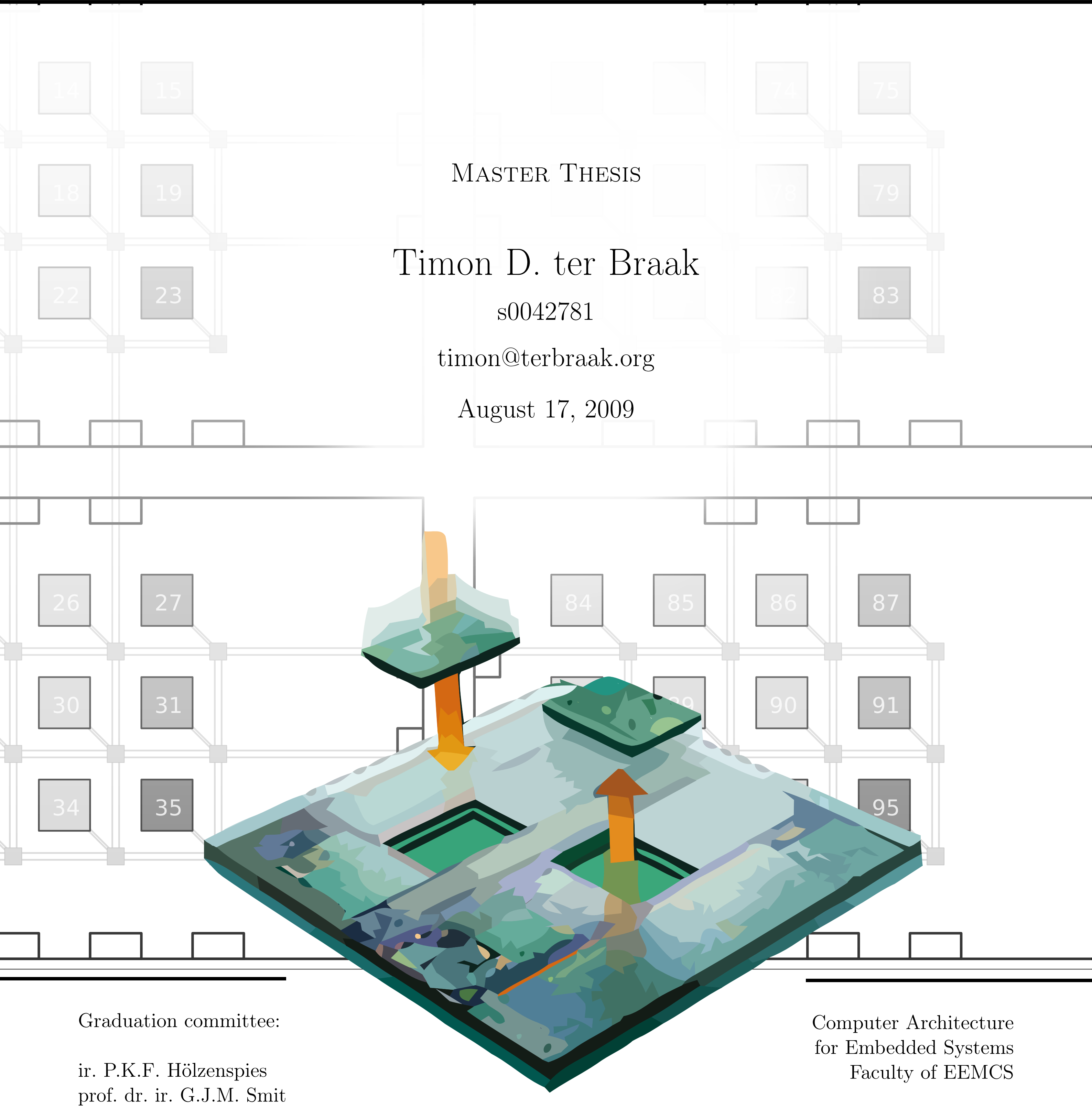
Run-time Spatial Resource Management in Heterogeneous MPSoCs



Master Thesis

Timon D. ter Braak

s0042781

timon@terbraak.org

August 17, 2009

Graduation committee:

ir. P.K.F. Hölzenspies
prof. dr. ir. G.J.M. Smit
prof. dr. J.L. Hurink
dr. ir. J. Kuper

Computer Architecture
for Embedded Systems
Faculty of EEMCS

University of Twente
Enschede

Abstract

This thesis concerns the arbitration between multiple applications over the resources available in a multi-processor system. Sharing resources can have negative effects on the performance of an application, due to scheduling and inter-task communication. Targetting the domain of streaming applications, many applications require a guaranteed performance for correct behavior. Therefore, we use dataflow models that allow analysis of an application, before it is admitted to the system.

Larger systems introduce a spatial factor into an already complex problem. In this thesis, we propose multiple heuristics to tackle the resource management problem. These heuristics use cost functions to steer the outcome and the optimization direction. Our approach is implemented in a Linux kernel. On low-end ARM processors, a single resource allocation attempt takes tens of milliseconds. Due to the many factors that play a role, we experienced a large variability between admission and failure rates.

Opposed to design-time generated use cases, our approach makes flexible and fault tolerant multi-processor systems available, with a guaranteed performance for critical applications.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

One cannot buy, rent or hire more time. The supply of time is totally inelastic. No matter how high the demand, the supply will not go up. There is no price for it. Time is totally perishable and cannot be stored. Yesterday's time is gone forever, and will never come back. Time is always in short supply. There is no substitute for time. Everything requires time. All work takes place in, and uses up time. Yet most people take for granted this unique, irreplaceable and necessary resource.

Peter F. Drucker, American educator and writer, 1909

Time can be a precious resource for humans. Some tasks that need to be done take a constant time, while the duration of other tasks may depend on the amount of other resources allocated to them. Likewise, applications in computer systems require a set of resources to perform their tasks; including time.

Computer systems are subject to continuously increasing performance demands. More work needs to be done in less time, preferably with the same energy budget. Energy consumption has become a critical issue, both for high-end large-scale parallel systems [1] as well for portable devices [2]. For some application domains, like digital signal processing, specific architectures proved to deliver more performance per watt than general purpose processors. Parallel composition of domain specific processing elements and general purpose processors resulted in heterogeneous multi-core systems. Research has shown that heterogeneous systems can deliver higher performance at a given energy budget than homogeneous multi-core solutions [3, 4]. Heterogeneous Multi-Processor Systems on Chip (MPSoCs) have the potential to become the preferred architectures for high performance, low power signal processing systems [5]. This thesis thus focuses on heterogeneous MPSoCs.

Currently, the usability of such systems is still very low; it requires extensive knowledge and effort to use such a system effectively. It is difficult to write applications that integrate enough parallelism to keep all the cores of such systems busy. Also, due to race conditions, communication and synchronization issues, it is known to be more complex to write parallel applications than sequential programs [6]. Decomposition of applications into multiple tasks allows engineers to specify and program smaller sub-problems, which can be allocated to separate processing cores. The functionality of such tasks can be implemented in various ways, depending on the target architecture, and trade-offs between performance, resource and energy budgets.

For some type of applications, especially in the domain of signal processing, it is important that some predefined performance can be guaranteed. An execution environment should be provided that allows real-time applications to meet their deadlines. To accomplish these conditions, each application should be shielded from interference with other applications in the system. The difficulty is that, for a large application domain, applications can be started or stopped by externally triggered events. At design-time, it is unknown what combinations of applications are requested to be executed during the life-time of the system, and whether enough resources can be allocated to each application. This makes it impossible to derive schedules for those applications at design-time, which is often done to ensure temporal isolation of tasks. We think of a schedule as a resource allocation plan, where each application may access resources during specified time intervals. In this sense, processor time is also considered to be a resource that is required to execute the application. In this thesis, we investigate methods to tackle this resource allocation problem at run-time.

1.1 Run-time spatial resource management

A run-time resource manager has to match the resource demand of applications with the resource provision of a platform. What resources are required during its execution should be specified per application. This *application specification* can be used to allocate sufficient platform resources to the application. Not only the amount and type of resources required matters, but the location of those resources as well. If an application consists of multiple tasks, they may also heavily rely on communication. The communication channels between tasks thus have to guarantee enough bandwidth with acceptable latency. This spatial factor increases the complexity of the resource allocation problem, making established scheduling algorithms unsuitable for this job.

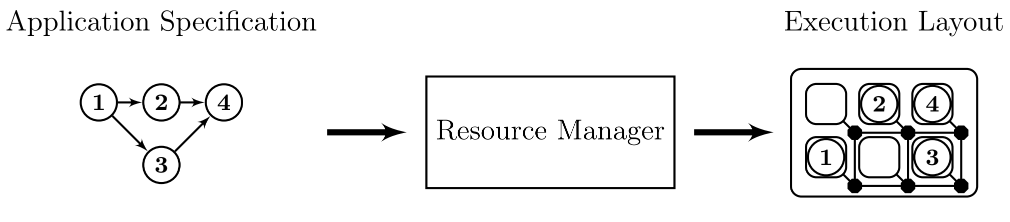


Figure 1.1: Context of run-time spatial resource management.

When successful, the resource manager provides an *execution layout* describing the amount and location of the resources allocated to that application. If no feasible execution layout can be found, the application must be rejected. Figure 1.1 shows the context of such an environment.

The described resource management process is commonly performed at design-time using semi-automatic tools, that are often still being researched. Aside from the necessity of run-time resource management, additional benefits can be identified:

- The ability to circumvent hardware faults (fault tolerance)
- Minimization of run-time system costs (energy consumption)
- Adaption to user demands (quality of service)
- Not restricted to specific combinations of applications (use-case flexibility)

The combination of these advantages with new hardware technology cause many research projects to focus on future MPSoCs.

1.1.1 The CRISP project

Within the Cutting edge Reconfigurable ICs for Stream Processing (CRISP) project [7], an MPSoC is being developed. The aim of this project is to create a scalable, reconfigurable and heterogeneous architecture that is designed to run streaming applications. Some software to support the hardware is required to deliver a working platform. The project specifically requested a resource manager, integrated in a Linux operating system, capable of running on an ARM-based processor. For demonstration purposes, it was also requested to look into methods to visualize the state of the MPSoC platform.

1.2 Research approach and methods

In this thesis, we investigate *spatial resource management* algorithms for run-time usage, targeting heterogeneous MPSoCs. These algorithms must run within a limited execution environment on the target platform. The main question in the evaluation of algorithms is what effort is allowed to derive an execution layout at run-time. A good understanding of the relation between the effort of the resource manager and the quality of the execution layout is desired.

Therefore, the research goal of this thesis is to create a resource manager that consists of a workflow from application specification to execution layout, as sketched in Figure 1.1. We propose a taxonomy of various phases in resource management, including a validation of the execution layout before the application is admitted to the platform; this guarantees that a minimum performance level is reached. We implement a modular prototype that allows for each phase to switch between algorithms with different approaches and various MPSoC platforms. As the number of parameters, the various algorithmic approaches, and different applications and platforms can be overwhelming, we limit this research to some reasonable alternatives.

Results are obtained using a selection of applications and three different platforms. These results are not only quantitative, as in the run-time of the resource manager, but also qualitative. For a better understanding of the resulting execution layouts, we examine possibilities to visualize the state of a platform.

1.3 Structure of the thesis

A state-of-the-art survey of related resource management algorithms is described in Chapter 2. A taxonomy for these algorithms is given, which splits the resource management process into multiple phases. For each phase, we integrate ideas from related work with our own insight and requirements. In Chapter 3, a coherent set of resource management heuristics is proposed. These heuristics form the first three phases of the resource management. Traversing these phases results in a preliminary execution layout. In Chapter 4, methods are described to check whether that execution layout meets the performance constraints of the application.

We created a prototype that implements the proposed algorithms in a Linux kernel. Details of the prototype are given in Chapter 5. We present some test results using this prototype with various platforms and applications in Chapter 6. This thesis ends with some conclusions and recommendations for further research.

CHAPTER 2

On run-time spatial resource management

In Section 1.1, we stated the research motivation to develop algorithms for allocation of resources in a running MPSoC. In this chapter, we survey the state-of-the-art in resource management at run-time. We have special interest in solutions that are tailored to heterogeneous architectures, although we investigate approaches for homogeneous architectures as well. This survey is organized in terms of relevant themes, rather than as a project-by-project summary. The discussion is presented around a taxonomy for the algorithms found in literature.

In this chapter, we separate the preparations made at design-time from the operations performed at run-time. Design-time exploration of applications is briefly discussed in Section 2.1. This process results in a set of tasks, each annotated with their resource requirements and performance characteristics. Section 2.2 states the restrictions we apply to the run-time part of the resource management process. Then in Section 2.3, a taxonomy of phases in run-time resource management is proposed. This taxonomy is used in the rest of this chapter to discuss related work.

2.1 Design-time preparations

An application is modeled and implemented at design-time. At this stage, characteristics of an application can be analyzed, in order to specify the resource budgets required to gain certain performance levels. We require such a specification of each application, to be used as input for the run-time resource management algorithms. Therefore, we first identify what preparations we expect to be done at design-time.

Numerous reasons exist why programming parallel architectures is difficult. The first problem is to find and to identify the parallelism in an application, known as task decomposition. Other steps involved are resource mapping, dealing with communication and synchronization issues, and avoiding race conditions. None of these steps are required to program a sequential algorithm. Some related work looks into methods to automatically extract parallelism from sequential applications [4, 8], but manually developing applications that employ multiple processors efficiently is still more common.

At design-time, it must be determined what resources an application minimally requires to meet its performance constraints. In general, there is a trade-off between multiple, conflicting objectives, like performance levels, resource requirements and costs. *Multi-objective optimization* may lead to a set of locally optimal solutions in the trade-off.

Definition 1 ⌊Multi-objective optimization⌉ *Multi-objective optimization is the process of simultaneously optimizing two or more conflicting objectives, subject to constraints.*

The best solutions for multi-objective problems are called the Pareto set. For solutions from the Pareto set, it is not possible to select another solution to improve one of the objectives, without worsening at least one of the others [9]. Ykman-Couvreur et al. [10, 11] use design-time exploration to construct a Pareto set that contains multiple implementations of an application. We assume that in a similar way, design-time exploration can produce a Pareto set for each task within an application. Various implementations of a task may require a different architecture to run on, expressed in the resource requirements. Figure 2.1 shows that each implementation is characterized by an optimal combination of performance constraints, used platform resources and costs. The result of this design-time exploration can be used as input for a resource manager.

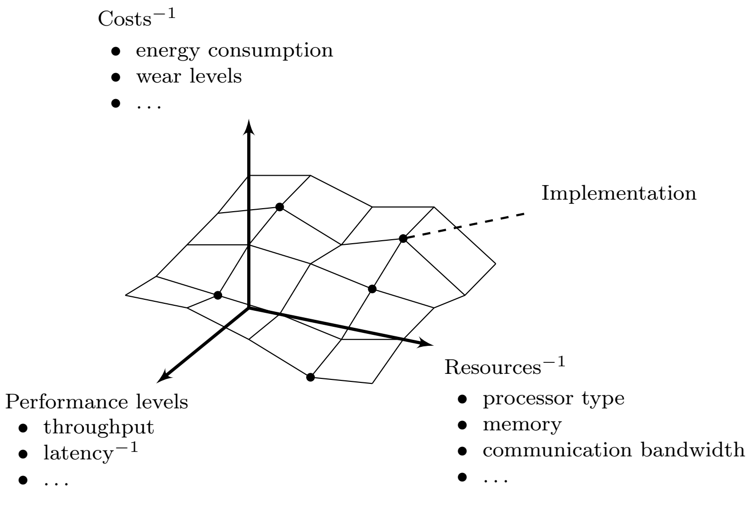


Figure 2.1: Design-time exploration of the application results in various implementations.

2.1.1 Application specification

We assume that after partitioning, every application can be expressed as a dataflow model, as they are commonly used as a design abstraction for parallel architectures [12]. More specifically, Synchronous Data Flow (SDF) graphs are widely used to specify applications, for example in [13, 14]. An entire application can be specified with an SDF graph, where the nodes in the graph represent individual *tasks*. The edges between the nodes model the *communication channels* between the tasks of the application. SDF graphs are a subclass of Petri nets. In this class, firing rules are independent of data values, so that the execution order can be determined at compile time. This ordering allows for a semi-static assignment scheduling strategy, where processor assignment can be done at the start of the application. These conditions are fit for a run-time resource manager. In Chapter 4, we reason about applications as SDF graphs, and the reader finds a more extensive introduction into SDF in Section 4.1.

2.1.2 Performance guarantees and multi-tasking

A non-preemptive scheduler can be used in single-tasking execution environments. Fortunately, non-preemptive scheduling algorithms are easier to implement than preemptive algorithms, and also have less run-time overhead [15]. Once the required resources are allocated to the application, it should run without interruption, and therefore meet the performance constraints posed on the application. This assumes that the resource budgets required by the application are correctly determined at design-time.

To support multi-tasking with real-time constraints, dataflow analysis must be used to derive schedules that guarantee that hard real-time tasks will meet their deadline. As design-time analysis of all possible use-cases is infeasible when the number of applications increases, the effects of run-time scheduling have to be considered. Schedulability analysis takes worst-case waiting times into account, resulting in a very pessimistic result [14, 16]. Kumar et al. compare various analysis techniques in [14]. They show that all the proposed techniques in the multi-processor domain that provide guarantees, have a low utilization. Kumar et al. [14] worked on a technique that improves utilization, by sacrificing the ability to provide hard real-time performance guarantees. Therefore, most known solutions that support multi-tasking do not provide hard real-time guarantees.

Wiggers et al. [17] show that the accuracy of the analysis can be improved by modeling run-time scheduling with latency-rate servers. The amount of improvement over a model with response times is not be given for the general case. Suitable scheduling algorithms for dataflow analysis are Time-Division Multiplex (TDM) and round-robin, which are both latency-rate schedulers [16].

Although we state that a multi-tasking environment adds an undesired complexity in the context of this research, we must allow applications to share resources where possible, because otherwise the flexibility and efficiency of the system would be too low. One can think of shared memory, communication lanes and other system support functions. The resource manager must arbitrate over these resources, and grant applications access to them to fulfill their resource requirements.

2.2 Assumptions for spatial resource management

The resource manager usually runs on one of the Processing Element (PE) of the MPSoC, having authority over the other resources in the system. Work presented in [18] states that centralized Run-time Spatial Resource Management (RSRM) has disadvantages in the context of hundreds or even thousands of cores that may be integrated in a System on Chip (SoC)[1]. Arguments are scalability issues, a single point of failure and communication hot-spots.

In [18], Faruque et al. propose a distributed solution that uses a two-step approach. An application is first forwarded to a cluster with enough available resources, and then the cluster agent has to deal with a problem similar to the general, centralized approach. While a distributed solution for resource management may be required for future designs, we first have to find a good approach to solve the problem for small systems (i.e. a cluster).

When a new application arrives, the RSRM must select suitable implementations from each Pareto set of the application's tasks. Picking the most suitable implementation depends on the current state of the MPSoC, and the optimization objectives the end-user has in mind. The resource manager determines if the resource requirements of the application can be fulfilled.

With arbitrary intervals, applications are incrementally added to the MPSoC, as done in [13, 19]. The resource manager must isolate the tasks that are already present on the system, from the unmanaged application. A resource management policy, mentioned in [14, 20], should enforce such conditions:

- *admission control* – an application is only allowed to start if the system can allocate, upon request, the resource budget it requires to meet its performance constraints.

- *guaranteed resource provisions* – the access of a running task to its allocated resources cannot be denied by any other task.

If an application can not be added to the system without violation of this policy, the application must be rejected. To resolve a failure, either the application or the platform state have to be changed. Changing the resource requirements of the application will be discussed in Section 2.3.1. Altering the platform state can be done by migration of running tasks.

2.2.1 Task migration

In the long run, it may be beneficial for the resource manager to be able to migrate tasks from one PE to another. Such a migration may result in less operational costs, allow Quality of Service (QoS) changes and load-balancing, or increase the success rate of future resource allocation attempts. Another reason may be that a PE must be cleared for dependability test-access [21].

1. Note that of today (2009), few heterogeneous SoCs are production ready, and those that are, contain no more that 10 processing elements.

New processor technology tends towards incorporating hardware support for *live migration* between processors [22]. To the best of our knowledge, this is only done in high performance processors, aiming at usage in virtualization servers. A hypervisor, also called Virtual Machine Monitor (VMM), initiates and controls the migration process. Even with hardware support, first a virtual machine is suspended; its state is captured and moved to another virtual machine, where it is resumed. In high performance systems, the latency involved is accepted, such that this migration process is considered to be "seamless". Also, the hypervisor plays an essential role to handle difference between types of processors properly. Currently, it is only possible to migrate virtual machines within a subset of x86-based processors. The `CPUID` instruction is used to query the processor's capabilities, which is processed by the hypervisor to avoid incompatibility issues [23].

In the way we consider MPSoC platforms, no hypervisor is running on the individual PEs. A migration solution then requires external intervention. For hard real-time applications, migration points should be defined so that the guarantees are not violated. A *migration point* in a task is a state in which all relevant information can be captured and transferred to another PE. Figure 2.2 shows a task migration process. After a migration request is issued, a *reaction time* is experienced until a migration point "M" is reached. At a migration point, the state "I" can be extracted and transported to another element. The performance of the task suffers from a *freeze time*, until the task resumes its execution.

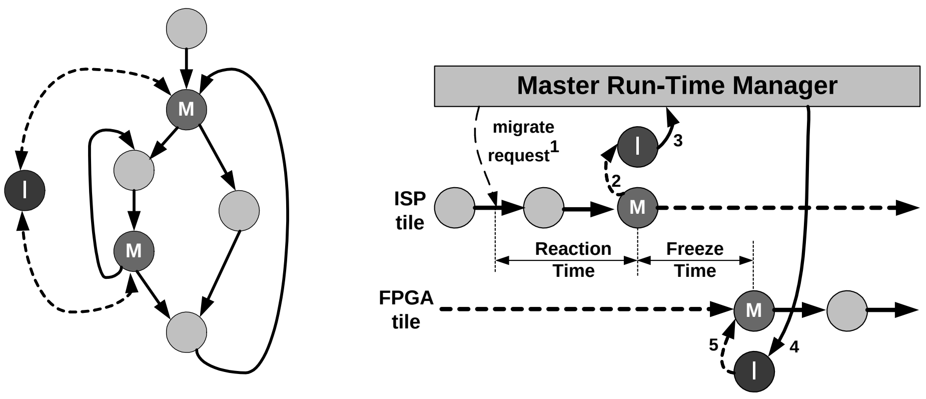


Figure 2.2: A migratable task containing two migration points M and an interrupted state I. At a migration point, the state is transferred to another PE, where the task continues [24].

Considering streaming applications, it is difficult, or even impossible, to integrate this approach in combination with performance guarantees. To reduce the delay between a migration request and a migration point, a task should have many migration points. The resulting overhead of checking for pending migration requests at every migration point is the main issue in task migration [24]. All software-based approaches have a significant performance overhead during normal execution. Nollet et al. [24] propose a hardware supported task migration technique for heterogeneous MPSoCs. This solution is hardware specific and lacks support in compilers and other tools.

In [25], experiments with the migration of various applications on an ARM architecture were analyzed. In the baseline case, the average downtime during migration was 22 seconds, ranging from almost 0 seconds to 95 seconds. The larger part of the downtime was devoted to serialization and deserialization of the task state. Zhang and Pande [25] developed a static compiler analysis method to decrease the migration overhead by pre-serializing the state in a program. For their test cases a little performance degradation of 2% was experienced. With this improvement, the system state still needs to be transmitted and deserialized, which still took, on average, 8.5 seconds, with 36 seconds being the worst case. Indicated by related work, task migration must be considered infeasible, and is therefore not considered in this research.

In the next section, we define multiple phases in resource management. We assume that once these phases are successfully traversed, an application will be executing on the target platform until the application is completed. As task migration is not possible, any decision made during the resource allocation process is fixed for the lifetime of the application.

2.3 Taxonomy for spatial resource management

Finding an optimal solution to add applications to an MPSoC is very complex. To avoid this complexity, most researchers split the problem into multiple sub-problems. In this section, we present a taxonomy for these sub-problems. For brevity reasons, a specific ordering is assumed, but different orderings exist as well. Each sub-problem is attacked iteratively in a separate phase, with reduced complexity compared to the original problem. Hölzenspies et al. [13] names such a work-flow *hierarchical search with iterative refinement.* The heuristics we found in literature can be classified into four phases:

1. **Binding:** for each task of the application, an implementation is selected that is able to execute the task with low cost or high performance. The resource requirements of the chosen implementations, including the type of architecture for execution, must be satisfied to admit the application.

2. **Mapping:** taking locality into account, specific resources are assigned to each task, such that the resource requirements of the implementation chosen in phase 1, are fulfilled.

3. **Routing:** for pairs of tasks that need to communicate, communication links are established between the processing elements assigned to them in phase 2. The amount of communication resources required is taken into account.

4. **Validation:** the performance constraints that are specified in the application specification are validated against the performance that is guaranteed by the execution layout derived in the previous phases.

Figure 2.3 illustrates this taxonomy. With the “partitioning” phase, we indicate the software development effort at design-time, as described in the beginning of this chapter. We only consider the *binding*, *mapping*, *routing* and *validation* phase to be part of a run-time resource manager. As a result of these phases, an execution layout is provided. Based on those instructions, platform specific software can configure the hardware accordingly and start the application, which we indicate with the “bootstrapping” phase.

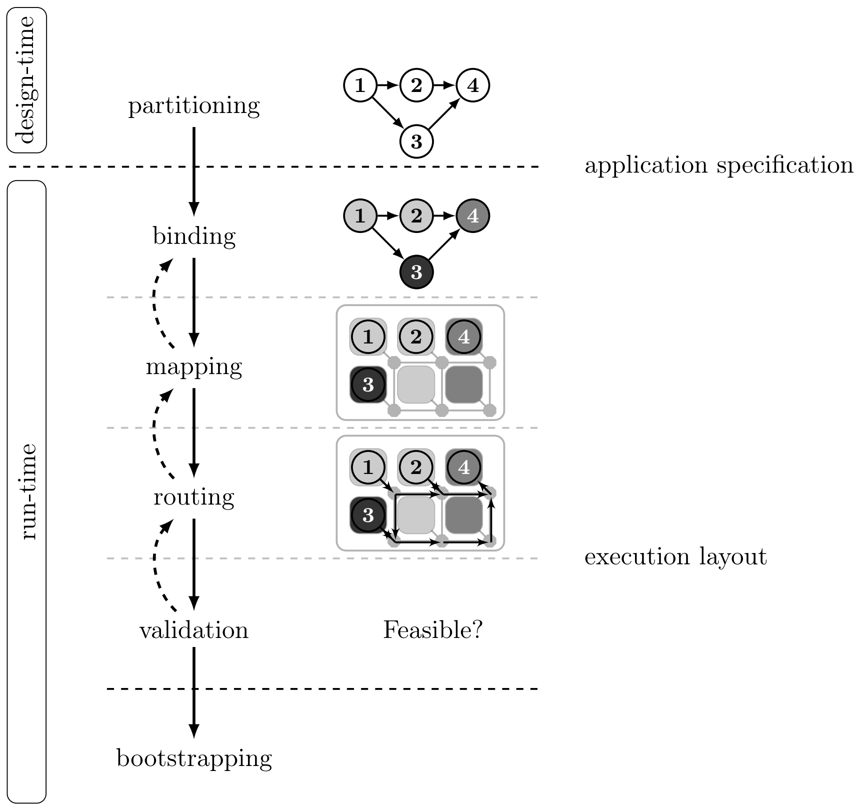


Figure 2.3: Various phases of a resource management algorithm. Note that application partitioning is typically done at design-time, while the other phases are performed at run-time.

In works that consider homogeneous architectures, the binding phase is irrelevant. That is also the case for applications that only have a single implementation for each task in the application, which may be considered as binding at design-time. All related work considers the mapping phase. In some approaches, the routing phase is not considered as part of the resource management problem or it is considered trivial, whereas in other works [19, 26, 27], the mapping phase is subordinated to the routing principle. The following sections categorize and describe the various approaches found in related work, according to this taxonomy.

2.3.1 Binding

The binding phase is the first step in the resource manager. Every task of the unbound application should be executable by at least one type of PE. From the set of unoccupied PEs, the resource manager should pick a PE type that is supported by the task. When PEs of multiple types fit, the Run-time Spatial Resource Manager (RSRM) makes a choice based on the task's desirability for some PE type. The most important part of heuristics in this phase of resource management are the criteria that express this desirability. This choice also defines the order in which tasks are bound, which is important for the final result, especially when the target system has few available PEs left.

To make the best binding choice possible, the entire set of tasks to be bound could be considered at once. This problem can mathematically be modeled as a Multi-dimensional Multiple-choice Knapsack Problem (MMKP). In [11], an MMKP formulation is used to obtain exact or near-optimal solutions for allocation of resources. However, an MMKP is NP-hard, making it not very suitable for run-time usage. Considering every possibility in the search space, Ykman-Couvreur et al. [11] developed a fast heuristic for solving the MMKP. Their heuristic reduces the multi-dimension resource requirements of each item into a single resource combination with a certain cost. Sorting them by *value per resource combination* can be considered as a notion of desirability. The last step is a greedy algorithm that optimizes the total value of the chosen resource combinations until there is no better feasible solution available. This greedy selection can be applied to any sorted optimization points. While this approach is derived from a mathematical formulation of the optimal solution, simpler sorting algorithms may lead to similar results. Other works use energy consumption [13, 18] or execution time [28] as an ordering criteria. Faruque et al. [18] specify the energy consumption of a PE type with two values, namely static and dynamic energy. Hölzenspies et al. [29] differentiate in energy consumption per architectural type, normalized by their processing capacity.

Generally, after ordering the unbound tasks by their desirability, the tasks are bound to the preferred architecture. We found no related work that explicitly mentions the case where the system runs out of resources during the binding phase. Kim et al. [30] evaluate heuristics for scheduling in a heterogeneous environment. They consider an oversubscribed system, such that not all tasks can be performed by the system. But the tasks considered in [30] are independent, so that no communication or dependency exists between tasks. A task that cannot be executed, does no harm to the other tasks in the system. Considering an application that consists of multiple dependent tasks, it is probably not useful to bind less than the whole application.

Binding at design-time

In [31], two types of tasks are considered: hardware tasks and software tasks. In this approach, the binding of tasks to a specific architecture is inherited from the application specification. Hardware tasks must be executed in reconfigurable logic or dedicated IPs, while software tasks can run on any processor. While there is an architectural difference in PE on a hardware level, the mapping algorithm of [31] only differentiates between hardware and software tasks.

Combining the binding and mapping phase

Nollet et al. [28] calculate the normalized task execution time variance (V_{Ni}) for each supported PE. Tasks with a high V_{Ni} value are very sensitive to which PE they are assigned to, and those tasks should preferably be bound before other tasks. The downside of this approach is that the execution time is decoupled from the other characteristics of task implementations. For example, in terms of energy consumption it may be beneficial to bind a task to a PE that takes more execution time, as long as it does not violate the task's constraints. The binding priority of the unbound task is defined by $V_{Ni} \times C_i$, where C_i is a communication requirements factor. Nollet et al. [28] sort the tasks to their priority, and order the PEs on their load and available communication resources. In a single iteration, the tasks are iteratively mapped to the best fitting PEs. In their case, reconfigurable hardware often has the highest preference. Therefore, an adhoc heuristic re-evaluates the choice, to avoid wasting this scarcely available PE type. We think it is difficult to apply this algorithm in a system with more architectural types.

2.3.2 Mapping

Many works that consider the mapping problem assume a homogeneous architecture. This clearly provides a high flexibility in assignment of tasks to PEs. Because heterogeneous systems pose more restrictions on the mapping procedure, it is presumably harder to obtain a good mapping. In general, the same optimization criteria may apply for both cases. Given the binding decision from the previous phase, optimization towards locality is generally used during the mapping phase. The intention is to minimize communication costs and to prevent a high fragmentation of the available resources after the mapping phase. In this section we describe various approaches in mapping applications onto an MPSoC.

Fragmentation

As tasks can not be migrated, a mapping is fixed for the lifetime of an application. This requires a mapping algorithm to consider fragmentation of the platform's resources. In Figure 2.4, two applications are mapped to a platform, using a simple linear allocation method. The first application is denoted with "x" and the second with ".". In each step, the resources required by one of the applications are allocated or de-allocated, while their precise resource requirements are varied over time. As can be seen in the figure, the platform is highly fragmentation after 25 time steps. The sum of the resources required by the application may be smaller than the available resources in the platform, but the routing may fail if the resources are too widely spread over the platform. Resource fragmentation must thus be taken into account, as future resource allocation attempts are more likely to fail when the fragmentation increases.

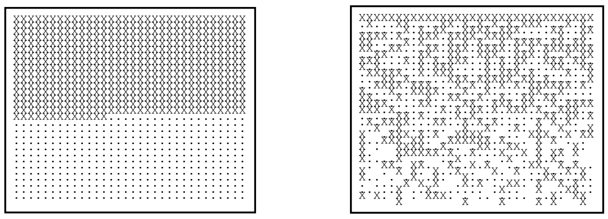

(a) Initial state **(b)** 25th reconfiguration

Figure 2.4: Resource fragmentation of a platform with two applications [32].

Platforms suffer both from internal and external fragmentation of available resources. A task may not employ an element to its fullest, leaving some percentage of the available resources unused. *Internal fragmentation* is the result of allocating a set of resources that does not precisely fit a certain amount of elements. *External fragmentation* results from allocation of non-adjacent PEs, as in Figure 2.4. In [33], these terms are expressed as ratios to the total amount of resources. In related work, fragmentation is not always considered, or sometimes only dealt with implicitly. For example in [20], algorithms are "adjusted to fill up partially filled PEs" to reduce internal resource fragmentation. The trade-off between fragmentation reduction and other objectives like minimal energy consumption or communication usage is not mentioned.

Cluster-based mapping

An implicit cluster-based approach is proposed by [31]. Each application has a "master" task that issues requests for the mapping of "slave" tasks. To initialize the system, only the master tasks are mapped. Carvalho et al. [31] illustrate that random mapping of these tasks may result in undesirable scenarios. For more careful mapping, a method is adopted that defines clusters of PEs for each application, and the initial master tasks are placed inside a cluster. In the example of Figure 2.5, the platform has been divided into nine clusters and each cluster is composed of nine PEs. Because these virtual clusters overlap, an application can use resources from another cluster. Unfortunately, this approach can not support more applications than the number of defined clusters, as this would require some sort of re-clustering strategy. Carvalho et al. [31] argue that this constraint is required to prevent *application deadlock* when the system runs out of resources.

Master tasks communicate with the *manager processor* (MP) to request and release resources. When a slave PE is requested, the mapping algorithm has to select a PE from within the application's cluster. This simplifies the mapping procedure, where mapping decisions in the overlapping zones of clusters are most interesting. This view of application mapping is very different from the general

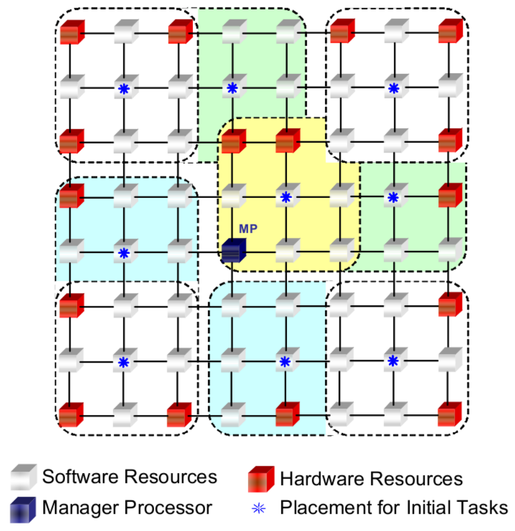


Figure 2.5: Cluster-based mapping. Dashed lines denote the cluster limits [31].

perspective. In general, the applications are mapped incrementally, but in this approach the tasks that compose the application are mapped incrementally, upon request of the master task.

The overall performance highly depends on the initial placement. The research of Carvalho et al. [31] lacks details of the method used for creation of clusters, and how to adapt to arrival and departure of applications. Also, some responsibility for bootstrapping the application is moved from the resource manager to the application's master task. This results in less distribution transparency and increased communication overhead, which are both not desirable.

Faruque et al. [18] also use a cluster-based approach to narrow the search space when the MPSoC is large. First, a cluster negotiation algorithm is started to find a cluster for the application. If no suitable cluster can be found, task migration requests are issued to free resources in a particular cluster. If no suitable cluster can be delivered, a re-clustering algorithm negotiates between the requesting cluster and its neighbors to exchange some PEs. This may or may not result in enough available resources to map the application. For resource management inside a cluster, Faruque et al. [18] use the algorithm of Hansson et al. [34]. That algorithm was originally developed for usage at design-time. A cost function is used to decide to which PE of a particular architectural type a task should be mapped. For a task t_i, the selection of PE n_j yields a certain cost $c(t_i, n_j)$:

$$\begin{aligned} c(t_i, n_j) &= \alpha(D(n_j) + bw_t(n_j) + RR(n_j)) \\ &+\beta \sum_{k \in T_{con,m}} d(k) vol(k) \end{aligned}$$

where $D(n_j)$ is the average distance of n_j to all other PEs of the cluster, $bw_t(n_j)$ is the total bandwidth requirement of the tasks on n_j, and $RR(n_j)$ is the resource requirement of n_j. The last factor is the sum over the set of all connected and mapped tasks $T_{con,m}$ of the Manhattan distance $d(k)$ between the mapped tasks, times their communication volume $vol(d)$ [18]. The unmapped tasks are sorted first by availability of PE types, and secondly by their bandwidth requirements. By evaluation of the cost function $c(t_i, n_j)$, a PE is selected iteratively for every task. The weights α and β in $c(t_i, n_j)$ are not given in [18], but they give some flexibility for optimization of the cost function for a specific system. Faruque et al. [18] claim to have a less complex mapping algorithm than the first algorithm described in this section (Carvalho et al. [31]), but they do not give results in terms of any other performance measure.

First Fit mapping

In some works [13, 20], the low-complexity First Fit (FF) algorithm is used for the initial mapping of tasks to PEs. After this greedy assignment, Hölzenspies et al. [13] try to improve the quality of the mapping by relocating tasks. Iteratively for each task, an attempt is made to find a better assignment by probing other locally available PEs of the same type. In this operation, it is allowed to swap two tasks that are mapped to the same PE type. When the assignment does not improve, the previous mapping is restored.

Moreira et al. [20] support multi-tasking by mapping tasks to intermediate Virtual Tiles (VTs). Then, each VT is assigned to a physical PE, taking communication channels into account. The First Fit with Clustering (FFC) algorithm of Moreira et al. [20] is more tailored to their architecture than a general FF algorithm. In their architecture, each router is connected to three PEs, making it desirable to map heavily communicating tasks to the same cluster (of three PEs). Although the FFC algorithm decreases bandwidth usage, it has an adverse effect on the mapping success rate when the system becomes saturated [20]. Therefore, Moreira et al. [20] use an unclustered version of their algorithm when FFC fails. We think that a more general mapping algorithm can also perform well on this architecture, if the internal communication links in the cluster are considered less costly than a single hop, increasing the preference to use local elements first. Doing so, the desirability remains to map heavily communicating tasks close to each other.

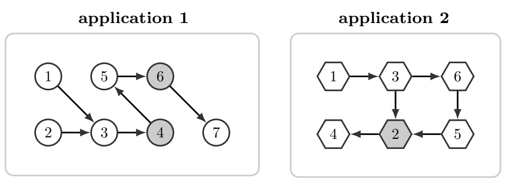
Mapping with region selection

Figure 2.6 shows two alternative mappings of two applications, where after mapping the second application, fragmentation of the available PEs becomes significant (see: Figure 2.6b). Chou and Marculescu [19] use a *region selection* procedure to allow for a more effective mapping for any subsequent applications (see: Figure 2.6c). The region is obtained by picking PEs around the initially chosen PE that have the lowest total value by the following definitions:

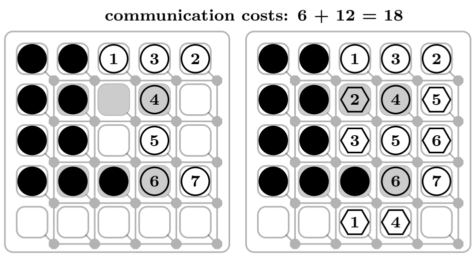
Definition 2 ⌊Dispersion factor⌉ *The* dispersion factor $D(e)$ *of a* PE *e, is defined as the number of idling neighbors of that* PE*.*

Definition 3 ⌊Centrifugal factor (C)⌉ *The* centrifugal factor $C(e)$ *of a* PE *e, is defined as the Manhattan distance between* PE *e and any* PE *at the border of the region occupied by the current application.*

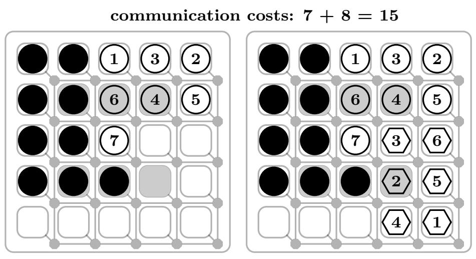
A PE that has most of its neighbors utilized, *i.e.* it has a small dispersion value, is very likely to be later isolated; selecting this PE to be utilized for the application mapping helps reducing the external fragmentation. A PE with the smaller centrifugal value is better suited for selection, since it increases locality of the tasks in the current application.



(a) Two applications that have to be mapped



(b) Greedy mapping approach.



(c) Mapping approach with region selection.

Figure 2.6: An example of two approaches in incremental application mapping on a heterogeneous MPSoC, where both the degree of fragmentation and communication costs play a role.

After a region has been determined by selecting the right amount of PEs according to these definitions, the resource manager attempts to find the best mapping within that region. Any of the procedures described earlier can be used to allocate PEs within the selection region to some optimal goal. Chou and Marculescu [19] propose the *node coloring* method, described in the next section, which aims at low inter-processor communication. The region selection algorithm shows a 25% reduction in inter-processor communication compared to the same mapping algorithm without the region selection strategy. The proposal of Chou and Marculescu [19] best suits dense application graphs, while a sparse graph could have a straight line of PEs as a very good solution. Thus, for optimizing inter-processor communication, it is sufficient (and more general) to minimize the distance between neighboring nodes, like in [13]. Unfortunately, the authors do not evaluate the mapping success rate, which we expect to increase when the degree of fragmentation is lowered.

Node coloring

Chou and Marculescu [19] use a node coloring technique to map an application on the PEs of the selected region. Note that colors used are unrelated to Figure 2.6, but indicate the progress of the algorithm. Every task that needs to be mapped is initially white. When a suitable PE is found, it is grayed, and is made black when the allocation is fixed. Tasks that have to be mapped are sorted per PE type, and ordered by decreasing communication demands. Starting with the first task of the smallest PE set, select available PEs of the correct type to map this task. Then the task is colored gray. If any neighbor (in the application graph) is either gray or black, the task is assigned to a specific PE, such that the distance to its neighbors is minimized. Afterwards this task is colored black.

2.3.3 Routing

After each task has been mapped to an element in the platform, we have to make sure that the positions of those elements allow tasks to communicate with each other. Communication links between tasks may be subject to latency constraints and bandwidth requirements. The communication data must be routed through the network in such a way that these requirements are met. If larger communication volumes are spatially clustered, non-optimal routes may have to be used to provide the required amount of communication resources. If tasks have a non-constant communication volume during their lifetime, routing decisions have to be made in both the spatial and the temporal domain.

When the capacity of a physical link between two routers is depleted, an alternative, probably longer route may work around this network congestion. This is a suitable solution to fulfill the bandwidth requirements. However, latency constraints are very sensitive to longer routes, and therefore should be given a higher priority over bandwidth constraints. Srinivasan and Chatha [27] have a priority based policy in their solution for design-time routing on Network on Chip (NoC) architectures. Their approach ensures that links with tight latencies are given priority over links with high bandwidth requirements (and more relaxed latencies). Every communication link c that must be routed is ordered to descending priorities:

$$\rho(c) = \frac{bandwidth(c)}{latency(c)^k}$$

An integer k is determined, such that the communication link with the highest bandwidth requirement, c_i, has a lower priority than the link with the tightest latency constraint, c_j:

$$\frac{bandwidth(c_i)}{latency(c_i)^k} \leq \frac{bandwidth(c_j)}{latency(c_j)^k} \tag{2.1}$$

Alternatively, Chou and Marculescu [19] define an architecture that has a separate *data* and *control* network. These networks are separated to ensure that data transmission does not interfere with the control messages generated by the Operating System (OS), which are considered to be more sensitive to latency. In their algorithm, the routing phase is not considered, because in the mapping phase it is assumed that PEs are only selected when the bandwidth requirements are met. Chou and Marculescu [19] do not explain how this check is performed.

In [31], the occupation of the NoC is measured using five mapping heuristics. To compare the approaches, the NoC topology, the routing algorithm and the communication delays are modeled. The best performing mapping heuristic is “Path Load”, which minimizes the occupation of the internal links of the NoC for each new task inserted in the system.

Moreira et al. [20] use a network graph model that incorporates time slots. A router consists of T nodes, where each node represents a time slot. Physically connected routers are modeled by connecting the time slots of the routers in the graph, shifting 1 slot to represent the 1-slot delay between routers. This approach reduces path finding and slot allocation to a single problem, at the cost of a larger and denser network graph. The problem of finding several paths in this network graph is known as the *Directed Edge-Disjoint Paths Problem.* Because this problem is NP-complete, Moreira et al. [20] use the *Shortest Path* approach instead, finding one route at a time instead of all paths at once. What percentage of the bandwidth can be reserved depends on the control overhead introduced by the resource manager.

NoC architectures

A NoC has a limited number of *physical channels* between each router. For example in [35], two routers are connected by two unidirectional links in opposite direction. There is often a demand for more than one communication channel between two routers. Like we discussed in the previous paragraph, time-sharing can be used to split communication resources between multiple applications. Doing resource management at run-time requires accounting of all time slots distributed over the interconnect at a central point.

In [35], a physical channel is time-shared using *virtual channels.* The resource manager allocates for each communication channel a virtual channel through each router in the communication route. The router arbitrates over a physical channel, splitting the available bandwidth between the allocated virtual channels. This arbitration can be implemented for example with round-robin, optionally with different budgets per virtual channel.

Due to hardware restrictions, there are a limited number of virtual channels per physical channel, the number of connections is also still limited. Kavaldjiev et al. [35] propose the “GuarVC” NoC router, and they state that 4 virtual channels per physical link is a good choice when considering router area and communication throughput [36] (see Figure 2.7 for a schematic of the “GuarVC” architecture). Each virtual channel can be configured to provide *guaranteed throughput* (GT) or *best effort* (BE) services. A GT channel should provide services to a single connection only, so that a certain bandwidth with a maximum latency can be guaranteed. Less demanding traffic can use a BE channel, that can be shared adhoc among multiple connections.

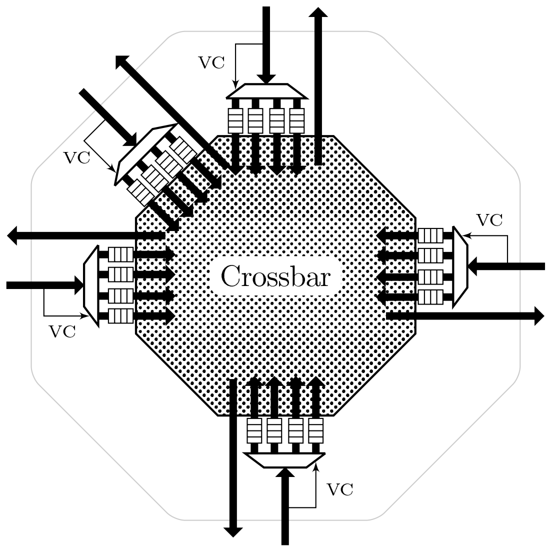


Figure 2.7: The "GuarVC" NoC router.

2.3.4 Validation

Even if enough communication resources were allocated to the application, the question remains whether the application performs well enough. We use the validation phase to validate the performance of the execution layout against the performance requirements of the application. This phase may be skipped by applications that do not have any performance constraints.

In [14, 20], an application is specified as an SDF graph. Kumar et al. [14] evaluate the performance of an application mapped to a MPSoC platform. A combined model of the platform and the application can generate performance figures, such as throughput and processor utilization. These results are obtained at design-time, and are used to create a resource manager. The resource manager then monitors the applications at run-time, and intervenes to ensure that all the applications are able to meet their throughput requirements. It is however not guaranteed that every constraint is honored at all times.

All related work on validation methods are still used at design-time. In Chapter 4 we discuss some related work that we use to realize the validation phase at run-time. That approach uses SDF graphs as well.

2.4 Optimization criteria

For comparison of the proposed algorithms in various works, a performance measure has to be defined. In some works [11, 18], computational effort is regarded as a performance measure. However, we consider algorithm complexity a design constraint like in [27], rather than an optimization criteria. An acceptable delay for the application to start executing on the system is at least 10ms, as this is the order of magnitude to start a new application in a Linux OS [11, 19]. This is due to the interrupt timer being set to 100 Hz. Depending on the target platform, this tight constraint may be relaxed. An upper bound on the algorithm complexity is that a solution must be computed faster than the anticipated average arrival rate of new applications [30]. As this period is relatively long in embedded systems, it is also not a realistic estimation of the acceptable delay that may be introduced by the resource manager. We claim that no universal constraint can be formulated in terms of computational complexity.

If it is possible to fulfill the request to execute a certain application, the system has to allocate the resources to that application, no matter what costs are involved. At the point where the system becomes saturated, both the availability of resources and the fragmentation degree of resources determines whether an application is still mappable. Benchmarking the algorithm with numerous scenarios, as done in [20, 27, 28], gives an estimation of the *mapping success rate*. We think that this rate is the most important factor for measuring the quality of the solution. Given a constant success rate, we can optimize the solution towards other performance measures, such as *energy consumption*.

We mentioned in this chapter that energy consumption is the main advantage of heterogeneous platforms. However, estimating energy consumption is very complex, due to the many relevant factors [37]. This is illustrated by the fact that all works that consider energy consumption use different energy consumption models. For example, Wolkotte et al. present in [38] some energy models that compare two NoC architectures with a shared-bus architecture. The derivation of those models shows that the final equation is highly dependent on the architecture and the technology of implementation. But once such an equation is derived, it is rather simple to use. However, if we do not want to know absolute energy consumption values, we can simplify the energy model in [38] by stating that less hops in the NoC cost less energy, which is trivial. Unfortunately, for the most important decision that takes energy consumption into account, the binding decision, it is unclear what energy model could be used as an optimization criterion.

Altogether, it is hard to compare the various approaches described in this chapter in qualitative terms. We think that there is a need for a generally applicable algorithm, that can be fine-tuned by specific objective functions. In the next chapter we describe heuristics for each resource management phase. At various points, these heuristics can be instructed to optimize towards some objective, such as high performance or low energy consumption.

CHAPTER 3

Heuristics for spatial resource management

The previous chapter discussed related work on spatial resource management. Following the decomposition of spatial resource management into phases, we propose a coherent set of heuristics for the implementation of a resource manager. Where relevant, we refer to the previous chapter to show similarities between this work and related work.

In the first part of this chapter we introduce some notation and assumptions about platforms and applications. We model both a platform and an application as a directed graph, but we need a distinct notation for both graph types to show relationships between them. Using this notation, we give specific algorithms that implement the binding, mapping and routing phase of spatial resource management.

3.1 Hardware platforms

In this section we describe the MPSoC platforms that can be managed by the resource manager. We do not want to limit our research to a specific platform. Therefore, we introduce an abstraction of a platform and its resources as a graph. We think of a platform as a number of *elements*, each capable of performing some task. Each element has a *communication interface* to the *interconnect* of the platform. Within the interconnect, *routers* are able to direct traffic along communication paths. Various types of communication architectures are possible, but an interconnect always has to provide predictable services. We define the interconnect of a platform as a directed graph $\mathcal{N} = \langle \mathcal{R}, \mathcal{L} \rangle$, where $\mathcal{R}$ is a set of *routers*, and $\mathcal{L} \subseteq \mathcal{R} \times \mathcal{R}$ represents the *communication links* between routers. The set of elements $\mathcal{E}$ interfaces with the interconnect through their *communication interfaces* $\mathcal{F} \subseteq \mathcal{E} \times \mathcal{R}$. Combining all the components, we obtain a platform graph $\mathcal{P} = \langle \mathcal{E} \cup \mathcal{R}, \mathcal{L} \cup \mathcal{F} \rangle$.

Each component in such a platform provides a limited amount of resources. Like in [29], we express those resources as *capacity vectors.* For each component, a capacity vector $C(x)$ indicates the remaining resource provisions of component x. All capacity vectors for the same kind of components (elements, routers, lanes and interfaces) have the same dimension. These dimensions are orthogonal, i.e. the members of a capacity vector are independent. Thus, in our definition of a platform, we have the capacity vectors $C_{\mathcal{E}}(e)$, $C_{\mathcal{R}}(r)$, $C_{\mathcal{L}}(l)$ and $C_{\mathcal{F}}(f)$, denoting the capacities of an element e, router r, communication link l and communication interface f respectively.

Architecturally different hardware components are used to compose heterogeneous platforms, or to build multiple different platforms. Each type of hardware component may have different capabilities. A *microarchitecture* is the realization of an Instruction Set Architecture (ISA) onto a processor. Processors with the same ISA are capable of executing the same tasks, not considering external restrictions such as limited memory capacities. The performance of a task might vary among microarchitectures with the same ISA.

Definition 4 ⌊Architecture⌉ *Let $\mathcal{X}$ be a finite set of different microarchitectures. We define the subset of elements $\mathcal{E}_\tau \subseteq \mathcal{E}$ to be of architectural type $\tau \in \mathcal{X}$. Likewise, heterogeneous interconnects can be modeled with architecturally different routers $\mathcal{R}_\tau$.*

Within the set $\mathcal{E}_\tau, \mathcal{R}_\tau, \mathcal{F}_\tau$ or $\mathcal{L}_\tau$, a component may have distinct amounts of remaining resource capacities, caused by design choices, asymmetric resource allocation or hardware failures. Whether a platform can support an additional application thus depends on the available architectures, remaining resource capacities, and their location. First we introduce the definition of an application, after which we continue to discuss the spatial resource management problem.

3.2 Applications

Each application defines a set of resource and performance constraints. Tasks pose resource constraints on the elements they are assigned to, while channels demand a certain bandwidth for communication between tasks. Altogether, they have to reach a minimum performance level.

We define an application to be either a single task, or multiple tasks connected with communication channels. An application can be formally described as a set $\mathcal{T}$ of *tasks* and a set $\mathcal{C}$ of *communication channels*: $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$. The application $\mathcal{A} = \langle \mathcal{T}, \mathcal{C} \rangle$ is a *weakly connected directed graph*, which means that no disjoint sub-graphs can exist in the application graph.

Definition 5 ⌊Connectivity of a directed graph⌉ *A directed graph is weakly connected if there is an undirected path between each pair of vertexes, and strongly connected if there is a directed path between every pair of vertices [39].*

For each task $t \in \mathcal{T}$, a nonempty set of *implementations* $\mathcal{I}(t)$ must be provided. The implementation of a task is an interpretation of its functionality, mapped to a specific architecture. Implementing the functionality of a task often entails a tradeoff between performance, energy consumption and development effort. A software developer can thus create multiple, reasonable implementations for a single task, shifting the tradeoff decisions from design-time to run-time. Other grounds to allow multiple implementations per task are ways to provide multiple QoS levels, and improved utilization by working around shortage of specific resources.

Analogue to the definition of elements, a set of implementations can be partitioned based on the architecture it requires for execution: $\mathcal{I}_\tau(t)$. Thus, the subset $\mathcal{I}_\tau(t) \subseteq \mathcal{I}(t)$ contains implementations that can be mapped to elements of architecture τ. An implementation i is then characterized by a *requirement vector* $R_\mathcal{I}(i)$, also known as demand vector [29, 40]. Such a requirement vector contains the minimal set of resources required to perform the task correctly. Opposed to the capacity vectors given in the previous section, we also have a requirement vector $R_\mathcal{C}(c)$ for each channel $c \in \mathcal{C}$.

Any arbitration over shared resources must ensure that every partioning of resources denoted by the capacity and requirement vectors ensures *composability.* A platform is defined to be composable if the functional and temporal behaviour of an application is not influenced by the presence or absence of other applications on the platform [41]. A composable platform allows us to redefine the resource allocation problem in terms of a single application and a set of remaining resources. This is an important assumption when we deal with the spatial resource management problem.

3.3 The spatial resource management problem

The resource management problem we address in this thesis is somewhat similar to the Multilevel Generalized Assignment Problem (MGAP), which was first described in [42]. The MGAP was formulated for a scenario where machines in a factory can perform jobs at different "levels". The same job may be executed with more or less accuracy, in more or less time, or with a larger or smaller energy consumption [43]. A specific costs value is associated with each job to machine assignment. The MGAP is a variation of the well-known Generalized Assignment Problem (GAP), but the MGAP itself is also used for a variety of problems [43–45]. A GAP is shown to be APX-hard [46], meaning that no polynomial-time approximation scheme exists that can find a solution within some fixed percentage of the optimal solution. The MGAP is formulated as a mathematical model as follows, using the context of our resource management problem:

$$\text{minimize} \quad \sum_{t\in\mathcal{T}} \sum_{e\in\mathcal{E}} \sum_{i\in\mathcal{I}(t)} c_{tei} x_{tei} \tag{3.1}$$

$$\text{subject to} \quad \sum_{e\in\mathcal{E}} \sum_{i\in\mathcal{I}(t)} x_{tei} = 1 \quad \forall t \in \mathcal{T}, \tag{3.2}$$

$$\sum_{e\in\mathcal{E}} \sum_{i\in\mathcal{I}(t)} R_\mathcal{I}(i) x_{tei} \le C_\mathcal{E}(e) \quad \forall t \in \mathcal{T}, \tag{3.3}$$

$$x_{tei} \in \{0,1\} \quad \forall t \in \mathcal{T}, \forall e \in \mathcal{E}, \forall i \in \mathcal{I}(t)$$

The binary decision variable x_{tei} is defined to be 1 if task t must be executed on element e using implementation i. The objective is to minimize the costs c_{tei} (Equation 3.1). Alternatively, this objective can be replaced to maximize a profit p_{tei}. The constraint in Equation 3.2 ensures that precisely one implementation and element combination is selected for each task. In Equation 3.3, the constraint expresses the limited resources available per element; the summed resource requirements of the implementations assigned to element e must not exceed the resource capacity provided by element e.

However, this problem formulation can only be considered as our resource management problem when we disregard the spatial factors. The interconnect $\mathcal{N}$ of a platform $\mathcal{P}$ introduces a topological relation between members of the set $\mathcal{E}$. Likewise, a topological relation exists between tasks within an application. These topological relations not only have an effect on the efficiency of a solution, but on the feasibility of a solution as well. Unfortunately, these relations are difficult to express in an MGAP formulation. Figure 3.1 illustrates the MGAP as a network problem, where arcs between tasks and elements have to be assigned an $x_{tei} \in \{0,1\}$, which indicates whether or not a task-with-implementation to element assignment exists. The dashed arcs within the sets of tasks and elements represent the spatial factors that need to be taken into account as well.

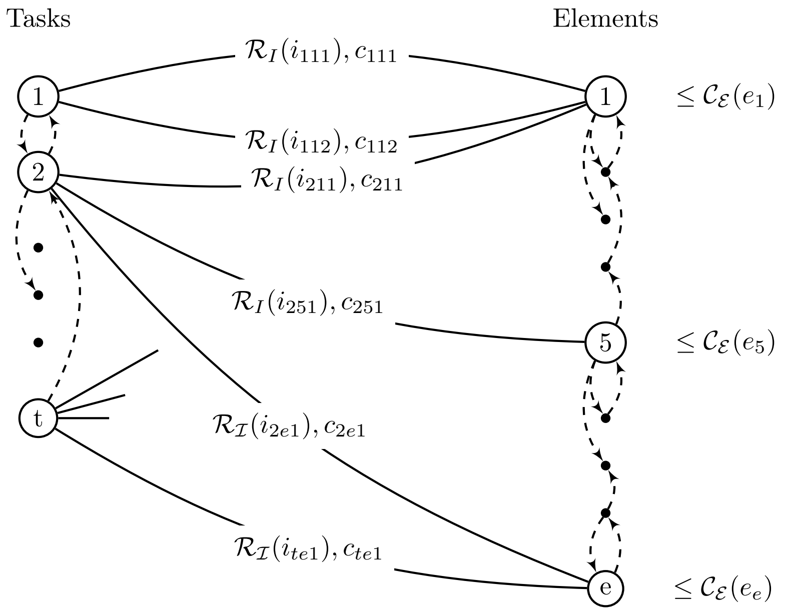

Tasks
Elements
$\mathcal{R}_I(i_{111}), c_{111}$
$\mathcal{R}_I(i_{112}), c_{112}$
$\mathcal{R}_I(i_{211}), c_{211}$
$\mathcal{R}_I(i_{251}), c_{251}$
$\mathcal{R}_\mathcal{I}(i_{2e1}), c_{2e1}$
$\mathcal{R}_\mathcal{I}(i_{te1}), c_{te1}$
1
2
t
1
5
e
$\leq \mathcal{C}_\mathcal{E}(e_1)$
$\leq \mathcal{C}_\mathcal{E}(e_5)$
$\leq \mathcal{C}_\mathcal{E}(e_e)$

Figure 3.1: The spatial resource management problem.

The MGAP in its original form has a one dimensional representation for size constraints. We use a vector representation for resource requirements and capacities. Vector bin packing is a related class of problems, but bin packing algorithms optimize towards using a minimal amount of bins (elements), which is not necessarily our goal.

Note that this model does not take *scheduling* of resources into account, as in time-sharing. Like in the original description of the MGAP in [42], we consider our problem to be a capacity planning problem. As we assume a composable system, providing sufficient resources implies that we can provide valid schedules to arbitrate over shared resources. Therefore, for simplicity and time constraints we leave scheduling over time out of this thesis.

Similar to the hierarchical search approach of [13], we first reduce the search space by choosing for each task the implementation that should be used to realize the application. This removes one dimension of the problem formulation in Equation 3.1-3.3, namely the set of implementations per task $\mathcal{I}(t)$. This process is called *binding* and is described in the next section.

3.4 Binding phase

We think of an application as a coherent set of tasks, having no reason to execute only a subset of tasks. For now, we see degradation of an application to a different QoS level as a redefinition of the task graph. So, if the binding phase fails to select an implementation for any task within the application, the application must be rejected. We define a *binding* function to make that choice.

Definition 6 ⌊Binding⌉ *Let $\mathcal{B} : \mathcal{T} \rightarrow \mathcal{I}$ be an injective function that selects for each task $t \in \mathcal{T}$ in application $\mathcal{A}$ one of its implementations $i \in \mathcal{I}(t)$.*

We want to avoid implementations to be selected that cannot be supported by the platform. For a task t, we only consider implementation $i \in \mathcal{I}(t)$, if there exists at least one element $e \in \mathcal{E}$, that is *available* for i.

Definition 7 ⌊Availability⌉ *An element e is available, writing $av(e, i)$, for implementation i if the architectural type matches: $\tau_e = \tau_i$, and the capacity vector $C_{\mathcal{E}}(e)$ suffices for the requirement vector $R_{\mathcal{I}}(i)$: $C_{\mathcal{E}}(e) - R_{\mathcal{I}}(i) \geq 0$.*

The availability requirement does not guarantee that enough resources can be allocated to the entire application. We need an extended notion of availability, where a set of elements (the platform) is available for a set of implementations (the application). A binding $\mathcal{B}$ should guarantee that an *adherent* mapping $\mathcal{M}$ exists:

Definition 8 ⌊Mapping, Adherency⌉ *Let $\mathcal{M} : \mathcal{I} \rightarrow \mathcal{E}$ be a function that maps an implementation to an element in platform $\mathcal{P}$. A mapping $\mathcal{M}$ is adherent if $\forall t \in \mathcal{T} |\ C_{\mathcal{E}}(\mathcal{M}(\mathcal{I}_t)) \geq 0$.*

So, we have to find an assignment from tasks to implementations, while considering the resource vectors. In other words, we want to reduce the MGAP of Equation 3.1-3.3 into a GAP by selecting for every task a single implementation.

An optimal solution in terms of costs requires that each task is assigned an implementation with minimal costs:

$$\forall i \in \mathcal{I}(t) \mid costs(i) \geq costs(\mathcal{B}(t)) \quad \forall t \in \mathcal{T}$$

Referring to Figure 3.1, for each task we remove outgoing arcs that are too expensive to use. After the binding phase, for every task a set of outgoing arcs may remain that effectively point to the same implementation, which can be executed by at least one element.

For reduction of the MGAP, we use a heuristic proposed in [47]. This heuristic uses a cost function, which allows us to be flexible in the optimization direction. 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 an element [47]. In other words, if the second best implementation is more expensive, the desirability to map the task increases. The definition of desirability can easily be exteneded to take additional factors into account, such as the scarcity of specific resources, or the resource requirements of an implementation. Experiments are required to gain better understanding of a good definition.

3.4.1 Implementation

To sustain the adherence of $\mathcal{M}$, we virtually assign the chosen implementation to the best fitting element, which is determined in the desirability calculation. The binding choice for the remaining tasks is affected by this assignment, as the available resource capacities in the system are reduced. This guarantees that after this phase (if all tasks are bound), at least one adherent mapping exists if we do not consider the communication restrictions.

Procedure 1 searches for the best possible choice of implementations for the tasks of an application $\mathcal{A}$. The heuristic takes into account whether an element is available for an implementation, according to definition 7, and how expensive it is to use that element. Once task t *is bound to* an implementation, we write $\mathcal{B}(t)$ to indicate that specific implementation.

The time complexity of the binding procedure is bound by $O(\mathcal{T} \times \mathcal{I} \times \mathcal{E})$. One worst-case scenario occurs when the platform is homogeneous in its elements. In heterogeneous systems we can instantly select a pre-arranged subset of platform elements, because each implementation is annotated with an architectural type. However, this does not necessarily give a reduced time complexity, as applications designed for heterogeneous platforms are likely to have more implementations per task than homogeneous applications. The asymptotic time complexity of *BindApplication* may be reduced to $O(\mathcal{I} \times \mathcal{E} log \mathcal{E} + \mathcal{I}^2)$, by sorting for each task, the implementations in decreasing desirability [47]. However, for small numbers this may not be benificial, due to increased memory usage.

Remark: If the presented approach is applied to applications consisting only of *elementary tasks*, where precisely one implementation per task is provided, some time could be saved by skipping the binding phase, which includes a check that

Procedure 1 BindApplication($\mathcal{A} = \langle \mathcal{T}, \mathcal{C} \rangle$, $\mathcal{P} = \langle \mathcal{E}, \mathcal{L} \rangle$)

```
Ensure: $\mathcal{T}$ ordered by increasing $|\mathcal{I}(t)|$
    $\mathcal{B} \leftarrow \emptyset$
    repeat
        $best \leftarrow \langle \emptyset, \emptyset, \emptyset, \infty, \infty \rangle$ {task, implementation, element, fstCost, sndCost}
        for all $t \in \mathcal{T}$ do
5:          $curr \leftarrow \langle \emptyset, \emptyset, \emptyset, \infty, \infty \rangle$
            for all $i \in \mathcal{I}(t)$ do
                for all $\{e \in \mathcal{E} \mid av(e, i)\}$ do
                    if BindingCost($i$, $e$) $< fstCost_{curr}$ then
                        $curr \leftarrow \langle t, i, e, \text{Cost}(i, e), fstCost_{curr} \rangle$
10:                 else if BindingCost($i$, $e$) $< sndCost$ then
                        $sndCost_{curr} \leftarrow$BindingCost($i$, $e$)
            if $curr = \emptyset$ then
                fail
            if $(sndCost_{curr} - fstCost_{curr}) > (sndCost_{best} - fstCost_{best})$ then
15:             $best \leftarrow curr$
        $\mathcal{B} \leftarrow \mathcal{B} \cup \langle \mathcal{T}_{best}, \mathcal{I}_{best} \rangle$
    until $best = \emptyset$
    return $\mathcal{B}$
```

determines if the remaining resource capacities suffice. However, the downside of this approach is that somewhere in the more expensive mapping phase the application may get rejected. This would not hurt the application itself, because it was already going to fail, but it may reversely affect the delay of other queued events.

3.5 Mapping phase

In this section we propose a heuristic to map the implementations bound to the tasks of application $\mathcal{A}$ to specific elements in platform $\mathcal{P}$. Now for each implementation $\mathcal{B}(t)$, the requirement vector $R_{\mathcal{I}}(\mathcal{B}(t))$ should be satisfied without over-utilizing any part of the system. But unlike Beck and Siewiorek [40], where a vector bin packing approach is used for a platform with a bus-based interconnection, we assume a network-on-chip, which is far more scalable and therefore introduces a spatial factor into the problem. To reason about spatial properties, we first introduce some terminology.

3.5.1 Spatial properties

In the context of platforms, we define the *degree* $d(e)$ of element $e \in \mathcal{E}$, as the number of links connected to element e. If we want to be more specific, then $d^{+}(e)$ denotes the *out-degree*, the number of links leaving element e, and the *in-degree* $d^{-}(e)$ the number of incoming links to element e Additionally, we define the *maximum degree* $\Delta(\mathcal{E})$ as the largest degree over all elements; the *minimum degree* $\delta(\mathcal{E})$ is the smallest. Likewise, the notion of degree can be applied to task graphs. Furthermore, we often reason about the neighbors of an element (or task), not including the element itself, which is known as the neighborhood.

Definition 9 ⌊Neighborhood⌉ *In a graph, the out-neighborhood $N^+(v)$ and in-neighborhood $N^-(v)$ of vertex v define the direct successors and predecessors of v, respectively. The neighborhood $N(v)$ of vertex v is then equal to the union of the out- and in-neighborhood. Vertex v itself is never a member of the set $N(v)$. The i^{th} neighborhood of v, written as $N_i(v)$ contains the elements that lie on a distance i of v, defining distance as the number of edges on the shortest path through the graph.*

The definition of neighborhood is used to partition the set of tasks $\mathcal{T}$ of an application. The next section describes how these smaller task sets are used to attack the mapping problem.

3.5.2 Divide-and-conquer

Reducing a problem to multiple smaller sub-problems is known as divide-and-conquer [48]. Our mapping heuristic breaks the mapping problem into sub-problems of variable size, depending on the density of the task graph. For now, assume that we select a random task t in the task graph. Each sub-problem i is then defined as a subset of tasks $\mathcal{T}_i \subseteq \mathcal{T}$, such that $\mathcal{T}_i$ is the i^{th} neighborhood of t. In other words, we do a breadth-first traversal of the task graph:

step 1) We group the tasks in sets with equal distance to the origin t.

In [19], a *node coloring* method is used to achieve a similar ordering. Maintaining the order of increasing distance i, each sub-problem is then resolved by these two steps:

step 2) Search the platform for enough elements $\mathcal{E}_i \subseteq \mathcal{E}$ close to $\mathcal{E}_{i-1}$, such that the resource requirements of the tasks in $\mathcal{T}_i$ are met.

step 3) Find a mapping $\mathcal{M}_i$ of the tasks in $\mathcal{T}_i$ to the elements in $\mathcal{E}_i$.

In these steps we have to reason about locality, and therefore a dependency exists between iterations of the algorithm. This *iterative mapping* approach is illustrated in Figure 3.2. The notation we use to describe this approach is defined in definition 10.

Definition 10 ⌊Iterative mapping⌉ *Let $\mathcal{T}_i \in \mathcal{T}$ be a set of tasks in iteration i, and $\mathcal{E}_i \subseteq \mathcal{E}$ likewise. Then $\mathcal{M}_i$ is a mapping from $\mathcal{T}_i$ to $\mathcal{E}_i$ in iteration i, such that $\forall i \neq j \mid \mathcal{M}_i \cap \mathcal{M}_j = \emptyset$. For convenience, we define $\mathcal{M}_i^* = \bigcup_{\forall j \leq i} \mathcal{M}_j$.*

Procedure *MapApplication* implements step 1 (§ 3.5.3), procedure *SearchElements* will search the platform for elements as in step 2 (§ 3.5.4), and finally procedure *SolveGAP* performs step 3 (§ 3.5.5). More details of these procedures are discussed in the following sections.

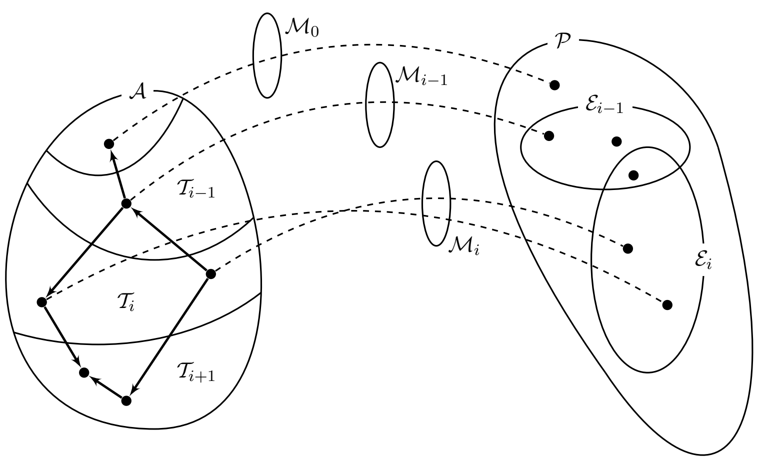


Figure 3.2: For every subset $\mathcal{T}_i$ of tasks in application $\mathcal{A}$, a subset $\mathcal{E}_i$ of the elements in platform $\mathcal{P}$ is selected to form mapping $\mathcal{M}_i$.

3.5.3 Task graph traversal

Mapping the tasks of an application consecutively may yield different solutions, depending on the ordering of the tasks. Known mapping algorithms start by selecting the most special tasks among the tasks in an application (see Section 2.3.2). It is shown for many cases that this approach results in a higher probability of generating a successful or better mapping. For homogeneous platforms, the most special task is the task with the highest communication volume. Dealing with heterogeneous architectures, tasks are often ordered by the scarcity of the required resources in the system. In [19], both metrics are used to determine the order of the tasks to be mapped.

When the binding of a task t limits the possible mappings of $\mathcal{B}(t)$ to a single option, we substantiate that mapping immediately. These restrictions affect the locality constraints for the other tasks that have multiple mapping options. We use this set of mapped tasks $\mathcal{T}_0$ as a starting point for the graph traversal.

Alternatively, when $\mathcal{T}_0$ is empty, we select a task $t \in \mathcal{T}$ that has the lowest degree $\delta(\mathcal{T})$ among the tasks in the application. Source or sink vertices in the task graph generally have a low degree. These tasks are special as well, because the location of an input and output of an application is often less flexible than processing tasks. Once a task $t \in \mathcal{T}$ is selected, it is considered to be the *root* of the task graph. Our heuristic requires a starting point in the platform as well, and therefore we immediately map task t to an element $e \in \mathcal{E}$. A cost function is evaluated on every available element in the system to find an element $e \in \mathcal{E}$ with a neighborhood that matches the requirements of the neighborhood of t, such that we minimize:

$$abs(\sum_{e' \in N(e)} C_{\mathcal{E}}(e') - \sum_{t' \in N(t)} R_{\mathcal{E}}(t'))$$

As task t has the lowest degree $\delta(\mathcal{T})$, an element $e \in \mathcal{E}$ will be selected for that task that has neighbors with low capacities. This is desired behavior, as such element has a higher probability to become isolated when the system utilization increases. Selecting such an element reduces the *external fragmentation* of the remaining resources in the system.

Definition 11 ⌊External fragmentation⌉ *Let α be the number of adjacent element pairs, where the first element is available, but the second element is not available for some $t \in \mathcal{T}$:*

$$\alpha = |\{e \in \mathcal{E}, e' \in N(e) \mid av(e,t) \wedge \neg av(e',t)\}|$$

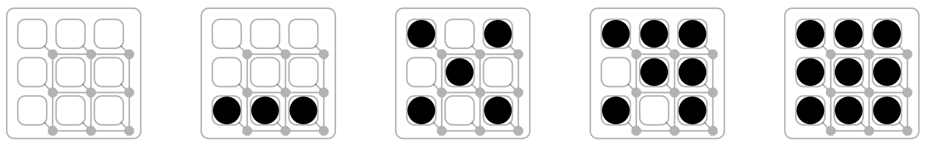
and θ the total number of adjacent elements in the system:

$$\theta = \frac{1}{2} \sum_{\forall e \in \mathcal{E}} |N(e)|$$

Then the external fragmentation of a platform $\mathcal{P}$ is expressed as:

$$f_{ext}(\mathcal{P}) = \frac{\alpha}{\theta}$$

With this definition, the external fragmentation of a completely empty or fully utilized platform is 0%. The worst external fragmentation possible is a chessboard pattern, illustrated with Figure 3.3c. According to the definition, the external fragmentation of such patterns is 100%. In [19], external fragmentation is formulated with the dispersion factor of elements (see Section 2.3.2).



(a) $f_{ext} = 0\%$ **(b)** $f_{ext} = 25\%$ **(c)** $f_{ext} = 100\%$ **(d)** $f_{ext} = 50\%$ **(e)** $f_{ext} = 0\%$

Figure 3.3: External fragmentation on some example platforms.

If for start task t_s an element e_s is selected, we add this pair to the first mapping of the application $\mathcal{M}_0$. We use element e_s as a starting point for the next iteration. Now iteratively, a set of unmapped tasks $\mathcal{T}_i$ is composed of the neighborhood of the tasks in the previous mapping (step 1):

$$\mathcal{T}_0 = \{t_s\}$$
$$\mathcal{T}_i = \{N(t) \mid t \in \mathcal{T}_{i-1}\}$$

The procedure *SearchElements* searches for a set of candidate elements to add to the mapping, starting to search from the elements of the previous mapping $\mathcal{E}_{i-1}$ (step 2). With sets $\mathcal{T}_i$ and $\mathcal{E}_i$, mapping $\mathcal{M}_i$ is constructed (step 3). If not all tasks in $\mathcal{T}_i$ can be mapped to an element in the system, the mapping procedure fails. This concept is listed in procedure 2.

Procedure 2 MapApplication($\mathcal{A} = \langle \mathcal{T}, \mathcal{C} \rangle, \mathcal{P} = \langle \mathcal{E}, \mathcal{L} \rangle$)

```
    $\mathcal{M}_0 \leftarrow \{\langle t, e \rangle \mid t \in \mathcal{T}, e \in \mathcal{E}, |\{e \mid av(e, \mathcal{B}(t))\}| = 1\}$
    if $\mathcal{M}_0 = \emptyset$ then
        $\mathcal{M}_0 \leftarrow \{\langle t, e \rangle \mid t \in \mathcal{T}, e \in \mathcal{E}, d(t) = \delta(\mathcal{T}), av(e, \mathcal{B}(t)), \min_{\forall e' \in \mathcal{E}} \text{Map.Costs}(\mathcal{A}, \mathcal{B}(t), e)\}$
    repeat
5:      $\mathcal{T}_i \leftarrow \{n \mid n \in N_i(t), t \in \mathcal{T}(\mathcal{M}_0)\}$
        $\mathcal{M}_i \leftarrow$ SearchElements($\mathcal{A}$, $\mathcal{P}$, $\mathcal{T}_i$, $\mathcal{M}_{i-1}$)
        if $\mathcal{T}_i \neq \mathcal{T}(\mathcal{M}_i)$ then
            fail
    until $\mathcal{T}_i = \emptyset$
10: return $\mathcal{M}_i^*$
```

3.5.4 Searching for elements

While traversing the task graph, we have to find for every $\mathcal{T}_i$ a set of suitable elements. In every iteration, we start searching in the neighbourhood of the elements that were allocated in the previous iteration. From the location of those elements a Breadth-First Search (BFS) is started. When the partial mapping $\mathcal{M}_{i-1}$ contains more than one element, we effectively start searching at depth one of the BFS if we consider a virtual root. In the BFS, we take the direction of communication channels between tasks into account. We also keep track of the distance between a discovered element and its origin.

Remark: Because we have multiple optimization objectives in the mapping phase, we do not stop searching for elements if we have exactly enough elements. This would facilitate only the cost minimization objective (like communication minimization), and make, for example, the fragmentation objective less effective. Thus, once we have discovered enough elements in the platform to map the tasks in $\mathcal{T}_i$, a single additional search step is performed. This results in a set of candidate elements that is likely to contain more elements than there are tasks in $\mathcal{T}_i$. Based on the ratio between computation and communication costs, the local search can be extended to gather even more candidate elements.

Procedure 3 SearchElements($\mathcal{A} = \langle \mathcal{T}, \mathcal{C} \rangle$, $\mathcal{P} = \langle \mathcal{E}, \mathcal{L} \rangle, \mathcal{T}_i, \mathcal{M}_{i-1} = \langle \mathcal{T}_{i-1}, \mathcal{E}_{i-1} \rangle$)

```
    $\mathcal{E}_{i-1}^{+} \leftarrow \{e_1 \mid \exists \langle t_1, t_2 \rangle \in \mathcal{C}, \langle t_1, e_1 \rangle \in \mathcal{M}_{i-1}, t_2 \in \mathcal{T}_i\}$
    $\mathcal{E}_{i-1}^{-} \leftarrow \{e_1 \mid \exists \langle t_2, t_1 \rangle \in \mathcal{C}, \langle t_1, e_1 \rangle \in \mathcal{M}_{i-1}, t_2 \in \mathcal{T}_i\}$
    for all $j \in \mathbb{N}$ do
        $\mathcal{E}_{i,j} \leftarrow \{n \mid n \in N_j^{\phi}(e), e \in \mathcal{E}_{i-1}^{\phi}, \phi \in \{+, -\}\}$
5:      if $\mathcal{E}_{i,j} = \emptyset$ then
            fail
        $\mathcal{M}_{i,j} \leftarrow$ SolveGAP($\mathcal{A}, \mathcal{T}_i, \mathcal{E}_i^*$)
        if $\mathcal{T}_i = \mathcal{T}(\mathcal{M}_{i,j-1})$ then
            return $\mathcal{M}_{i,j}$
```

3.5.5 Assigning tasks to elements

Up to this point, we described a search method that breaks the larger resource allocation problem into smaller sub-problems. We still have a set of tasks and elements, but much smaller than the entire application or platform. For each task $t \in \mathcal{T}_i$, an element $e \in \mathcal{E}_i$ has to be selected. Due to resource constraints, not all solutions are feasible. Additionally, we want a solution that respects our optimization criteria.

This problem is known in applied mathematics as the GAP, which is again NP-hard. A GAP describes a problem where a number of items have to be placed in a number of bins, in which both items and bins have a certain size. The size and the costs of an item may vary over bins. When the GAP has only one bin, the problem reduces to a knapsack problem. In our case, we consider elements to be bins with the resource capacities being the size of the bin. The tasks are the items that have to be placed in those bins, such that the resource requirements are met and a maximum profit is gained. In our case it is more convenient to reason about costs instead of profits, which can be transformed into one another. The GAP described below is similar to the problem in Equation 3.1-3.3, but with the binding choice already fixed.

$$\begin{aligned}
\text{minimize} \quad & \sum_{t\in\mathcal{T}}\sum_{e\in\mathcal{E}} c_{te}x_{te} && (3.4)\\
\text{subject to} \quad & \sum_{e\in\mathcal{E}} x_{te} \le 1 \quad \forall t \in \mathcal{T} && (3.5)\\
& \sum_{e\in\mathcal{E}} R_{\mathcal{I}}(\mathcal{I}_t)x_{te} \le C_{\mathcal{E}}(e) \quad \forall t \in \mathcal{T}, && (3.6)\\
& x_{ei} \in \{0,1\} \quad \forall t \in \mathcal{T}, \forall e \in \mathcal{E}
\end{aligned}$$

In [46], an efficient algorithm for GAPs is presented, with a time complexity of $O(\mathcal{E} \cdot k(\mathcal{T}) + \mathcal{E} \cdot \mathcal{T})$, where $k(\mathcal{T})$ indicates the time complexity of a subroutine that solves knapsack problems. A feasible solution is an *r-approximate* solution if it lies within a factor r of the optimal solution. This algorithm guarantees an $(1+\alpha)$-approximation solution, where α is the approximation ratio of the knapsack subroutine. These characteristics state that both the quality and time complexity of this approach mostly depend on the knapsack solver.

We implemented this approach to run in parallel with the *SearchElements* routine. At a lower level, a costs function is used that incorporates the optimization criteria. What follows is an explanation of this approach and its implementation.

Approximation of the generalized assignment problem

The algorithm of [46] lets every element choose a set of tasks by using a cost function. In *SolveGAP*, we iterate over the elements $\mathcal{E}_i$ that were discovered in *SearchElements*. For every $e \in \mathcal{E}_i$, we calculate for each $t \in \mathcal{T}_i$ the cost of mapping task t to element e. We put these values in a vector of length $|\mathcal{T}_i|$, and pass that vector on to a knapsack routine that picks the least expensive tasks for that specific element. The knapsack subroutine *Knapsack* selects for a single element a subset of tasks that minimizes the total costs. This approach works for both a single-tasking environment, as well for a multi-tasking environment.

Once the best task t for element e is determined using *Knapsack*, the cost of that combination is stored as $c^1(t)$. Any subsequent evaluations consider the cost improvement and compare that with the best known cost improvement. Thus, we only consider remapping a task t, if the cost improvement $c^1(t) - c^2(t)$ is positive. Most of the time, picking a yet unmapped task takes precedence over remapping a task to an other element.

The procedure *SolveGAP* gathers all the mappings proposed by *Knapsack*, and returns them to caller *SearchElements*. If insufficient elements were supplied to map every task, then *SearchElements* will invoke *SolveGAP* again, but with a larger set of elements. Figure 3.4 shows the growth of the set of elements $\mathcal{E}_i$, until *SolveGAP* manages to map all tasks $\mathcal{T}_i$. During this process, the set of tasks remains unchanged, allowing us to reuse the mappings and their associated costs, as determined in the previous invocation. Note that when the cost function depends on the state of the partial mapping $\mathcal{M}_i$, it must be re-evaluated every time $\mathcal{M}_i$ changes. This results in an increased complexity; the knapsack subroutine in Procedure 5 has time complexity $O(\mathcal{T}^2)$. If lower run-time is desired, a simpler costs function may result in a cost matrix that can be cached, as proposed in [46].

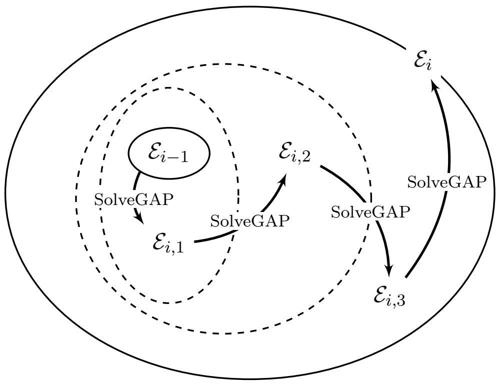


Figure 3.4: Starting from the elements of the previous iteration $\mathcal{E}_{i-1}$, the set of candidate elements $\mathcal{E}_i$ is expanded, until a feasible mapping $\mathcal{M}_i$ has been found.

Procedure 4 SolveGAP($\mathcal{A}, \mathcal{T}_i, \mathcal{E}_i$)

```
    $\mathcal{M}_i = \emptyset$
    $\forall t \in \mathcal{T}_i : c^1(t) \leftarrow \infty$
    for all $e \in \mathcal{E}_i$ do
        for all $t \in \mathcal{T}_i$ do
5:          $c^2(t) = Costs(\mathcal{A}, t, e)$
        $\mathcal{M}_i \leftarrow \mathcal{M}_i \cup Knapsack(\mathcal{T}_i, c, e)$
    return $\mathcal{M}_i$
```

Procedure 5 Knapsack($\mathcal{T}_i$, c, e)

```
    $\mathcal{M} \leftarrow \emptyset$
    **repeat**
        $best \leftarrow \langle \emptyset, 0 \rangle$ {task, costs}
        **for all** $t \in \mathcal{T}_i$ **do**
5:          **if** $c^1(t) - c^2(t) > best_{costs} \wedge av(e, \mathcal{B}(t))$ **then**
                $best \leftarrow \langle t, c^1(t) - c^2(t) \rangle$
        $c^1(best_{task}) \leftarrow best_{costs}$
        $\mathcal{M} \leftarrow \mathcal{M} \cup \langle t, e \rangle$
    **until** $best_{task} = \emptyset$
10: **return** $\mathcal{M}$
```

This approach gives a method to solve the GAP. The actual result depends mostly on the definition of the cost function. While it is hard to define a good cost function, it also provides flexibility by the possibility to switch between optimization criteria or to support multiple QoS levels. In our case, we assume that every task has to be mapped to avoid failure. We optimize towards both external fragmentation and communication minimization. This tradeoff can be influenced at run-time by changing weight parameters. The next section defines the cost function we use to make the actual decisions.

3.5.6 The cost function

A cost function is used to measure the quality of a task t to element e mapping. Currently, we take two cost factors into account; external resource fragmentation and communication distances.

The fragmentation factor makes better connected elements more costly to use, given by the size of the neighborhood of element e. These fragmentation costs are reduced when elements $e' \in N(e)$ in the neighborhood of element e are used by tasks $t' \in N(t)$ in the neighborhood of task t, by tasks of the same application $t' \in \mathcal{T}$, and finally by tasks from other applications. Note that tasks that are more related to the subject task, have more influence than unrelated tasks; the tasks $N(t)$ may occur in all of the three mentioned sets.

While searching the platform, we keep track of the distance between newly discovered elements and their predecessors $e' \in \mathcal{E}_{i-1}$. When the set $\mathcal{E}_{i-1}$ contains multiple elements, the sparse distance matrix may not contain all required distance information. Therefore, we initialize a distance matrix with a certain value UNKNOWN_DISTANCE_COST, that serves as a penalty, when a distance lookup requests a distance value that is unknown. If a distance value $d(e', e)$ is not available, then element e is relatively far away from one of its origin elements $e' \in \mathcal{E}_{i-1}$. Instead of employing a costly routing algorithm, we give a penalty to replace that unknown distance costs.

Figure 3.5 shows the iterations of a mapping process, where the top row shows the partitioning of tasks into iso-distant subsets (step 1), and the bottom row shows the final search state for elements (step 2). Indicated with dashed nodes, *SolveGAP* assigns $\mathcal{T}_i$ to the elements $\mathcal{E}_i$, composing mapping $\mathcal{M}_i$ (step 3).

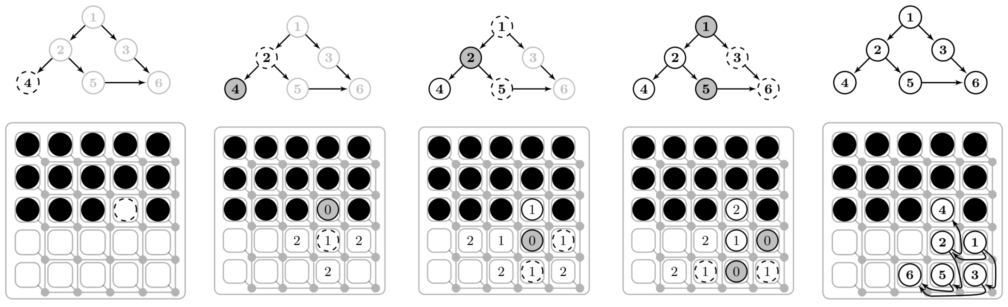
Function 6 Costs($\mathcal{A} = \langle \mathcal{T}, \mathcal{C} \rangle, t, e$)

```
if $\neg av(e, \mathcal{I}_t)$ then
    return $\infty$

for all $e' \in N(e)$ do
    $f \leftarrow f + 3 - |\{\langle t', e' \rangle \in \mathcal{M}_i^*, t' \in N(t)\}| - |\{\langle t', e' \rangle \in \mathcal{M}_i^*, t' \in \mathcal{T}\}| - |\{\langle t', e' \rangle \in \mathcal{M}\}|$

$c \leftarrow \sum_{t' \in N(t)} \begin{cases} d(e, e') & : \langle t', e' \rangle \in \mathcal{M} \\ \text{UNKNOWN\_DISTANCE\_COSTS} & : \langle t', e' \rangle \notin \mathcal{M} \end{cases}$

return $f \cdot W_{frag} + c \cdot W_{comm}$
```



(a) $i = 0$ **(b)** $i = 1$ **(c)** $i = 2$ **(d)** $i = 3$ **(e)** finished

Figure 3.5: Mapping state after each iteration in *MapApplication*, where gray nodes represent $\mathcal{T}_{i-1}$ and dashed nodes $\mathcal{T}_i$. On the bottom row the final search state of *SearchElements* is given.

3.6 Routing phase

There are three methods to communicate through an interconnect: unicast, broadcast and multicast. Unicast is used to setup point-to-point routes between two elements, and will often be the preferred communication method for tasks. Broadcast traffic is forwarded to every element in the system. This might be useful to implement some features in the operating system managing the platform, but is seldom used within real-time streaming applications. Multicast traffic is delivered to a subset of elements on the platform. As this method is the most difficult to implement, it is not always available in NoC interconnects. On the other hand, we could model a communication bus as a multicast router.

We consider each channel in the task graph to be a point-to-point communication channel, that requires an unicast route through the interconnect. After the mapping phase, the endpoints of these channels are known. For each channel we want to find the least expensive route between its two endpoints, which should both be a network interface of an element. Thus, a possible route can be expressed as $\langle f_1, l_1, \ldots, l_n, f_2 \rangle$, where $f_1, f_2 \subseteq \mathcal{F}$ and $l_{1..n} \subseteq \mathcal{L}$. The notion of cost is expressed with a cost function, which can, for example, optimize towards low energy consumption or low network congestion [35].

Considering all channels at the same time is rather complex. Like in [20], we search one route at a time, using the routing algorithm described in the next section. We do not order the channels, for example by decreasing bandwidth requirement, as in preliminary tests we experienced that the limited number of simultaneous connections causes more problems than the bandwidth limitation.

3.6.1 Routing algorithms

*A*search* [49] is a well known routing algorithm. This algorithm is based on Dijkstra's shortest path algorithm, extended with a heuristic that estimates the remaining distance to the destination. A*search proved to be optimal, if an *admissible* heuristic $h(n)$ can be defined. An admissible heuristic never overestimates the cost of reaching the destination; it must be conservative. The worst-case time complexity of A*search is exponential in the length of the solution. A disadvantage of A*search is that an exponential number of nodes must be remembered, resulting in high memory requirements.

We found it difficult to formulate a generic heuristic to estimate the remaining distance. For specific platforms, this would not have to be so hard. For example, in a regular mesh structured platform, coordinates can be used to quickly determine the remaining distance.

The routing algorithm we use is Uniform Cost Search (UCS), which employs Dijkstra's algorithm [50], but it terminates when the destination is reached. Dijkstra's shortest path algorithm can be considered as a special case of A*search, where the estimation heuristic is constant; $h(n) = 0$. This allowes us to make A*search optional; a platform specific heuristic can easily be plugged in to optimize the routing algorithms.

In [35], a simple breadth-first search algorithm is compared to Dijkstra's algorithm. No noticeable performance difference in term of successful routes and communication energy is experienced. Therefore, they recommend breadth-first search, as it has a lower computational complexity of $O(N)$ compared to $O(N^2)$ of Dijkstra's algorithm. We can implement breadth-first search as well, by omitting the ordering step of UCS.

3.6.2 Multicast routing

Multicast routing may be useful in some application classes, where tasks need to send identical data to multiple receivers at the same time. An example is the beamforming application that will be discussed in Chapter 6. If multicast routing is not employed, the same data is transferred over multiple point-to-point connections to all its receivers. This is not convenient, if all those receivers are located far away from their common source, but close to each other. Using multicast routing, the transceiver only sends the data once and, at a certain point in the network, the data is replicated and sent in different directions towards the receivers. If we place those *rendezvous points* strategically, we reduce the amount of allocated communication resources and we lower the energy consumption, provided that the energy overhead for multicast routing is not substantial.

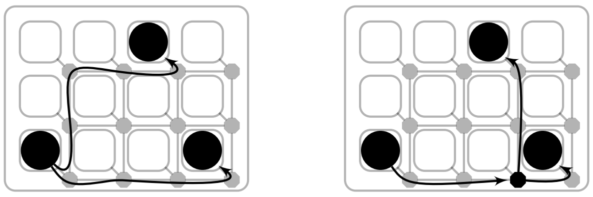


Figure 3.6: Modeling multicast routing in the task graph may result in an execution layout with less communication resources allocated.

At design-time, we have to specify the task graph of the application. If we know that two channels from a single task send identical data to multiple receivers, we may want to benefit from multicast routing. Therefore, we may add a "router task" r_1 to the task graph, creating an additional channel as well. This way the resource manager knows that the channels can be considered the same. As each task requires at least one implementation, we add an implementation to r_1 as well. This implementation requires a router with multicast capabilities. Additionally, we may add another implementation to r_1 as a fallback mechanism. This implementation must be executable on the same architecture as the source task. When multicast is not supported by the target platform, then the router task can be mapped on the same element as the transceiver, effectively reverting to the old situation by using multiple unicast connections.

Location of the rendezvous points

Determining the best location of the rendezvous point of multicast traffic is a problem similar to the mapping problem. Anticipating this problem, we propose to model multicast router tasks in the task graph as "normal tasks". With some minor adjustments we can integrate the mapping of those routers into the task mapping algorithms. We start by mapping all the non-router tasks using algorithm 2 to traverse the task graph. We skip any router task we find in the task graph. Afterwards, we iterate over the unmapped router tasks. As communication distance to neighbor tasks is taken into account in the mapping algorithms, the router is placed at a location that results in short routes relative to its neighbor tasks.

3.7 Execution layout

If all three phases described in this chapter succeed, an execution layout for the application is known. This execution layout guarantees that enough resources are allocated to every task and channel within the application. However, it is possible to derive an execution layout that contains resources that are unreachable from the operating system's perspective. There must be a path through the network that allows those resources to be configured. One way to deal with this problem is to assume that best effort traffic is always available. This requires that at least one virtual channel per physical link is configured for best effort services. Another approach is that the resource manager allocates a communication route from the operating system's processor to the application. This route can be used together with the communication resources allocated to the application for configuration purposes.

Even with these extra requirements, an execution layout may be infeasible due to the performance constraints of the application. If communication latencies would be zero, and processing elements would be infinitely fast, than the application executes without problems. As these assumptions are invalid, we have to validate the proposed execution layout against the performance constraints of the application. In the next chapter, we describe the last phase of spatial resource management; the constraint validation phase.

CHAPTER 4

Constraint validation

Streaming applications have a high potential to benefit from MPSoC platforms. Their most important constraint is usually the minimal throughput that should be achieved. However, in some application domains, such as communication, medical imaging and robotics, also the maximum allowable latency is a hard constraint. In [51], it is shown that it is not always possible to optimize both throughput and latency simultaneously. This implies that there is a trade-off between throughput and latency, which can be extensively analyzed at design-time to determine the most desirable scenario.

Based on such analysis, we assume that for each application a set of performance constraints is specified. When those performance constraints are too tight, or the resource requirements too high, an application is probably not suitable for the platform it is executed on. Also with a larger set of constraints, or tighter constraints, the probability that an execution layout is not feasible increases. For example, certain combinations of implementations may require too much resources, or communication channels may introduce too much latency.

In this chapter, we discuss methods that can verify whether an execution layout provides conditions for the application to meet its performance constraints. This knowledge is required to support predictable behavior of the application once it is being executed on the platform. The methods presented in this chapter use the application specification, as provided by an application engineer, to create a Synchronous Data Flow graph that models the structure and behavior of the application. The application specification only contains information about the application that is known at design-time.

In the following section, we first introduce the reader to some terminology and notation of SDF. Using SDF graphs, some modeling techniques are described that complement the application specification with details of the execution layout that describe the influence of the platform limitations on the behavior of the application [52]. Then in Section 4.5, an analysis method is discussed to determine the minimum throughput of such a graph, which is equal to the throughput of the corresponding application. We end this chapter by showing how both latency and throughput constraints can be checked on this graph simultaneously.

4.1 Synchronous Data Flow

Synchronous Data Flow (SDF) [53] is commonly used as a modeling technique for data flow applications. In an SDF graph, the nodes are called *actors*, and in the MPSoC domain they typically represent tasks in an application. The edges between actors model data or control dependencies, being the communication channels between tasks. Edges may contain *tokens*, which are per edge predefined units of data or control. At the start of an actor execution, called a *firing*, tokens are consumed from those edges. When the execution ends, tokens are produced on the outgoing edges. An actor is *enabled*, and thus allowed to execute, if enough tokens are available on its incoming edges. Therefore, an edge may contain *initial tokens* before any firing occurred.

Definition 12 ⌊Initial token placement⌉ *An SDF graph has an initial token placement function $\delta : \mathcal{C} \rightarrow \mathbb{N}$, that gives for each edge $c \in C$ in the graph, the amount of tokens it holds.*

An example SDF graph is given in Figure 4.1. For every pair of actors connected with an edge, the production and consumption *rates* of those actors are indicated at the start and end of that edge respectively. Also, the number of initial tokens on each edge is indicated.

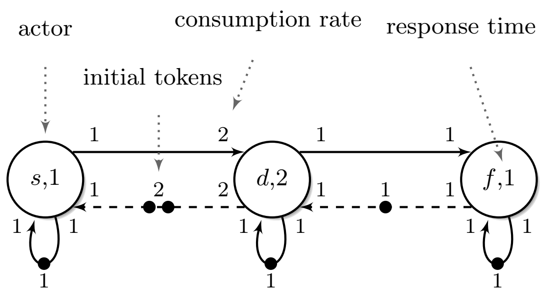


Figure 4.1: An example timed SDF graph.

4.1.1 Graph notation

Similar to the definition of an application in Section 3.2, an SDF graph $\mathcal{G} = \langle \mathcal{T}, \mathcal{C}, \delta \rangle$ consists of a set of actors $\mathcal{T}$, and a set of channels $\mathcal{C} \subseteq \mathcal{T}^2 \times \mathbb{N}^2$; additionally it contains the token placement function δ. The definition of a channel is extended with communication rates, such that $\langle s, d, p, c \rangle \in \mathcal{C}$ specifies that per firing, actor s produces p tokens on this channel, while actor d consumes c tokens. If the communication rates in an SDF graph are not properly balanced, the graph may deadlock, or an increasing amount of tokens may build up in a channel. This balance relation between actors is defined as the *repetition vector* of an SDF graph.

Definition 13 ⌊Repetition vector⌉ *The repetition vector q of an SDF graph $\mathcal{G} = \langle \mathcal{T}, \mathcal{C} \rangle$ is a vector of length $|\mathcal{T}|$, containing such values, that for each channel $\langle s, d, p, c \rangle \in \mathcal{C}$ from actor $s \in \mathcal{T}$ to $d \in \mathcal{T}$, $p \cdot q_s = c \cdot q_d$. A repetition vector is non-trivial if it has no zero entries.*

A non-trivial repetition vector of an SDF graph indicates that a finite schedule exists, that returns the graph to its original token placement [54]. We call an execution of such a schedule a *graph iteration.* For the example graph in Figure 4.1, the repetition vector q is given by $\langle 2, 1, 1 \rangle$. If a non-trivial repetition vector of an SDF graph $\mathcal{G}$ exists, then $\mathcal{G}$ is *consistent*, meaning that the graph does not deadlock and that the channels have bounded capacities. This ensures that the application that is modeled does not deadlock either, if all details concerning the execution of that application are correctly modeled in the SDF graph. Later in this chapter we describe various modeling techniques that result in a correct SDF graph.

We use SDF graphs to model applications, and most actors in the graph represent tasks within that application. A firing of an actor is therefore not instantaneous. In Figure 4.1, the response time of each actor is given as well. The response time is given by a function $\rho : \mathcal{T} \rightarrow \mathbb{N}$, that gives for each actor a the number of time units required to complete a firing. When resources are time-shared between multiple actors, the arbitration of a scheduler results in increased response times for the actors involved, compared to their Worst Case Execution Time (WCET) in isolation. For example, if for the execution of task t a budget R in a time interval Q is reserved, then the response time may be calculated from the execution time $x(\mathcal{I}_t)$ of its implementation:

$$\rho : t \rightarrow x(\mathcal{I}_t) + (Q - R) \cdot \left\lceil \frac{x(\mathcal{I}_t)}{R} \right\rceil$$

Combining the SDF graph of an application with the response times of the its tasks results in a *timed* SDF graph.

Definition 14 ⌊Timed SDF graph⌉ *A timed SDF graph $\mathcal{G} = \langle \mathcal{T}, \mathcal{C}, \delta, \rho \rangle$ is an SDF graph annotated with timing information. When an actor a fires, it takes $\rho(a)$ time units to complete. The function $\rho : \mathcal{T} \rightarrow \mathbb{N}$ gives the amount of time an actor $a \in \mathcal{T}$ requires to complete a firing.*

Paths and cycles in an SDF graph

Sometimes we need to argue about connectivity in a graph. The following terms are used throughout the rest of this chapter, analogously to [55]. A *path* through an SDF graph is a sequence of edges $\langle e_1, e_2, \ldots, e_n \rangle$. A path is *simple* if every edge is only traversed once. When the *source* and the *sink* node of the path are the same, the path is a *cycle.*

4.2 Throughput limitations

A timed SDF graph has certain analytical properties. If an SDF graph correctly models the layout of an application onto a platform, temporal analysis on the SDF graph gives bounds on the performance that will be achieved during execution of that application. Two interesting performance measures are the throughput of the application and the end-to-end latency in pairs of actors. Analysis techniques available in the SDF domain can be used to derive those properties. But before we resort to such analysis techniques, we have to make sure that the SDF graph models every aspect of the application that is subject to validation. In the this section, we describe the factors that limit the throughput of an application.

The most obvious throughput limitations of an application are the tasks within an application that have a non-zero response time. These limit the throughput of the application, as for each token some operation has to be performed. The response time of a task may also be larger than its execution time, when multiple tasks have to be scheduled on a single processing element.

Another closely related limitation is the latency of communication between tasks. Tasks may have to wait until all data required for processing is available on their input channels. This phenomenon may contribute to an overall decreased throughput. How much latency each channel has, depends on the assigned route, which is only known after the routing phase. In Section 4.4 we show how we adapt the SDF graph to model this limitation.

Probably the most researched throughput limitation of applications involves buffering. At design-time, an application engineer has to implement the functionality of each task within the application, including a communication infrastructure. One approach to establish communication (and synchronization) between tasks is to communicate through buffers. Circular buffers are used instead of FIFOs, as the amount of memory is limited. The application engineer has to compute the capacity of those *application buffers*. Buffering occurs in the network as well, as arbitration in the routers requires data to be stored for short intervals. How both types of buffering have to be modeled in the SDF graph will be discussed in the next sections.

4.3 Modeling buffering

Communication between tasks within the application can be realized in various ways. The amount of buffering can be programmed on a per task basis. If application buffers are defined, this must be correctly modeled in the application specification. Here, we give a specific method as an example implementation and modeling technique for application buffers. Various other methods may be suitable as well, but they may require a different representation in the SDF graph.

4.3.1 Streaming memory consistency model

Bijlsma et al. [56] propose a method to compute sufficient buffer capacities for tasks that communicate via circular buffers, guaranteeing deadlock free execution of the application. These circular buffers are located in shared memory, for example in a Scratch Pad Memory (SPM) located near a processing element. The producer initiates communication by writing into the consumer's SPM, resulting in low latency memory access for the consumer task. Figure 4.2 illustrates this approach.

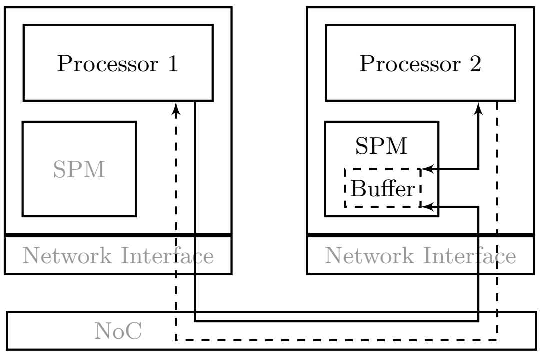


Figure 4.2: Streaming memory consistency model.

This *streaming memory consistency model* provides synchronization via *acquire* and *release* statements. A location in the buffer has to be acquired before it can be accessed and released after it has been written or read for the last time. Bijlsma et al. [56] introduce sliding windows to support rereading, out-of-order reading and writing and location skipping. The main advantage of this model (with sliding windows) is that it supports *posted writes*, where the producer can pipeline write operations without waiting on previous operations to complete first. This way, a higher throughput can be gained; especially on long communication paths. Note that this approach assumes that the underlying network is predictable and capable of maintaining ordering within a data-stream.

4.3.2 Modeling application buffers

Assuming the streaming memory consistency model of [56], for each "forward" channel a "backward" channel must be added to the graph to model the synchronization statements of the consumer. This way, back-pressure of the buffers in an application is dealt with explicitly. In Figure 4.1 these "backward" channels are modeled by dashed arrows to give better insight into the application, but otherwise being equal to "forward" channels. How a "backward" channel, i.e. the synchronization statements from the consumer to the producer task, is modeled depends on the implementation of the tasks.

A scenario, modeled by the example in Figure 4.1, could be that the producer (actor s) writes into a buffer of the consumer (actor d), having a buffer size of two tokens. After actor d has fired, the entire buffer is released at once, resulting in two tokens on the “backward” channel.

Because various scenarios are possible, the buffers in the application have to be modeled explicitly in the data flow graph. For each buffer in the application an extra channel is modeled, which in effect results in a throughput limitation. Therefore, this buffer has the same effect on the analysis on the SDF graph as it has on the behavior of the application.

4.3.3 Network buffering

We build upon the latency and throughput guarantees provided by the NoC, which depend on every arbitration point in the platform and upon the buffers in the Network Interfaces (NIs). If those buffers are not sufficient, guarantees may be violated. Simply providing large buffers is not desirable, as they are not energy efficient and occupy area in the chip. Therefore, many related papers aim at minimal buffer size estimation techniques for developing an application specific platform, such as in [57].

On the contrary, we deal with an already established platform, where the size of the buffers was fixed at design-time. Assuming static buffer sizes and predictable arbitration in the network, every path through the network guarantees a minimum throughput [35]. The minimum guaranteed throughput is determined by the component with the lowest throughput. So, if we ensure that a certain task in the application limits the throughput of the application, and not one of the communication channels, we can discard the influence of communication buffering on the application throughput. Buffering in the interconnect does introduce some additional communication latency. That will be discussed in the next section.

Concluding, an application demands of each communication channel from task t_1 to task t_2, a minimum throughput of the network path between the processor of t_1 and t_2. This approach facilitates the modeling of heterogeneous communication architectures. For example, an Advanced High-performance Bus connecting an ARM processor to a NoC may guarantee a certain throughput by considering worst-case response times of bus accesses. Some notes about the effects of run-time arbitration are found in Section 4.7.

4.4 Modeling communication latency

For efficiency reasons, an interconnect is not only shared in a spatial manner, but we allow time-sharing of communication links as well. Routers have to arbitrate between multiple channels that use the same communication link, while introducing some latency in every data stream. Some applications require a guaranteed upper bound on the end-to-end latency between a pair of tasks. We define *end-to-end latency* as the delay between the moment the first task produces a sample (or message), and the point in time at which the second task generates the corresponding output. Goddard and Jeffay [58] define two components of latency, where the total latency is the sum of the latency *inherent* from the communication and the latency *imposed* by the scheduling and execution of the tasks.

Each communication channel in the task graph has a corresponding route through the NoC, which is assigned during the routing phase. On this route, latency is inherent to the arbitration of the NIs and the routers. This impacts the performance of the application, and must be taken into account during constraint validation. This latency can be calculated by summing the worst-case latencies that may occur at every arbitration point on that route. As discussed in the previous section, we only assign routes to channels that provide sufficient throughput. The throughput of the tasks connected with a channel is thus less than or equal to that of the channel itself. This means that the channel never is the throughput limiting factor. Now, we combine the buffering and latency aspects into one model.

4.4.1 Combined communication and buffer model

We assume that an application buffer is orders of magnitude larger than the buffer capacity in a network route. The model we propose does not take the network buffering into account, but it still provides guarantees because of the more restricted buffer size. In Figure 4.3, we have modeled a channel between two actors, that captures both a limited buffer capacity, a throughput limitation and network latency. The tokens produced by actor s experience network latency through actor sd. Due to the throughput limitation of actor d, tokens may accumulate on the channel between sd and d, which can be considered as the application buffer of actor d. Actor s can thus produce as much data as the application buffer in consuming actor d can contain. This amount of data is limited by the backward channel between d and s, which should contain the number of initial tokens that fit precisely into the application buffer of actor d. Note that the synchronization actions communicated via the backward channel may experience latency as well (by actor ds).

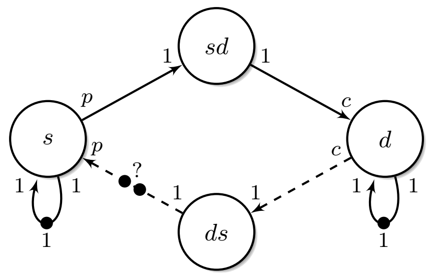


Figure 4.3: SDF graph modeling a routed channel.

In the routing phase, the resource manager routes the communication channels through the network. At run-time, the application graph is altered to incorporate the latency introduced by these routes. Figure 4.4 shows the SDF graph from Figure 4.1, extended with the channel model we described. In this example, we assume all the routes have a latency of 4 time units. We modeled the minimum possible amount of buffer space in the application, so as to guarantee deadlock free execution, resulting in a decreased throughput compared to the graph in Figure 4.1.

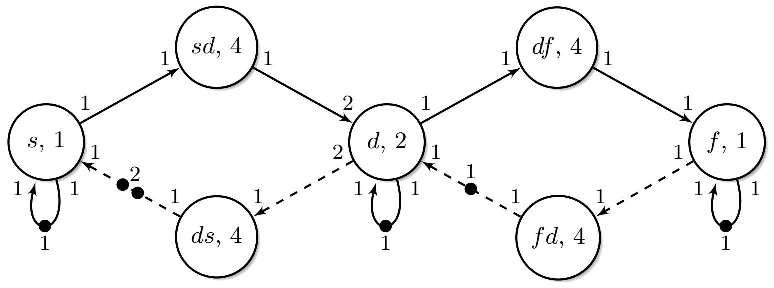


Figure 4.4: The example SDF graph, where the channels now capture the latency introduced by the routes through the NoC.

Strongly connected graphs

As described in Section 3.2, we require the task graph to be at least weakly connected, because each task within the application must have some dependency on another task. For the validation phase, we strengthen that requirement by requiring a *strongly connected* graph (see definition 5). The example graph in Figure 4.4 is strongly connected, as there is a directed path between every pair of actors in the graph. This requirement ensures that no strongly connected component of the graph deadlocks, while other parts of the graph are still able to execute. This is not a real restriction, as in practice, applications must be able to execute within bounded memory. Therefore, we can convert any task graph into a strongly connected graph by modeling application and communication buffers. The throughput of the resulting graph can be calculated with the methods we describe next.

4.5 Throughput analysis

An application engineer may require that a correct execution of an application must achieve a minimum throughput. For example, in communication protocols, data packets may be missed or the connection can be lost. To ensure a correct execution of the application, a throughput constraint may be specified in the application specification. As the throughput of an application is determined by the actor with the lowest throughput, this constraint is specified with a single value for the entire graph. Now, we describe two methods to analyze the SDF graph and to check whether a throughput constraint is satisfied.

4.5.1 Maximum cycle mean analysis

Throughput analysis of SDF graphs is widely researched [59–61], although it is only common practice for design time purposes, due to its computational complexity. Generally, an SDF graph is converted to a Homogeneous SDF (HSDF) graph, which is a special case of SDF graphs where all communication rates consist of a single token. For every actor a that has a higher communication rate in the SDF graph, exactly q_a copies are created in the HSDF graph, all having the same execution time $\rho(a)$. After the conversion, a Maximum Cycle Mean (MCM)[1] analysis is performed on the HSDF graph. The MCM is indicated with $\mu(\mathcal{H})$, and equals one over the throughput of $\mathcal{H}$:

$$\mu(\mathcal{H}) = \max_{\gamma \in \mathit{Simple\ Cycles}(\mathcal{H})} \frac{\sum_{a \in \mathcal{T}(\gamma)} \rho(a)}{\sum_{c \in \mathcal{C}(\gamma)} \delta(c)}$$

For graphs with few actors and low communication rates, this approach may fit the conditions of a run-time environment. However, there is no restriction on the number of tasks in an application, and even worse, on the communication rates of the channels. Therefore, even a small application may lead to an explosion in the size of the corresponding HSDF graph. Converting our example SDF graph from Figure 4.1 results in an HSDF graph with 4 actors, shown in Figure 4.5. When we change the consumption rate of channel $\langle s, d, 1, 2\rangle$ to 5 and the consumption rate of channel $\langle d, f, 1, 1\rangle$ to 4, the repetition vector becomes $\langle 20, 4, 1\rangle$. Converting this graph results in a corresponding HSDF graph with 25 actors. For larger applications, the required memory capacity and execution time of the analysis algorithms preclude usage at run-time. Additionally, some well-known algorithms seem to fail in finding a solution when the size of the graph increases [62]. Thus, the conversion of an SDF graph to an HSDF graph is the major disadvantage of this approach.

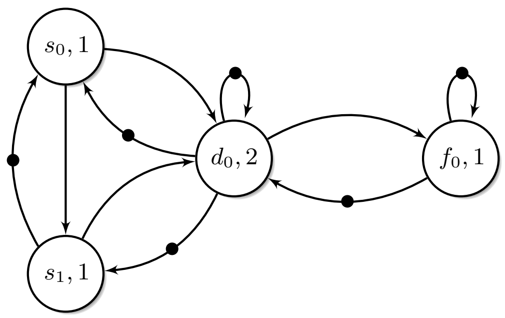


Figure 4.5: The example SDF graph converted to a homogeneous SDF graph.

To the best of our knowledge, no method exists that efficiently gives the throughput of an SDF graph. Using an approach that avoids the costly conversion to HSDF, Ghamarian et al. [62] developed a technique that works directly on SDF

1. Also known as Maximum Cycle Ratio

graphs. Considering a worst case scenario, this approach may still be infeasible for run-time usage, but Ghamarian et al. [62] state that in practice this method works well. For now, that statement is not verified, but the results in [62] show that this method outperforms other analysis techniques. The execution time of their method is acceptable for run-time analysis, being less than 1 ms for graphs up to 20 actors[1]. In the section that follows we explain this approach.

4.5.2 Self-timed execution of SDF graphs

We assume that an SDF graph executes in a *self-timed* manner, which requires each actor to fire as soon as it is enabled [54]. A timed SDF graph contains enough information to simulate this behavior. For simulation, we need to identify the state of the graph at each time step. The state of an SDF graph is given by the set of actors that fired at a time step, and the number of tokens that reside on every channel. At each time step in the simulation, we first check whether any actor finishes its firing, while producing tokens at the outgoing channels. Then, we check for every actor whether it is enabled, and thus will fire at the current time step, while consuming tokens from its incoming channels. The simulation continues until a state is reached that has been visited before. Then we known we have explored the entire state-space. This simulation is a *self-timed execution* of an SDF graph, resulting in a state-space exploration of the SDF graph. Figure 4.6 illustrates the simulation result of our example in Figure 4.1. The arcs denote a transition of a single time step, annotated with the set of actors that fired at that time step.

Most streaming applications require a start-up period (the *transient* phase), before they resort to a repeating pattern (the *periodic* phase). In the specification and analysis of applications we target the periodic phase, as we assume that the periodic lifetime of the application is orders of magnitude longer than the transient phase, and it better reflects the desired functionality of the application.

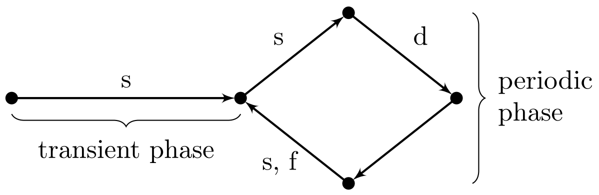


Figure 4.6: A state-space exploration of the example graph.

Throughput calculation

Similar to the throughput calculation using MCM, the throughput of an actor a is defined as the average number of firings of a per time unit.

1. No details are available about the host system that is used for benchmarking

Definition 15 ⌊Actor throughput⌉ *The throughput $Th(a)$ of an actor $a \in \mathcal{T}$ is equal to the average number of firings per time unit in the periodic part of the self-timed state-space.*

So, for the example in Figure 4.6 we see that the periodic phase is four time units long. Actor s fires on two of the four edges of the periodic phase, denoting that the throughput $Th(s)$ of actor s is $\frac{2}{4}$. Likewise $Th(d)$ and $Th(f)$ are $\frac{1}{4}$.

Note that there is a relationship between the throughput of actors in an SDF graph. Dividing the throughput of an actor by its value in the repetition vector gives the throughput of the SDF graph. This means that we can determine the throughput of an SDF graph using an arbitrary actor. Definition 16 specifies the throughput of the entire SDF graph.

Definition 16 ⌊SDF graph throughput⌉ *For a consistent and strongly connected SDF graph $\mathcal{G}$ with repetition vector q and actors $a, b \in \mathcal{T}$, $Th(a) \cdot q_b = Th(b) \cdot q_a$. The throughput of $\mathcal{G}$ is defined as $Th(G) = \frac{Th(a)}{q_a}$, for an arbitrary $a \in \mathcal{T}$.*

Up to this point we have discussed how we can model the execution layout of an application as an SDF graph. The self-timed execution method then is used to calculate the throughput of that graph. This analysis gives a guaranteed throughput of the corresponding application. We can compare this value with a throughput constraint that may be specified for that application. For an extended formalization and proofs of the throughput analysis technique in this section, we recommend [62].

We also want to allow application engineers to be able to specify latency constraints. In the following section, we describe how latency constraints may be specified and how they are checked during the validation phase.

4.6 Latency analysis

For introductory purposes, we first discuss some related work on latency analysis in SDF graphs. Using the ideas of [55], we argue that a more convenient method exists to check whether the application meets its latency constraints. Our contribution lies in an integration of this idea with other dataflow components, so that at run-time all throughput and latency constraints can be validated simultaneously.

A latency constraint must be specified between two actors; a *source* and *sink* actor. Using the definition of latency given in Section 4.4, it is debatable what the corresponding output of an input sample is, when the repetition vector entries of the source or sink actors differ. For example, in a video processing application, it may not be clear if a latency constraint is specified for the first output sample, i.e. a scanline, or that it requires a entire video frame as output. We distinguish between early latency and late latency. *Early latency* is the first firing of the *sink* actor in reaction to the enabling of the *source* actor. A *late latency* response indicates the moment at which all corresponding output data has been produced.

As an example, we use the graph from Figure 4.1, with the communication rates of d on the channels to and from f increased to 8, resulting in the repetition vector $\langle 2, 1, 8\rangle$. In Figure 4.7, the firings of each actor are given, starting in the transient phase. In the bottom row, the reader can see at time 14 and 21 an Early Latency Response (ELR) and a Late Latency Response (LLR) of actor f to the firing of actor s at time 4.

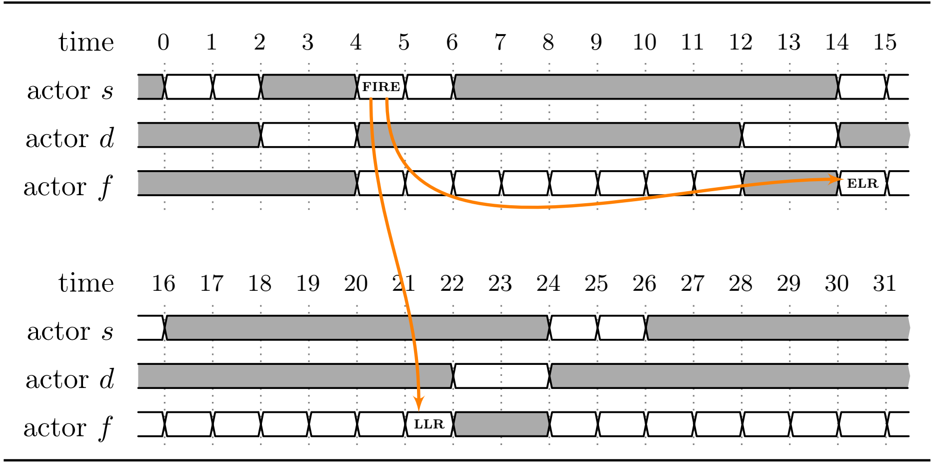


Figure 4.7: Actor firings during self-timed execution of the example.

In [51], the concept of a *latency graph* is introduced, which assumes that only complete iterations of graphs are meaningful. An iteration of a graph is precisely the set of actor firings, such that each actor fired the number of times of its repetition vector entry. A graph iteration indicates that an implicit period of task executions has been performed, which may be seen as a predefined amount of data having been processed. An LLR occurs when a graph iteration is completed.

If an actor is the source or sink of a latency constraint and its repetition vector entry is larger than one, we can add an additional actor to the graph. A latency constraint is then defined within a graph iteration over the first firing of the source and the last firing of the sink actor. This extended SDF graph is known as a latency graph [51]:

Definition 17 ⌊Latency graph⌉ *Let* $\{a, b\} \in \mathcal{T}$ *be two actors of a timed SDF graph* $\mathcal{G} = \langle \mathcal{T}, \mathcal{C}, \delta, \rho\rangle$ *with repetition vector* q*, and* $\{src, snk\} \notin \mathcal{T}$ *be two new actors. We define the latency graph for actors* $\{a, b\}$ *as* $\mathcal{G}_{L(a,b)} = \langle \mathcal{T}_L, \mathcal{C}_L, \delta, \rho_L\rangle$*, with* $\mathcal{T}_L = \mathcal{T} \cup \{src, snk\}$*,* $\mathcal{C}_L = C \cup \{\langle src, a, q_a, 1\rangle, \langle b, snk, 1, q_b\rangle\}$*, and* $\rho_L = \rho \cup \{\langle src, 0\rangle, \langle snk, 0\rangle\}$*.*

Figure 4.8 gives the latency graph $\mathcal{G}_{L(s,f)}$ of the example used for Figure 4.7, where actors s and f are chosen to be subject to a latency constraint. Such a constraint denotes an LLR, as actor *src* fires at the first input sample, and actor *snk* fires when the complete output is available. If a constraint on an ELR is required, other actors should be chosen to specify the latency constraint; in this case s and d. Next, we describe two methods to derive the maximum latency between the source and sink actor of the latency graph.

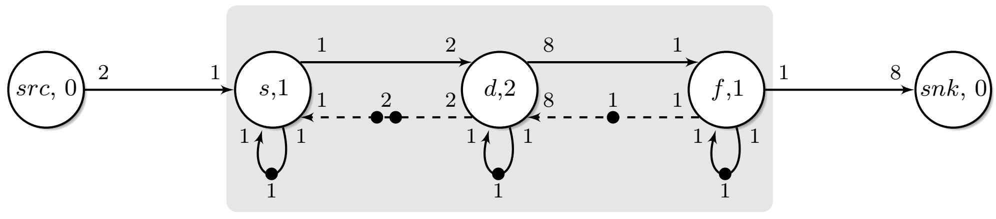


Figure 4.8: A latency graph is an SDF graph (gray) extended with actors that fire once per graph iteration.

4.6.1 Tracking corresponding firings

In [63], a latency analysis technique is implemented that employs the state-space exploration as used for throughput analysis. This approach keeps track of the firings of the source and sink actors of a latency constraint, gathering information similar to Figure 4.7. The latency is determined by the maximum time difference between corresponding firings. In this method, the definition of a corresponding firing of a source and sink actor can be adapted, allowing both early and late latency constraints to be analyzed.

The downside of this approach is that multiple latency constraints have to be checked sequentially, multiplying the required run-time of the validation phase by the number of constraints. Another difficulty is that latency constraints have to be checked using integral values. In some cases, it is difficult to come up with a number for such constraints. The following section describes a method that allows latency constraints to be specified in terms of graph iterations.

4.6.2 Modeling latency constraints as throughput constraints

Illustrated by Figure 4.7, we may have a requirement that the latency between actors s and f may not exceed 18 time units. We can rephrase this requirement by stating that actor f must finish its corresponding firings within the duration of a graph iteration. This duration can be formulated as a minimum throughput constraint on the graph.

In [55], the idea is presented to model a latency constraint in terms of a throughput constraint. This is accomplished by limiting the throughput of the source actor by only enabling that actor if the sink actor does not lag behind. This throughput limitation is created with a “backward” edge from the sink actor to the source actor. A latency constraint is specified by inserting an actor l on that edge; this is shown in Figure 4.9. When the response time $\rho(l)$ equals zero, the source actor is limited by the throughput of the latency cycle. The latency between the source and sink actor is then guaranteed to be bound by the MCM $\mu(\mathcal{G})$ of the graph.

Increasing the response time $\rho(l)$ eventually results in a decreased throughput of the entire graph. To maintain the same throughput, the other actors on the latency cycle have to use less time to perform the same functions. We may specify $\rho(l)$ such, that precisely L time units remain for those actors subject to the latency constraint. Given that the source actor has a period $\mu(\mathcal{G})$, we allow latency actor l a response time of $\mu(\mathcal{G}) - L$, so that the sink actor has L time units left to produce the corresponding output. When the throughput of the graph drops below the specified threshold $Th(\mathcal{G})$, the latency constraint L is violated.

Placing more initial tokens β_l on the latency edge specifies that a latency L is allowed to be spread over a number of graph iterations. For every β_l consecutive firings of actor l, the average latency is guaranteed to be less or equal to L time units. This is a more relaxed latency constraint, as it allows for more jitter in the processed data streams, compared to a smaller latency constraint specified for each graph iteration. Thus, the latency constraint can be modeled more generally with [55]:

$$\rho(l) = \beta_l \cdot \mu(\mathcal{G}) - L, \quad \beta_l \geq 1$$

A latency constraint is fulfilled, if the throughput of the entire application is still sufficient, even with this additional throughput limitation. If a latency constraint is violated, the throughput of the application will drop below the minimum required throughput. Figure 4.9 shows how such a latency constraint can be added to the latency graph of Figure 4.8.

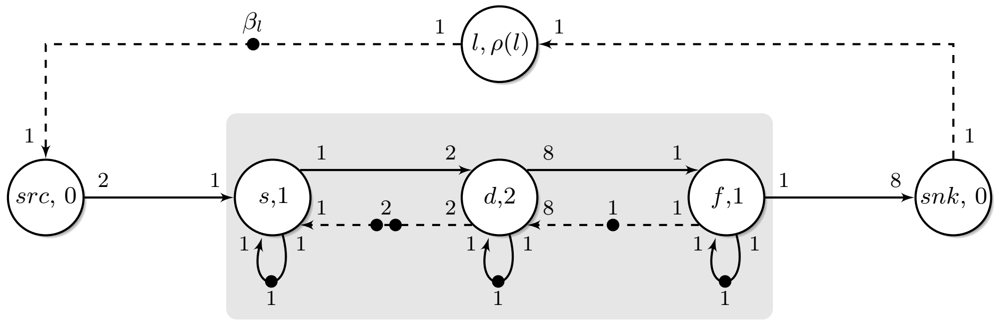


Figure 4.9: A latency constraint specified as a throughput limitation, using the corresponding latency graph.

Simplifying the latency graph

For clarity reasons, we used a latency graph to illustrate the method discussed in the previous section. When multiple latency constraints have to be specified, a latency graph may add undesired complexity to the graph. The purpose of the latency graph is to convert communication rates, such that a latency constraint can be specified over actors with a repetition vector entry of one. This conversion can also be expressed in the communication rates of the actors themselves. Figure 4.10 shows this approach using the repetition vector for conversion.

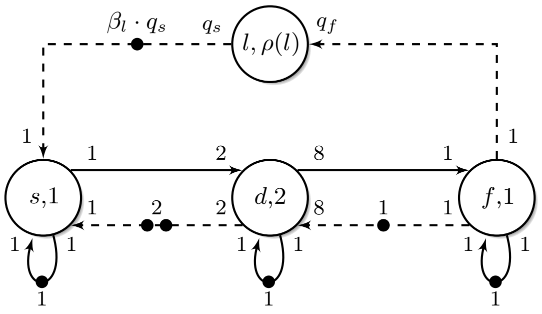


Figure 4.10: A latency constraint specified as a throughput limitation, using the repetition vector for rate conversion.

Applications with a periodic source

Each application submitted to the resource manager for execution contains a specification of the tasks and the communication channels within that application. This specification must describe a strongly connected graph, which can be achieved by modeling buffering with backpressure. However, some applications are specified as a weakly connected graph, as no explicit buffering method is used. This is exemplified by applications that are dictated by a periodic task; an A/D converter can, for example, be used as an input task. For such *periodic applications*, a steady data stream is assumed to flow through the tasks. These applications indeed have a strong throughput requirement, requiring each task within the application to follow the steady pace dictated by a periodic task.

To ensure that these conditions are provided, latency constraints have to be added to the application specification. Each latency constraining actor l in a graph $\mathcal{G}$ is then specified with $L = \mu(\mathcal{G})$ and $\beta_l = 1$, resulting in a response time $\rho(l) = 0$. This ensures that a single graph iteration must finish within the period of the periodic actor. For streaming applications, the amount of tokens β_l may be increased to fill the pipeline. These additions compose a strongly connected graph, satisfying the requirement for constraint validation. Appendix B describes a beamforming application illustrating this scenario.

4.7 Improved accuracy with latency-rate servers

Up to this point, we used response times to model the effects of arbitration over resources onto the application. However, a model with response times assumes that every actor firing experiences the worst-case scenario; that its budget for execution is depleted just before the actor is enabled. The fact that multiple actor firings may execute consecutively within the same period cannot be taken into account [17]. Wiggers et al. [17] present a more accurate dataflow model, that results in a non-decreasing guaranteed throughput. In that model, a task is represented by two actors, which form a *latency-rate server.* In Figure 4.11, the relation is shown between a task, modeled as a single SDF actor using response times, and a model as latency-rate server. Actor a_0 models the latency that results from the worst-case enabling time of the actor; that is when a contiguous execution budget is depleted for the current period. The latency part of the model has no self loop, which allows an entire datastream to experience the worst case scenario simultaneously. This results in more accuracy, compared to a worst-case assumption for every actor firing. Actor a_1 models a limitation on the rate of task executions. After the initial worst-case latency, the corresponding task may be executed at a rate that is equal to or higher than the response-time of the previous model.

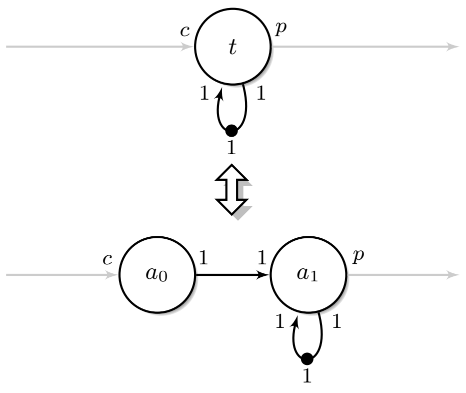


Figure 4.11: Model run-time scheduling of an actor as a latency-rate server.

The following definition gives the firing durations of the actors in a latency-rate server that models the scheduling effects on a task:

Definition 18 ⌊Conservative latency-rate dataflow component⌉ *Given a task t with execution time $x(t)$, that is guaranteed to have a minimum execution time R in an interval of time Q. If, in a corresponding latency-rate server lr, the latency component of lr has a firing duration $Q - R$, and if the rate component of lr has a firing duration $\frac{Q \cdot x(t)}{R}$, then the model lr is temporally conservative to task t [17].*

For illustrative purposes, we add a schedule to the example from Figure 4.4, shown in Table 4.1. Using Definition 18, we derive the properties of the corresponding latency-rate servers, shown in Table 4.1 as well. Using these properties, we improve the model from Figure 4.4 with latency-rate servers, resulting in the model in Figure 4.12.

Table 4.1: Given an application and its schedule, the properties of the corresponding latency-rate dataflow component are obtained.

Application		**Scheduling**		**Dataflow model**	
Task	WCET (x)	Budget (R)	Interval (Q)	Latency	Rate
s	1	1	3	2	3
d	2	3	4	1	2
f	1	2	5	3	2

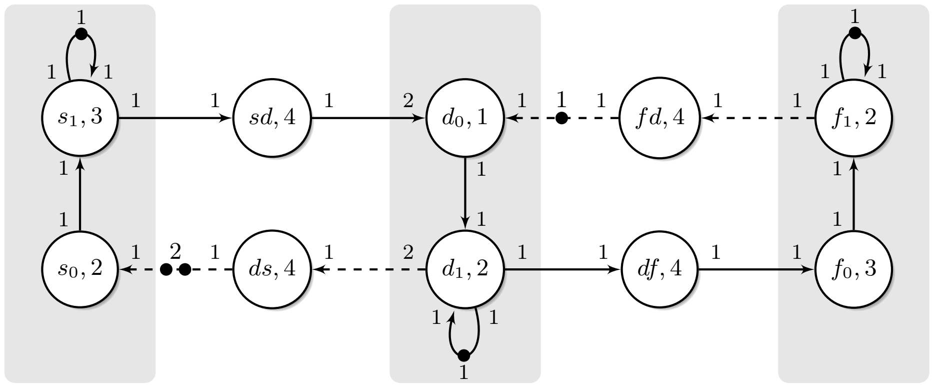


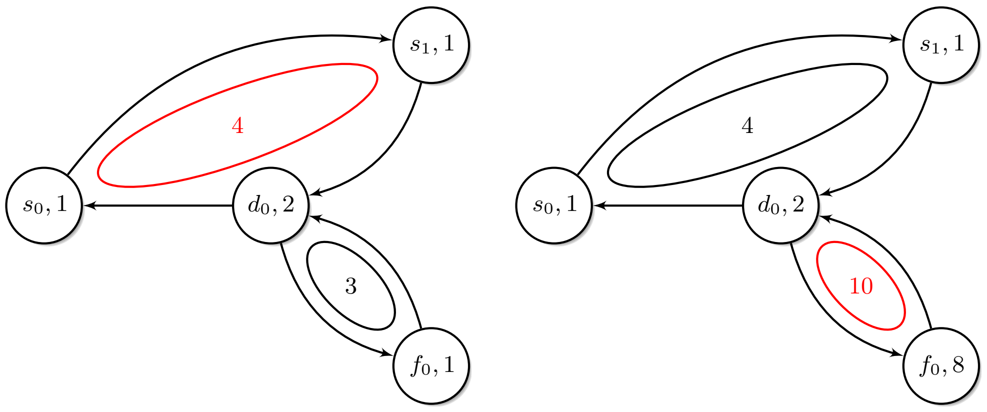
Figure 4.12: Improved accuracy by using latency-rate servers instead of response times to model run-time scheduling.

4.8 Validation of the execution layout

If the first three phases of resource management succeed, then a preliminary execution layout is known. The validation phase combines the application specification with this execution layout to create an SDF graph containing details about communication latencies and scheduling properties of actors. With the modeling techniques described in this chapter we can integrate all details of the execution layout of an application into an SDF graph. A self-timed execution of the resulting SDF graph gives a guaranteed throughput of the application. If the guaranteed throughput is not lower than specified, all constraints are met, including the latency constraints. Thus, the constraint validation algorithms can check with a single self-timed execution of the SDF graph whether the specified constraints are met.

4.8.1 Feedback

If the execution layout fails to provide sufficient performance, we may revise decisions made in earlier phases. Therefore, we have to identify the critical cycle within the SDF graph. A reverse traversal of the final state-space after validation yields a dependency of the actors in the SDF graph, as illustrated by Figure 4.13. A critical cycle (multiple cycles may exists) gives information on what combination of actors and communication paths is limiting for the throughput of the entire application. This information can be used as feedback to the binding and mapping phase. The problem with this approach is that the explored state-space alone is not sufficient to create the dependency graph. Additionally, all the simple cycles in the dependency graph have to be traversed to find the critical cycles. This is a very complex operation [64], which makes it hard to create an efficient feedback mechanism.



(a) The example from Figure 4.1 **(b)** The example from Figure 4.7

Figure 4.13: The dependency graphs indicate the critical cycles (red) in two examples used in this chapter.

CHAPTER 5

Implementation

The algorithms described in the previous two chapters are integrated into a prototype. The prototype was designed to be deployed onto a hardware platform designed for the CRISP project [7]. Because the platform was still being designed in parallel with this research project, we deployed a virtualized environment similar to that platform. This chapter briefly describes the prototype we developed, giving the reader an insight into the engineering effort and the resulting ideas.

5.1 System overview

The main requirement of the CRISP project is that the run-time resource manager has to be integrated into a Linux kernel. Our prototype requires a description of the platform, and a specification for each application that may be executed on that platform. The software stack we propose is illustrated in Figure 5.1. The platform description (dashed) is compiled together with the rest of the Linux kernel. With some development effort, this description can be replaced with a platform driver that informs the resource manager about the available resources on the platform at startup of the system.

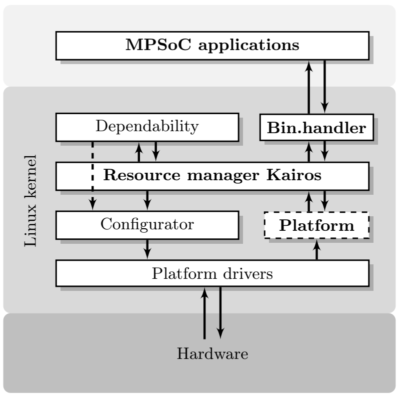


Figure 5.1: The software stack of a run-time resource manager. Kairos and the other bold typed parts were developed in this project.

A user should be able to execute applications on this platform. Each application should have some metadata that describes the "contents" of the application. We consider such application specifications to be the input of our prototype.

5.2 Application specifications as input

We think of an application as a set of tasks, connected with communication channels. This structure can be represented with a task graph, which is at run-time converted to an SDF graph. Figure 5.2 shows the relation between the data structures that are used to model an application. An application engineer can specify such an application in a predefined XML format. We developed a parser that converts that textual representation of an application into a binary format, which we call an MPSoC application. Future work should incorporate methods that add the binary data of the implementations, as referred to in the application specification, to the MPSoC application as well. In the following sections, we describe the format of an MPSoC application and why such a binary format is convenient.

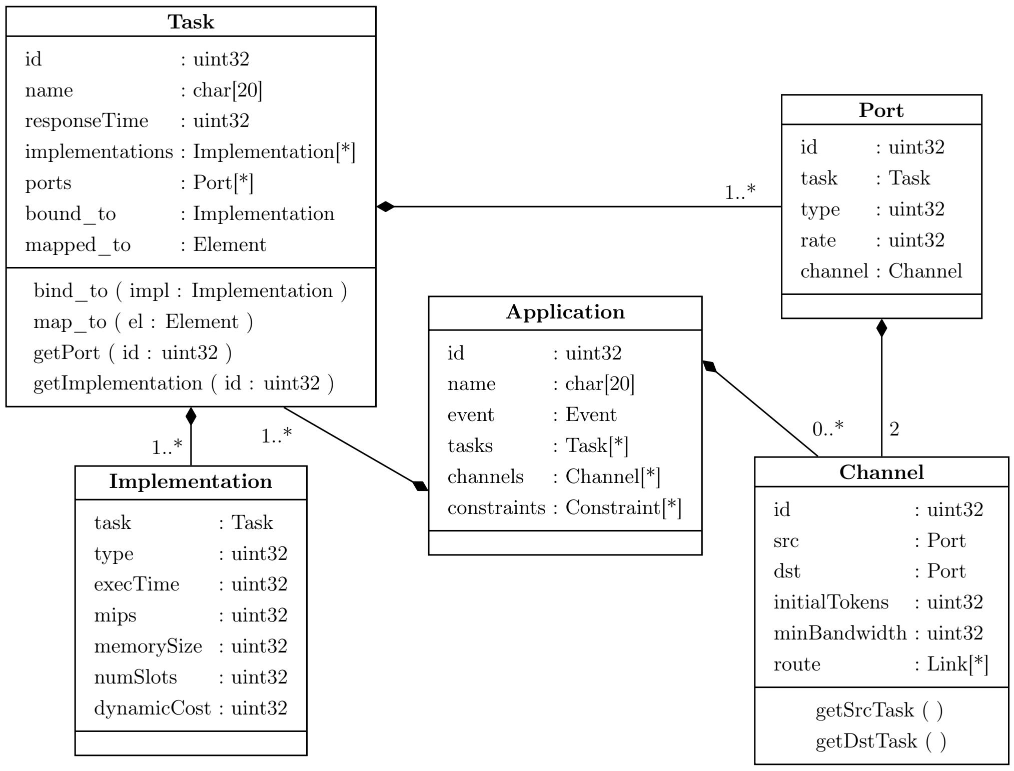


Figure 5.2: Data structures modeling an application.

Binary file format

The Macho-O file format used by Apple is used to store programs and libraries in a single Application Binary Interface (ABI), providing data and both statically linked and dynamically linked code [65]. Analogue to this concept, we can pack the implementations of each task within an application into a single binary file. A Mach-O file contains code and data for a single architecture. But as the hardware that Apple developed evolved over the years, moving from Motorola processors to PowerPC-based computers, and finally switching to Intel-based CPUs, while also dealing with 32-bit and 64-bit systems, a file archive format was introduced to cope with the architectural differences. A *universal binary* is an archive that contains binaries targeted at more than one architecture. A special file header allows run-time tools to quickly find the code appropriate for the requested architecture.

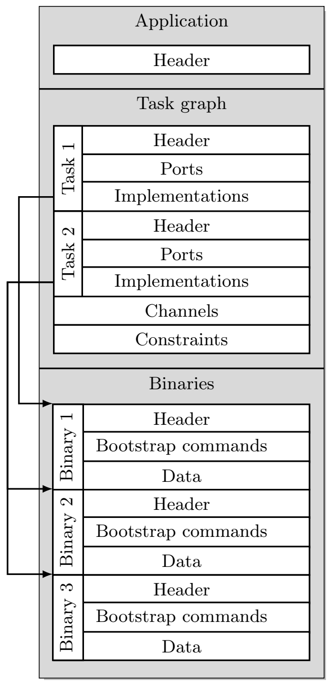


Figure 5.3: The MPSoC binary format.

We can combine both concepts into a single binary file format, the *MPSoC binary format*. Such a file contains the binaries that implement the functionality of a task, together with the required annotations and the task graph. Figure 5.3 shows how this file format is constructed. We prefer to put all the information required by the resource manager in the first part of the file, as this prevents unnecessary seeking in the file. Once a specific binary within the ABI is needed (during bootstrapping), the lower part of the file has to be accessed, but at a known file-offset. Appendix A contains a reference of this format.

From the information in this ABI, we must be able to construct the data structures as show in Figure 5.2. Besides having a neat solution to manage all the components of an MPSoC application in a single place, it also provides a solid entry point into the kernel. The next section describes how the kernel handles this new binary format.

5.3 Workflow within the Linux kernel

A basic operation of a kernel in general, is the ability to execute applications on demand. When a user starts an application, the system call `exec` is invoked in the Linux kernel. The role of this entry point is to fill a "binary parameter structure" (`struct linux_binprm`) with some information regarding the application a user wants to be executing, and to find the matching binary handler for that application. Linux supports multiple binary formats, and we introduced a new binary handler that supports the MPSoC binary format. The function `search_binary_handler` scans the list of registered binary formats, until a handler accepts the `linux_binprm` structure. This way, both standard tools and applications intended to manage the operating system can co-exist with applications for the MPSoC platform.

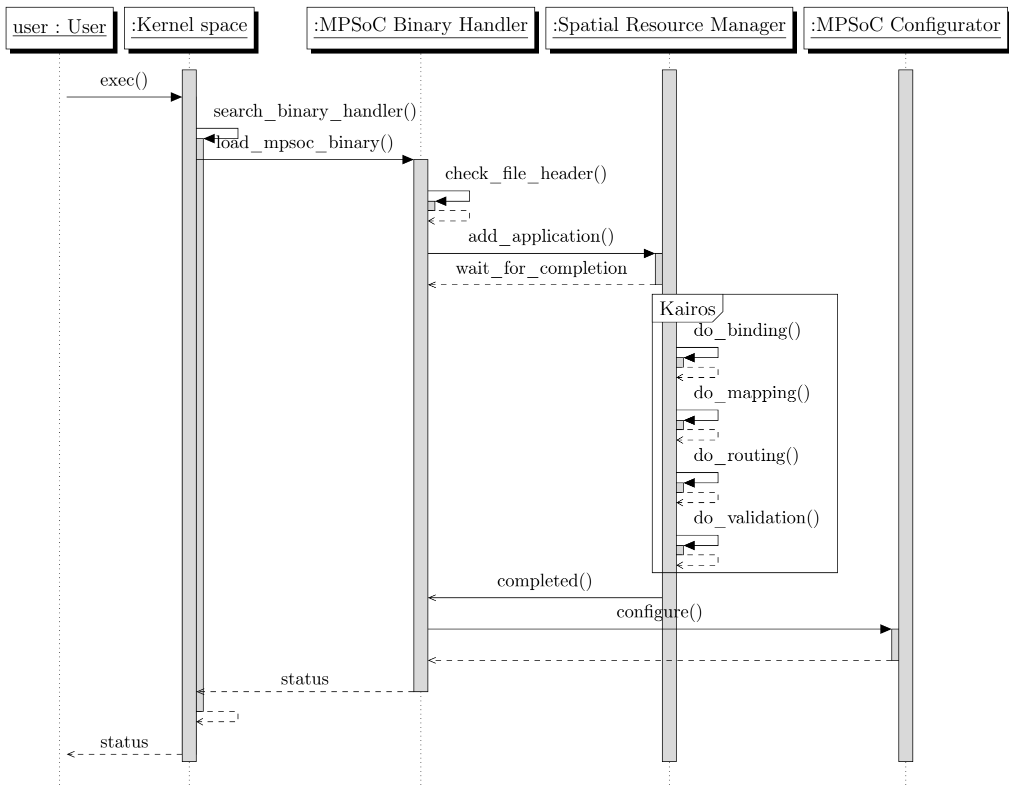


Figure 5.4: A typical sequence of an application being submitted to the resource manager.

Our binary handler uses the MPSoC file header to initialize the data structures of Figure 5.2. This application is then submitted to the resource manager Kairos, which adds the the application to a priority queue at first. We use a priority queue, because Kairos can not allocate resources to multiple applications simultaneously. It also makes sense to handle removal events, freeing up resources on the platform prior to addition of new applications. Future work may integrate more intelligent logic to reason about the order in which application and events should be handled. Depending on the amount of work Kairos is doing at that moment of enqueing, the application is served within a certain *response time*. During that time, the user's request is suspended, using the `completion` structure to block on.

Once the application is being handled by Kairos, the various stages of resource management are executed. The final result is twofold. Feedback, whether a feasible execution layout was found, is returned to the user as a return value interpreted by the terminal, which is standard Unix behavior. If successful, the execution layout is presented to the configuration layer that instructs the platform drivers to configure the hardware and bootstrap the application. A sequence diagram of this workflow is given in Figure 5.4.

5.3.1 Simulating hardware failures

A big advantage of resource management at run-time is that resources can be added to, and removed from the platform at any time. This allows for some degree of fault tolerance. For example, if an element breaks down, the application that is using that element may fail, but the platform can still be used for other applications. The *dependability* part in Figure 5.1 monitors the platform and removes resources that are faulty. The resource-manager should be able to work around these hardware problems. Therefore, we create a simple substitution for the dependability part. We use the fault injection framework [66] of the Linux kernel to generate hardware faults. Figure 5.5 illustrates a faulty chip that is still capable of running an application.

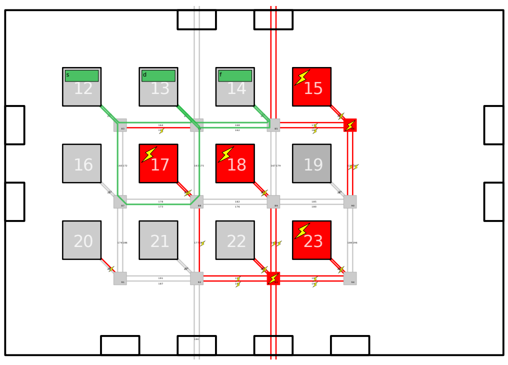


Figure 5.5: Resource allocation should avoid faulty hardware (red).

5.4 Execution layouts as output

If all the resource management phases of Kairos are traversed successfully, a feasible execution layout is known. The details of this execution layout are embedded in the various data structures that compose an application (see Figure 5.2). A platform driver (that remains to be developed) must configure the platform according to this information. Currently, the data structures are directly accessible for such a driver. An Application Programming Interface (API) may be developed to facilitate some glue logic between Kairos and the platform driver. The interface to the platform is not part of this project, but we do not see fundamental problems regarding interaction with Kairos. A different aspect is the output that should be directed to the user controlling the platform.

5.4.1 User interface

In the previous section, we described how MPSoC applications can be started with a standard user perspective on terminal usage and application execution. Managing applications running on the platform is more complicated. All standard available tools shipping with a minimal operating system environment assume that an active application is listed in the process table, which can be accessed through the virtual filesystem "procfs". Providing a standard user perspective on management of MPSoC applications requires Kairos to add those applications to the process table. However, listing applications as processes requires Kairos to provide scheduling parameters and a memory map of the application. Providing that information is not desired, as the application does not have to be scheduled on the processor running the Linux kernel. This approach must carefully be examined, as we do not want the user to crash the system or let him think some operation is performed on an MPSoC application, while it is not supported.

For now, Kairos creates entries in the "procfs" filesystem to provide user interaction with the applications on the MPSoC platform, but in a non-standard way. Additionally, the platform state and other useful information is exported as well. This allows the user to instruct the resource management algorithms, and to monitor the platform (see Figure 5.6).

5.5 Virtual environment

Embedded systems commonly have ARM-based processors. An MPSoC architecture developed in parallel to this research contains an ARM processor as well. This processor runs the operating system that manages the entire MPSoC platform. The framework and algorithms we developed are integrated into that operating system. As the hardware was not available during this research, we have created a virtualized environment that should give a reasonable idea of the performance and limitations we have to deal with.

5.5.1 Virtualization technologies

Multiple ways of virtualization exists, but we identify the three most used methods. In the first method, a hypervisor close to the hardware provides some basic isolation, separating the guest operating systems from the hardware. This approach is called *native (or full) virtualization*, and often gives the best performance, but also being the most difficult method to work with. The best known example of this method is VMware ESX Server [67].

In the *paravirtualization* technique, a hypervisor provides an API to the guest OS. The restriction of this method is that the API provided must be similar, but not necessarily identical, to that of the underlying hardware. Therefore, this method is impractical for translation between instruction sets. Paravirtualization is used by both Xen [68] and the Linux Kernel-based Virtual Machine [69].

The last technique is called *emulation*, which virtualizes the guest platform by simulation of an entire hardware environment. Emulation has a high degree of freedom, and thus is implemented in variety of ways. Bochs is a platform emulator that has been around for some time, but it is restricted to the x86 instruction set [70]. Various ARM emulators exists [71, 72], but emulating an entire OS is more difficult than running a single ARM binary. We chose to create a development environment, in which QEMU [73] is used for emulation of a hardware environment. QEMU is an open source project, which has the advantage of being well supported in the Linux community. In a standard QEMU distribution, a number of hardware environments are provided[1]. Additionally it provides the means to boot and configure a Linux kernel, and to connect to the emulated platform with the Linux kernel debugger KGDB.

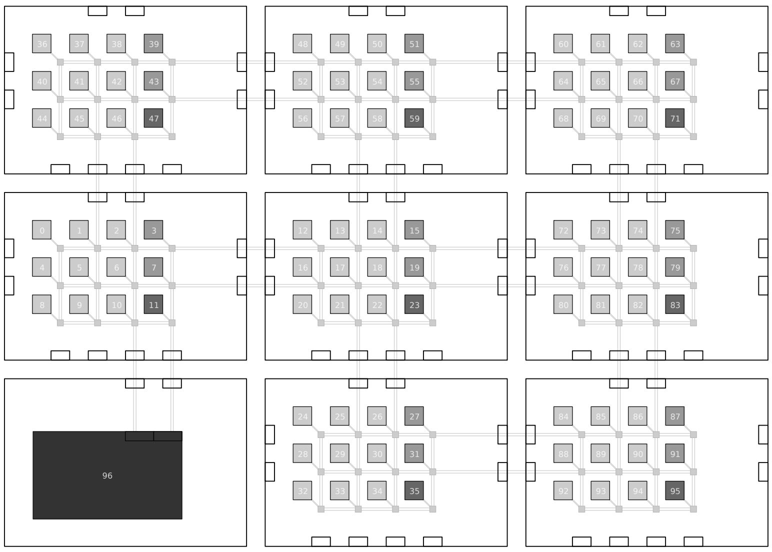
Emulated hardware platform

From the list of machines available for emulation in QEMU, we picked the Realview Verstatile Platform Baseboard [74], which is supported by the entire toolchain we used during this project. This hardware platform contains an ARM926Ej-S processor, which is commonly used in embedded systems, and is likely to be of the same class as the processor of the CRISP hardware platform. During the startup of a Linux kernel, a "BogoMIPS" value is determined by timing a simple calculation loop. On ARM processors, this loop consists of two instructions; the "BogoMIPS" value of any ARM processor is thus equal to half its frequency. We may take the "BogoMIPS" value as reported by the emulator, and use it to provide realistic numbers on the performance to be expected of a physical system.

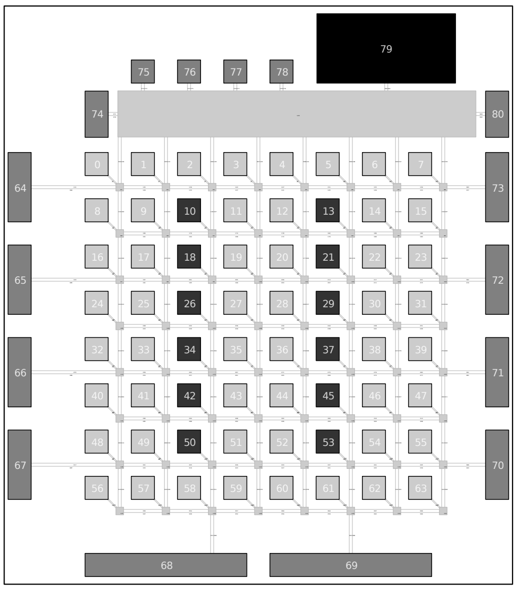
Figure 5.7: Realview Versatile Platform Baseboard.

Preliminary results show that the developed prototype also runs on a physical system. The test results that are discussed in the next chapter, are still performed in the virtual environment.

1. QEMU version 0.10 provides 20 ARM-based machines to choose from.



(a) The CRISP platform with 8 GSPs and 1 GPD.



(b) The “Mesh 64” platform.

Figure 5.6: Kairos provides a way to monitor the platform on an external system.

CHAPTER 6

Results

This chapter presents results from various benchmarks of our prototype “Kairos”. Most results are obtained by large scale benchmarks with synthetic datasets. We have a limited number of quality measures available to monitor the benchmarks. These measures are used to indicate the quality of certain aspects of the implementation. Table 6.1 lists the measures available, and what quality indicators can be assessed with them. We start this chapter with a description of the datasets we used for benchmarking the prototype. Due to the lack of reference material, we present some specific scenarios and variations within the algorithms themselves. After presentation of the results, we discuss the interpretation of them in Section 6.3.

Table 6.1: Quality measures available in Kairos.

Indicator	**Measure**
Energy consumption	Energy costs
Remaining resources	External resource fragmentation
	Required resource overhead
Running applications	Mapping success rate
	First failure
Robustness algorithms	Run-time of an allocation attempt

6.1 Case study: a 16-channel beamforming application

In Appendix B, we describe a beamforming application that is being developed in the CRISP project [7]. This application is tailored to the hardware platform that is designed in parallel. This application does exceed the dimensions of the application class we foresee to be used in the systems discussed in this research project. We think that it gives nonetheless a good impression of how our prototype behaves when the complexity of the applications increases.

6.2 Synthetic benchmark

To test our algorithms more thoroughly, we need a large set of applications. As the availability of models of real world applications is low, we use synthetic datasets. Similar to task graph generator “Task Graphs For Free” [75], an in-house developed application generator was used to generate six synthetic datasets. In this tool, the structure of an application can be specified with the number of input tasks, the number of output tasks and the total amount of tasks. Also the maximum in-degree and out-degree of tasks gives direction to the generated communication structure. For each task, a number of implementations are generated from a subset of architectures, annotated with bounded random resource requirements.

We generate applications that are either *computation intensive* or *communication oriented*. Tasks of applications in the first set use between 70% and 100% of the element’s resources, and tasks in communication oriented applications use between 10% and 70%. This allows for communication oriented applications to time-share elements, eventually resulting in communication bottlenecks. Note that we allow multi-tasking during this synthetic benchmarking, but we do not generate local schedules for elements (see Figure 6.2). Within this characteristic, we categorize applications based on their size, namely *small* (< 5 tasks), *medium* (6-10 tasks) and *large* sized (11-16 tasks) applications.

We use thee different platforms in the benchmark; the “crisp51” platform from Figure B.2, the “mesh64” platform from Figure 5.6b, and the “ring23” platform from Figure 6.2. For each platform, we change the mapping cost function to observe the influence of the mapping objective. We optimize towards communication minimization, fragmentation reduction, and a combination of both objectives. Also, we disable the cost function, indicated with “None” in the various figures. The resulting execution layouts then depends on the communication minimization that is inherent to the search method.

Each of the six datasets contains 100 applications. As a preprocessing step, we remove applications from the dataset that are not admitted to an empty platform; this is required to filter out any extraneous applications. For each dataset, we generate 30 random sequences of those, at most, 100 applications, containing no duplicate entries. We benchmark five variations of three different platforms with each dataset, by adding iteratively the application from that dataset, using the random sequence. Thus, our benchmarks consists of 3 $platforms \times$ 5 $variations \times$ 6 $datasets \times$ 30 $sequences \times$ 100 $applications$ resource allocation attempts. Relatively early in the sequence, all the platform resources are allocated, resulting in failure of the remaining applications. Between sequences the platform is emptied.

Now, we present some results obtained with these synthetic datasets.

6.2.1 Run-times of Kairos

In Section 2.4, we argued that the time required to derive a feasible execution layout for an application is merely a design constraint, rather than an optimization criteria. Also, it is difficult to qualify the required run-time; this is partly caused by the lack of any quantitative material for comparison. So, the first tests we perform give an impression of what run-times can be expected of a spatial resource manager. The results reflect execution times on a low-end embedded ARM processor, running at 200 MHz. Figure 6.1 shows the run-times of Kairos as a function of the number of tasks in an application, gathered on the three different platforms, averaged over all five variations and all sequences. At some points, the variability is very large, especially in the validation phase.

6.2.2 Admission versus failure

In the proposed heuristics, cost functions are used to steer the optimization direction, possible with multiple weighted factors. The most basic optimization goal is admission of the application to the platform. We think that the most difficult test case available, is to allocate resources for the CRISP application in the "crisp51" platform. Shown in Figure 6.3, we plotted whether certain combinations of weights in the mapping cost function results in a feasible execution layout. We have two weights, where the fragmentation takes the values $[0, 1, .., 25]$ and the communication weight is given the values $[0, 10, .., 1000]$. We also give the frequency of successful admissions in specific bands of the ratio between fragmentation and communication weights; listed in Table 6.2.

Table 6.2: Only certain ratio (intervals) of fragmentation versus communication lead to successful admission for the CRISP scenario.

Frequency	**Ratio**
24	4
425	$[34, 46]$
196	$[62, 73]$
55	$[75, 79]$

For our synthetic dataset, it was difficult to generate reasonable throughput constraints. We do traverse the validation phase, but an application from the synthetic dataset is never rejected due to violation of performance constraints. For the other phases, we give the percentage of failures per dataset, relative to the total number of applications in each dataset. Table 6.3 shows per platform, the characteristics of the datasets we used, the number of applications per dataset and the percentage of failures in the binding, mapping and routing phase.

6.2.3 External resource fragmentation

In every test, a sequence of applications is submitted for execution on the platform. The amount of available resources in the platform is depleted relatively early in the sequence. Figure 6.4 shows the external fragmentation of the elements in the platform, as a function of the sequence ordering. Later in the sequence (more right on the horizontal axis), less resources are available.

6.2.4 Communication overhead

For the platforms “mesh64” and “crisp51” holds, that two tasks that are located on different elements in the platform, have to communicate through at least one, but often at least two, routers. A communication channel between such tasks, has a corresponding route through the interconnect of at least three hops. Due to the fact that we also have source, sink and router tasks, combined with multi-tasking capabilities, routes may be smaller than three hops.

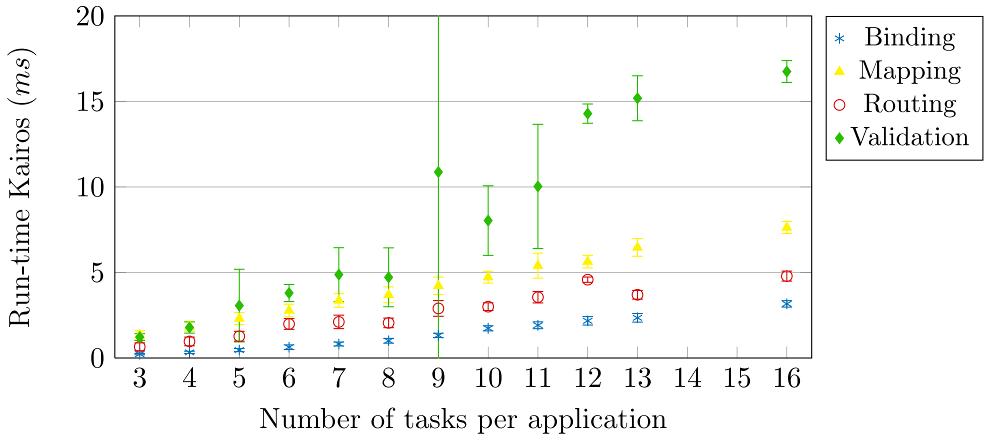
Figure 6.5 plots the average number of hops required per channel, as a function of the order in the application sequence. The communication resources per channel in the “mesh64” platform is on average 3.32±0.48 hops; in the “crisp51” platform it is on average 3.58±0.46 hops.

6.3 Discussion

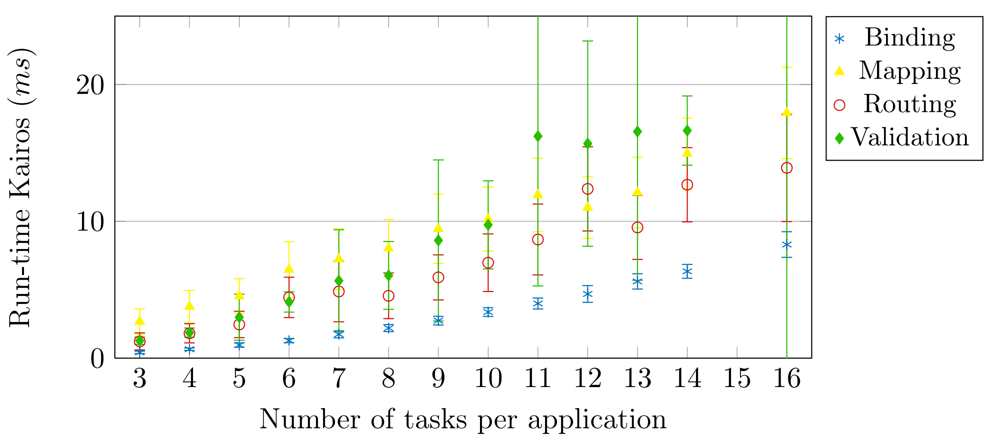
The results we presented in the previous sections show that the mapping cost function does influence the results in a deterministic way. Optimizing towards low communication leads to less communication overhead (see Figure 6.5), and trying to reduce fragmentation has effect as well (see Figure 6.4). Setting the cost function to both objectives, gives results that lie between the results of the individual objectives. When the cost function is disabled, denoted with “None”, the algorithms fall back to the communication minimization inherent to the search method.

Figure 6.3 shows for a specific case that not all optimization directions result in feasible execution layouts. In fact, for that specific case only certain ratios between the weights of the fragmentation and communication objectives leads to admission of the application. Disabling either one of them never gives a succesful result. That both factors are important is backed by Figure 6.5, where the allocated communication resources decrease for applications further in the application sequence. When less resources are available, one would expect that the communication routes become longer. On the contrary, we observe that an application is only admitted to a platform with few remaining resources, when those resources are not too widely spread over the platform. It seems likely that the admission chances increase, wehn the fragmentation objective tries to maintain clusters of available resources. This effort results in slightly more allocation of communication resources, but at a more constant ratio over time compared to the other cost functions (see Figure 6.5).

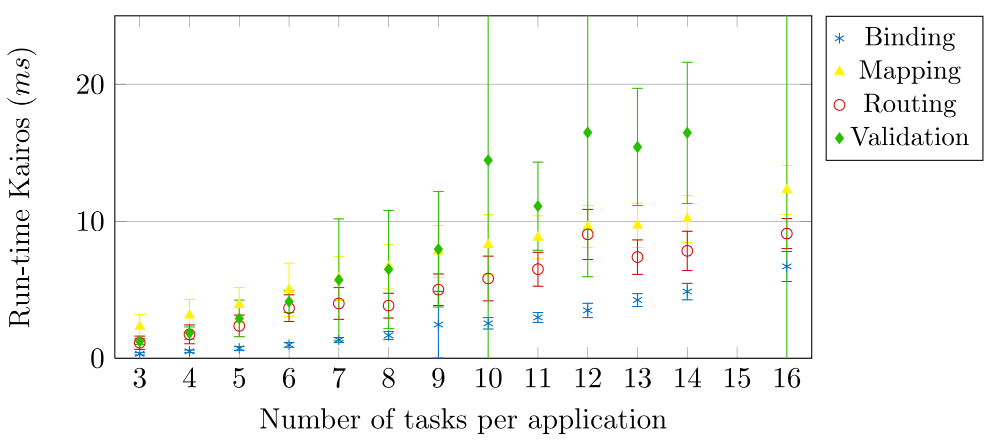
When we compare the topology and structure of the “mesh64” and “crisp51” platforms, we see that the “crisp51” is less connected than the regular meshed platform “mesh64”. Figure 6.4 shows that the “crisp51” platform suffers less from external fragmentation. This is inherent to its topology, because the borders of the individual packages do not contribute to the fragmentation. The penalty for this structure is its communication requirements, which are higher than that of the "mesh64" platform (see Figure 6.5). Table 6.3 shows this as well; the "mesh64" platform has a clear separation between computational intensive applications failing in the binding phase, and communication oriented application failing in the routing phase. Looking at the "crisp51" platform, significantly more computational intensive applications fail in the routing phase, while there are still computational resources available. Following these arguments, it is no surprise that the "ring23" platform shows results that lie somewhere between the other two platforms.



(a) The "ring 23" platform.



(b) The "mesh 64" platform.



(c) The "crisp 5.1" platform.

Figure 6.1: Runtimes of Kairos for the applications in the synthetic datasets.

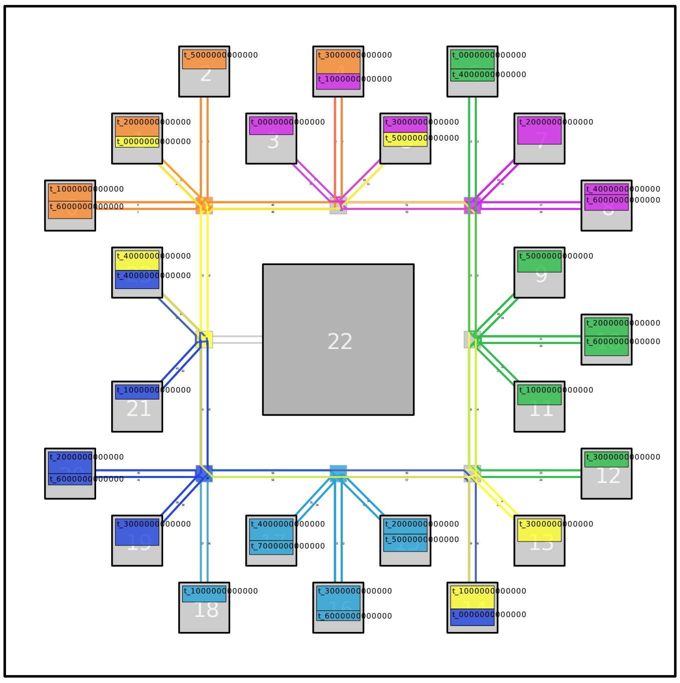


Figure 6.2: Multi-tasking is enabled for benchmarking purposes. This figure shows the fictional "Ring 23" architecture, inspired by [20].

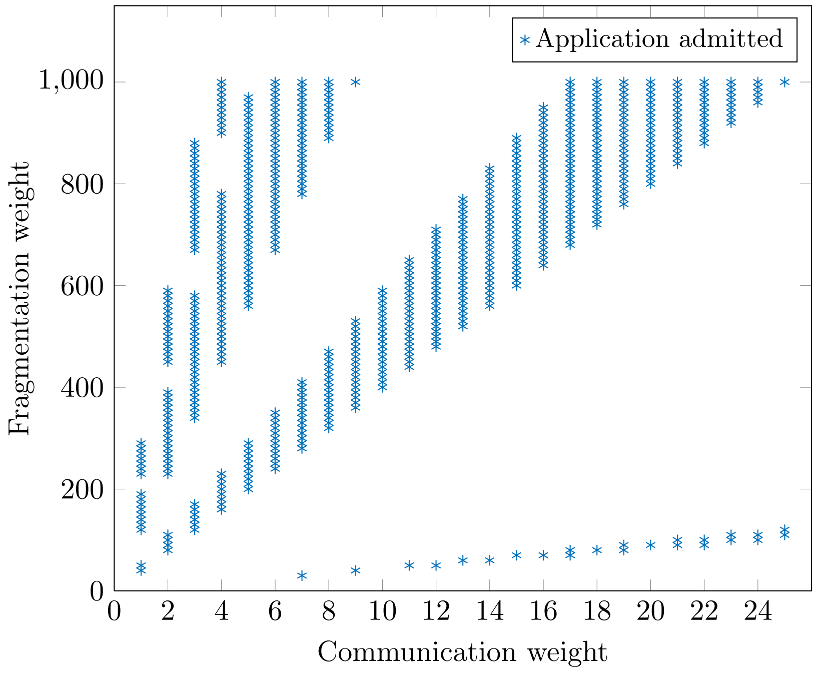


Figure 6.3: Admission of the CRISP application on the "crisp51" platform with varying mapping parameters. Every point in $[0, 1, .., 25] \times [0, 10, .., 1000]$ is sampled.

Table 6.3: Percentage of failures per phase.

(a) The "ring23" platform.

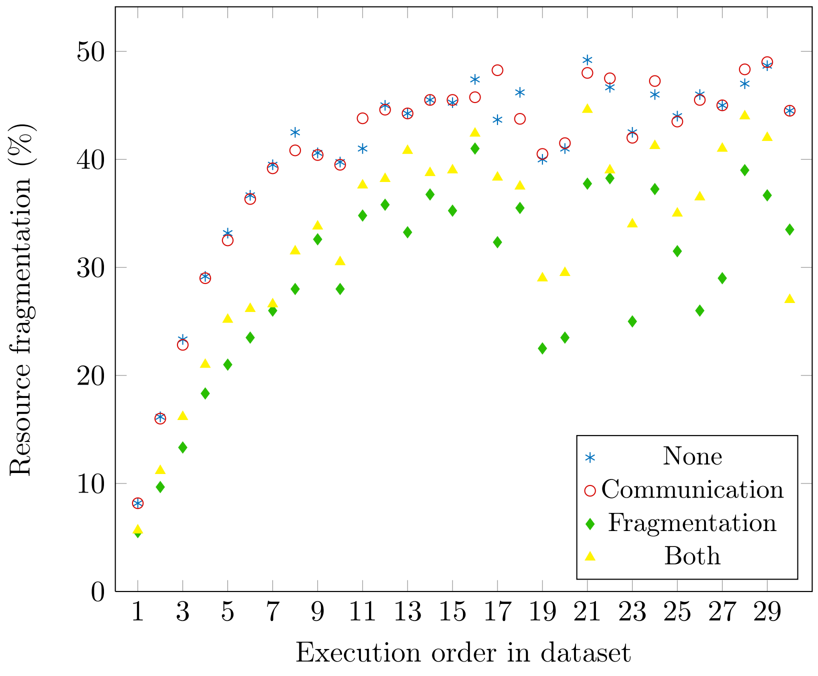
Dataset			**Percentage of failure**		
Characteristics		Size	Binding	Mapping	Routing
Communication	Small	9900	6.73%	1.13%	73.01%
Communication	Medium	4800	38.27%	5.98%	41.35%
Communication	Large	750	5.47%	0.40%	52.93%
Computation	Small	9300	89.60%	0.13%	0.13%
Computation	Medium	6600	89.18%	0.39%	3.03%
Computation	Large	900	49.00%	0.00%	20.11%

(b) The "mesh64" platform.

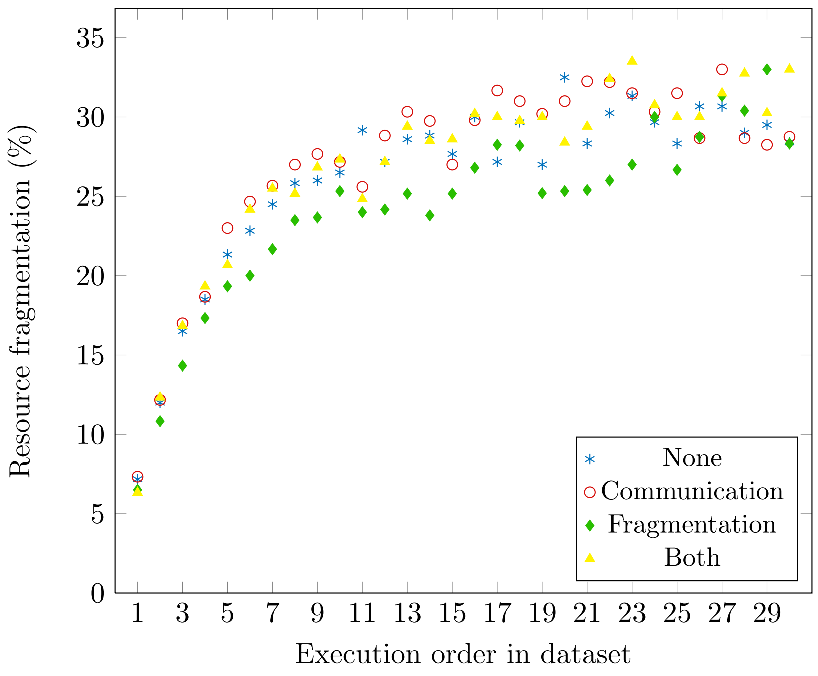
Dataset			**Percentage of failure**		
Characteristics		Size	Binding	Mapping	Routing
Communication	Small	12300	0.41%	0.03%	62.07%
Communication	Medium	7350	23.13%	1.74%	51.24%
Communication	Large	1200	0.00%	0.00%	46.00%
Computation	Small	13050	83.14%	0.00%	0.00%
Computation	Medium	11250	88.89%	0.04%	0.02%
Computation	Large	6450	88.42%	0.12%	0.70%

(c) The "crisp51" platform.

Dataset			**Percentage of failure**		
Characteristics		Size	Binding	Mapping	Routing
Communication	Small	14550	0.46%	0.00%	69.92%
Communication	Medium	8550	10.98%	1.49%	68.90%
Communication	Large	3300	2.91%	0.28%	81.48%
Computation	Small	14850	80.95%	0.29%	3.95%
Computation	Medium	14100	79.95%	0.02%	11.65%
Computation	Large	11400	58.31%	0.01%	36.00%

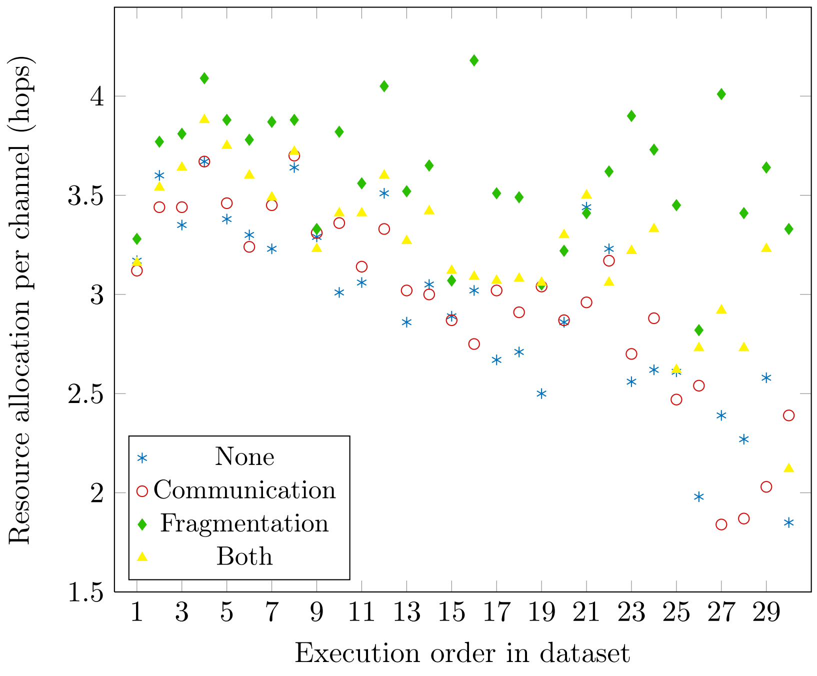


(a) The "mesh 64" platform.

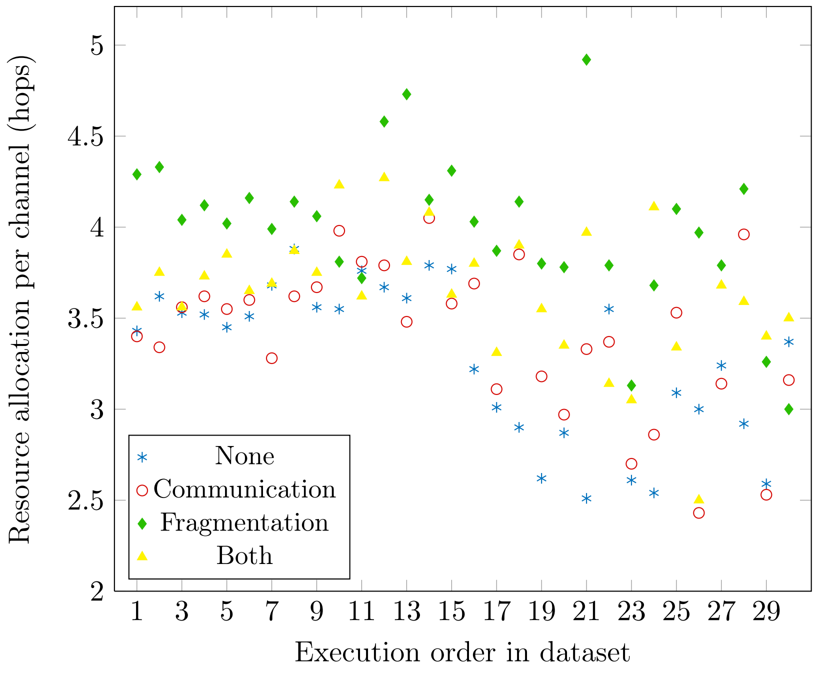


(b) The "crisp 5.1" platform.

Figure 6.4: External fragmentation of platform resources, averages over all datasets, using various optimization criteria.



(a) The "mesh 64" platform.



(b) The "crisp 5.1" platform.

Figure 6.5: Average communication resources allocated per channel.

CHAPTER 7

Conclusion

This thesis shows that run-time resource management for applications on large heterogeneous MPSoCs (>100 elements) is possible with acceptable delays. For large applications (±20 tasks) the calculation time on a low-end processor is in the order of tens of milliseconds, growing to a few hundred milliseconds if multiple iterations are required.

This thesis introduced a framework with instrumentation to influence the direction in which solutions are sought. The proposed resource manager can therefore be tailored to a variety of platforms or application domains. We experienced that even on an application-basis, parameters may be altered to improve the solution.

A larger search space, for example empty platforms or highly heterogeneous applications, increases the expectation for good results, whilst increasing the complexity of the problem for the resource manager. With less flexible applications or fewer available resources, the admission rate may decrease, but the verdict is produced faster. Limiting the amount of flexibility, for example by giving I/O tasks only a few options instead of all compatible elements, benefits the resource management process. In practice, it also seems reasonable that the location of inputs and outputs of the application is fixed before the application is started.

For the proof-of-concept platform, we developed a binary format for MPSoC applications. While the specification still needs to evolve, it is the first step towards an interface transparent system. We showed that from a user perspective an MPSoC platform does not have to be dealt with differently than a classical platform.

7.1 Recommendations for further research

This thesis covers many aspects of spatial resource management at run-time. Therefore, we have not covered every detail in depth, leading to the following list of suggestions for further research and improvement of the implementation we presented.

7.1.1 Quality-of-service and partial relocation

Some applications may be specified with multiple QoS levels; this is not uncommon for streaming applications. Instead of a binary admission control scheme, we can introduce an admission controller that takes multiple QoS levels into account. When a submitted application is accepted, the resource manager guarantees that one of the QoS levels is served. Switching between QoS levels while the application is running is optional; this transition likely violates the performance guarantees for a short period of time.

Implementing support for multiple QoS levels is tightly coupled with the ability to do partial relocation. When hardware fails while running an application, some parts of the application should be relocated to other elements in the system, but preferable not the entire application. The same scenario may occur when parts of the system have to be cleared for testing purposes. We think that support for partial location can be implemented within the proposed framework without fundamental changes. Real world applications may be useful to aid the development of ways to specify QoS levels.

7.1.2 Failure resolution

The current implementation Kairos only supports a single iteration through the resource management phases. If an attempt to derive a feasible execution layout fails, the application is rejected. After identification of the problem areas, we may choose a different binding in the first phase by disabling or increasing the cost of certain implementations. We think however, that the weakest point in the algorithms we proposed, is the starting point in the mapping phase, where we choose an element for the task with the lowest degree in the application. At that point, a different choice may lead to a solution that is unrelated to the failed attempt.

Identification of multiple application classes may increase the chance that we make a good decision for a starting point in the task graph. Alternatively, an application engineer may provide good starting points to the run-time spatial resource manager in the application specification.

The proposal in the following section may also aid in a suitable feedback mechanism.

7.1.3 Alternative validation algorithm

The validation algorithm we proposed in Chapter 4 has at least two drawbacks; the algorithm may be slow in some cases, and it is complex to generate feedback information. The authors of our current validation methods proposed an alternative approach for throughput calculation of SDF graphs in [76]. This approach has the potential to solve both problems we identified for our current implementation.

At design-time the values of the response time of actors may be unknown, due to the choice of multiple implementations per actor and the variability in communication latencies. We may however, specify the unknown response times with parameters. With a throughput analysis at design-time we can derive a system of linear equations expressing the throughput of the application in terms of the response times of individual actors. Note that at design-time, for each communication channel, a parametrized actor has to be defined to model the communication limitations. Like our current approach, this set of equations can be evaluated at run-time, to obtain the throughput of the application, by filling in the parameters with the response times of the actors, given by the execution layout.

Various approaches are possible to implement this idea [76], and we illustrate one approach here with the example SDF graph of Chapter 4. At design-time we convert the SDF graph (application specification) to an HSDF graph, resulting in the graph in Figure 7.1. Each cycle in the HSDF graph is then represented by a linear combination of parameters, where each parameter denotes the response time of an actor. Such an expression denotes the cycle main λ_c, which is the inverse of the throughput of that cycle.

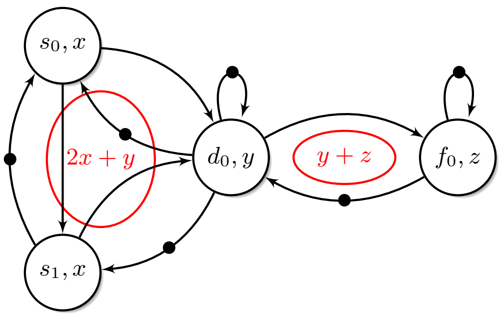


Figure 7.1: Parametrized HSDF graph of our example.

In [76], two methods are given to reduce the set of linear combinations to a minimum set of dominating expressions. Removing all redundant expressions results in *dominant cycle mean set*. In Figure 7.1, we indicated the dominant cycles and annotated them with their *cycle mean expression* (compare to Figure 4.13). Each cycle mean expression $\bar{e}_c$ should also contain a constant factor, which represents the total response time of the non-parametrized actors.

$$\lambda_c(\bar{p}) = \bar{e}_c \cdot (\bar{p}, 1)$$

$$\lambda_1(\bar{p}) = \langle 2, 1, 0, 0 \rangle \cdot (\bar{p}, 1)$$
$$\lambda_2(\bar{p}) = \langle 0, 1, 1, 0 \rangle \cdot (\bar{p}, 1)$$

Now we can fill in any point $\bar{p} \in \mathbb{R}^d$ from the parameter space into these expressions to obtain for each dominant cycle its cycle mean. The cycle with the largest cycle mean is then the critical cycle in the execution layout of the application.

$$\begin{aligned} &\text{with } \bar{p} = \langle 1,2,1\rangle \\ \lambda_1(1,2,1) &= \langle 2,1,0,0\rangle \cdot \langle 1,2,1,1\rangle = 4 \\ \lambda_2(1,2,1) &= \langle 0,1,1,0\rangle \cdot \langle 1,2,1,1\rangle = 3 \end{aligned}$$

$$\begin{aligned} &\text{with } \bar{p} = \langle 1,2,8\rangle \\ \lambda_1(1,2,8) &= \langle 2,1,0,0\rangle \cdot \langle 1,2,1,1\rangle = 4 \\ \lambda_2(1,2,8) &= \langle 0,1,1,0\rangle \cdot \langle 1,2,8,1\rangle = 10 \end{aligned}$$

Using this approach, the complexity of the throughput analysis is moved from run-time to design-time, making the validation approach a lot faster. To obtain the actual throughput of the application, a single iteration over the set of dominant expressions is required. This provides means to evaluate the performance of the application at various stages during the resource allocation process. Additionally, feedback information on critical cycles in the execution layout is easily available.

The most interesting aspect of this approach is the interaction between performance evaluation and scheduling. Evaluation of the throughput expressions requires all arguments to be known. However, the parametrized response time of the tasks includes the effects of run-time scheduling. A schedule should satisfy the throughput constraints, while we may want to restrain the greediness of the scheduler.

7.2 Acknowledgments

I want to express my gratitude to all who gave me the possibility to complete this thesis. Philip, I would like to thank you for your introduction into the subject, you continuous involvement in the project, and your criticism towards the sense and nonsense I brought to the table. Also, your reviews and assistance helped me to greatly improve the quality of my writings. Gerard Smit, in your role of professor, you gave me the opportunity to work on this subject. I enjoyed the trip to Rheinfels, and I hope for more memorable events in the coming years. Furthermore, I am grateful for the other members of my graduation committee: Johann Hurink and Jan Kuper, thanks for the many comments and advice.

APPENDIX A

MPSoC File Format

This reference document describes the structure of the MPSoC file format, which is definition of the way to store applications developed for MPSoC platforms. The MPSoC file format combines individual implementations for tasks into a single binary file, providing the required information to construct an SDF graph of the application. As a convention, MPSoC binaries use the `.mpsoc` extension in their filenames.

Basic Structure

An MPSoC binary consists of three segments, as illustrated by Figure A.1. The first segment contains the file header; the second segment contains the application structure and an index for the implementation data files, which are located somewhere in the third segment. In this research project, the third segment never was fully developed, as this requires more details of both hardware and configuration methods.

Here we describe the first two segments, which can contain four major types of blocks, and two minor block types. The order in which these blocks appear in the file is fixed:

- At the beginning of every MPSoC file is a `mpsoc_header` structure that identifies the file as an MPSoC file. The header contains information about the number of other blocks per type that can be expected in the second segment.
- Directly following the header are a series of `mpsoc_task` structures that specify the tasks that compose the application embedded in the file.
  - Any number of `mpsoc_port` structures may be specified here. A port is the interface between the task and a channel.
  - Each task should have at least one implementation to choose from. Following the ports, `mpsoc_impl` structures can be used to provide implementations for the task.

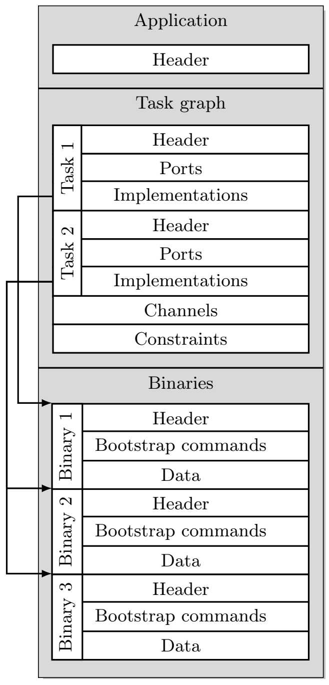


Figure A.1: Basic structure of the MPSoC file format.

- Following the task structures, `mpsoc_chan` structures specify the channels present in the application.
- The last block contains `mpsoc_constraint` structures, used to specify performance constraints on the execution of the application.

In an MPSoC binary, precisely one `mpsoc_header` and at least one `mpsoc_task` with at least one `mpsoc_impl` should be defined. Other blocks are optional, and the number of blocks per type are bounded by the `uint32_t` data type. Regardless of the implementation data files this binary contains, all its fields are stored in little-edian byte order.

mpsoc_header

The `mpsoc_header` data structure is placed at the start of a binary that contains an application for an MPSoC platform. This header defines the layout of the binary, and is declared in `linux/mpsoc.h`.

```
struct mpsoc_header {
        uint32_t magic1;
        uint32_t magic2;
        char name[16];
        uint32_t no_tasks;
        uint32_t no_ports;
        uint32_t no_impls;
        uint32_t no_channels;
        uint32_t no_constraints;
};
```

Fields

`magic1`
: A value that identifies this file as an `MPSoC` file. Use the constant `MPSOC_MAGIC_1` if the file is intended to run on an MPSoC platform.

`magic2`
: A second file identification field. Use the constant `MPSOC_MAGIC_2` defined in `linux/mpsoc.h` to identify this binary of type `MPSoC`.

`name`
: A human readable representation of the application embedded in this binary. This field must be null-terminated, so the maximum length of the name is 15 characters.

`no_tasks`
: An integer specifying the number of `mpsoc_task` data structures that follow. This is the number of tasks within the application.

`no_ports`
: An integer indicating the total number of ports attached to the tasks in the application.

`no_impls`
: An integer indicating the total number of implementations for the tasks in the application.

`no_channels`
: An integer specifying the number of `mpsoc_chan` data structures that follow. This is the number of channels in the application.

`no_constraints`
: An integer specifying the number of `mpsoc_constraint` data structures that follow. This is the number of constraints posed on the application.

mpsoc_task

```
struct mpsoc_task {
        uint32_t id;
        char name[16];
        uint32_t no_ports;
        uint32_t no_impls;
};
```

Fields

`id`
: An integer that uniquely identifies this task within an application. A valid range for this field is `[0..no_tasks-1]`.

`name`
: A human friendly representation for this task. This field must be null-terminated, so the maximum length of the name is 15 characters.

`no_ports`
: An integer specifying the number of `mpsoc_port` data structures that follow. This should correspond with the number of incoming and outgoing channels to this task.

`no_impls`
: An integer specifying the number of `mpsoc_impl` data structures that follow. This is the number of implementations available for this task.

mpsoc_port

An `mpsoc_port` block has three required attributes.

```
struct mpsoc_port {
        uint32_t id;
        kairos_port_t type;
        uint32_t rate;
};
```

Fields

`id`

An integer that uniquely identifies this port within an application. This id is used when connecting the port to a channel. A valid range for this field is `[0..no_ports-1]`.

`type`

An enumeration value of type `kairos_port_t`, declared in `linux/kairos/application/port.h`. This field specifies whether this port is an input or output port for the task it is attached to. When a port is of type `PORT_IN`, the actor will during a firing read tokens from the channel to which the port is connected. When a port is of type `PORT_OUT`, the actor will during a firing write tokens to the channel to which the port is connected.

`rate`

An integer that indicates the communication rate of the task.

mpsoc_impl

Describes the characteristics of an implementation for a specific architecture, and its location within the MPSoC binary. If an implementation can be used for multiple tasks, its data file has to be provided only once, but for each task it implements an `mpsoc_impl` structure should be generated.

```
struct mpsoc_impl {
        kairos_arch_t arch;
        uint32_t mips;
        uint32_t memorySize;
        uint32_t numSlots;
        uint32_t execTime;
        uint32_t dynamicCost;
        uint32_t offset;
        uint32_t size;
        uint32_t align;
};
```

Fields

`arch`

An enumeration value of type `arch_type_t`, declared in `linux/kairos/platform/arch.h`. This specifies the architecture required to execute this implementation.

`mips`

An integer denoting the amount of processing capacity required to achieve some functionality within `execTime` time.

`memSize`

An integer indicating the maximum size (in bits) of the state of the implementation's task on the specified architecture.

`numSlots`

An integer indicating the number of processes this implementation will spawn when executed. Note that some architectures or platforms are not capable of multi-tasking.

`execTime`

An integer indicating the (worst-case) execution time (in time-units) of the implementation's task on the specified architecture.

`dynamicCost`

An integer indicating the increased energy consumption of the specified architecture during execution of this implementation, as opposed to the static energy consumption of the specified architecture.

`offset`

Offset to the beginning of the data file for this implementation.

`size`

Size of the implementation data file.

`align`

The power of 2 alignment for the offset of the implementation data file for the architecture specified in `arch` within the binary. This is required to ensure that, if this binary is changed, the contents it retains is correctly aligned for virtual memory paging and other uses.

mpsoc_chan

Defines a communication channel between two ports. The ports of a channel can be attached to the same task (self-loop) or they can connect two distinct tasks.

```
struct mpsoc_chan {
        uint32_t id;
        uint32_t src;
        uint32_t dst;
        uint32_t initialTokens;
        uint32_t tokenSize;
        uint32_t minBandwidth;
};
```

Fields

`id`

An integer that uniquely identifies this channel within an application. A valid range for this field is [0..`no_channels`-1].

`src`

The source port of this channel. Should be specified with an id of an `mpsoc_port`. A valid range for this field is [0..`no_ports`-1].

`dst`
: The destination port of this channel. Should be specified with an id of an `mpsoc_port`. A valid range for this field is [0..`no_ports`-1].

`initialTokens`
: The number of tokens that reside on this channel when the application is not yet started.

`tokenSize`
: The size of the tokens that are communicated over this channel.

`minBandwidth`
: The minimal amount of bandwidth that should be reserved for this channel.

mpsoc_constraint

Defines constraints posed on the mapping of the application in this binary.

```
typedef enum {
        CONSTRAINT_THROUGHPUT = 0,
        CONSTRAINT_LATENCY = 1
} kairos_constraint_t;

struct mpsoc_constraint {
        kairos_constraint_t type;
        uint32_t numerator;
        uint32_t denominator;
        uint32_t src;
        uint32_t snk;
};
```

Fields

`type`
: An enumeration of type `kairos_constraint_t`, declared in `linux/kairos/application/constraint.h`. Two types of constraints can be specified. A thoughput constraint (`CONSTRAINT_THROUGHPUT`) specifies the minimal throughput that should be achieved by the application. A latency constraint (`CONSTRAINT_LATENCY`) is gives the maximum allowed latency between two tasks.

`numerator`
: The number of fractional units that determines the value of this constraint.

`denominator`
: The divisor of the fractional value of this constraint.

`src`
: The source task as one end of a latency constraint.

`snk`
: The sink task as the other end of a latency constraint.

APPENDIX B

Case study: Beamforming application

Beamforming is used in radar, wireless communication and astronomy. It is a signal processing technique that processes data from sensor or antenna arrays for directional signal reception (or transmission). As the signal of interest arrives from a certain angle from the perspective of an antenna array, each antenna receives the signal at a different time. Some processing is required to calculate the time delay and to apply a phase shift. Also, some filtering and equalization may improve the quality of the signal. Beamforming is a streaming application, characterized by a high data rate and critical timing requirements. An application that works on multiple channels and beams usually has a regular task structure. As a case study, we modeled a beamforming application with 16 input channels and 8 output beams. This application was developed in the CRISP project [7].

B.1 Specification

At the time of modeling, no exact execution times for tasks were known, as the application was still in development. For each task, some preliminary numbers of the load on processing elements was available. We used these numbers to derive an execution time for each task. We assume that a Xentium processor running at 200 MHz, which is used as a DSP on the CRISP platform, has a processing power of 800 Million Multiply Accumulate (MMAC) operations per second. For example, a processing load of 640 MMAC multiplied with the token rate of 2.5 MHz then gives an execution time of 320 ns:

$$\frac{640}{800}\ MMAC \cdot \frac{1}{2.5 \cdot 10^{6}\ Hz} = 320 \cdot 10^{-9}\ s = 320\ ns$$

Likewise the execution times of tasks are derived that have a different processing load. As every processing element is assigned a single task at maximum, the response time of each task equals its execution time. The response times can be found in Figure B.1, using the SDF notation introduced in Chapter 4.

B.1.1 Performance constraints

The throughput constraint of this application is dictated by the throughput of the input tasks, which produce tokens at a periodic rate. The application must achieve at least the throughput of those input tasks to avoid incorrect behavior. The throughput is at the same time bounded by these input tasks. So, if a feasible execution layout is presented, it will have exactly that throughput.

In the beamforming application, an input actor fires $2.5 \cdot 10^6$ times per second. As actor execution times are given in ns, we thus require a throughput of $2.5 \cdot 10^{-3}$ tokens per nanosecond. The application thus requires a throughput of 1 sample per 400 ns.

If the communication routes of the channels have different latencies, some channels or tasks need to buffer data streams for a while. Unfortunately, we had no information regarding the amount of buffer space available within each task. To overcome this problem, we applied the approach described in Section 4.6.2. We specified some latency constraints from the output tasks to the input tasks of the application. These constraints limit the difference in amount of firings between the inputs and outputs. The number of firings that an output may lag behind the input is determined by the amount of data the application can contain in its buffers.

A conservative approach is to assume that each task can only contain a single sample in its buffer. To fill the pipeline in the application, we place 8 initial tokens on the latency constraint; having $\beta_l = 8$ in Figure 4.10. The total execution time of every processing chain is 1840 ns. The lowest response time in the chain is 80 ns, and by the assumption of buffer sizes of 1 sample, the maximum latency between the input and output of the processing chains may be 1840 + 80 = 1920 ns. This results in a response time of the latency actors l_{ij} of:

$$\begin{aligned}\rho(l_{ij}) &= \beta_l \cdot \mu(\mathcal{G}) - L, \quad i = 0..3, j = 0..3 \\ &= 8 \cdot 400 - 1920 \\ &= 1280\end{aligned}$$

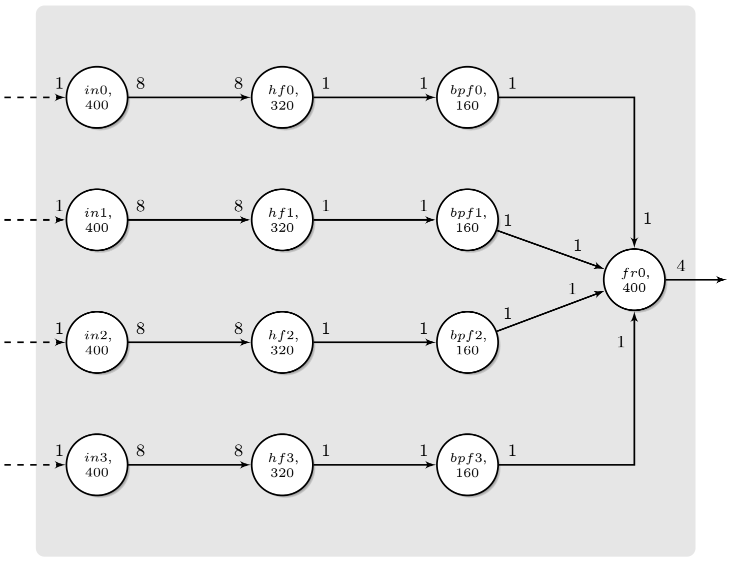
Note that for practical reasons we multiplexed every four "latency cycles" into a single arc in Figure B.1.

B.2 Resulting execution layout

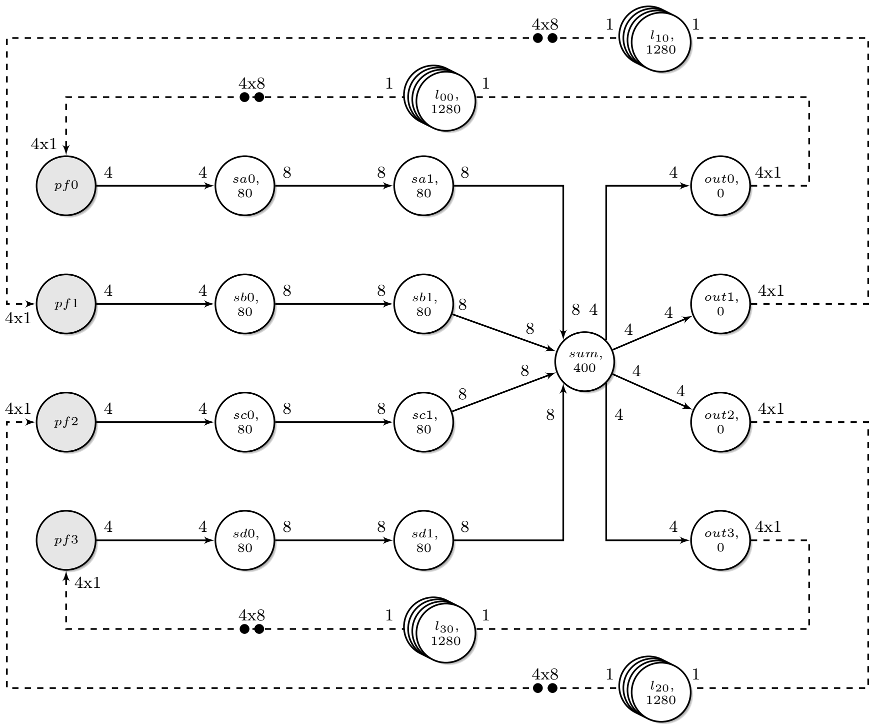
The execution layout of the beamforming application produced by Kairos is given in Figure B.2. The leftmost block models an Field Programmable Gate Array (FPGA), which holds all the input and output tasks of the application. The rightmost block is a general purpose processor, and that should be the ARM processor where the operating system runs. The other blocks are packages that contain 9 Xentium processors, 2 smart memory tiles and an undefined element.

In this research project, we tried to maintain a hardware agnostic view of a platform, maybe at the cost of some optimality in the proposed heuristics. The abstraction used to define the platform has an effect on the resulting execution layout. In an earlier attempt, the FPGA was modeled to have 10 generic I/O ports, instead of 10 distinct ports with the same capabilities. In the latter case, an input port within an application may require a more specific subset of hardware ports to be mappable to. The amount of restriction to apply to the location of inputs and outputs of a tasks within a platform is scenario specific. But, such restriction often implies a human made ordering, and therefore increases the "quality" of a mapping.

In the beamforming case, it was difficult to distinct the I/O ports of the FPGA in terms of locality. Starting from an arbitrary FPGA port, the topology of the platform is very homogeneous, only discovering at the very end that the mapping of I/O tasks does make a huge difference. Therefore, we restricted each input and output task to two FPGA ports, such that each processing block of Figure B.1a is mapped to single package (block of 9 Xentiums).



(a) Antenna processing and filtering block for 4 input channels.



(b) A beamformer with 4x4 input channels (grey blocks) and 2x4 output beams.

Figure B.1: Specification of a beamforming application. The block in (a) is used 4 times in (b). At run-time, a self-loop is added to every actor (omitted). Tokens are 32-bit long, and execution times are in *ns*.

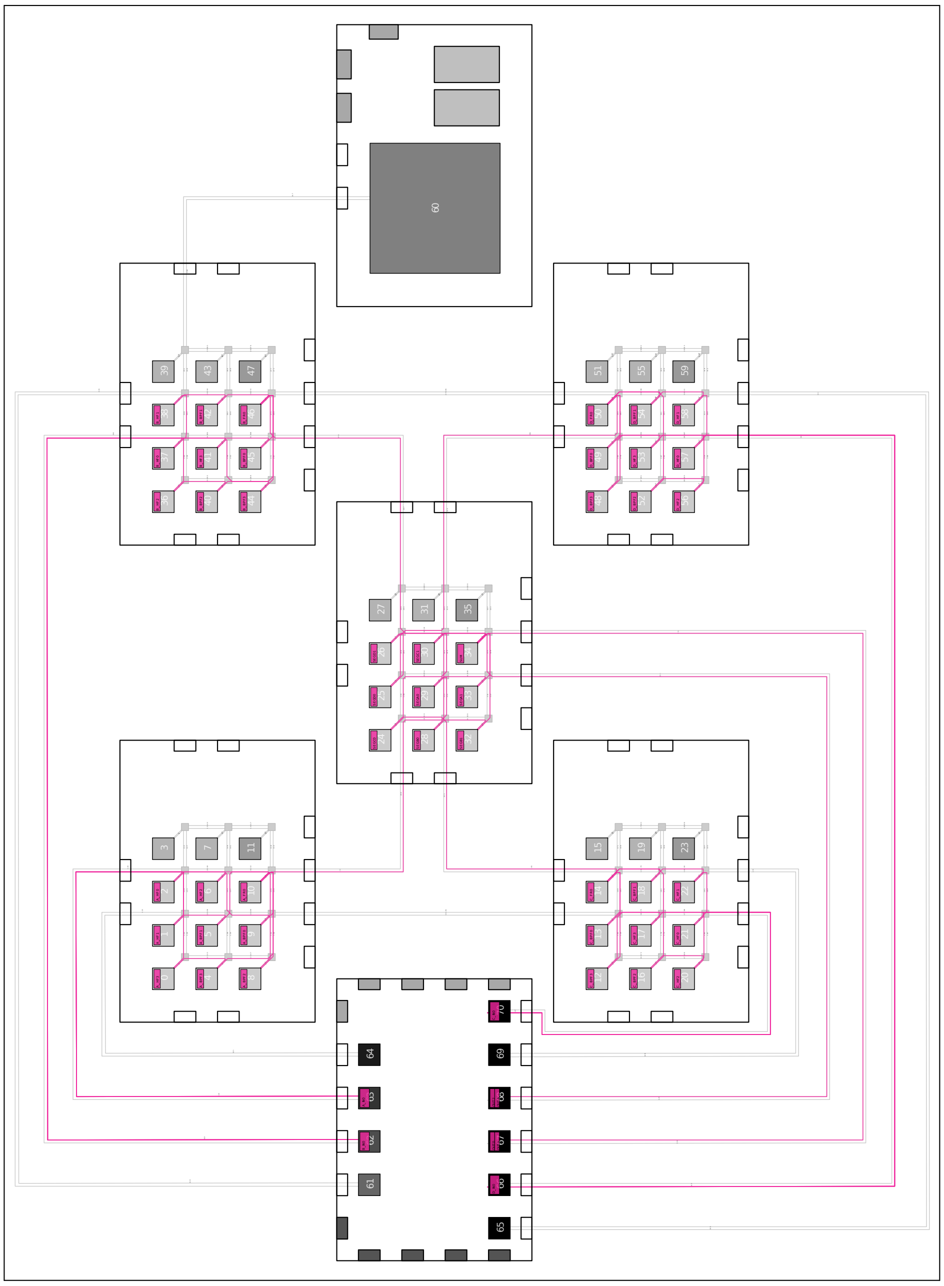


Figure B.2: Execution layout of the beamformer on the CRISP hardware platform.

References

[1] J. S. Chase and R. P. Doyle, "**Balance of Power: Energy Management for Server Clusters**," in *In Proc. of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[2] C. Rusu, R. Melhem, and D. Mossé, "**Maximizing rewards for real-time applications with energy constraints**," *Trans. on Embedded Computing Sys.*, vol. 2, no. 4, pp. 537–559, 2003.

[3] R. Kumar, K. I. Farkas, P. Ranganathan, and D. M. Tullsen, "**Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction**," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 81–92.

[4] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "**Daedalus: Toward composable multimedia MP-SoC design**," *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 574–579, Jun. 2008.

[5] M. Horowitz and W. Dally, "**How scaling will change processor architecture**," in *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, vol. 1, Feb. 2004, pp. 132–133.

[6] D. A. Patterson and J. L. Hennessy, **Computer organization and design (2nd ed.): the hardware/software interface**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[7] Recore Systems BV. (2008, Feb.) **CRISP - Cutting edge Reconfigurable ICs for Stream Processing**. FP7-ICT-215881. [Online]. Available at: http://www.crisp-project.eu. [cited 2009-07-14]

[8] S. Verdoolaege, H. Nikolov, and T. Stefanov, "**pn: a tool for improved derivation of process networks**," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 19–19, 2007.

[9] R. Steuer, **Multiple Criteria Optimization: Theory, Computation, and Application**. Krieger Publishing Company, Jan. 1986.

[10] C. Ykman-Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor, and H. Corporaal, "**Design-time application mapping and platform exploration for MP-SoC customised run-time management**," *Computers & Digital Techniques, IET*, vol. 1, no. 2, pp. 120–128, Mar. 2007.

[11] C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal, "**Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management**," *System-on-Chip, 2006. International Symposium on*, pp. 1–4, Nov. 2006.

[12] W. B. Joerg, "**A subclass of Petri Nets as design abstraction for parallel architectures**," *SIGARCH Comput. Archit. News*, vol. 18, no. 4, pp. 67–77, 1990.

[13] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit, "**Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSOC)**," *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 212–217, Mar. 2008.

[14] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "**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.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 33–38.

[15] K. Jeffay, D. F. Stanat, and C. U. Martel, "**On non-preemptive scheduling of period and sporadic tasks**," *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pp. 129–139, Dec. 1991.

[16] M. H. Wiggers, "**Aperiodic multiprocessor scheduling for real-time stream processing applications**," Ph.D. dissertation, Enschede, Jun. 2009. [Online]. Available at: http://doc.utwente.nl/61568/

[17] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "**Modelling run-time arbitration by latency-rate servers in dataflow graphs**," in *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems.* New York, NY, USA: ACM, 2007, pp. 11–22.

[18] M. A. A. Faruque, R. Krist, and J. Henkel, "**ADAM: run-time agent-based distributed application mapping for on-chip communication**," in *DAC '08: Proceedings of the 45th annual conference on Design automation.* New York, NY, USA: ACM, 2008, pp. 760–765.

[19] C.-L. Chou and R. Marculescu, "**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.* New York, NY, USA: ACM, 2007, pp. 161–166.

[20] O. Moreira, J. J.-D. Mol, and M. J. G. Bekooij, "**Online resource management in a multiprocessor with a network-on-chip**," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing.* New York, NY, USA: ACM, 2007, pp. 1557–1564.

[21] O. J. Kuiken, X. Zhang, and H. G. Kerkhoff, "**Built-In Self-Diagnostics for a NoC-Based Reconfigurable IC for Dependable Beamforming Applications**," in *DFT '08: Proceedings of the 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 45–53.

[22] A. M. Devices. (2008, Mar.) **Live Migration with ADM-V Extended Migration Technology**. White paper. [Online]. Available at: http://developer.amd.com/assets/LiveVirtualMachineMigrationonAMDprocessors.pdf. [cited 2009-03-26]

[23] VMware. (2008, Jun.) **VMware VMontion and CPU compatibility**. Information Guide. [Online]. Available at: http://www.vmware.com/files/pdf/vmotion_info_guide.pdf. [cited 2009-03-26]

[24] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, "**Low Cost Task Migration Initiation in a Heterogeneous MP-SoC**," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 252–253.

[25] K. Zhang and S. Pande, "**Minimizing downtime in seamless migrations of mobile applications**," in *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems.* New York, NY, USA: ACM, 2006, pp. 12–21.

[26] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner, "**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.* New York, NY, USA: ACM, 2005, pp. 33–38.

[27] K. Srinivasan and K. S. Chatha, "**A Technique for Low Energy Mapping and Routing in Network-on-Chip Architectures**," *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pp. 387–392, Aug. 2005.

[28] V. Nollet, T. Marescaux, P. Avasare, and J.-Y. Mignolet, "**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.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 234–239.

[29] P. K. F. Hölzenspies, J. Kuper, G. J. M. Smit, and J. L. Hurink, "**Demonstration of Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSoC)**," in *Dagstuhl Seminar Proceedings 07101, Dagstuhl Wadern, Germany*, B. R. H. M. Haverkort, J. P. Katoen, and L. Thiele, Eds., vol. 07101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Oct. 2007.

[30] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. D. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. S. Yellampalli, "**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.* Washington, DC, USA: IEEE Computer Society, 2003, p. 98.1.

[31] E. Carvalho, N. Calazans, and F. Moraes, "**Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs**," in *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 34–40.

[32] D. I. Kang, J. Suh, O. McMahon, and S. Crago, "**Preliminary Study towards Intelligent Run-time Resource Management Techniques for Large Multi-Core Architectures**," University of Southern California – Information Sciences Institute, Tech. Rep., Sep. 2007, high-Performance Computing Workshop.

[33] K. Li and K. H. Cheng, "**A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system**," in *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation.* New York, NY, USA: ACM, 1990, pp. 22–27.

[34] A. Hansson, K. Goossens, and A. Rădulescu, "**A unified approach to constrained mapping and routing on network-on-chip architectures**," in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* New York, NY, USA: ACM, 2005, pp. 75–80.

[35] N. Kavaldjiev, G. J. M. Smit, P. T. Wolkotte, and P. G. Jansen, "**Providing QoS Guarantees in a NoC by Virtual Channel Reservation**," in *ARC*, 2006, pp. 299–310.

[36] N. K. Kavaldjiev, "**A run-time reconfigurable Network-on-Chip for streaming DSP applications**," Ph.D. dissertation, University of Twente, Enschede, Jan. 2007.

[37] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, "**The performance and energy consumption of embedded real-time operating systems**," *Computers, IEEE Transactions on*, vol. 52, no. 11, pp. 1454–1469, Nov. 2003.

[38] P. T. Wolkotte, G. J. M. Smit, N. Kavaldjiev, J. E. Becker, and J. Becker, "**Energy Model of Networks-on-Chip and a Bus**," *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pp. 82–85, Nov. 2005.

[39] S. Skiena, **Implementing discrete mathematics: combinatorics and graph theory with Mathematica**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1991.

[40] J. Beck and D. Siewiorek, "**Modeling Multicomputer Task Allocation as a Vector Packing Problem**," in *ISSS '96: Proceedings of the 9th international symposium on System synthesis.* Washington, DC, USA: IEEE Computer Society, 1996, p. 115.

[41] A. Hansson, K. Goossens, M. J. G. Bekooij, and J. Huisken, "**CoMPSoC: A template for composable and predictable multi-processor system on chips**," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–24, 2009.

[42] F. Glover, J. Hultz, and D. Klingman, "**Improved Computer-Based Planning Techniques. Part II**," *Interfaces*, vol. 9, no. 4, Aug. 1979.

[43] A. Ceselli and G. Righini, "**A Branch-and-Price Algorithm for the Multilevel Generalized Assignment Problem**," *Oper. Res.*, vol. 54, no. 6, pp. 1172–1184, 2006.

[44] M. Laguna, J. P. Kelly, J. González-Velarde, and F. Glover, "**Tabu search for the multilevel generalized assignment problem**," *European Journal of Operational Research*, vol. 82, no. 1, pp. 176–189, Apr. 1995. [Online]. Available at: http://dx.doi.org/10.1016/0377-2217(93)E0174-V

[45] A. P. French and J. M. Wilson, "**Heuristic Solution Methods for the Multilevel Generalized Assignment Problem**," *Journal of Heuristics*, vol. 8, no. 2, pp. 143–153, 2002.

[46] R. Cohen, L. Katzir, and D. Raz, "**An efficient approximation for the generalized assignment problem**," *Inf. Process. Lett.*, vol. 100, no. 4, pp. 162–166, 2006.

[47] S. Martello and P. Toth, **Knapsack problems: algorithms and computer implementations**. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[48] A. Levitin, "**Do we teach the right algorithm design techniques?**" *SIGCSE Bull.*, vol. 31, no. 1, pp. 179–183, 1999.

[49] R. Dechter and J. Pearl, "**Generalized best-first search strategies and the optimality of A***," *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985.

[50] E. W. Dijkstra, "**A note on two problems in connexion with graphs**," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available at: http://dx.doi.org/10.1007/BF01386390

[51] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen, "**Latency Minimization for Synchronous Data Flow Graphs**," *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pp. 189–196, Aug. 2007.

[52] P. K. F. Hölzenspies, G. J. M. Smit, and J. Kuper, "**Mapping streaming applications on a reconfigurable MPSoC platform at run-time**," in *Proceedings of the International Symposium on System-on-Chip (SoC 2007), Tampere, Finland.* Tampere, Finland: IEEE Circuits and Systems Society, Nov. 2007, pp. 74–77.

[53] E. A. Lee and T. M. Parks, "**Dataflow process networks**," pp. 59–85, 2002.

[54] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. J. G. Bekooij, "**Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis**," *IET Computers & Digital Techniques*, 2009.

[55] O. M. Moreira and M. J. G. Bekooij, "**Self-Timed Scheduling Analysis for Real-Time Applications**," in *EURASIP Journal on Advances in Signal Processing*, vol. 2007. Hindawi Publishing Corporation, Apr. 2007, pp. 24–37.

[56] T. Bijlsma, M. G. J. Bekooij, P. Jansen, and G. J. M. Smit, "**Communication between nested loop programs via circular buffers in an embedded multiprocessor system**," in *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*, 2008, pp. 33–42.

[57] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. J. G. Bekooij, "**Applying Dataflow Analysis to Dimension Buffers for Guaranteed Performance in Networks on Chip**," in *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 211–212.

[58] S. Goddard and K. Jeffay, "**The synthesis of real-time systems from processing graphs**," in *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, Nov. 2000, pp. 177–186.

[59] R. M. Karp, "**A characterization of the minimum cycle mean in a digraph**," *Discrete Mathematics*, vol. 23, no. 3, pp. 309–311, 1978.

[60] A. Dasdan, "**Experimental analysis of the fastest optimum cycle ratio and mean algorithms**," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 4, pp. 385–418, 2004.

[61] A. Dasdan and R. K. Gupta, "**Faster maximum and minimum mean cycle algorithms for system-performance analysis**," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 10, pp. 889–899, Oct. 1998.

[62] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi, "**Throughput Analysis of Synchronous Data Flow Graphs**," *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp. 25–36, Jun. 2006.

[63] S. Stuijk, M. C. W. Geilen, and T. Basten, "**SDF^3: SDF For Free**," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings.* IEEE Computer Society Press, Los Alamitos, CA, USA, Jun. 2006, pp. 276–278. [Online]. Available at: http://www.es.ele.tue.nl/sdf3

[64] H. Weinblatt, "**A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph**," *J. ACM*, vol. 19, no. 1, pp. 43–56, 1972.

[65] Apple. (2009, Feb.) **Mac OS X ABI Mach-O File Format Reference**. [Online]. Available at: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/Mach-O_File_Format.pdf. [cited 2009-03-26]

[66] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L. Scott, "**Fault injection framework for system resilience evaluation: fake faults for finding future failures**," in *Resilience '09: Proceedings of the 2009 workshop on Resiliency in high performance.* New York, NY, USA: ACM, 2009, pp. 23–28.

[67] VMware. (2008, Nov.) **VMware ESX 3.5 Product Datasheet**. [Online]. Available at: http://www.vmware.com/files/pdf/esx_datasheet.pdf. [cited 2009-03-25]

[68] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "**Xen and the art of virtualization**," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles.* New York, NY, USA: ACM, 2003, pp. 164–177.

[69] I. Habib, "**Virtualization with KVM**," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[70] K. P. Lawton, "**Bochs: A Portable PC Emulator for Unix/X**," *Linux Journal*, p. 7.

[71] S.-Y. Cho, Y. Chung, and J.-B. Lee, "**A Flexible Virtual Development Environment for Embedded Systems**," in *Intelligent Solutions in Embedded Systems, 2007 Fifth Workshop on*, Jun. 2007, pp. 37–47.

[72] S. Singhai, M. Ko, S. Jinturkar, M. Moudgill, and J. Glossner, "**An integrated ARM and multi-core DSP simulator**," in *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems.* New York, NY, USA: ACM, 2007, pp. 33–37.

[73] F. Bellard, "**QEMU, a Fast and Portable Dynamic Translator**," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2005, pp. 41–46.

[74] B. S. Ltd. (2008) **Realview Versatile/PB926EJ-S**. [Online]. Available at: http://www.bluewatersys.com/development/doc/realview/versatile/pb.php. [cited 2009-03-25]

[75] R. P. Dick, D. L. Rhodes, and W. Wolf, "**TGFF: task graphs for free**," in *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign.* Washington, DC, USA: IEEE Computer Society, 1998, pp. 97–101.

[76] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuijk, "**Parametric throughput analysis of synchronous data flow graphs**," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe.* New York, NY, USA: ACM, 2008, pp. 116–121.

Kronos ($\kappa\rho o\nu o\sigma$) – refers to sequential or linear time. We know kronos time as chronology, ticking time. It is measured by hours, minutes and seconds. It is the time in which we make appointments and face deadlines. It tends to be more of a nemesis or taskmaster than a friend. Most people speak of never having enough of it...

Kairos ($\kappa\alpha\iota\rho o\sigma$) – refers to the right time, opportune time or seasonable time. It cannot be measured. It is the perfect time, the qualitative time, the perfect moment, the "now". Kairos brings transcending value to kronos time. Kairos is the right moment of opportunity which requires proactivity to achieve success. It is significant and decisive. To miscalculate kronos is inconvenient. To miscalculate kairos is lamentable.

Even though kairos is a bit illusive, it is at the same time, alluring. The Greeks knew kairos intersected kronos time. In *Panathenaicus*, Isocrates writes that educated people are those *"who manage well the circumstances which they encounter day by day, and who possess a judgment which is accurate in meeting occasions as they arise and rarely misses the expedient course of action."*

During my research, an experimental spatial resource manager has been implemented, based on the ideas described in this thesis; it is called *Kairos.*

Inspired by M.R. Freier,
Whatif Enterprises – 2007