Int J Parallel Prog (2010) 38:68–83
DOI 10.1007/s10766-009-0120-y

Run-time Spatial Mapping of Streaming Applications to Heterogeneous Multi-Processor Systems

Philip K. F. Hölzenspies · Timon D. ter Braak · Jan Kuper · Gerard J. M. Smit · Johann M. Hurink

Received: 29 February 2008 / Accepted: 23 September 2009 / Published online: 12 November 2009

Abstract In this paper, we define the problem of *spatial mapping*. We present reasons why performing spatial mappings at run-time is both necessary and desirable. We propose what is—to our knowledge—the first attempt at a formal description of spatial mappings for the embedded real-time streaming application domain. Thereby, we introduce criteria for a qualitative comparison of these spatial mappings. As an illustration of how our formalization relates to practice, we relate our own spatial mapping algorithm to the formal model.

Keywords Reconfigurable computing · Heterogeneous multi-processor systems · Run-time spatial mapping · Resource management

1 Introduction

Academia as well as industry 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 real-time streaming applications, running on heterogeneous multi-processor systems, with a special focus on MPSoC cases, where energy efficiency is a key goal.

The remainder of this section introduces the concepts relevant to this paper. A description of the contribution of this paper is given in Sect. 2 as well as an overview of the related work. Our formal description of the spatial mapping problem is given in Sect. 3. As an illustration of how this formal description relates to the real world, Sect. 4

P. K. F. Hölzenspies (✉) · T. D. ter Braak · J. Kuper · G. J. M. Smit · J. M. Hurink
Department of Electrical Engineering, Mathematics and Computer Science,
University of Twente, Enschede, The Netherlands
e-mail: p.k.f.holzenspies@utwente.nl

shows how our spatial mapping algorithm corresponds to it. The implementation in a concrete system and some experimental results are presented in Sect. 5. Finally, conclusions are drawn in Sect. 6.

1.1 Real-Time Streaming Applications

Real-time streaming applications are implemented and used in portable and otherwise energy constrained (embedded) systems and require energy-aware tools and an energy-efficient processing architecture. Typical examples of such applications involve Digital Signal Processing (DSP) algorithms and are found in phased array antenna systems (for radar and radio astronomy), wireless (baseband) communication (for wireless LAN, digital radio, UMTS [6,19,30]), multi-media, medical imaging and sensor networks.

A key characteristic of what is referred to here as "streaming" applications is that they can be modeled as dataflow graph (DFGs) with streams of data items (the edges) flowing between computational kernels (the vertices) [5]. For the remainder of this paper, computational kernels will be referred to as *tasks* and the data streams flowing between them as *channels*. Applications are represented by *task graphs* consisting of these tasks and channels. The qualification "real-time" implies that timeliness is part of correctness. As a consequence, throughput, latency and jitter are *constraints* rather than (optimization) *objectives* [25]. In hard real-time systems no deadline may be missed, as that may lead to dangerous situations. In soft real-time systems, missing a deadline is not catastrophic, but does degrade the system's total performance. Even though no firm guarantees are given for such systems, the goal is to keep the Quality of Service (QoS) high. In short, an important property of real-time systems is that nothing is gained by delivering a higher QoS than the application asks for.

For any kind of real-time behaviour (soft or hard), applications need to have predictable behaviour in terms of time and spatial (i.e. hardware) resource usage [3] so that at least some QoS prediction can be made. Predictable behaviour means that execution time and resource usage are bounded. Tighter bounds give better-or-equal predictability. Typical real-world applications that fall into this category display a high degree of regularity in the communication between tasks and have a semi-static life-time [30, Ch. 3], i.e. typically in the order of minutes, rather than milliseconds.

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. Examples of such MPSoC designs are Cell [10], Tilera64 [27], Intel's experimental 80-tile [28] and Avispa [8]. On a more conceptual level, MPSoC design templates have been developed, such as Pleiades [1] and Chameleon [9,22]. For a more detailed overview, we refer to [9].

What is referred to as a *tiled system* in this paper, is a multi-processor architecture, where the individual processors can be considered autonomous and composable. Autonomicity means that a processor can be programmed separately from other processors. Separate Arithmetic and Logic Units (ALUs) or pipelines in a superscalar

processor are not considered autonomous. Composability means that a processor can be assigned a task—or tasks already running on the processor can be changed or removed—without directly affecting (unrelated tasks on) other processors. In other words, the operational feasibility of unrelated tasks is not affected, i.e. they still do their jobs correctly and within their guaranteed resource bounds. The same autonomicity must hold for other resources in the tiled system, like memories with a communication assist or Direct Memory Access (DMA), I/O modules (A/D converters, etc.), or application specific circuitry. For these (spatial) resources to form one system, they must be interconnected. The combination of an autonomous resource and its interface to the system's interconnect is referred to as a *tile*. When an MPSoC contains different types of tiles (i.e. different resources), it is considered heterogeneous.

For the sake of composability, a system's interconnect must also provide QoS guarantees [11]. The Network-on-Chip (NoC) paradigm [2], which is gaining popularity in the MPSoC world, has interconnect architectures that provide such guarantees [30], but is by no means the only applicable paradigm. Conventional busses and mixed NoC-and-bus interconnects are all acceptable, as long as their behaviour is predictable, e.g. using latency-rate schedulers [26]. This is especially relevant when extending systems from MPSoC to (SiP) and even to multiple chips on a Printed Circuit Board (PCB).

1.3 Run-time Spatial Mapping

Generally, spatial mapping is the allocation of spatial resources to applications. In the context of tiled systems, spatial resources are tiles and communication resources. Thus, spatial mapping is the assignment of tasks and channels from the application's task graph to tiles and the interconnect, respectively. A *feasible* spatial mapping satisfies the mapped application's QoS constraints. A spatial mapping's *quality* depends on the extent to which it optimizes resource usage and extra-functional costs like energy consumption. The quality of a spatial mapping *algorithm* depends on the trade-offs of the platform on which it is used, but is typically a combination of response time, all mappings' qualities and the success rate of finding mappings for applications.

A downside of heterogeneous tiled systems is that even when only a few tiles are allocated to applications, there may be no more tiles of the correct type available to execute a specific task of the application being mapped. When there are different types of tiles with the same functionality (e.g. different types of processors, memories with different types of communication assists, etc.), the same task could be implemented for different types of tiles. Having multiple implementations for the same tasks thus increases the flexibility of the resource allocation in a heterogeneous system. Even when an additional implementation of a task is less energy-efficient, the application's overall energy-efficiency might still benefit from its use, when the closest (in terms of the interconnect) available tile required for the preferred implementation is far away. The same holds for the latency imposed by computation and communication. For sufficiently large systems, communication costs (in terms of latency or energy) might supersede the added computation cost from a less efficient implementation on a nearby tile.

1.3.1 Necessity and Advantages

We argue that performing the spatial mapping at *run-time* is both necessary and desirable. Preliminary experiments [23] were promising with respect to the feasibility of run-time spatial mapping in general. Our implementation (see Sect. 5) shows that it is feasible for the concrete systems and applications we have developed. However, the more formal analysis presented in this paper is required to make qualitative comparisons between different spatial mapping algorithms.

Performing the spatial mapping at run-time is necessary, whenever the application set is not known completely at design-time. This happens for a wide variety of reasons, e.g. when the platform allows the user to use software from any vendor, developed for that platform. When different software vendors produce software for the same platform independently, no one knows the complete application set.

A further reason for applying the spatial mapping at run-time, results from the dependency of the availability of the resources on the set of applications running simultaneously, on variations in QoS requirements due to changes in the environment, and on user initiated changes. Because of these dependencies, a design-time spatial mapping needs to know and consider all possible combinations of applications at design-time.

Performing the spatial mapping at run-time, thus offers desirable flexibility. Unforeseeable changes in applications (modifications of standards, bugfixes, etc.) can be taken into account. Moreover, defective tiles can be avoided, which both increases production yield 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, storage (i.e. memory) and communication. In principle, the spatial mapping is performed only when a new streaming application is started. This does not strictly exclude dynamic structural changes in an application, e.g. when the signal of a wireless broadcast degrades, the control system of a receiver may be specified to start an extra error-correction task. When new tasks are dynamically added to an application, the mapping of tasks already running is a constraint for the mapping of the new tasks. A core assumption for run-time spatial mapping, though, is that applications are quasi-static, so that the benefit of the flexibility gained outweighs the added cost of the run-time mapping. Furthermore, run-time spatial mapping algorithms must be fast, because start-up time is often bounded by the application as well (e.g. answering a phone).

To be able to perform the mapping of an application to tiles, a spatial mapping algorithm needs a model of the hardware platform and, for the application, the task graph with the corresponding QoS constraints and available implementations of the tasks with their resource requirements, energy costs and behavioural bounds. Some performance figures can already be determined at design-time, e.g. the execution time and energy consumption of various implementations of tasks on specific tile types. However, some figures can only be determined at run-time. This requires simple

performance models (simple in the computational sense, since there may be tight constraints on the time required to find the mapping).

Performing the spatial mapping at run-time implies that *more* performance figures can not be determined at design-time. It is, after all, only known after the mapping on which tile a task will be executed, which means that inter-task communication parameters (e.g. estimated latency, energy consumption), for example, need to be determined at run-time. Likewise, it is only known at run-time which tasks are already running on a tile. Therefore, the response time of a task is only known at run-time and schedulers must not just guarantee their own QoS constraints, but it must also be guaranteed that the constraints of applications are not violated. This requires schedulers to be asynchronous servers with bounds on preemption [3]. However, the choices at run-time are restricted to a finite set of implementations, all of which have properties that are determined at design-time.

The constraints of the application can only be fully checked after it has been mapped: Only after a spatial mapping has been determined and latencies and throughputs of tasks running on tiles are known, the constraints can be checked. We use a dataflow analysis [7,29] for this check, which is beyond the scope of this paper. As previously stated, only a spatial mapping that lets the application meet its QoS constraints is considered to be *feasible*.

2 Contribution and Related Work

Run-time spatial mapping of real-time streaming applications is a very young research topic. As such, opinions vary on what it does and does not comprise. Current practice in the embedded systems world is to perform both the spatial and temporal mapping of applications to tiled systems simultaneously at design-time (e.g. [20]). 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. With a formal analysis of the problem of spatial mapping, this paper attempts to fence off the territory of spatial mapping.

To our knowledge, this is the first attempt at a formalization of this type of spatial mapping. The problem described is explicitly different from spatial mapping in the High Performance Computing (HPC) field, where the primary goal is to achieve optimal load balancing [14,15] and applications do not have tight structural constraints.

Many solutions have already been proposed for subproblems of the spatial mapping problem. Unfortunately, these solutions are very hard to compare, because they are commonly tailored to rather specific systems.

The current research can be grouped based on the computation/communication trade-off. Work in which communication is assumed to be most costly, treats the mapping of tasks to tiles as a cost factor in routing algorithms [4,16,24]. On the other hand, work that stresses computational costs, mostly does not discuss or even consider communication resource management [18].

Many papers also perform specific analyses at design-time or make assumptions as to what factors are detrimental to performance. Their run-time spatial mapping approaches are then related to how well their results are improved by the design-time preparations [31,32] or how well (assumed) detrimental factors are avoided [17]. The work presented in [12] only considers independent tasks, i.e. without inter-task communication or QoS constraints over multiple tasks.

In [13,17], some conditions that must be met by resource management policies for real-time applications are described. The conditions mentioned are *admission control* (applications are only allowed to start if sufficient resources can be allocated) and *guaranteed resource provisions* (running tasks are always allowed access to the resources allocated to them).

In the following, we present a model, which provides a means for qualitative comparison of different run-time spatial mapping approaches.

3 A Formal Definition of Spatial Mapping

In this section, we first describe the elements of a system, both hardware and software. Next, we introduce an abstraction over a system's interconnect that allows us to define spatial mappings. Finally, the constraints, in terms of capacities and requirements are described.

3.1 Hardware Platform

In this section, we formally describe tiled systems. In such a system, a tile is any resource that can perform a task. A tile can be connected to a router through an interface. Routers connect to other routers through lanes. The term 'router' is used to denote any kind of interconnection element that controls the direction in which data flows. This includes, but is by no means limited to, NoC-routers, bus controllers and bridges.

To clarify notation, consider the example architecture depicted in Fig. 1. This system is heterogeneous with regards to both tiles and interconnection elements. The bus controllers, bridge and routers depicted, are all modelled as routers, since they all influence the direction of data streams in the interconnect. Furthermore, the system is also heterogeneous with regards to the connections between routers, i.e. the lanes. The bus shown allows communication between all connected components, but all communication shares the bus as a single resource. Therefore, the bus should be represented by a single object in our formal description. The same holds for the bidirectional connections depected in the NoC between the DSPs. Both connections between the bridge and the NoC are unidirectional. Thus, the representation of lanes should reflect direction. Because of these requirements, we use *hypergraphs* to model interconnection. Hypergraphs are a generalization of graphs, where edges (referred to has *hyperedges*) can connect more than two vertices. A hyperedge is described as a set of vertices. A hypergraph is *directed*, if hyperedges are described by pairs of sets of vertices, viz. a 'from' set and a 'to' set.

Formally, an *interconnect* is represented by a directed hypergraph $\mathfrak{C} = \langle \mathcal{R}, \mathcal{E} \rangle$, where $\mathcal{R}$ is a set of *routers* (vertices) and $\mathcal{E} \subseteq \mathbb{P}\mathcal{R} \times \mathbb{P}\mathcal{R}$ (where $\mathbb{P}$ denotes the

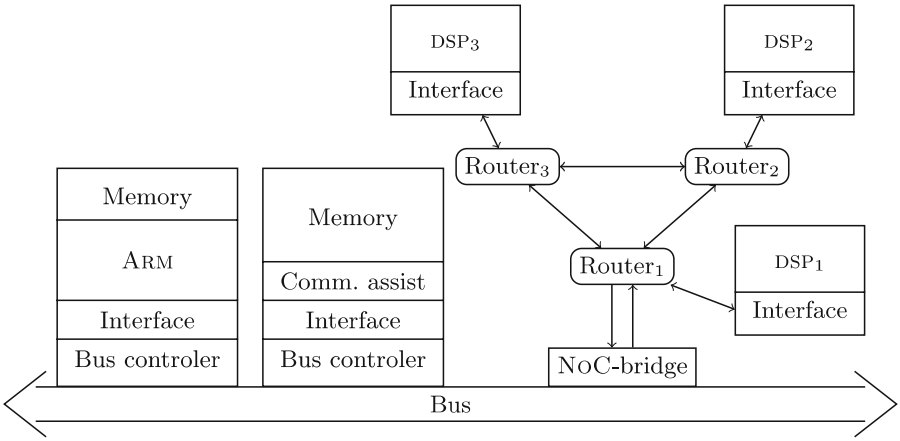

Fig. 1 Example architecture

powerset) is a set of *lanes* (directed hyperedges) between routers. Lanes are defined as hyperedges, so that both busses and bi-directional point-to-point links can be expressed in this model, besides 'simple' unidirectional point-to-point links. Furthermore, let $\mathcal{T}$ be a set of *tiles* and $\mathcal{F} \subseteq \mathcal{T} \times \mathcal{R}$ be a set of *interfaces* of tiles with routers. Interfaces allow bidirectional communication. A *tiled system* $\mathfrak{T}$ now is a quadruple $\langle \mathcal{T}, \mathcal{R}, \mathcal{F}, \mathcal{E} \rangle$, which, again, is a hypergraph: $\langle \mathcal{T} \cup \mathcal{R}, \mathcal{F} \cup \mathcal{E} \rangle$. Note that, following from this definition, there are no direct connections between tiles. Furthermore, we assume that $\mathfrak{T}$ is *weakly connected*, i.e. there are no unconnected non-empty subgraphs.

Coming back to the example architecture given in Fig. 1, it can be modelled in the way described above.

The resulting hypergraph is depicted in Fig. 2. The hyperedges, representing lanes between routers, are shown in grey. Interfaces are shown as bidirectional double arrows. This figure shows the difference between busses, bidirectional links (between the routers) and unidirectional links (between the NoC-bridge en Router_1.

3.1.1 Capacities

All elements in $\mathfrak{T}$ represent resources with finite capacities. One can think of computational and memory capacities, but also of the maximum number of tasks that can be assigned to it; e.g. Application Specific Integrated Circuits (ASICs) can not switch between tasks, so they have a maximum of one task assigned to it, while an ARM may be able to serve as many tasks as there are slots in its Time Division Multiple Access (TDMA) scheduler. Examples of such capacities for the interconnect are lane bandwidth, router TDMA slots, number of virtual channels etc.

Thus, *all* relevant (local) capacities of a tiled system can be expressed by *capacity vectors*. Let $C_{\mathcal{T}}(t)$, $C_{\mathcal{R}}(r)$, $C_{\mathcal{E}}(l)$ and $C_{\mathcal{F}}(f)$ denote the capacity vectors of every tile t, router r, lane l and interface f respectively. All capacity vectors for the same kind of elements (tiles, routers, lanes and interfaces) are considered to be of the same 'shape,'

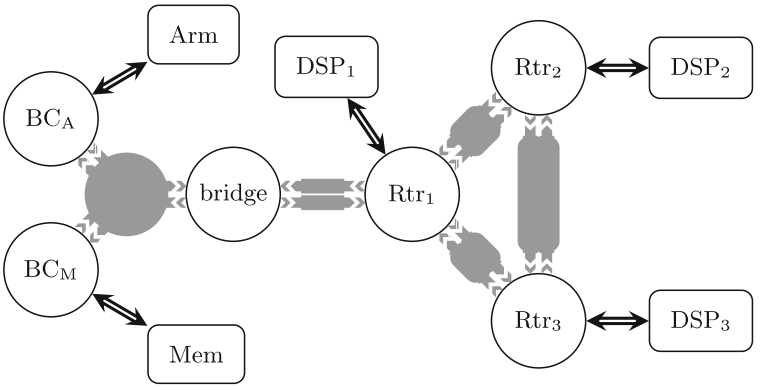

Fig. 2 The hypergraph representation of the example architecture

i.e. every capacity vector of any tile always has the same dimension. For simplicity, we assume vectors of independent dimensions.

3.2 Software Applications

An *application* is represented by a directed graph $\mathfrak{P} = \langle \mathcal{P}, \mathcal{C} \rangle$ where $\mathcal{P}$ is a set of *tasks* and $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ a set of *channels* between tasks along which tasks communicate with each other. Tasks are functions of input streams to output streams. Implementations are realisations of tasks. In other words, implementations are executable units that compute the function that is their corresponding task, i.e. for any task p and implementation $i \in \mathcal{I}(p)$, the semantics of i is p.

For all tasks, several implementations may exist, though not necessarily for all types of tiles in the tiled system. The set of implementations for task p is denoted by $\mathcal{I}(p)$. The subset $\mathcal{I}_\tau(p) \subseteq \mathcal{I}(p)$ denotes the set of implementations of p that can be allocated to tiles of type τ.

3.2.1 Requirements

As a dual to the notion of capacities of the hardware, software applications have resource requirements, described by *requirement vectors*. For task p, every implementation $i \in \mathcal{I}(p)$ has a requirement vector $R_{\mathcal{I}}(i)$. Similarly, every channel c has a requirement vector $R_{\mathcal{C}}(c)$.

3.3 Paths

For every task, one implementation is chosen and this implementation is mapped onto a single tile. To get a connection between mapped tasks, a channel must be mapped to a *sequence of* elements of the interconnect. We introduce an abstraction from the interconnect to *paths*, so that every channel from an application can be mapped to a

single path. This abstraction leads to a 'higher order graph,' in which edges *are* the paths in $\mathfrak{T}$.

Let $\mathcal{E}^*$ be the set of all cycle-free paths[1] over a tiled system $\mathfrak{T} = \langle \mathcal{T}, \mathcal{R}, \mathcal{F}, \mathcal{E} \rangle$. Because paths connect tiles, a path starts and ends with an interface, connecting tiles to routers. The number of routers on a path is arbitrary (albeit ≥ 1), but between every two routers there has to be a lane. A path is only considered *valid* if every consecutive pair of elements (interface, router or lane) is connected in $\mathfrak{T}$. As a result, we get a *pathed tiled system* $\mathfrak{T}^* = \langle \mathcal{T}, \mathcal{E}^* \rangle$, which is a directed multi-graph (a graph that may have several edges between any pair of vertices).

For the description of the proposed methods, it is helpfull to have the notion of 'capacity of a path $p \in \mathcal{E}^*$'. We denote such a capacity vector by $C_{\mathcal{E}^*}(p)$, where all relevant capacities of routers, lanes and interfaces have an own component in this vector. For a given path $p \in \mathcal{E}^*$, each component in $C_{\mathcal{E}^*}(p)$ represents the minimum value for this component in the relevant elements of path p, e.g. bandwidth in bits per second.

The dual of a capacity vector of a path is the requirement vector of a channel. As such, the vectors in $R_{\mathcal{C}}$ have the same size as those in $C_{\mathcal{E}^*}$.

3.4 Spatial Mapping

If a software application has to run on a tiled system, we have to associate tiles to tasks and paths to channels. Clearly, this has to be done in such a way that the necessary implementations exist and that the capacity of the tiled system is not exceeded.

An *assignment function* α is a function which maps an application $\mathfrak{P}$ to a pathed tiled system $\mathfrak{T}^*$. More precisely, for every task $p \in \mathfrak{P}$, $\alpha(p)$ is a tile in $\mathcal{T}$ and for each channel $\langle q, r \rangle \in \mathcal{C}$, $\alpha\langle q, r \rangle$ is an edge from $\alpha(q)$ to $\alpha(r)$ in $\mathcal{E}^*$. Where required, the part of α that maps tasks onto tiles is referred to as α_π and the part of α that maps channels onto paths is referred to as α_γ.

An *implementation selector* $I : \mathcal{P} \rightarrow \mathcal{I}$ is a function which projects tasks onto implementations. A *spatial mapping* $\mathfrak{m}$ can now be defined as a tuple $\langle \alpha, I \rangle$ of a task assignment function α and an implementation selector I.

Suppose, for a task p, that $\alpha(p) = t$, where t is a tile of type τ. In this case, an implementation of p for a tile of type τ should exist. A spatial mapping is considered *adequate* if every task is mapped to a tile of a type for which an implementation is available. In other words, a mapping $\mathfrak{m} = \langle \alpha, I \rangle$ is called adequate, iff for every task $p \in \mathcal{P}$ it holds that $I(p) \in \mathcal{I}_\tau(p)$, where τ is the type of $\alpha(p)$.

3.4.1 Adjusted Capacities

When an application is mapped to a system, obviously, the resources required for that application will no longer be available for the next application to be mapped. This system state is reflected in the capacity vectors. In other words, the capacity vectors reflect the *currently available capacity* of the system, rather than the capacity of a

[1] In a hypergraph, a path is described as an alternating sequence of vertices and edges. It is cycle-free, if no two vertices occur twice in it.

system after boot. Dually, when applications are stopped, the resources assigned to them are added to the capacity vectors again.

3.5 Cumulative Resource Requirements

The definitions so far only relate individual implementations and channels to requirements. No definitions have yet been given to express the cumulative resource requirements of a mapped application. For these definitions, the inverse of a task assignment function α is required. This inverse is defined in two parts: One part for the assignment of tasks to tiles and one part for the assignment of channels to paths.

The inverse of the task assignment function with regards to tasks is defined as a function from tiles to sets of tasks:

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

Using this inverse, the cumulative requirement $\mathcal{L}_\pi^{\mathfrak{m}}(t)$ imposed on tile t by mapping $\mathfrak{m}$ can be expressed as

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

Analogously, the inverse of the task assignment function with regards to channels is defined as a function from paths to channels, viz.

$$\alpha_\gamma^{-1}(p) = \left\{c \in \mathcal{C} \middle|\ \alpha_\gamma(c) = p\right\}$$

with cumulative requirement $\mathcal{L}_\gamma^{\mathfrak{m}}(p)$ imposed on path p by mapping $\mathfrak{m}$

$$\mathcal{L}_\gamma^{\mathfrak{m}}(p) = \sum_{p \in \alpha_\gamma^{-1}(p)} R_{\mathcal{C}}(p)$$

With this notion of cumulative requirement, spatial mappings can be checked against the capacity vectors of a tiled system to see whether no capacities are exceeded. A spatial mapping $\mathfrak{m} = \langle \alpha, I \rangle$ of application $\mathfrak{P} = \langle \mathcal{P}, \mathcal{C} \rangle$ to the pathed tiled system $\mathfrak{T}^* = \langle \mathcal{T}, \mathcal{E}^* \rangle$ is called *adherent* if the following constraints are met:

$$\begin{aligned} &\mathfrak{m} \text{ is adequate} \\ \forall t : \mathcal{T} \quad &(\mathcal{L}_\pi^{\mathfrak{m}}(t) \leq C_{\mathcal{T}}(t)) \\ \forall p : \mathcal{E}^* \quad &(\mathcal{L}_\gamma^{\mathfrak{m}}(p) \leq C_{\mathcal{E}^*}(p)) \end{aligned}$$

4 Algorithm

Even when only considering the assignment of processes to a heterogeneous tiled system, which is a Generalized Assignment Problem (GAP) and this is known to be

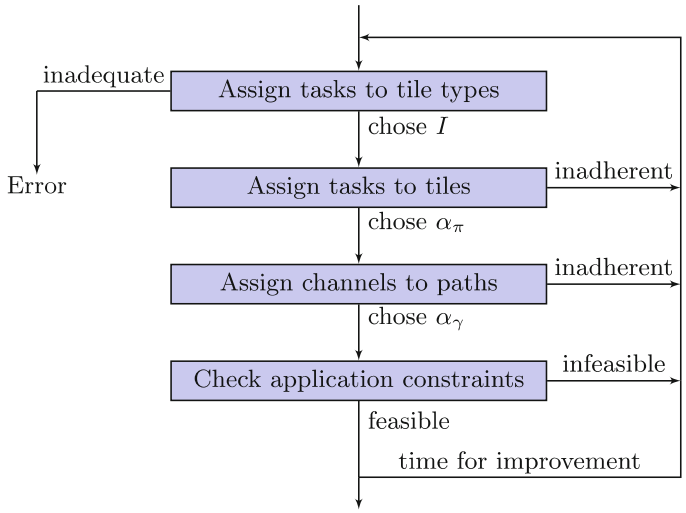

Fig. 3 Hierarchical search with iterative refinement

NP-complete. Considering the prohibitive 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 step. 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 only comes to light in later steps, we propose a strategy for iterative refinement. Figure 3 shows the hierarchical decomposition into steps used in our run-time spatial mapping algorithm for heterogenous MPSoCs. We now describe each of these steps in more detail.

1. The goal of the first step is to choose an implementation (and thereby tile type) for every task, 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 all components of $C_t - R_i$ are ≥ 0.
   The order in which we pick an implementation for each task is based on its *desirability*. We define the desirability of a task as the difference between the cheapest assignment and the second cheapest assignment of one of its implementations to a tile. In other words, if the second best implementation is more expensive, the desirability to map the task increases.

To sustain the adherence of $\mathfrak{m}$, we virtually map the chosen implementation to the best-fitting tile, which is determined in the desirability calculation. The implementation choice for the remaining tasks is affected by this mapping, as the available resource capacities in the system are reduced. This guarantees that after this step (if this step manages to map all tasks), at least one adherent α_π exists, although $\mathfrak{m}$ might still be inadherent due to the communication restrictions. If the ordering on desirability does not result in an adequate solution, alternative orderings can be tried.

2. Resulting from step one, we have an implementation assigned to every task in the application and we know that an adequate α_π exists. In this step, we take more detail into account, aiming at finding a task assignment α_π with minimal cost. Besides cost factors based solely on the mapping of a task to a tile, we also award assignments with a bonus for proximity of neighboured tasks in the application's data flow graph. This stimulates locality, causing the communication routes, assigned in the next step, to likely be short.
   We define a start point in the application's data flow graph as the task with the lowest communication degree. For this start task t, we evaluate the costs of all possible assignments $\alpha_\pi(t)$ to tiles in the system matching the task's type. After assigning this task to the tile with the lowest cost, we proceed with an iterative mapping process. At each iteration i, the tasks are mapped that lie in the data flow graph iso-distance i away from the start task. Using local search in the proximity of the tasks from the previous iteration, we map each task to the best available tile of the required type. Deciding when to stop the local search once a suitable tile has been found can be based upon the ratio between computational costs and communication costs.
   Again, we prevent immediate inadherence in the next step, by only considering tiles for a task that have sufficient communication resources to facilitate the task's communication requirements, at least, locally.
3. For the realization of step three, the channels are sorted by non-increasing throughput. Then, 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 interface of the channel is determined, where only those routers 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 router.
   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. The last step checks the constraints posed on the application using techniques developed by Wiggers et al. [29]. When any such constraint is violated, mapping $\mathfrak{m}$ is *infeasible* and feedback should be given to higher steps to try and improve those characteristics of the mapping that violate the constraint(s). If no constraint is violated, mapping $\mathfrak{m}$ is *feasible*. We also may decide to improve upon the current solution. In this case, possible points of improvement should be identified

and fed back into the corresponding step. After each iteration we select the best solution among the feasible mappings.

In general, a 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 immediately generates 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 Implementation and Results

We have implemented the algorithm described in Sect. 4 in a Linux kernel. This kernel runs on a QEMU [21] simulated ARM926EJ- S processor. The test set consists of three architectures of multi-tiled systems: a homogeneous ring of 16 tiles (HORING16), a heterogeneous mesh of 28 tiles (HEMESH28) and a multi-chip architecture (MCHIP89), consisting of 89 tiles spread over nine chips. Furthermore, we used three applications. These applications differ most significantly in the structure of their data flow graphs (see Fig. 4). The first application is a chain of six tasks (i.e. every task has at most one incoming and one outgoing stream). The second application has a total of five tasks, where one task produces output for two others and one consumes input from these two. Finally, the third application is an unbalanced composition of two chains; one chain of two tasks, alongside a chain of four tasks, with a task producing to both chains at the beginning and a task consuming from both chains at the end. For every applications, we used two instances: One where all tasks had just a single implementation, i.e. homogeneous task implementations (HO), and one where all tasks were implemented for as many tile types as possible, i.e. heterogeneous task implementations (HE).

Every application is mapped onto a system multiple times, until the algorithm fails to find a mapping. We verify manually for every failure that there is, indeed, no way to map yet another copy of the application onto the system. When the algorithm finds equal cost alternatives for a choice, the choice is made at random. To eliminate possible lucky guesses, every measurement is repeated seven times. The execution time of the algorithm's steps was measured for these applications and averaged per application, per architecture, per repeated measurement. The result is given in Fig. 5. The execution times, as shown in the graph, of the algorithm are very low

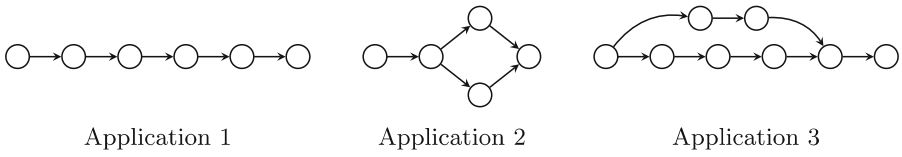

Fig. 4 Applications in the test set

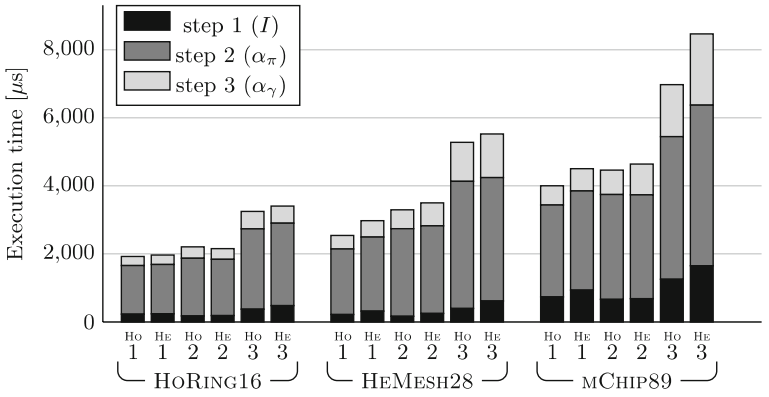

Fig. 5 Execution times for the algorithm per application and architecture

(compared to exhaustive placement algorithms that guarantee optimality). The graph further suggests that with growing complexities (either in the structure of the application or of the platform), the execution time of the first and third step grows relatively larger than that of the second step. During the tests, we never achieved situations where a mapping existed, but the algorithm did not find it, i.e. the algorithm never produced false negatives for the test set. Furthermore, all mappings found by the algorithm were feasible, i.e. the algorithm never produced false positives for the test set.

6 Conclusions and Future Work

We have presented a formal model of spatial mapping. The definitions of adequacy and adherence give testable criteria of spatial mappings. However, the notion of feasibility can only be defined formally, if the constraints of the application are defined formally as well. An application independent formalization of application constraints and feasibility is content of future work.

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 analyzed and compared qualitatively. Furthermore, a *quantitative* comparison of spatial mapping algorithms is required to compare algorithms that are similar under qualitative comparison. Current System-on-Chip (SoC) benchmarks focus on computational issues, much more than on resource management, and they largely ignore future, large scale applications. Currently, we are working on a benchmark suite tuned specifically for this purpose.

It has been shown, that the presented algorithm implements the introduced formalism. 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.

Acknowledgements We would like to thank the reviewers for some very helpful comments, questions and suggestions.

References

1. Abnous, A.: Low-power domain-specific processors for digital signal processing. PhD thesis, University of California, Berkeley (2001)
2. Benini, L., De Micheli, G.: Networks on chips: a new soc paradigm. Computer **35**(1), 70–78 (2002)
3. Buttazzo, G.C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, Norwell (1997)
4. Chou, C.-L., Marculescu, R.: Incremental run-time application mapping for homogeneous nocs with multiple voltage levels. In: CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis, pp. 161–166. ACM, New York, NY, USA (2007)
5. Dally, W.J., Kapasi, U.J., Khailany, B., Ahn, J.H., Das, A.: Stream processors: programmability and efficiency. Queue **2**(1), 52–62 (2004)
6. ETSI: Broadband Radio Access Networks (BRAN); HiperLAN type 2; Physical (PHY) layer, ETSI TS 101 475 v1.2.2 (2001–02), (2001)
7. Ghamarian, A.H., Geilen, M.C.W., Sander, S., Basten, T., Moonen, A.J.M., Bekooij, M., Theelen, B.D., Mousavi, M.R.: Throughput analysis of synchronous data flow graphs, pp. 25–36 (2006)
8. Held, I., Vandewiele, B.: Avispa ch—embedded communications signal processor for multi-standard digital television (2006)
9. Heysters, P.M.: Coarse-grained reconfigurable processors—flexibility meets efficiency. PhD thesis, University of Twente, Enschede, The Netherlands (2004)
10. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4/5), 589–604 (2005)
11. Kavaldjiev, N.: A run-time reconfigurable network-on-chip for streaming DSP applications. PhD thesis, University of Twente (2006)
12. Kim, J.-K., Shivle, S., Siegel, H.J., Maciejewski, A.A., Braun, T.D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R.B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., Yellampalli, S.S.: Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, p. 98.1. IEEE Computer Society, Washington, DC, USA (2003)
13. Kumar, A., Mesman, B., Theelen, B., Corporaal, H., Yajun, H.: Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In: ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, pp. 33–38. IEEE Computer Society, Washington, DC, USA (2006)
14. Keqin, L.: Optimal load distribution in nondedicated heterogeneous cluster and grid computing environments. J. Syst. Archit. **54**(1–2), 111–123 (2008)
15. Kai, L., Subrata, R., Zomaya, A.Y.: On the performance-driven load distribution for heterogeneous computational grids. J. Comput. Syst. Sci. **73**(8), 1191–1206 (2007)
16. Marcon, C., Borin, A., Susin, A., Carro, L., Wagner, F.: Time and energy efficient mapping of embedded applications onto nocs. In: ASP-DAC '05: Proceedings of the 2005 Conference on Asia South Pacific Design Automation, pp. 33–38. ACM, New York (2005)
17. Moreira, O., Jan-David Mol, J., Bekooij, M.: Online resource management in a multiprocessor with a network-on-chip. In: SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing, pp. 1557–1564. ACM, New York (2007)
18. Nollet, V., Marescaux, T., Avasare, P., Mignolet, J-Y.: Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In: DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 234–239. IEEE Computer Society, Washington, DC, USA (2005)
19. Ojanpera, T., Prasad, R.: An overview of air interface multiple access for imt-2000/umts. IEEE Commun. Mag. **36**(9), 82–95 (1998)
20. Primentel, A.D.: The artemis workbench for system-level performance evaluation of embedded systems. Int. J. Embed. Syst. **3**(3), 181–196 (2008)

21. QEMU homepage. http://www.nongnu.org/qemu/ [cited 2009-03-13]
22. Smit, G.J.M., Kokkeler, A.B.J., Wolkotte, P.T., Hölzenspies, P.K.F., van de Burgwal, M.D., Heysters, P.M.: The chameleon architecture for streaming dsp applications. EURASIP J. Embed. Syst. 78082 (2007)
23. Smit, L.T., Hurink, J.L., Smit, G.J.M.: Run-time mapping of applications to a heterogeneous soc. In: Proceedings of the 2005 International Symposium on System-on-Chip, pp. 78–81. IEEE Computer Society (2005)
24. Srinivasan K., Chatha, K.S.: A technique for low energy mapping and routing in network-on-chip architectures. In: Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005. ISLPED '05, pp. 387–392 (2005)
25. Stankovic, J.A.: Misconceptions about real-time computing: a serious problem for next-generation systems. Computer **21**(10), 10–19 (1988)
26. Stiliadis, D., Varma, A.: Latency-rate servers: a general model for analysis of traffic scheduling algorithms. IEEE/ACM Trans. Netw. **6**(5), 611–624 (1998)
27. Tilera Corporation: Tile64™ processor product brief. Corporate product brief (2008)
28. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y., Borkar, N.: An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In: Proceedings of the IEEE International Solid State Circuits Conference (2007)
29. Wiggers, M., Bekooij, M., Jansen, P.G., Smit, G.J.M.: Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In: Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS'07, pp. 281–292. IEEE Computer Society, Los Alamitos, CA, United States, April (2007)
30. Wolkotte, P.T.: Exploration within the network-on-chip paradigm. PhD thesis, University of Twente, Enschede, January (2009)
31. Ykman-Couvreur, Ch., Nollet, V., Catthoor, Fr., Corporaal, H.: Fast multi-dimension multi-choice knapsack heuristic for mp-soc run-time management. International Symposium on System-on-Chip, 2006, pp. 1–4, November (2006)
32. Ykman-Couvreur, Ch., Nollet, V., Marescaux, Th., Brockmeyer, E., Catthoor, Fr., Corporaal, H.: Design-time application mapping and platform exploration for mp-soc customised run-time management. Comput. Digit. Tech., IET **1**(2), 120–128 (2007)