Predictable embedding of large data structures in multiprocessor networks-on-chip

Sander Stuijk, Twan Basten, Bart Mesman and Marc Geilen
Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
s.stuijk@tue.nl

Abstract - Predictable, tile-based multiprocessor networks-on-chip are considered as future embedded systems platforms. Each tile contains one or a few processors and local memories. These memories are typically too small to store large data structures (e.g. a video frame). A solution to this is to embed tiles with large memories in the architecture. However, fetching data from these memories is slow because of the large network delays. The delay can be hidden by using prefetching. Our main contributions are models that allow timing analysis to provide guaranteed quality and performance when using remote memories and prefetching. We use two realistic video applications to show that our models can be used in practice to derive a predictable system using large memory tiles and prefetching, and to provide guaranteed real-time performance.

1. Introduction

Current developments in settop-box products for media systems show that chips are becoming memory dominated (estimated 90% in 2010) for two reasons. Firstly, logic scales faster with chip technology than memory. Secondly, current media applications require increasingly more memory.

For cost reasons, we can no longer afford different subsystems to have separate, large memories. This suggests the need for a high level of re-use of these memories. It is therefore expected in the electronic design community that future electronic systems re-use platforms that integrate many IP-blocks and memories. These platforms will concurrently execute many applications and (sub-)tasks. The number of possible *use cases* is enormous. For example, in a modern television platform 60 applications are running in parallel, corresponding to an order of 60! possible use cases. It is clearly impossible to verify the correct operation of all these situations through testing and simulation. The product divisions in large companies currently already report 60% to 70% of their effort being spent in verifying potential use cases and this number will only increase in the near future. This has motivated researchers to emphasize the ability to analyze and predict the behavior of applications and platforms without extensive simulations. Real-time requirements in media systems have put the main focus on predicting the *timing* behavior of complex media systems. This development has accelerated the use of *models of computation* as a class of them allows analysis of the system at design time. There are two requirements for using these models of computation, namely the development of good analysis tools that exploit these models, and the ability to capture real-world behavior.

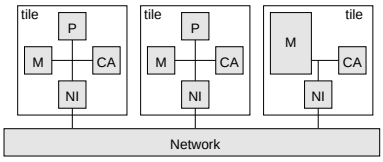


Figure 1. MP-SoC architecture template.

Untill now, designers lived comfortably with dedicated memory close to the computational logic, thereby allowing predictable access times. In future platforms, as explained, memories will be *distant* and *shared* among potentially many computational resources. This problem was first identified in [16]. That abstract presents a first solution on how to model memory accesses on a distant and shared memory in an important class of future embedded platforms, namely network-on-chip-based multiprocessors. The current paper adds to this analyzable models to predict timing properties of computations operating on large memories in network-on-chip-based multiprocessors. The models incorporate prefetching techniques, block fetching, and provide the basis for an application programming interface hiding the complexity of using shared, distant memories. In two case studies on realistic media applications, we show how our models can be used to derive predictable and acceptable timing numbers for an application when distant, shared memories are used.

2. Background

2.1. Architecture template

Culler et al. describe in [4] a general template for multiprocessors. It consists of multiple processing tiles connected with each other by an interconnection network. Each *tile* contains one or a few processor cores and local memories. The tile may contain communication buffers, accessed both by the local processors and the network. The tile has a small controller, called communication assist (CA), that performs buffer accesses on behalf of the network. It decouples the communication and computation. The archi-

tecture template in our work is shown in Figure 1. Many multiprocessor systems-on-chip (MP-SoCs), e.g. Daytona [1], Eclipse [15], and StepNP [11], fit nicely into this template. The interconnection network in our template is a network-on-chip (NoC). Each processing tile is 'plugged' into the network through a network interface (NI). The NoC must offer unidirectional point-to-point connections. The connections must provide guaranteed bandwidth, and a tightly bounded propagation delay per connection - i.e. they must provide a *guaranteed throughput*. The connections must also preserve the ordering of the communicated data. Further details of the NoC are not relevant. Examples of NoCs providing these properties are Æthereal [14] and Nostrum [9].

Video processing applications perform operations on large data structures like frames. A single frame in an HDTV consists of over 2 million pixels. The memories near the processor on a tile will not be big enough to store this amount of data [13]. Therefore, we introduce tiles into the architecture template that contain large memories and no computational elements. These tiles are called *memory tiles* as opposed to the *processing tiles* which do contain computational elements. The data in a memory tile can be accessed through the communication assist of that tile.
In the rest of the paper, we refer to a memory tile as the *remote tile* and to its memory as the *remote tile memory*. The processing tile that runs the code segment we are concerned with is referred to as the *local tile* and its memory is called the *local tile memory*.

2.2. Model of computation

Synchronous dataflow (SDF) [8] is a model of computation that allows design-time analysis of multiprocessor applications [7, 13]. Nodes in an SDF graph, called *actors*, typically correspond to functionality, code, that must be executed, or to other actions performed by the system being modeled. Edges show data dependencies (*data edges*) or execution order (*sequence edges*). Every edge can carry an infinite number of *tokens* between two actors, and contain *initial tokens* (present at the edges at start time). Each actor in the model has a *firing rule* which specifies the number of tokens that must be present on each of its inputs before it can *fire*. A constant number of tokens is produced to and consumed from all edges in first-in-first-out order by each firing.

If an actor firing represents the execution of a code segment, the result of this execution may be data that is needed by other actors and/or results needed for the next firing of the actor itself. The first situation is modeled via data edges. For the latter situation, *self-edges* are used. A self-edge is a data edge of an actor to itself. The token sent over the self-edge models the state that is communicated (stored) between two consecutive actor firings (*state saving*).

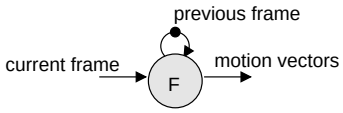


Figure 2. SDF actor for FSBM.

Typically, dataflow models see a token as an indivisible element. We assume sometimes that a token, after being read by an actor, may be divided into a set of *data elements*. The actor has random access to the data elements inside the token. For example, a video frame consists of many pixels. The frame can be communicated between two actors as a single token. But the actor may operate on the individual pixels (data elements).

Following common practice, when doing timing analysis, we extend SDF models with *actor execution time* annotations (real numbers), representing the time span between consumption and production of tokens. These annotations can be either fixed or variable.

In this paper, we also use Boolean dataflow (BDF) [3]. The main difference between BDF and SDF is that in BDF we have a *switch* and *select* construct. The *switch* copies the data it reads from its input to one of its outputs based on the value of the control token. The *select* reads, based on the value of a boolean *control token*, data from one of its inputs and copies this data to the output. The switch and select introduce data-dependent behavior, which makes a BDF graph in general non-analyzable at design time.

2.3. Example: full-search block matching

Full-search block matching (FSBM) is the basic algorithm for detecting motion vectors in video sequences. Many variants exist, like 3D recursive search [5]. Motion vectors describe the motion of a block of pixel data from one frame to another. They are typically used to perform spatial up-conversion or to achieve a higher data compression (e.g. in H.263 and MPEG-2 encoders). The algorithm divides the current frame into a sequence of blocks of typically 8 by 8 pixels. It then takes for each block in the current frame a window of typically 4 by 4 or 8 by 8 blocks centered around the block's position from the previous frame. The motion vectors are determined by the best match, using the sum-of-absolute differences, between the pixels in the block from the current frame and the pixels in the window.

The computation of all motion vectors for a frame can be represented by the SDF actor as shown in Fig. 2. The actor reads in the current frame from the input and the previous

frame from the self-edge. Next, it computes and outputs all motion vectors, and it outputs its state (i.e. the current frame) on the self-edge. The current frame becomes in this way the previous frame in the next firing.

3. Approach

To get a system with predictable timing properties, we need an appropriate design flow. The starting point of this flow is an SDF model of the application and a predictable architecture as described in Sec. 2.1. We use stepwise refinement of the application SDF model to include mapping decisions and to model the effects of architecture details. The result is a combined SDF model of the application and architecture with predictable behavior wrt timing, memory usage etc. See [13] for an example of such a flow. The mapping of data structures onto memories is one of the decisions that should be taken into account in this flow.

The FSBM application illustrates the problem. The available storage space typically forces a designer to store the state and input (frames) in a remote tile memory; the output (motion vectors) may be stored in the local tile memory. Note that the precise mapping of input, output, and state data to local and remote memory may differ per application. In this paper, we focus on the situation where input and state are stored in remote memory, and output in local memory; other situations can be handled similarly.

To be able to reason about the timing properties of this mapping decision, we need a model of the memory accesses to the remote memory. Such a model is presented in Sec. 4. The communication between the remote memory and the local actor is handled by the communication assist (CA). Its role is to provide (pre)fetching of data from a remote memory in a transparent manner to a programmer. Support for prefetching is important as it allows hiding of large delays experienced when data is fetched over the network. The CA cannot be ignored when we want to build a predictable system. Therefore, we need to refine the memory access model with an SDF model, presented in Sec. 5, of the (pre)fetching functionality of the CAs for accessing data in a remote tile memory. When a remote memory is accessed, data is sent over the network. Therefore, we also have to take the timing properties of the network into account. In the experimental evaluation of Sec. 6, we show how an existing model of the network [10] can be combined with the models developed in this paper to reason about the timing behavior of an application and the dimensioning of system components (e.g. network interface buffer sizes and communication bandwidth).

4. Memory model

4.1. Memory access model

Assume an actor A with a single input i, a single output o and a self-edge for its state. The following approach can

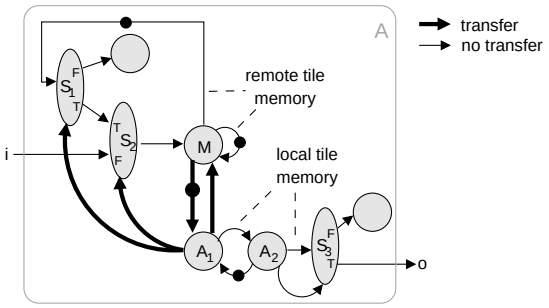


Figure 3. BDF model for remote memory accesses of state and input data.

be used to model the mapping decision of state and input to remote memory. The basic idea is that the remote tile memory is modeled through a separate actor M. An actor can send a token to M to request a read or write of data stored in memory M. On its turn, M returns the requested data elements to the requesting actor. A BDF model based on this idea is shown in Fig. 3. The state of A stored in the remote tile memory is modeled via the self-edge on actor M. The input data for A, also stored in the remote tile memory, is modeled via the loop going through the switch S_1, the select S_2 and the actor M. The switch S_1 and select S_2 are used to keep the existing input token (both control tokens true) or to read a new input token into the remote tile memory (both control tokens false). In the situation that the control tokens are false, the current input token is discarded and a new input token is read via the input i. This approach allows for (pre)fetching of data elements to be incorporated later.

The functionality of actor A is split over two actors A_1 and A_2 which allows tighter bounds on the time at which tokens are consumed/produced because A_2 and M can operate in parallel. When actor A_1 fires, it reads the data elements it requested from the remote tile memory into the local memory. The local tile memory is modeled by the edges from A_1 to A_2 and from A_2 to switch S_3. A_1 simultaneously sends a new request to the remote tile memory to read and/or write data. It further outputs control tokens for S_1 and S_2, both with the same value. As soon as A_1 finishes its firing, it hands over control to A_2. Actor A_2 reads the local state and performs a transformation on it, which is equal to the transformation performed by the original actor A. At the end of its firing, it outputs a control token to switch S_3. This switch is used to control the production of an output token for the original actor A on its output o. The actor A_2 indicates to the switch whether the produced data, stored in local memory, is valid or not. If it is not valid, the token is discarded; if it is valid, the token is put on output o. Actor A_2 hands back control to A_1 as soon

as it has to read or write data that is not stored in the local tile memory.

It is important to note that not all edges in the graph shown in Fig. 3 represent an actual transfer of data in the system. Only tokens that are sent over the bold edges must physically be transfered from one memory location to another. Tokens sent over the non-bold edges require no actual transfer of data (i.e. transformations can be done in place). In the remainder, we use this convention for all edges.

4.2. Sharing remote memories

As explained, remote memory tiles will be shared among many processing tiles. To get a predictable system, each processing tile must get guarantees on the response time of the remote memory. This can be realized by using a TDMA scheme to control access to the remote memory [2]. Using such a scheme, different processing tiles using the same remote memory can be considered independently. For each actor A whose data is mapped to a remote memory we can use the model shown in Fig. 3 (or variants of it for different mapping decisions). The sharing of the remote memory is taken into account via the timing behavior of actor M (memory).

4.3. Typical access patterns

To allow a refinement of the general model of Sec. 4.1 with prefetching, we briefly discuss typical memory access patterns for the data elements contained in remotely stored data. Data elements needed by actor A_2 (computation) must explicitly be fetched from actor M (memory). The subsequent data elements needed by A_2 may be at arbitrary positions in the remote data token. Thus, we have a *global* (possibly random) access pattern. Opposed to this, we see the situation in which subsequent data elements needed by A_2 are *local* to each other. For example, an actor doing some video processing may need access to many pixels (data elements) within a certain window of a frame (see e.g. the FSBM case). If the required data elements are local to each other, then all these elements can be fetched from the memory at once. In case we know in advance which data elements will be accessed, we can even prefetch the data from the memory. The access pattern will of course be known at *run-time*, but for some applications we can also derive it at *design-time*. The latter case allows design-time analysis of the timing behavior.

4.4. Design-time analysis

The BDF model shown in Fig. 3 cannot be analyzed at design-time [3] when the number of iterations between M, A_1 and A_2 before an output is produced is unknown. In case we know how often and when the memory needs to be

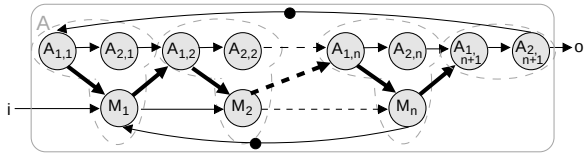


Figure 4. SDF model for remote memory accesses.

accessed, we can translate the BDF model via an unfolding into an SDF model that can be analyzed at design-time (see Fig. 4). The basic idea is that each iteration i results in its own set of M_i, $A_{1,i}$ and $A_{2,i}$ actors (encircled in gray in the figure). All M_i are then connected to each other via sequence edges. Each actor $A_{2,i}$ is also connected via a sequence edge to the actor $A_{1,i+1}$. During the last firing of $A_{1,i}$ ($i = n + 1$) it is not necessary to sent a new request from $A_{1,n+1}$ to the memory; thus the actor M_{n+1} can be discarded.

In the situation that we have further design-time knowledge, we can further reduce the number of actors. (This may be useful for speeding up timing analysis.) Each actor $A_{1,i}$ is responsible for accessing data from the memory. In case the whole memory access pattern is known at design-time, then we can assume the M_i actors to send the correct data. (It remains to realize this assumption via appropriate prefetching, as explained in the next section.) As a result, all $A_{1,i}$ actors can be removed. The $A_{2,i}$ actors can read the correct data directly from M_i. We illustrate this transformation using the FSBM application. FSBM requires access to a window from a video frame. This window moves in a pattern defined at design-time through the frame. Hence, the memory access pattern is fully known at design-time. The original FSBM actor (Fig. 2) can be translated to the model with explicit memory accesses (Fig. 3). The next step is to translate the BDF model to an SDF model using our knowledge of the number of iterations through the actors M, A_1 and A_2. The resulting SDF graph is shown in Fig. 5. It shows n memory actors M_i which send data to n actors F_i. Each F_i computes a single motion vector. A frame size of 352 by 288 pixels would make n equal to 1584.

4.5. Expressiveness of the models

Section 4.1 presents a general BDF graph to explicitly model memory accesses to a remote memory. BDF graphs may exhibit data-dependent behavior, which in general makes it impossible to analyze their behavior at design-time without analyzing all possible input streams. Being able to analyze the system at design-time without extensive simulation of many possible inputs is key for predictable multiprocessor systems. SDF models on the other hand can be analyzed at design-time. However, SDF graphs cannot

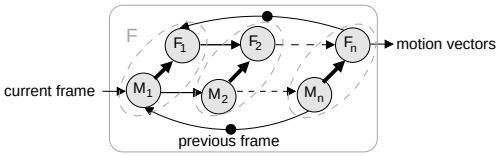


Figure 5. SDF model for FSBM with remote tile memory.

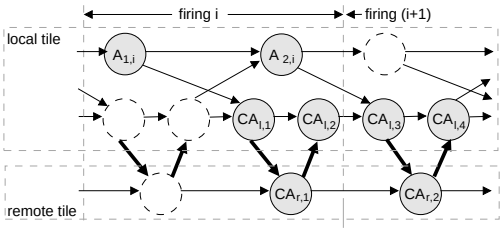


Figure 6. SDF model for the CAs.

model data-dependent behavior (i.e. all actors have a constant rate). This may seem to impose a very severe restriction to the applicability of our SDF model. However, the SDF model is only used to analyze the behavior of the application when mapped onto the multi-processor system under certain conditions. For instance, an MPEG decoder may be modeled as a BDF graph. Our BDF memory model can be used to make accesses to a frame memory explicit. To analyze the worst-case behavior of the MPEG decoder, the graph may be transformed into an SDF model; all variable (data-dependent) rates in the BDF model are replaced in the SDF model by their worst-case rates. The SDF model can then be used to analyze, for instance, the worst-case timing behavior when this remote frame memory is used. More in general, one can use the concept of scenarios as presented in [12, 17] to capture data-dependent behavior and certain control aspects. Using the concept of scenarios, a *set* of SDF graphs can be derived from the BDF model, one for each typical mode of execution. They can be analyzed individually. The results can be used, for example, for design-time mapping decisions and run-time resource and QoS management. The concept of scenarios is orthogonal to this work. So, our approach is not limited to only applications that can be modeled as SDF graphs. It can handle applications with data-dependent behavior that can be bounded, which is true for many multi-media applications.

5. Prefetching

5.1. Prefetching model

In Sec. 3, we already argued that the (pre)fetching of data from a remote memory as handled by the communication assist (CA) must be taken into account. Therefore, we need to extend the memory access model of Sec. 4 with an SDF model of (pre)fetching functionality of the CAs. Before introducing the model, we first discuss the typical behavior of the local and remote CA for the ith sequence of firings of the actors $A_{1,i}$ and $A_{2,i}$ in the memory access model ($A_{1,i}$ and $A_{2,i}$ in Fig. 4). A firing of actor $A_{1,i}$ produces data to $A_{2,i}$ and it requests data that is needed for firing $A_{2,i+1}$. The local CA sends this request to the remote CA. On its turn, the remote CA will return the requested data. Next, the local CA has to copy the data into the local tile memory. However, it might be that not all requested data can be stored in the local memory when $A_{2,i}$ is firing (and thus occupies part of the local memory). For that reason, the local CA might have to break the request to the remote CA into two steps. First, the local CA requests and receives the data that can be *prefetched* while $A_{2,i}$ is firing. Next, it *fetches* the data that could not be stored in the local memory while $A_{2,i}$ was firing. The SDF model for the behavior of the CAs is shown in Fig. 6. The actors $CA_{l,1}$, $CA_{r,1}$ and $CA_{l,2}$ model the prefetching of data and the actors $CA_{l,3}$, $CA_{r,2}$ and $CA_{l,4}$ model the fetching of data. Note that the remote tile memory actor M_i has been abstracted away. Its (timing) behavior is included in the remote CA. The SDF model can be simplified in the situation that all data can be buffered - i.e. the second access to the remote memory ($CA_{l,3}$, $CA_{r,2}$, $CA_{l,4}$) can be removed. In the situation that the complete data access pattern is known at design time, we can apply a simplification similar to the one discussed at the end of Sec. 4.4. The actor $A_{1,i}$ can be removed and the data access pattern can be directly implemented in $CA_{l,1}$ and $CA_{l,3}$. For the FSBM example, reverting back to the SDF model shown in Fig. 5, we have to do the following to take the CAs into account. Each pair of M_i and F_i should be replaced by the SDF model of Fig. 6 without the $A_{1,i}$ and with the F_i replacing $A_{2,i}$.

5.2. Communication assist

It is interesting to observe that the (pre)fetching model is independent of the prefetching strategy. A designer can choose which strategy to use and use our model to analyze its timing behavior. However, to support re-use, it is desirable that a multi-processor platform supports some typical prefetching strategies via its CAs. In the domain of multi-media systems, we see that applications typically have similar memory access patterns. For example, many video processing applications use a window that moves over a video frame. They only differ in the used window, frame size, or direction in which the window moves over the frame. A parametrized CA can be designed that implements the correct prefetching behavior for all these applications. In the video example, the parameters used to configure the CA are the window and frame size and the direction of movement of the window.

6. Experimental results

Through the FSBM application and an H.263 decoder we show how the models presented in the previous sections can be used to analyze the timing properties of an application.

6.1. Experimental setup

In our experiments, we used the architecture template as described in Sec. 2.1. Each processing tile contains one ARM7TDMI core running at 133MHz. The processor has single-cycle access to the local memory and no cache is available. The memory access model deals with a single actor whose state and input token must be mapped onto a remote memory. The sizes of the local memories are chosen such that in our experiments we always have only one actor that requires the use of a remote memory. This actor is mapped onto a processing tile without other actors. The mapping of the other actors is chosen such that they do not form a bottleneck in the system.
The functionality of the communication assist can be implemented in hardware (speed) or software (flexibility). For the FSBM application, we chose to use a software solution as this is fast enough. In the H.263 decoder, we had to assume a hardware implementation to avoid that the communication assist becomes the bottleneck in the system.
The tiles in our MP-SoC are connected with each other using Æthereal guaranteed throughput channels. The network runs at 100MHz and provides a bandwidth of 100MByte/s. There exist two SDF models for reasoning about the timing properties of such a channel [10, 13]. These can be used to refine the edges between the CA_l and CA_r actors in our model. We have chosen to use the model from [10] as this fits best to our approach. Data which is sent through the channel travels first through the CA at the writing side, then it goes through the network interface, the network and finally through the network interface and CA at the reading side. (When sending a request to the remote memory, the local CA acts as the writing side and the remote CA is the reading side.) A TDMA scheme is used for the CAs and network interfaces to provide a guaranteed throughput and resource sharing [2]. The delay of a single data element (de) traveling through the channel can be upper bounded by the following formula:

$$T_{NW,de} = \lceil (2 \cdot (T_{CA,w} + T_{NI}) + T_L + T_{CA,r})/N \rceil \quad (1)$$

$T_{CA,(w|r)}$ models the fraction of time that a particular task has access to the communication channel through the CA at the writing respectively reading side. T_L represents the time that a packet is traveling through the network from one tile to another. T_{NI} is used to model the bandwidth allocated for this channel in the network. T_{NI} and $T_{CA,w}$ are counted twice - once for the data and once for the control flow token going back. The reading CA is only involved in reading the data and is therefore counted once. The factor N is the number of data elements transfered in one burst.

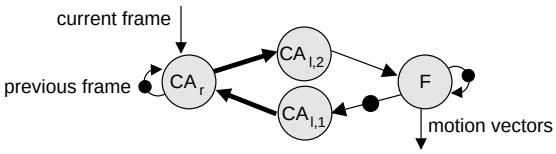


Figure 7. SDF graph for FSBM.

6.2. FSBM: manual analysis

Model. The SDF model for the computation of all motion vectors of a single frame was derived in Sec. 4.4. In Sec. 5.1, we discussed how this model can be refined to include the behavior of the CA. The result is an SDF graph containing many actors (11088). This number can be reduced significantly by using application knowledge. All actors F_i in Fig. 5 have the same functional and timing behavior, and are executed in sequence; they can be modeled with a single actor F that is repeatedly fired via a self-edge. We can apply a similar reasoning for the actors modeling the local and remote CA. The resulting graph is shown in Fig. 7. Note that such a simplification is in general not possible. However, in this case, it is convenient because it allows a manual analysis that can be used to illustrate the potential of our approach. Note that in case these simplifications are not possible, the automated analysis technique presented in the next section is not affected.
Let's compare Fig. 7 to the original FSBM actor shown in Fig. 2. Actor F still has a self-edge, but the size of the token that it carries is now much smaller. The token fits in the local tile memory. The large token present on the self-edge of the original F actor is now put on the self-edge of actor CA_r.
We measured, using an ARM instruction set simulator, an execution time of $6ms$ to compute a single motion vector in the F actor for a block of 8 by 8 pixels and a window of 24 by 24 pixels.

Prefetching. After a firing of actor F, the window moves to a new position in the frame. During the firing of F, this data should be prefetched from the remote memory. The worst situation that can happen is that the window moves down to a new row and the left of the frame. In this case, the complete window must be discarded and a new window must be (pre)fetched. To handle this situation, we should be able to prefetch a complete window. We never have to prefetch more than this as the FSBM actor has no jitter in its execution time that must be accommodated for.

Network interface buffer size. We want to prefetch all data during the firing of actor F. This is possible if the

sequential firing of the actors $CA_{l,1}$, $CA_{r,1}$, $CA_{l,2}$ in our model and the actors in the channel model takes less time than the execution time of F ($T_F = 6ms$). The constraint that must be satisfied is:

$$T_F \geq T_{CA_{l,1}} + T_{NW,req} + T_{CA_{r,1}} + T_{NW,upd} + T_{CA_{l,2}} \quad (2)$$

The execution time of the network is given by the time needed to send a request to the remote CA ($T_{NW,req}$) and the time needed to send the data from the remote CA to the local CA ($T_{NW,upd}$). The request to the memory is only a single data element ($T_{NW,req} = T_{NW,de}$). An update consists of multiple data elements (e.g. a window requires an update of 192 data elements (pixels)). It holds that $T_{NW,upd} = T_{NW,de} \cdot \#(pixels\ in\ update)$. Let's assume that each access of the CA to the network takes $2\mu s$ ($T_{CA,w} = T_{CA,r} = 2\mu s$). The latency of a token in the network is $20\mu s$, and for the local CA requesting an update, $T_{CA_{l,1}}$, it is $100\mu s$. The remote and local CA which send and receive the update, $T_{CA_{r,1}}$ and $T_{CA_{l,2}}$, need $2 \cdot \#(pixels\ in\ update)\mu s$. Using Eqn. 1 and Eqn. 2, we find that the following constraint should be satisfied: $27\mu s \geq (2 \cdot T_{NI} + 26\mu s)/N$. Let's choose a network interface buffer size of $N = 10$ elements. Note that we can choose a smaller N, but this will increase the required communication bandwidth as computed below. T_{NI} should then be at most $122\mu s$.

Communication bandwidth. Using Eqn. 1, we compute that the maximum time available for prefetching the data ($T_{NW,upd}$) is equal to $5ms$. The number of data elements that must be prefetched is at most one window (576 data elements of one byte). The required bandwidth is then equal to $576byte/(5ms \cdot 100Mbyte/s) = 0.11\%$ of the bandwidth available in the channel.

Analysis result. By reserving the calculated buffersize and communication bandwidth, all data can be transfered from the remote to the local memory in time so that the computation of all motion vectors for a frame takes equally long when using a local or remote tile memory.

6.3. FSBM: automatic analysis

The FSBM algorithm and the prefetching mechanism have been implemented in HAPI. HAPI is an MP-NoC simulator based on YAPI [6] that can analyze the timing and simulate functional behavior of an SDF graph when mapped onto our architecture template. It allows for co-simulation between (timed) SDF actors running on the native processor and SDF actors running on cycle accurate simulators. An SDF model for an Æthereal guaranteed throughput channel is also provided. A design-space exploration is performed by repeatedly analyzing the SDF graph for different communication bandwidth allocations, network interface buffer sizes and prefetching sizes. Each analysis is equal to a complete analysis as done in the previous subsection.

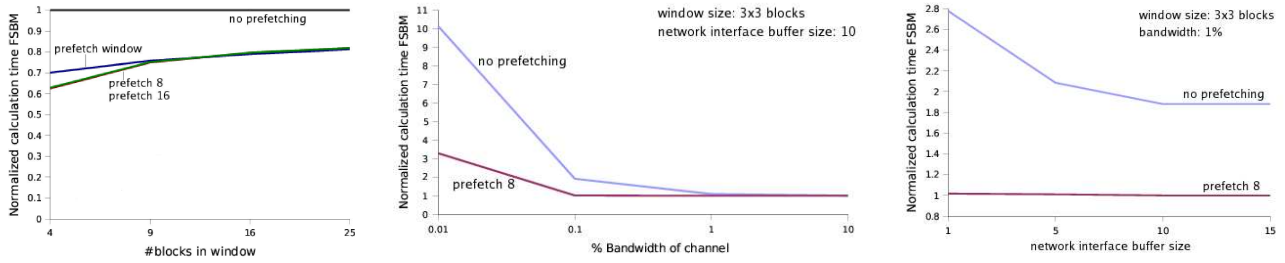
We first analyzed the impact of prefetching data elements from the remote memory for different network interface buffer sizes in the local memory and different sizes of the window. The results, shown in Fig. 8a, have been normalized to the time needed to compute all motion vectors for a frame without prefetching. They clearly indicate that the time needed to calculate all motion vectors improves considerably when prefetching is used. The results show further that prefetching two columns of the window (prefetch 16) or the entire window (prefetch window) does not improve the time required to calculate all motion vectors for a frame compared to prefetching one column (prefetch 8).

We also used the network channel model in conjunction with our SDF models to analyze the bandwidth and network interface buffer size requirements. The results are normalized to the smallest time needed to compute all motion vectors of a frame. Fig. 8b shows that if we prefetch one column, then we need 0.1% of the available bandwidth. Fig. 8c shows a very small difference (2%) in the time needed to calculate all motion vectors for a frame when prefetching one column and using a network buffer size of 1 or 10 elements. With more than 10 elements the time does not decrease. Note that both results match with our analysis made in the previous subsection.

6.4. H.263 decoder

H.263 is a standard video-conferencing codec optimized for low data rates and relatively low motion. The codec was used as a starting point for the development of the MPEG-2 codec which is optimized for higher data rates. Part of the H.263 decoder is a sub-pixel accurate motion compensation block. It uses the motion vectors calculated by a motion estimator (e.g. FSBM) to construct a frame based on already decoded frames (reference frame). In this experiment, we mapped the motion compensator on an ARM7. The other parts of the decoder were mapped in such a way onto other resources that the motion compensation is the bottleneck in achieving maximal throughput.

The motion compensation needs access to a reference frame and outputs a new frame. These frames must be stored in remote memory. The input data is stored in the local memory. To model this situation, we used a BDF model for remote accesses of state and output data (not shown in the paper, but similar to Fig. 3). The motion compensation works on a block-by-block basis; the prefetching is limited to only a



(a) **(b)** **(c)**

Figure 8. Simulation result.

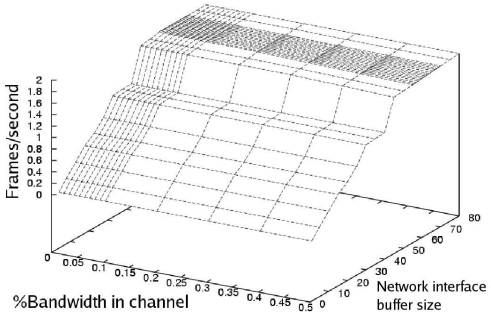


Figure 9. H.263 design space.

single block as not more is needed. We performed a design-space exploration for the network interface buffer size and allocated communication bandwidth. The result of the exploration is shown in Fig. 9. The experiment shows that the number of frames per second is not influenced by the allocated bandwidth (this is always sufficient), but the network interface buffers need to be large (64 elements) to obtain the maximal throughput.

7. Conclusions

In this paper, we have presented an approach to deal with large data structures in on-chip multiprocessors while guaranteeing performance. We use a generic tile-based multiprocessor architecture with tiles containing large memories that are shared by different subsystems and applications. We developed a number of SDF models that allow reasoning about the timing aspects of using memory tiles in the system and to reason about buffer sizes and communication bandwidth requirements. An integrated prefetching mechanism hides the latency introduced by using memory tiles. The analytical properties and the usefulness of the prefetching have been demonstrated in two case studies.

References

[1] B. Ackland, et al. A single chip 1.6 billion 16-b mac/s multiprocessor dsp. *IEEE Journal of Solid-State Circuits*, 35:412–424, 2000.

[2] M. Bekooij, et al. Predictable multiprocessor system design. In *SCOPES'04*, p. 77–91. Springer, 2004.

[3] J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California, Berkeley, CA, 1993.

[4] D. Culler, et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[5] G. de Haan, et al. True motion estimation with 3-d recursive search block-matching. *IEEE Transactions on CSVT*, 3(5):368–388, 1993.

[6] E. de Kock, et al. YAPI: Application modeling for signal processing systems. In *DAC'00*, p. 402–405. ACM, 2000.

[7] R. Govindarajan, et al. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *VLSI Signal Processing*, 31:207–229, 2002.

[8] E. Lee, et al. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[9] M. Millberg, et al. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE'04*, p. 890–895. IEEE, 2004.

[10] A. Moonen, et al. Timing analysis model for network based multiprocessor systems. In *Progress'04, Workshop, Proc.*, p. 122–130. STW, October 2004.

[11] P. Paulin, et al. Application of a multi-processor SoC platform to high-speed packet forwarding. In *DATE'04*, p. 58–63. IEEE, 2004.

[12] P. Poplavko, et al. Mapping of an mpeg-4 shape-texture decoder onto an on-chip multiprocessor. In *PRORISC'03*, p. 140–147, 2003.

[13] P. Poplavko, et al. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES'03*, p. 63–72. ACM, 2003.

[14] A. Radulescu, et al. An effi cient on-chip network interface offering guaranteed services shared-memory abstraction, and flexible network confi guration. In *DATE'04*, p. 878–883. IEEE, 2004.

[15] M. Rutten, et al. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, 2002.

[16] S. Stuijk, et al. Predictable embedding of large data structures in multiprocessor networks-on-chip (extended abstract). In *DATE'05*, p. 254–255. IEEE, 2005.

[17] P. Yang, et al. Managing dynamic concurrent tasks. In *ISSS'02*, p. 112–119. ACM, 2002.