks graph, Omphale derives the read and the write window and the phases for the corresponding edges in the CSDF model once more.

When the CSDF graph is derived for a job and the phases for the actors are determined, the code in the corresponding tasks should be extended to use the circular buffers. The acquire and release statements are inserted in the code of the actors, using the phases found for the CSDF graph.

The combination of a task graph with circular buffers naturally results in a pipelined implementation, if at least the minimum buffer sizes are met. In [22] a method is described to derive the minimum buffer sizes from a CSDF model, given the end-to-end throughput and latency constraints of

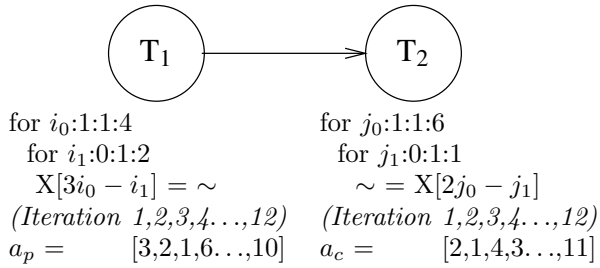

Figure 6: Task-graph with a producing task T_1 and a consuming task T_2

a job. Starting from the minimum circular buffer size, the mapping tool can assign more memory to increase the amount of pipelining. A job has a maximum amount of memory that can be assigned, after which the throughput will not increase anymore. The trade-off between the total amount, the type of memory assigned and the achieved throughput determines the cost of the solution.

4 Window solution

This section shows how Omphale derives a CSDF model from a task-graph, where the lead-in and lead-out are used to annotate the CSDF model. Furthermore it shows how the code in the task-graph is extended with acquires and releases.

From the nested-loop of the consuming task T_2, in figure 6, an *address list* a_c can be derived, containing the addresses in the order that they are read. The first element in the address list a_c is 1, the last element in the list is a_c^h, which equals the number of reads that are performed. In the address list a_c of the nested-loop it can be seen that at iteration 3 the address $a_c[3] = 4$ is read. In a similar way an address list a_p with a_p^h items can be derived for the producer.

The circular buffer in the SPM may be smaller than the shared data structure and the acquires not necessarily start at the first location of the buffer. Furthermore the circular buffer is probably not located at the first location in the memory. Therefore the address of data in the shared data structure needs to be translated to the address in the buffer in the SPM.

The address the consumer has to read in the shared data structure is called r. A function t can be defined that gives the location of address r in the SPM. This functions contains three variables, the variable l is the location of the buffer in the SPM, q the location in the buffer where the acquires started and z the size of the buffer.

Definition 4.1. *An address r in the shared data structure can be translated to the physical address $t(r)$ in the SPM with the function:*

$$t(r) = (r + l + q)\%z$$

In the buffer at the consumer a *lead-in* d_1 is needed to cover addresses that have to be acquired in advance, so to make sure that for the i^{th} read the address $a_c[i]$ is acquired even if $a_c[i] > i$. For example, in the first iteration of task T_2, from figure 6, address 2 should be acquired, so $a_c[1] = 2 > 1$.

A graphical method to find the lead-in is shown in figure 7, for the consumer from figure 6. The producer releases consecutive addresses in the SPM and the consumer reads them in a different order. The lead-in d_1 is determined by shifting the list with read addresses to the right in relation to the list with acquired addresses such that the addresses are acquired earlier or at least at the same moment as they are read. In the figure address 1 needs to be acquired before address 2 can be acquired and read, the same occurs for the addresses 4 and 6. Thus a d_1 of 1 is needed such that the addresses are acquired before they are read.

Lemma 4.1. *A lead-in of $d_1 = \max_i(a_c[i] - i)$ acquires, with $1 \leq i \leq a_c^h$, ensures that every address is acquired before or when it is read.*

Proof: The proof is by construction. In an iteration j address $a_c[j]$ is read, so at least $a_c[j]$ addresses should be acquired in the buffer. Each iteration performs 1 acquire, so initially at least $a_c[j]-j$ acquires should have been performed, when $a_c[j] \geq j$. To make sure that in each iteration of the loop-nest the read address is acquired, the maximum number of required initial acquires d_1 is found by $\max_i(a_c[i] - i)$, with $1 \leq i \leq a_c^h$. □

When we want to reuse space in the buffer a *lead-out* d_2 is needed before starting to release addresses that are acquired, to make sure that the address at which data is stored is not released before it is read for the last time. The lead-out is the number of reads after which the consumer can start releasing addresses in the buffer. For the example in figure 6, address $a_c[1] = 2$ is acquired after $a_c[2] = 1$, but

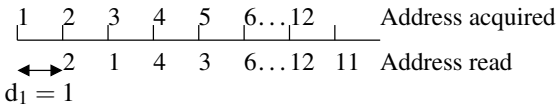

Figure 7: The lead-in guarantees that the used data *has been written* in the circular buffer

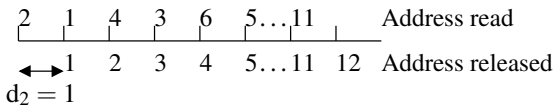

Figure 8: The lead-out guarantees that the data *still is* in the circular buffer till it is used

read earlier. The read from address $a_c[1] = 2$ cannot be followed by a release, because than address 1 is released before it is read.

In figure 8 the d_2, for the consumer in figure 6, is graphically determined. The list with released addresses is shifted right such that data is read earlier or at least at the same moment as it is released. Note that the order of the acquires is also the order in which the addresses are released.

Lemma 4.2. *A lead-out $d_2 = \max_i(i - a_c[i])$, with $1 \leq i \leq a_c^h$, is the number of reads after which reads can be combined with releases, making sure that the released data has been read for its last time.*

Proof: The proof is by construction. In an iteration j, at least the address $a_c[j]$ should still be acquired in the buffer. After an initial number of iterations each iteration releases 1 address. To make sure that address $a_c[j]$ is still acquired in iteration j, at least the first $j - a_c[j]$ iterations should release no address, when $j \geq a_c[j]$. To make sure that in each iteration the read address is still acquired, the first $\max_i(i - a_c[i])$ iterations, with $1 \leq i \leq a_c^h$, of the loop-nest should not release an address. □

Figure 9 shows a circular buffer containing a window. The 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 $w = d_1 + d_2 + 1$. In case of read multiplicity it is possible that $d_1 + d_2 + 1$ is larger than the number of data items produced. In this situation the window size is chosen as the number of data items produced, $w = a_p^h$. With the found d_1 and d_2 for the consumer from figure 6, a read window with size 3 would be required.

Theorem 1. *When a window size $w = d_1 + d_2 + 1 \leq a_p^h$ is used, the data that will be read by the consumer is always in the window. So $i - d_2 \leq a_c[i] \leq i + d_1$*

Proof: The lemmas 4.1 and 4.2 contain the proof that shows that for each of them individually the buffer will always contain the address to be read. Lemma 4.1 shows that the lead-in is determined in such a way that $a_c[i] \leq i + d_1$. Lemma 4.2 shows that the lead-out is determined such that $i - d_2 \leq a_c[i]$. Therefore the combination of both ensures that the address is always in the window. □

It is possible to derive the write window for a producer, in a similar way as the read window for the consumer. In this case d_1 denotes the number of addresses that should be acquired such that the address that has to be written is always in the window. The d_2 denotes the number of iterations the loop-nest should wait with releasing addresses in the circular buffer, such that the address contains valid data. The producing task T_1 from figure 6, has a $d_1 = 2$ and a $d_2 = 2$, so a write window with the size 5 is required.

Figure 10 shows a combination of figure 7 and 8, including when acquires, reads and releases should be performed. This information can be used to derive the CSDF actor and to extend the code of the task with acquires and releases.

Figure 11 shows the CSDF graph that corresponds to the task graph from figure 6. The actor A_1 corresponds to task T_1 and actor A_2 to task T_2 from figure 6. The number besides the black dot on the edge in the CSDF graph denotes the initial number of tokens. In this example there are 7 initial tokens, 3 for the read window plus 5 for the write window minus one. Every actor has an implicit self edge with one initial token, to make sure that the actor is finished before it is started again. In its first phase actor A_2 consumes 2 tokens and releases none. In its second phase A_2 consumes a third token and at the end of this phase a token is released

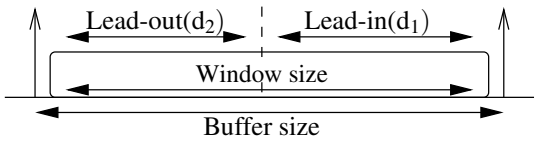

Figure 9: A circular buffer containing a window, build up by a lead-in, a lead-out and the current address

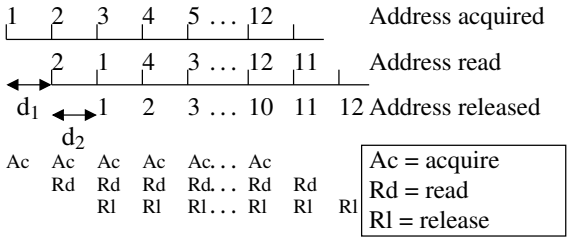

Figure 10: The lead-in and the lead-out and the pattern of acquires and releases

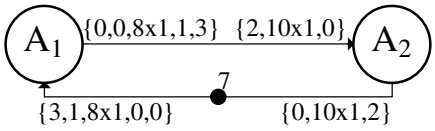

Figure 11: The CSDF model derived from the task graph in figure 6

again. In each of the following 9 phases a token is acquired and released. In the task graph this corresponds to task T_2 starting with acquiring two containers in the circular buffer after which the first read from the body of the loop-nest is performed, a third container is acquired for the second read, after which the first container is released, followed by 9 iterations that acquire and release a container. The scheme of producing and consuming tokens in the CSDF model shows the sender-initiated communication at a word level granularity, that is used to hide the communication latency. Furthermore the minimum buffer capacity for the producer consumer pair in isolation is shown by the number of initial available tokens. In a CSDF model that contains cycles, it is possible that more initial tokens are required on an edge. This requires an analysis considering the whole CSDF model, as shown in [22].

In figure 12 the tasks are extended with acquires and releases, according to the found pattern in the CSDF graph. Both tasks start with acquiring an initial number of d_1 locations in the circular buffer $CB1$ and than acquiring one location in each iteration untill all locations of the data structure have been acquired. Both loop-nests are extended with a counter to count the iterations. The releases are started after d_2 iterations. At the end of the loop-nest the d_2 locations that are still acquired in the circular buffer are released. To access the correct


```
int tp = 0                          int tc = 0
q = writepointer(CB1)               q = readpointer(CB1)
acquire(2,CB1)                      acquire(1,CB1)
for i0:1:1:4                        for j0:1:1:6
  for i1:0:1:2{                       for j1:0:1:1 {
   if(tp < 11)                         if(tc < 12)
     acquire(1,CB1)                      acquire(1,CB1)
   write(CB1,                          ~ = read(CB1,
   (3i0 − i1+l+q)%8,~)                 (2j0 − j1+l+q)%8)
   if(tp > 2)                          if(tc > 1)
     release(1,CB1)                      release(1,CB1)
   tp++                                tc++
}                                   }
release(2,CB1)                      release(1,CB1)
```

Figure 12: Task-graph with tasks that are extended with acquires and releases

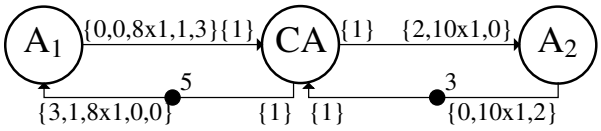

Figure 13: The CSDF model from figure 11 extended with a CA actor

location in the shared data structure, the read and write addresses of the tasks are translated, according to definition 4.1. Note that the parameter l, the location of the buffer in the SPM, is determined by the compiler. The parameter q, the location where the first address of the shared data structure is stored in the buffer, is the write pointer of the producing actor before it performs the initial acquires, which is similar to the read pointer before the initial acquires of the consumer.

The addresses released from the write window are acquired in the same order by the read window, allowing the write and the read window to be separated. Figure 13 shows a CSDF graph where a CA is inserted, such that the producer and the consumer have separate circular buffers, with a smaller buffer in the SPM of the consumer. When an address is released from the write window, the CA copies the data to the corresponding address in the circular buffer of the consumer and releases the ad-

dress there. To keep the CSDF graph similar to the task graph, the task graph should also be extended with a CA. The code of the CA task, represents the copy operation the CA performs.

5 Conclusion and Future work

The previous sections have discussed and shown the Omphale tool flow. The tool flow performs the first steps in mapping a job to a target architecture. The tool delivers a task graph where the communication is organized via circular buffers and a CSDF model that shows the communication pattern.

The NLPs of the tasks in the input job implicitly contain the patterns in which the shared data structures are communicated between tasks. By deriving the CSDF graph from a task graph, the communication pattern of the shared data structure is made explicit. The phases of an CSDF actor are used to extend the corresponding task in the task graph to use a sliding window in a circular buffer. The sliding window covers out-of-order communication and read multiplicity. Furthermore, by placing the circular buffer in the SPM of the consuming task, sender initiated communication is performed such that communication latency for the consumer is hidden. The communication latency for the producer is hidden because it can perform posted writes, due to the used streaming consistency model.

When a task and its circular buffer require more memory space than available in the SPM of the consumer, Omphale reorganizes the task graph and the CSDF model. Either the circular buffer is split and a part of it is located in a memory tile, or additional parallelism is derived from the task. If a memory tile is used for the circular buffer, the task graph is extended with a task and the CSDF model with an actor for the memory tile. If additional parallelism is derived, the single task and actor are replaced by multiple tasks and actors in the task graph and the CSDF model, respectively. Using the resulting CSDF model and the task graph, other tools in the mapping flow can map the job to the target architecture by determining the buffer sizes, scheduler setting and the connections via the NoC and assigning the tasks to tiles.

An Integer Linear Problem formulation to determine the window size for two communicating tasks, with their NLPs, is under development. This formulation considers no parameters and input data dependency, in the bound or index expressions. It is possible to extend the formulation to allow parameters. The formulation can be further extended to allow if-statements in the loop body. The trade-off between a hardware and a software implementation for the transformation of the index expression and the acquire and release statements can be investigated, in order to improve the performance of the resulting mapping. An interesting optimization is to change the pattern in which the data structure is written and read, by changing the index expressions, such that the write window becomes larger and the read window smaller. This allows a solution in which less memory is used in the SPM and more in the memory tile.

References

[1] M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. *to be published at the Bits & Chips Embedded Systemen symposium*, October 2007.

[2] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, 1966.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *Signal Processing, IEEE Transactions on*, 44[2]:397–408, 1996.

[4] J. v. d. Brand, M. Bekooij, and A. Moonen. Streaming memory consistency for efficient MPSoC design. 2007.

[5] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl. Sprint: A tool to generate concurrent transaction level models from sequential code. *EURASIP Journal on Advance in Signal Processing*, 2007.

[6] D. Culler, A. Gupta, and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1997.

[7] 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. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 14(3):279–291, 2006.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[9] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design & Test of Computers, IEEE*, 22(5):414–421, 2005.

[10] J. Kang, A. van der Werf, and P. Lippens. Mapping Array Communication onto FIFO Communication-Towards an Implementation. *Proceedings of the 13th international symposium on System synthesis*, pages 207–213, 2000.

[11] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, pages 13–17, 2000.

[12] P. Lippens, J. van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle. PHIDEO: a silicon compiler for high speed algorithms. *Design Automation. EDAC. Proceedings of the European Conference on*, pages 436–441, 1991.

[13] 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. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[14] G. Ottoni, R. Rangan, A. Stoler, M. Bridges, and D. August. From Sequential Programs to Concurrent Threads. *IEEE Computer Architecture Letters*, 5(1), 2006.

[15] P. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1998.

[16] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 177–188, 2004.

[17] T. Stefanov and E. Deprettere. Deriving process networks from weakly dynamic applications in system-level design. *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign & system synthesis*, pages 90–96, 2003.

[18] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, 2004.

[19] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007:Article ID 75947, 13 pages, 2007.

[20] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, J. van Meerbergen, and A. van der Werf. Modelling periodicity by phideo streams. *Sixth International Workshop on High Level Synthesis*, pages 256–266, November 1992.

[21] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, 2004.

[22] M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS'07, Seattle, WA, United States*, pages 281–292, Los Alamitos, CA, United States, April 2007. IEEE Computer Society.