View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Demonstration of Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSoC)

Philip K.F. Hölzenspies, Jan Kuper, Gerard J.M. Smit, Johann Hurink
University of Twente
Department of Electrical Engineering, Mathematics and Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
p.k.f.holzenspies@utwente.nl

Abstract

In this paper, the problem of spatial mapping *is defined. Reasons are presented to show why performing spatial mappings at run-time is both necessary and desirable and criteria for the qualitative comparisson of spatial mappings are introduced. An algorithm is described that implements a preliminary spatial mapper. The methods used in the algorithm are demonstrated with an illustrative example.*

1 Introduction

Academia and industry alike recognize the trend towards parallelism in computation. Although many techniques exist for the analysis of (data and temporal) dependencies between parallel processes, programming models for multi-processor architectures are subject of current research. This paper deals with models for streaming applications on heterogeneous multi-processor systems, with a special focus on MPSoC cases, where energy efficiency is a key requirement.

The remainder of this section introduces the concepts relevant tot this paper. Section 2 describes the contributions made by this paper. Section 3 describes a formal model of spatial mapping, including quality criteria, followed by a description of an algorithm implementing this formal model in Section 4. To provide some intuation, a full case example is given in Section 5, before conclusions are drawn in Section 6.

1.1 Streaming DSP applications

Streaming DSP algorithms are implemented and used in portable and otherwise energy constrained embedded systems and require an energy-efficient processing architecture. Typical examples are found in signal processing for wireless baseband processing (for wireless LAN, digital radio, UMTS[11]), multi-media processing, medical image processing and sensor processing. Streaming DSP applications can be modelled as task graphs with streams of data items (the edges) flowing between computation kernels (the nodes)[4].

Analyzing the common characteristics of these applications, we can observe that they:

- require relatively simple processing on huge amounts of data.
- display a high degree of regularity in the communication between tasks.
- have data flowing through the tasks in a pipelined fashion, thus allowing tasks to be executed in parallel, either on different processors or in a time-multiplexed fashion.
- often require real-time throughput and latency guarantees for both communication and computation.
- have a semi-static life-time, i.e. typically in the order of minutes, rather than milliseconds.

- display a high degree of periodicity (possibly dependent on data arrival times)

From these observations, we can conclude that these applications have a predictable behaviour, both temporally and spatially.

On a functional level, a streaming application can be described as a Kahn Process Network (KPN), because only the functional decomposition of an application and the data dependencies between the components is specified. Concrete implementations of these components can be specified with much more detailed information, i.e. Worst-Case Execution Time (WCET) and granularity of consumption and production of data (e.g. an implementation of a function on video frames may read the entire frame before executing, but it may also work only on the first few lines). These added details can be described in terms of Synchronous Data Flow [9] (SDF) graphs. Here the endings of the edges are labelled with consumption and production rates in terms of tokens. Also, annotations are added to the edges to signify the number of tokens that "currently reside on the edge" (i.e. are produced and not yet consumed). Finally, nodes are labelled with the WCET of a single execution.

When the difference between the time spent reading and writing large tokens and the time spent executing becomes large, a more fine grained specification by means of Cyclo-Static Data Flow [2] (CSDF) graphs will allow for more exact analysis[15]. In CSDF, the labels from the SDF graph are split into the different phases of an execution of the implementation.

1.2 Tiled architectures

Although multi-processor systems are not a new concept, the MPSoC concept is on the rise. Recently, considerable numbers of MPSoC designs have been proposed and built (e.g. [7, 14, 3]) and design templates have been developed (e.g.[1, 6, 12]).

What is referred to as a *tiled architecture* in this paper, is a chip made up out of multiple autonomous processing elements. Autonomicity means that tasks can be started and stoped on a processor without directly effecting (independent tasks on) other processors. In other words, the guaranteed resource bounds of other tasks are not threatened. For these processors to form one architecture, they must be interconnected. The combination of a processor and its interface to the architecture's interconnect is referred to as a *tile*. Autonomicity of the interconnect requires that guarantees can be provided with respect to throughput and latency[8]. The remainder of this paper assumes that the tiles on the chip are interconnected by a Network-on-Chip (NoC)[5].

1.3 Run-time spatial mapping

Generally, spatial mapping is the allocation of spatial resources for applications. In the context of tiled architectures, spatial resources are tiles and—in the case of a NoC—routers. Thus, spatial mapping is the assignment of tasks from the KPN describing the streaming application to tiles and channels to paths through the NoC. A *feasible* spatial mapping satisfies the mapped application's Quality of Service (QoS) (QoS) requirements. A spatial mapping's *quality* depends on the extend to which it minimizes cost (in our case: energy consumption) under the resource constraints. The objective of run-time spatial mapping is to find a feasible spatial mapping with the best quality (in our case: the lowest energy cost).

To be able to utilize heterogeneous multi-processor systems, tasks that are used often should be implemented for different processor types. For example, for a frequently used DSP kernel such as an FFT there are implementations for an embedded ARM +processor and for a reconfigurable core. Thereby, a flexibility is introduced that allows a task to be executed, even if there is no processor available of the preferred type.

1.3.1 Necessity and advantages

Performing the spatial mapping at run-time is arguably both necessary and desirable. Preliminary experiments[13] are promising with respect to the feasibility of run-time spatial mapping. Performing the spatial mapping at run-time is necessary, whenever the application set is not known completely at design-time, e.g. when the platform allows the user to use software from any vendor, developed for that platform.

Availability of resources depends on the set of applications running simultaneously. Also, variations in QoS requirements due to changes in the environment effect an application's resource demands. Because of these dependencies, all possible combinations of applications

need to be known at design-time, in order to do design-time spatial mapping with energy-efficiency as an optimization objective, which is—even in small systems—impossible.

Performing the spatial mapping at run-time offers very desirable flexibility. Unforeseeable changes in applications can be taken into account. Moreover, defective tiles can be avoided, which both increases yields and makes a system more robust against aging.

1.3.2 Goals and requirements

In our context, the objective of the spatial mapping is to minimize the energy consumption of the entire application: processing (including memory requirements thereof) as well as interprocess communication. In principle, the spatial mapping is performed only when a new streaming application is started.

To be able to perform the mapping of an application to tiles, a spatial mapping algorithm needs models of the application to be mapped and the (MPSoC) platform to be mapped to. Furthermore, the constraints of the application (e.g. throughput requirements and latency bounds) need to be known, as well as the resource requirements of the available implementations (e.g. time, memory, etc.)

When performing the spatial mapping at run-time, some figures can only be determined at run-time. Inter-process communication parameters (e.g. estimated latency, energy consumption), for example, need to be determined at run-time as these are dependent on the specific mapping. Likewise, it is only known at run-time on which tile a process will be executed and which processes are already running on this tile, so the actual response time of a process is only known at run-time. However, the choice made at run-time is from a finite set of implementations, all of which have properties that are determined at design-time.

The constraints of the application can only be checked after it has been mapped. Thus, when a spatial mapping has been determined and latencies and throughputs of processes running on processors are known, the constraints can be checked. We use a data flow analysis for this check, that is beyond the scope of this paper. Instead, we reference [15].

Finally, a spatial mapping is considered *feasible* if it is adherent and all the application's constraints are met.

2 Contribution

Run-time spatial mapping is a very young research topic. As such, opinions vary on what it does and does not comprise. Current practice is to perform both the spatial and temporal mapping of applications to multi-processor architectures simultaniously at design-time. Even at design-time, exhaustive search for optimal mappings is not always possible. Thus, heuristics are often used to perform this design-time mapping.

Run-time mapping poses much tighter time constraints on the search process. Therefore, better-tailored heuristics are required. The separation of spatial and temporal mapping is one such heuristic. This separation is made clear in this paper by means of a formal definition of spatial mapping (see Section 3), a description of the algorithm used (see Section 4) and an illustrative example (see Section 5).

3 A formal definition of spatial mapping

3.1 Hardware platform

In this section we will formally describe *tiled multi-processor systems*. In such a system, a *tile* may for example consist of a processor, some memory, and a router. Tiles can be connected to each other through *links*. A link between two tiles enters those tiles through the routers, such that communication with a processor is only possible through the router on the same tile. On the other hand, in order to let data, sent from one tile to another, pass through intermediate tiles, only the routers on these tiles need to be involved and the processors need not.

Furthermore, a tiled multi-processor system has a certain capacity to run software applications.

Thus, a *tiled multi-processor system* is a graph $\mathfrak{T} = \langle \mathcal{T}, \mathcal{E} \rangle$ where $\mathcal{T}$ is a set of *tiles* (nodes) and $\mathcal{E} \subseteq \mathcal{T} \times \mathcal{T}$ is a set of *links* (edges) between tiles.

Edges in a graph $\mathfrak{T}$ are unlabelled, i.e., an edge is completely determined by its source and destination. We will assume that a graph $\mathfrak{T}$ is directed. Furthermore, $\mathfrak{T}$ is *connected*, i.e., there is a path, possibly consisting of several edges, between each pair of tiles.

A tiled multi-processor system typically contains processors of various types, say there can be ARM-processors in a tiled system, FPGAs, DSPs, etc.

The main issue in this paper is the *mapping* of the processes of a software application on a system of processors. Since these processors need not be directly linked to each other, communication channels between processes are mapped onto *paths* between tiles. Therefore we consider a "higher order graph," in which the edges *are* the paths in $\mathfrak{T}$.

Thus, let $\mathcal{E}^*$ be the set of all cycle-free paths over a tiled system $\mathfrak{T} = \langle \mathcal{T}, \mathcal{E} \rangle$, then the graph $\mathfrak{T}^* = \langle \mathcal{T}, \mathcal{E}^* \rangle$ is called the *pathed* tiled system over $\mathfrak{T}$. A (possibly empty) path from t_0 to t_n will be denoted by $\langle t_0, t_1, \ldots, t_n \rangle$ (with $\langle t_i, t_{i+1} \rangle \in \mathcal{E}$), and may be considered as a label of an edge in $\mathfrak{T}^*$ from t_0 to t_n. The *length* of a path is the number of steps in it, i.e., the length of the path mentioned is n. Clearly, a pathed graph is a directed multi-graph.

3.1.1 Capacities

Each tile in $\mathfrak{T}$ has capacities. One can think of computational and memory capacites, but also of the maximum number of processes that can be assigned to it; e.g. ASICs can not switch between processes, so they have a maximum of one process assigned to it, while an ARM may be able to serve as many processes as there are slots in its TDMA scheduler.

Capacities concerning communication between processors such as bandwidth, are also supposed to be expressed as capacities of the *tiles*. For example, the bandwidth of a link between tiles can be expressed as the capacity of the outgoing port(s) on a tile connected to that link.

Thus, we consider *all* relevant (local) capacities of a tiled system as being expressed as capacities of tiles. That is to say, all capacities of a tile t are expressed simultaneously by its *capacity vector* $C(t)$. The 'shape' of every $C(t)$ is the same for all t, i.e. the capacity of the processor on every tile has the same dimension. These dimensions are *orthogonal*, i.e. the corresponding capacities are independent.

Given the above, it is possible to derive capacities of a path in $\mathfrak{T}^*$. For example, the bandwidth of a path is the minimum bandwidth of the hops in the path.

3.2 Software applications

An *application* (*task*) is a directed graph $\mathfrak{P} = \langle \mathcal{P}, \mathcal{F} \rangle$ where $\mathcal{P}$ is a set of *processes* and $\mathcal{F} \subseteq \mathcal{P} \times \mathcal{P}$ a set of *channels* between processes along which processes communicate with each other.

For all processes in an application, implementations have to exist such that a process can be executed on a processor. For one process several implementations may exist, though not necessarily for all available types of processors. The set of implementations for process p is $\mathcal{I}(p)$. The subset $\mathcal{I}_\tau(p) \subseteq \mathcal{I}(p)$ denotes the set of implementations of p that can be executed on processors of type τ.

3.2.1 Requirements

Any implementation poses requirements on the processor it is executed on. Examples of such requirements are the computational and memory loads. For a given implemenation i, its requirements are expressed simultaneously by its *requirement vector* $R_\pi(i)$. The subscript π indicates that in this case the requirements are posed by (an implementation of) a process. Below, requirements of channels will be dealt with.

The dimensions of a requirement vector are the same, and in the same order, as with capacity vectors. Hence, implementation i of process p can be executed on tile t of type τ if $i \in \mathcal{I}_\tau(p)$ and $C(t) - R_\pi(i) \geq 0$.

Likewise, the communication between processes along channels poses requirements on routers and links in a tiled system. Here too, the requirements of communication channels will be expressed as vectors of the same form as the capacity vectors. Note that such requirements vectors will contain zeroes on positions where they are not relevant, such as with memory requirements on the tiles.

Note that the communication through channels does not depend on the selected implementations for individual processes, but on the specification of the application as a whole. Requirements following from communiciation along a channel c will be expressed as $R_\gamma(c)$.

In order to determine whether there is sufficient capacity on a tile t for information that has to flow through this tile, it is important in which direction this information will flow through the tile. We will come back to that point below.

3.3 Spatial mapping

Software applications have to run on a tiled system, i.e. tiles have to be associated to processes, and paths in the tiled system have to to be associated to channels. Clearly, this has to be done in such a way that the necessary implementations exist, and the capacity of the tiled system is not exceeded.

A *task assignment function* α is a function which maps a software task $\mathfrak{P}$ to a pathed tiled system $\mathfrak{T}^*$. More precisely, for every process $p \in \mathcal{P}$ we have that $\alpha(p)$ is a tile in $\mathcal{T}$, and for each channel $\langle p, q \rangle \in \mathcal{F}$ we have that $\alpha\langle p, q \rangle$ is an edge from $\alpha(p)$ to $\alpha(q)$ in $\mathcal{E}^*$. Thus, $\alpha\langle p, q \rangle$ is a path from $\alpha(p)$ to $\alpha(q)$ in $\mathfrak{T}$. If needed, we will distinguish between α_π and α_γ, where α_π denotes that part of α that deals with processes, and α_γ deals with channels.

Let I be a function which selects an implementation for a process p. Then a *spatial mapping* $\mathfrak{m}$ is a pair $\langle \alpha, I \rangle$ of a task assignment function α and an implementation selector I.

Suppose that for a given process p we have that $\alpha(p) = t$, where t is of type τ. Then an implementation of p for a processor of type τ should exist. A spatial mapping is considered *adequate* if every process is mapped to a tile type for which an implementation is available. Formally, we call $\mathfrak{m} = \langle \alpha, I \rangle$ *adequate* if for every process p such that the type of $\alpha(p)$ is τ, we have that $\mathcal{I}_\tau(p)$ is non-empty, and $I(p) \in \mathcal{I}_\tau(p)$.

3.3.1 Computational load

Next, we discuss the computational load of a spatial mapping $\mathfrak{m} = \langle \alpha, I \rangle$ on a tile t. First, we discuss the load as resulting from mapping processors on tiles, later we come to the load resulting from cummunication.

Define the inverse of the task assignment function α concerning tiles as follows:

$$\alpha_\pi^{-1}(t) = \{\, p \in \mathcal{P} \mid \alpha(p) = t \,\}$$

Thus, $\alpha_\pi^{-1}(t)$ is the set of processes that is assigned by α to tile t.

The *computational load* $\mathcal{L}_\pi^{\mathfrak{m}}(t)$ of a spatial mapping $\mathfrak{m} = \langle \alpha, I \rangle$ on the processor of a tile t is given by

$$\mathcal{L}_\pi^{\mathfrak{m}}(t) \;=\; \sum_{p \in \alpha_\pi^{-1}(t)} R_\pi(I(p))$$

where $R_\pi(I(p))$ is the requirement vector of the implementation $I(p)$ of process p. Thus, the load of a tile on which several processes may run, is a vector of the same structure as the capacity vector of a tile, and also as the requirement vectors of the implementations of the individual processes.

Next, we turn to the load caused by communication along channels. First we define the corresponding inverses of a task assignment function α:

$$\alpha_\gamma^{-1}(t) = \{\, c \in \mathcal{F} \mid t \in \alpha(c) \,\}$$

Thus, $\alpha_\gamma^{-1}(t)$ yields the set of those channels in the application that are mapped on paths in $\mathfrak{T}$ that pass through the router of tile t.

In order to calculate the load on a tile t caused by the communication through the router of that tile t, we need to know from which channel this communication comes and in what direction it goes. To define the direction of the communication through tile t, caused by channel c, suppose

$$\alpha(c) = \langle t_0, \ldots, t, \ldots, t_n \rangle,$$

i.e., t is one of the tiles in the path associated to c by α.

We denote the immediate predecessor of t in $\alpha(c)$ by t_α^-, and the immediate successor of t by t_α^+. In case $t = t_0$ or $t = t_n$, we choose $t_\alpha^- = t_0$ and $t_\alpha^+ = t_n$, respectively.

Let $\varphi_{(t_\alpha^-, t, t_\alpha^+)}$ be a function that determines which components of the load vector of tile t are changed when information flows through tile t in the direction coming from tile t_α^- and going to tile t_α^+. Thus, if the requirement vector of a channel c in an application is $R_\gamma(c)$, then the load on a tile t resulting from the communication through t according to c, now can be expressed as

$$\varphi_{(t_\alpha^-, t, t_\alpha^+)}\left(R_\gamma(c)\right).$$

The total load on tile t caused by all communication in all channels that are mapped on a path through t, now is

$$\mathcal{L}_\gamma^{\mathfrak{m}}(t) = \sum_{c \in \alpha_\gamma^{-1}(t)} \varphi_{(t_\alpha^-, t, t_\alpha^+)}\left(R_\gamma(c)\right)$$

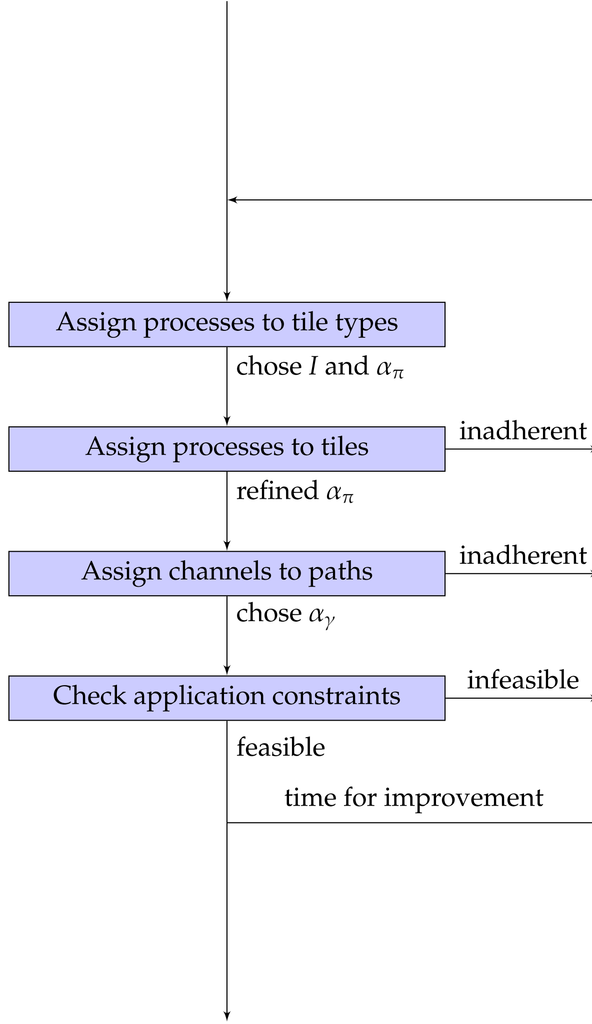

Figure 1: Hierarchical search with iterative refinement

The total load on tile t, covering both the load caused by processes and the load caused by channels, now is

$$\mathcal{L}^{\mathfrak{m}}(t) = \mathcal{L}^{\mathfrak{m}}_{\pi}(t) + \mathcal{L}^{\mathfrak{m}}_{\gamma}(t).$$

When a spatial mapping is adequate and no tiles are overloaded (the resources required by all the implementations mapped to a tile do not exceed the resources offered by the tile), it is considered *adherent*. In other words, we call $\mathfrak{m}$ *adherent* if $\mathfrak{m}$ is adequate, and if for all t we have that

$$C(t) - \mathcal{L}^{\mathfrak{m}}(t) \geq 0$$

4 Algorithm

Even when only considering the assignment of processes to a heterogenous multi-processor platform, we find a Generalized Assignment Problem [10] (GAP), which is NP-complete. Considering the prohibitve complexity of exhaustive search, we propose an application domain aware heuristic: *hierarchical search with iterative refinement*. We divide the search process in steps, starting with a very coarse grained perspective in the first step and gradually adding more detail. At each step decisions are made that shrink the search space in the next. Decisions made in previous steps are considered fixed in later steps.

As is to be expected of heuristics, this abstraction carries with it the danger that decisions made in early steps, using very high level abstract information, lead to search spaces in

later steps that contain no feasible solutions. Since this infeasibility only comes to light in later steps, we propose a strategy for iterative refinement. Figure 1 shows the hierarchical decomposition into steps used in our run-time spatial mapping tool for heterogenous MPSoCs. We will now describe each of these steps in more detail.

1. **Assign implementations to processes.** The goal of the first step is to choose an implementation (and thereby tile type) for every process, i.e. to choose I in $\mathfrak{m} = \langle \alpha, I \rangle$. By choosing I prior to α_π, this step implies a contract for α_π, i.e. inadequacy can be prevented later on by limiting the choice of $\alpha_\pi(p)$ to tiles of type τ, where $I(p) \in \mathcal{I}_\tau$.

   To prevent running into inadherence directly after this step, we only consider those implementations for which an adhering mapping exists, i.e. that fit on at least one tile in the system. Thus, we only consider $I(p) = i$ when there is at least one tile t of type τ, where $i \in \mathcal{I}_\tau$ and $C_t - R_i \geq 0$.

   We go about this choice iteratively. The choice of the next process to pick an implementation for is based on its *desirability*. The desirability of a process is the difference between the cheapest assignment and the second cheapest assignment of the process to a tile. In other words, if the alternative is more expensive, the desirability to map the process now increases.

   If a process has been chosen to be assigned next, not only do we choose this process' implementation, we also map it to the first tile we come across with sufficient resources (i.e. a first-fit packing). This guarantees that after this step (if this step manages to map all processes), at least one α_π exists that does not break the adherence of $\mathfrak{m}$, although $\mathfrak{m}$ might still be inadherent when no α_γ exisits.

2. **Assign processes to tiles.** From step one, we now have a chosen implementation for every process. The (greedy) assignment of processes to specific tiles α_π obtained in the previous step is now improved upon by taking more detail into account. Again, we iteratively choose what to do based on desirability. In perticular, for every iteration we try, for every implementation, to remove it from the tile it is mapped onto and, by local search, to map it onto the best available tile of the required type. The difference in cost between the original mapping and the best tile found in the local search is, again, the measure of desirability for choosing this reassignment. Only the best reassignment is actually performed every iteration. Because a process can only be reassigned to a tile with the same type as the tile it is already assigned to, this step maintains adequacy.

   The previous step simply iterated until all processes where assigned to a tile (assigning one process each iteration). Deciding when to stop at this step can be based upon a minimum gain from iteration (once an iteration improves the total solution by a lesser amount than a chosen threshold, we decide to stop) and/or by a maximum number of iterations. Besides cost factors based solely on the mapping of a process to a tile, an assignment should be awarded a bonus for proximity to the process' neighbours in the application graph. This stimulates locality, causing the communication routes, assigned in the next step, to likely be short. Moreover, we again prevent immediate inadherence in the next step, by only considering tiles for a process that have sufficient communication resources to facilitate the processes communication requirements, at least, locally.

3. **Assign channels to paths.** For the concrete realization of step three, the channels are sorted by non-increasing throughput. Next, iteratively for each channel, a corresponding path is determined, taking into account the loads resulting from the previously mapped channels.

   The sorting is done to increase the probability that a heavy demanding channel gets assigned a better path. In each iteration for a given channel, a shortest path between the source and destination tile of the channel has to be determined, where only such tiles are taken into account which still have enough capacity for the throughput requirement of the current channel. Thus, an α_γ is constructed iteratively, never overpacking communication capacities of a tile.

   Adding α_γ to the α_π and I from the first two steps, the result of this step is an *adherent* spatial mapping $\mathfrak{m} = \langle \alpha, I \rangle$ where $\alpha = \langle \alpha_\pi, \alpha_\gamma \rangle$.

4. **Check application constraints.** The last step checks the global application constraints. When any such constraint is violated, the $\mathfrak{m}$ is *infeasible* and feedback should be given to higher steps to try and improve upon those characteristics of the mapping

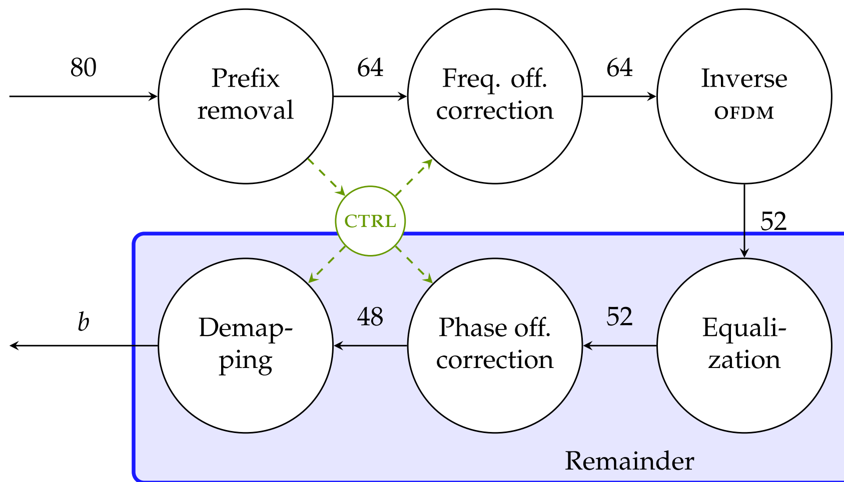

Figure 2: Decomposition of a HiperLAN/2 receiver

that violate the constraint(s). Should no constraint be violated, $\mathfrak{m}$ is *feasible*. When we decide we have time to look for improvements of this solution, possible points of improvement should also be identified here and fed back into the first step (keeping the current mapping in mind, should the feedback only result in infeasible results and feasible mappings that are further removed from the optimum).

In general, the production of feedback immediately triggers a new iteration, to prevent that multiple changes influence the mapping process. In other words, if any step fails to find a satisfactory result, it will immediately generate feedback so that 'higher' steps may generate a more suitable result.

It is important to realize that this proposed iterative hierarchical approach differs significantly from simple local search methods and global-local search methods that are often used in heuristics. The feedback from a lower level may result in a completely different mapping on a higher level in a next iteration.

5 Case: HiperLAN/2

In order to illustrate the above with an example, an implementation and mapping of a HiperLAN/2 receiver is described in this section.

5.1 Application Level Specification

The receiver's decomposition into communicating processes is shown in the KPN in Figure 2. The control part of the receiver application is included for completeness, but it is not part of the data stream. Orthogonal Frequency Division Multiplexing (OFDM) applications are based on (MAC) frames of OFDM-symbols, which, in turn, consist of samples (complex numbers). In the HiperLAN/2 case, frames consist of 500 symbols and every symbol consists of 80 32-bit complex numbers. The control part only comes into play briefly at the beginning of each frame, while all other processes in the KPN operate on every OFDM symbol.

The last three processes have been grouped to form one process. Not only do they fit well together in a single implementation, but treating them separately needlessly lengthens this example. The numbers shown on the edges of this KPN indicate the number of 32-bit complex numbers per symbol coming in at each process. The size of the output of the HiperLAN/2 receiver (b), depends on what 'mode' the receiver is in. The standard defines seven modes, that only differ with regards to the demapping (hence the input from the control process, which selects the demapping type). Depending on the chosen demapping type, the output can be 2 bits (Binary Phase-Shift Keying—BPSK), 4 bits (Quadriphase-Shift Keying—QPSK), 16 bits (Quadrature-Amplitude Modulation-16—QAM16) or 64 bits (QAM64) per sample. Thus the minimum output is 12 bytes and the maximum is 384 bytes (per OFDM symbol). One OFDM symbol is fed into the application once every $4\mu s$.

This is the Application Level Specification (ALS) of the OFDM receiver. It serves as a contract between the application and the implementations of processes. A lot of freedom is left for the implementations, because a lot of behaviour has *not* been specified in the ALS.

Table 1: Available implementations

Process	PE type	Phases[a] Input [token]	Output [token]	Execution Time [clockcycles]	Avg. energy [nJ/symbol]
Prefix removal	ARM$	$\langle 8^2, \langle 8,0\rangle^8 \rangle$	$\langle 0^2, \langle 0,8\rangle^8 \rangle$	$\langle 18^{18} \rangle$	60
	Montium	$\langle 1^{80}, 0 \rangle$	$\langle 0^{17}, 1^{64} \rangle$	$\langle 1^{81} \rangle$	32
Freq. off. correction	ARM$	$\langle 8,0,0 \rangle$	$\langle 0,0,8 \rangle$	$\langle 18,32,18 \rangle$	62
	Montium	$\langle 1^{64}, 0^2 \rangle$	$\langle 0^2, 1^{64} \rangle$	$\langle 1^{66} \rangle$	33
Inverse OFDM	ARM$	$\langle 64,0,0 \rangle$	$\langle 0,0,64 \rangle$	$\langle 66,4250,54 \rangle$	275
	Montium	$\langle 1^{64}, 0^{53} \rangle$	$\langle 0^{65}, 1^{52} \rangle$	$\langle 1^{64}, 170, 1^{52} \rangle$	143
Remainder	ARM$	$\langle 52,0,b \rangle$	$\langle 0,0,b \rangle$	$\langle 54,2250,b+2 \rangle$	140
	Montium	$\langle 1^{52}, 0, 0$	$\langle 0,0,1^b \rangle$	$\langle 1^{52}, 73-b, 1^b \rangle$	76

[a] We will use the notation $\langle x^n, y^m \rangle$ to denote $n+m$ phases, where the value for the first n phases is x and for the last m phases is y.

For example, the fact that an OFDM symbol is fed into the application every $4\mu s$ does not imply there is one burst of size b at the output every $4\mu s$. The output may be a continuous stream, or it may still be bursty, but its periodicity may be less strict (i.e. output may show jitter). Similarly, processes are semantically defined on OFDM symbols (in various states of abridgement), but concrete implementations may very well work on a per sample basis, or, adversely, on a group of symbols.

5.2 Implementations

Given a set of implementations of the processes in Figure 2, the spatial mapping algorithm can now choose implementations, map them to concrete processors, route the communication channels through the interconnect and construct a CSDF graph of the entire receiver. The description of any implementation should include a CSDF graph, describing its behaviour correctly and in as much detail as is relevant and possible. As stated above, many processes can be described as a single CSDF actor. Table 1 lists implementations of the processes in Figure 2. The phases described in the table are the phases of the CSDF actors corresponding to the implementations. In this table the notation for the inputs $\langle 0, a, b^{10} \rangle$ means: in phase 1, 0 tokens are read, in phase 2, a tokens are read, and in phase 3 through 12, b tokens are read at every phase (the superscript is a shorthand for the number of phases with equal parameters). For example: the inverse OFDM on the ARM has 3 phases; in phase 1, 64 tokens are read, 0 tokens are written and the WCET is 66 clock cycles; in phase 2, 0 tokens are read, 0 tokens are written and the WCET is 4250 cc; in phase 3, 0 tokens are read, 64 tokens are written and the WCET is 54 cc.

Control-flow has been omitted from this table, but *will* be taken into account in the verification process. Also, only ARMs using cache have been described here. It will be shown later that ARMs with Communication Assists (CAs) will have behaviour that requires multiple actors.

Note that the input and output token counts are in terms of symbols (i.e. one token corresponds to one complex number). In an actual description, the token size should be no bigger than the smallest word-size of the hardware, to avoid having to translate between a process emitting a 'complex number' and a piece of hardware taking in a 'byte'. Also, the execution times given in the table are in terms of clock cycles. Therefore, execution times have to be normalized by taking into account the clock frequency of the processor assigned to the implementation and, where applicable, scheduler settings need to be taken into account to translate *execution* times to *response* times.

5.3 Hardware

A few notes are required on the hardware of the test environment. The HiperLAN/2 receiver is mapped to a MPSoC, consisting of ARMs with cache, ARMs with CAs and their own Scratch Pad Memory (SPM) and Montiums.

Caches allow for burst reads, based on the locality principle. In the case of streaming applications, there is generally very little locality. However, reading ahead in the input

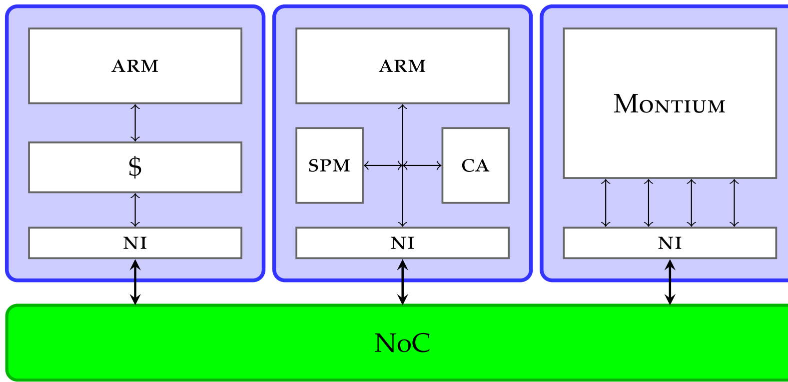

Figure 3: MPSoC architecture

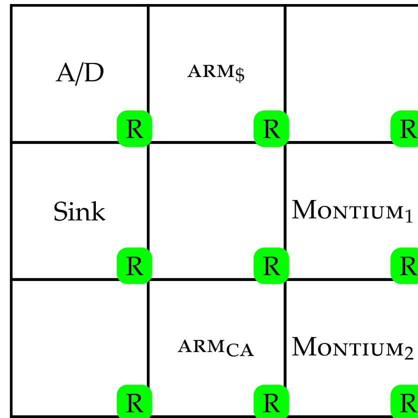

Figure 4: MPSoC layout

stream and caching that data does increase the speed of read operations. Since caches are non-deterministic, it is very hard to guarantee any behaviour in the general case, but when processes can be exclusively assigned a cached processor, the behaviour may very well become fully deterministic (locking of caches). In this text, caches will be assumed to incidentally cause lucky speed-ups, but execution times will be subject to worst-case assumptions.

A ca is essentially a local memory manager that can autonomously stream data into or out of the spm (the latter being a dual-port memory), i.e. independent of the arm on the same tile. Predominantly, cas decouple communication from computation, at least from the processor's perspective. The required number of tokens for the next firing is known to the ca, so it can gather that number of tokens before signalling the processor that the input data is ready. With regards to the final csdf model of an implementation, a ca allows communication to occur in parallel with computation, thus the ca should be modelled with a separate actor.

The interconnect consists of a NoC with routers that provide guarantees with regards to provided throughput and maximum latency. All tiles have their own Network Interface (ni) to connect to the NoC, but the NoC side of that interface is the same for every tile. The routers in the NoC have buffered inputs and round-robin arbitration on the output, which imposes a maximum latency of 4 clock cycles. For brevity, a homogeneous NoC is considered here, implying that every step in a communication path has the same behaviour and, thus, the same description in the csdf graph.

The architecture of the MPSoC is summarized in Figure 3. It shows one instance of each type of tile and the interconnecting NoC. The hypothetical MPSoC used for this example has two Montiums and two arms. Of the latter, one has cache and the other is communication assisted.

Figure 4 shows a possible MPSoC layout with these specifications. The tiles without labels in this figure are tiles of types not relevant to this example. The tile labeled 'A/D' is the source of all the incoming data. The tile labeled 'Sink' is the tile that has to receive the stream flowing out of the HiperLAN/2 receiver.

Step	ARM		MONTIUM		Cost	Remark
	CA	$	1	2		
0	Pfx.rem.	Frq.off.	Inv.OFDM	Rem.	11	Initial (greedy) assignment
1	**Frq.off.**	**Pfx.rem.**	Inv.OFDM	Rem.	11	No improvement, revert
2	Pfx.rem.	Frq.off.	**Rem.**	**Inv.OFDM**	9	Improvement, keep
3	**Frq.off.**	**Pfx.rem.**	Rem.	Inv.OFDM	7	Improvement, keep
						No further choices

Table 2: Processor assignment iterations

5.4 Mapping

The first step of the mapping process is to choose what implementation to use for which process. This choice is iterative, i.e. an implementation is chosen for one process before choosing an implementation for the next. The measure by which to determine the order by which to choose implementations is *desirability*. This is defined as the difference between the cost of the best (i.e. cheapest) implementation and that of the next-best.

In this example, the 'Inverse OFDM' process is the most desirable. Thus, it is assigned to its preferred processor type, being a MONTIUM. Likewise, the 'Remainder' process is assigned a MONTIUM. At this point, both MONTIUMS are occupied and thus, the available implementations for the MONTIUM architecture can no longer be assigned a processor. This means that all these implementations are ignored from here on. As such, both remaining processes only have ARM implementations and are thus chosen per default.

In the second step, the implementations chosen in the first need to be assigned to specific processors. This step uses heuristics to look ahead towards communication, but does not have exact knowledge of the status of the complete NoC. Since the first step already constructed a greedy assignment to processors, pairs of assigned implementations can now be swapped to find improvements. Improvements arise from having to communicate less (probably, since exact routing is not known here) and from being able to turn off tiles.

Given that multi-tasking processors are not considered in this example, minimizing the *number* of processors in use (so processors not used can be switched off) does not improve a mapping. As a look ahead heuristic, the Manhattan distance is used to estimate how much a channel's communication will cost. The total cost of assigning an implementation to a processor is the sum of the Manhattan distances of all the implementation's incoming and outgoing channels.

Table 2 shows the iterations that lead to the final implementation to processor assignment. Swaps can, of course, only occur between processors of the same type. The sum of all Manhattan distances of the application (the cost column) can increase or remain the same for any iteration. When this happens, that choice is rejected and another mutation from the previous assignment is evaluated. Currently, the algorithm commits to the first improvement it finds, i.e. when an iteration decreases the total cost, it is never revoked. This will potentially result in a local extreme that is not globally optimal.

As a last step in the mapping process, step three performs incremental routing. This means the channels from the ALS are routed incrementally with a point-to-point shortest path algorithm. Only those lanes in the NoC that can guarantee sufficient throughput are considered. Figure 5 shows the resulting CSDF graph. This graph can be checked (with [15]) to see whether the throughput of the mapping suffices to meet the constraints laid down in the ALS (step 4 of the spatial mapper). The buffer sizes B_i are calculated by the algorithm used in step 4. When they are smaller than the buffers reserved by the implementations, not further action is required. When they are larger, an attempt should be made to allocate the additional required buffer size on the tiles the consuming actor is mapped onto. If this additional buffer capacity can not be allocated, the mapping is infeasible and the spatial mapper should iterate. The buffer capacity of the Sink actor (x) is fixed by the specification of Sink.

6 Conclusions and future work

This paper presented a formal model of spatial mapping. The formal definitions of adequacy and adherence give testable criteria of spatial mappings. The notion of feasibility can only be defined formally, if the constraints of the application are defined formally as

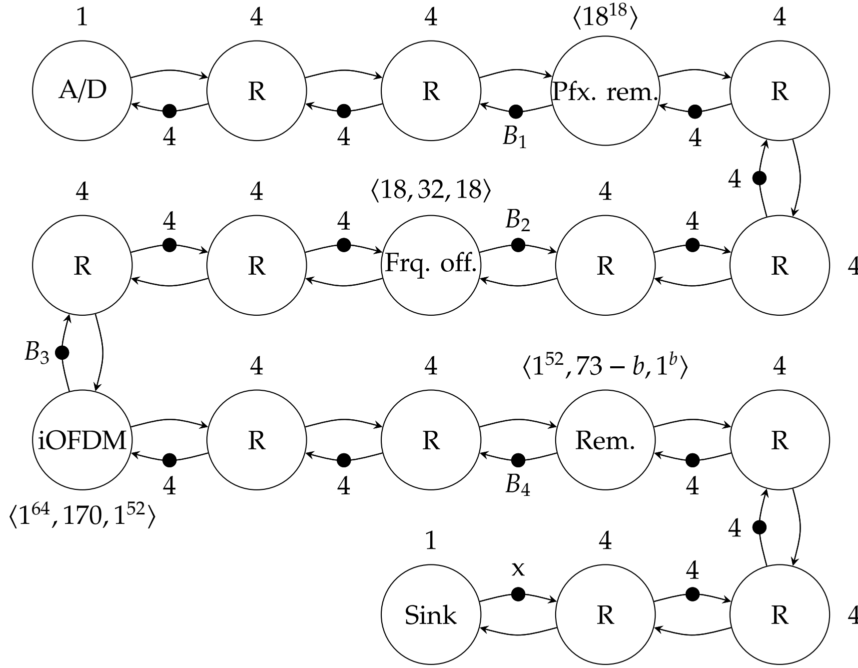

Figure 5: Final CSDF graph (production and consumption rates omitted to prevent clutter)

well. An application independent formalization of application constraints and feasibility is considered future work.

The algorithm we presented earlier has now been shown to implement the formalism we have presented. Other algorithms, designed for the purpose of spatial mapping, can now be related to the algorithm in this paper by relating it to the formalism.

Optimization objectives have not been treated formally in this paper. Future work should include a formalization thereof, so that different algorithms for spatial mapping can be analysed and compared qualitatively. Moreover, the formalism assumes point-to-point connections (NoC), but should be extended to deal with any kind of interconnect.

References

[1] Arthur Abnous. *Low-Power Domain-Specific Processors for Digital Signal Processing*. PhD thesis, University of California, Berkeley, 2001.

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

[3] Tilera Corporation. Tile64™ processor product brief. Corporate product brief.

[4] William James Dally, Ujval J. Kapasi, Brucek Khailany, and Abhishek Ahn, Jung Hoand-Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52–62, 2004.

[5] Giovanni de Micheli and Luca Benini. Networks on chip: A new paradigm for systems on chip design. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 418, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Paul M. Heysters. *Coarse-Grained Reconfigurable Processors – Flexibility meets Efficiency*. PhD thesis, University of Twente, Enschede, The Netherlands, sep 2004.

[7] James A. Kahle, Michael N. Day, H. Peter Hofstee, Theodore R. Johns, Charles R. and-Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[8] Nikolay Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. PhD thesis, University of Twente, 2006.

[9] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. In *Proceedings of the IEEE*, volume 75(9), pages 1235 – 1245, September 1987.

[10] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[11] T. Ojanpera and R. Prasad. An overview of air interface multiple access for imt-2000/umts. *IEEE Commun. Mag.*, 36(9):82–95, September 1998.

[12] G.J.M. Smit, Andre B.J. Kokkeler, Pascal T. Wolkotte, Philip K.F. Hölzenspies, Marcel D. van de Burgwal, and Paul M. Heysters. The chameleon architecture for streaming dsp applications. *EURASIP Journal on Embedded Systems*, 2007:78082, 2007.

[13] L.T. Smit, J.L. Hurink, and G.J.M. Smit. Run-time mapping of applications to a heterogeneous soc. In *Proceedings of the 2005 International Symposium on System-on-Chip*, pages 78–81, November 2005.

[14] Michael Bredford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[15] Maarten Wiggers, Marco Bekooij, and G.J.M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 658–663, New York, NY, USA, 2007. ACM Press.