A Platform-based Design Flow for Kahn Process Networks

Abhijit Davare
Qi Zhu
Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-30
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-30.html

March 28, 2006

Acknowledgement

Thanks to Professor Edward Lee for his guidance. His extensive knowledge of the formalisms underlying meta-modeling techniques has greatly impacted this work. Many thanks to the members of the Metropolis team. In particular, thanks to Yosinori Watanabe, Alex Kondratyev, Alvise Bonivento, Trevor Meyerowitz, and Douglas Densmore. Also thanks to the anonymous reviewers from the Fall 2004 offering of EE 290N for their comments, especially for the first step of the design flow.

A Platform-based Design Flow for Kahn Process Networks

Abhijit Davare, Qi Zhu, Alberto Sangiovanni-Vincentelli

Abstract

Effectively implementing multimedia applications on multiprocessor architectures is a key challenge in system-level design. This work explores automated solutions to this problem by considering two separate directions of research. First, the problem is placed within the context of a generalized mapping strategy and the concept of a common semantic domain is developed which is capable of reasoning about the automation techniques that are to be applied. Second, a specialized design flow and associated algorithms are developed to solve this problem.

The idea of a common semantic domain is described and its usefulness in other mapping problems is demonstrated. For this particular problem, a common semantic domain is identified and forms the basis of the algorithms which are developed in the design flow.

The design flow is divided into four clearly defined steps, to ensure the tractability of optimization problems while obtaining a good overall solution. The separation of the flow into these steps allows prior work from a variety of sources to be used. Efficient heuristics are developed for each step of the design flow. The effectiveness of the heuristics used in this design flow is demonstrated by applying them to an industrial case study.

1 Introduction

To enable effective design reuse at the system level, separating the functional and architectural aspects of a design is essential. To realize an implementation, the functional specification is captured and then mapped onto a particular architectural platform. The mapping step must ensure that the association between the two designs is legal and can be realized. Also, mapping must try to maximize system performance according to the desired metrics. This process is depicted in Figure 1, where a functional algorithm and an architectural platform are captured as models and then mapped together to produce a system implementation. If necessary, the functional or architectural models may need to modified if a suitable solution cannot be found – this is represented by the dotted arrows. The overall objective is obtain a suitable solution as quickly as possible without many iterations.

The task of mapping is to choose a suitable association between the functional and architectural designs from the set of all possible associations. Traditionally, this task has been carried out manually through designer intuition. While manual mapping may produce good results, there are

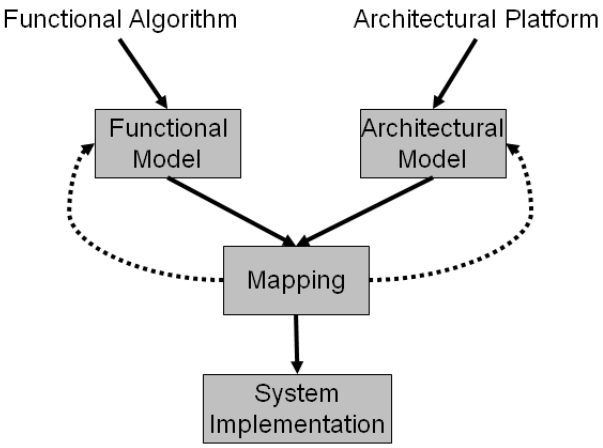

Figure 1: Overview of Mapping in System-Level Design

many challenges to face. Today, increasing design complexity and heterogeneity mean that the size of this design space is increasing. In order to shorten design time, automation is necessary. The broad focus of the research presented here is exactly this task of automated mapping. In this work, we will focus on a specific problem within the realm of automated mapping – that of efficiently implementing multimedia applications on heterogeneous multiprocessor architectures. This problem is pertinent for a whole host of systems, ranging from cellular phones, digital cameras, and portable audio players to high-end printers and broadcast video units.

The chief characteristic of multimedia applications is that they involve sets of transformations to streams of data. Data streams in these applications can represent any type of information, such as audio samples, image blocks or video frames. Typically, the streams have one source and one sink, and must be non-lossy. Usually, reordering of data items in streams is not acceptable. The transformations that are carried out on data streams can be quite complex and their granularity is design-dependent. These transformations may consume data from any number of streams and produce data to any number of streams.

Heterogeneous multiprocessor architectures are becoming common as power and nanometer-era challenges force designers to use several specialized cores instead of relying on a monolithic processor. The potential benefits of these architectures include increased throughput and lower power consumption. However, the relative novelty of these types of architectures along with their highly concurrent nature introduces many challenges. The paradigm of software development today is still based on sequential code, meaning that it is often difficult to implement algorithms on these types of architectures. This difficulty often precludes design space exploration, since the cost of evaluation is too high.

The platform-based design methodology [27] attempts to remedy some of these problems, by advocating a meet-in-the-middle approach to design, instead of a purely bottom-up or top-down view. In this methodology, an abstract model of the actual architectural platform is typically constructed which provides a set of services. The functional application is refined such that it makes use of the services. Different associations between the functional and architectural model reflect different points in the design space. Once an attractive point has been found, then the system

can be implemented using either automatic or manual approaches.

Within this methodology, finding the convenient representations for the architectural and functional models and finding a suitable point in the design space remain difficult problems. This work attempts to address both of these issues.

1.1 Contributions

The philosophy of the approach to automated mapping developed in this work is based on the concept of a Common Semantic Domain (CSD). Intuitively, a common semantic domain enables reasoning about the aspects of the behavior of the functional and architectural models. The hypothesis is that the concept of CSD is important in the development of the efficient mapping and allocation algorithms for system-level design problems. A contribution of this work is the initial description of the CSD concept along with a concrete definition of a CSD for the multimedia applications and heterogeneous multiprocessor architectures that we consider. In this CSD, both the functional and architectural models are represented with the Kahn Process Networks model of computation.

The major contribution of this research is the development of a design flow for Kahn Process Network specifications within the Platform-based methodology. This design flow is decomposed into four well-defined steps – reconfiguration, allocation, initial sizing, and runtime deadlock detection & resolution. This decomposition enables us to focus on specific optimization problems and leverage existing work for each step. A beneficial consequence of the separation of the design flow into these steps is that the previous work that we consider comes from a number of different research communities. For each step in the design flow, we develop novel algorithms and heuristics.

The design flow is aligned with the principles of the Platform-based design methodology [27] and therefore promotes orthogonalization, analysis, and refinement. To validate this design flow, we have implemented and applied the algorithms to an industrial design example within the Metropolis [4] framework.

1.2 Organization

The remainder of this paper is organized as follows: In Section 2, background information is given on models of computation that are suitable for multimedia applications – including Kahn Process Networks(KPN). Also, some of the salient characteristics of heterogeneous multiprocessor architectural platforms and examples of these are presented. In Section 4, we describe in further detail some of the system-level design environments that try to tackle the problem of implementing KPN specifications on heterogeneous multiprocessor platforms. In Section 3, an introduction is provided to the Metropolis design framework. This framework supports the Platform-based design methodology and has been used to validate the design flow described in this work. Section 5 explains the concept of a Common Semantic Domain and how this applies to the problem of automated mapping in general and the design flow developed in this work. The four-step design

flow itself is described in Section 6 with approaches and algorithms described at each step. The design flow is applied to an industrial case study in Section 7. Finally, conclusions and avenues for future work are presented in Sections 8 and 9.

2 Background

In this section, some background material for the task of mapping multimedia applications on heterogeneous multiprocessor architectures will be considered. In Section 2.1, models of computation that are suitable for expressing the behavior of multimedia systems will be covered. The motivation for focusing on the KPN model of computation will be provided as well. In Section 2.2, further information will be provided on why heterogeneous multiprocessor architectures are relevant for the topic at hand.

2.1 Models of Computation for Data-streaming systems

In this section, some of the models of computation that have been suggested for data-streaming systems will be described in further detail. The expressiveness and suitability for analysis of these models will be evaluated.

Kahn Process Networks [25] is a model of computation where concurrent processes communicate with each other through point-to-point one-way FIFOs. Read actions from these FIFOs block until at least one data item (or token) becomes available. The FIFOs have unbounded size, so write actions are non-blocking. Reads from the FIFOs are destructive, which means that a token can only be read once.

More formally, each channel in a KPN is a signal which carries a finite (possibly empty) or infinite sequence of tokens. The set of all possible signals is denoted S while the n-tuple of signals is denoted as S^n. The relation $\sqsubseteq$ is defined as the binary prefix relation on signals. For instance, $s_1 \sqsubseteq s_2$ means that the sequence of tokens contained in the signal s_1 is a prefix of the sequence of tokens contained in s_2. This definition generalizes to an element-wise prefix order, which can be defined on S^n. This element-wise prefix order is a CPO.

Any process P in the KPN with m inputs and n outputs is a mapping from its input signals to its output signals, $P : S^m \rightarrow S^n$. The semantics of KPN places a restriction on the type of mapping that a process can represent. A process must be monotonic in its mapping from input signals to output signals under the element-wise prefix order, $s_1 \sqsubseteq s_2 \Rightarrow P(s_1) \sqsubseteq P(s_2)$. This means that supplying additional inputs to a process results in additional outputs being produced, tokens which have already been produced cannot be retracted.

Under this restriction, and given the fact that the processes are monotonic functions on a CPO, the least fixpoint theorem tells us that a least fixpoint exists for a network of these processes. According to the denotational semantics of KPNs, this least fixpoint represents the behavior of the KPN [25]. This is the behavior that we want to find with simulation. However, the procedure for finding this least fixpoint is not given under monotonicity conditions alone. To find this procedure,

we must apply a stronger condition on processes.

A stronger condition that is required is that of continuity, which requires that the result of this function to an infinite input is the limit of its results to the finite approximations of this input. Under this stronger condition, a procedure for finding the least fixpoint behavior exists. This procedure involves an initial condition of empty channels. Then, the processes are allowed to act on the empty channels until no further change takes place. This is the procedure that we need for finding the behavior of the KPN since it corresponds directly to simulation. To guarantee continuity, the sufficient (but not necessary) condition imposed on Kahn processes involves blocking read semantics [26]. Since continuous functions are compositional, is suffices to ensure that each process in a KPN is continuous to guarantee that the entire network has a deterministic behavior.

This is the appealing characteristic of the KPN model of computation – that execution is deterministic and independent of process interleaving. Also, this model of computation allows natural description of applications since it places relatively few requirements on the designer other than blocking reads.

Implementing KPN specifications on resource-constrained architectures has a key challenge: that of realizing a theoretically infinite-sized communication channel with a finite amount of architectural memory. Indeed, a KPN implemented in this manner no longer satisfies the original definition of non-blocking writes, since a lack of storage space in the communication channel may force further write actions to be blocked. This additional constraint of blocking writes may possibly introduce deadlock into the execution of the system. This undesirable occurrence is referred to as *artificial* deadlock [19]. It is undecidable in general to determine if a KPN can execute in bounded memory, therefore deadlocks cannot be avoided statically.

The resolution of artificial deadlock requires dynamically supplying extra storage to some communication channel which is involved in the deadlock. This is the basis of Parks' algorithm [42]. However, choosing the channel and the amount of memory to allocate such that the deadlock is resolved with a minimum of extra memory is undecidable in general. A "bad" strategy will allocate memory to channels in such a way that the deadlock is not truly resolved, just postponed. In this case, the system will eventually run out of memory and the system will need to be reset.

Dataflow process networks are a special case of Kahn Process Networks where the execution of processes can be divided into a series of atomic firings [32]. This MoC in general suffers from the same undecidability as Kahn Process Networks [10]. In static dataflow [34], the number of tokens produced and consumed for each firing is statically fixed. Due to this restriction, aspects such as scheduling and buffer size can be computed statically and efficiently. The key limitation is that data-dependent behavior is difficult to express. This limitation makes this MoC unsuitable for many practical applications. Related work including Cyclo-static Dataflow [43], Heterochronous Dataflow [20], and Parameterized Dataflow [9] attempt to extend static dataflow but retain decidability by allowing structured data-dependent behavior.

2.2 Heterogeneous multiprocessor architectural platforms

Heterogeneous multiprocessor architectural platforms are gaining prevalence for embedded systems. The previously utilized technique of increasing performance by increasing clock frequencies for monolithic processors causes an unmanageable increase in power consumption. Since embedded devices are usually constrained by battery life and/or packaging cost, increased power consumption cannot be tolerated. The alternative is to increase throughput by adding more parallelism to the system in the form of multiple cores. Each processor can run at a relatively low clock frequency and perform a portion of the requested task. The overall power consumption of the system decreases since the critical path delays that determine the cycle time can be relatively higher for each processor.

With this new paradigm, programming an embedded system becomes more difficult. In order to take advantage of the increased throughput, the functional specification should consider concurrency as a basic construct. A functional specification approach which relies on single-threaded sequential code is not easy to deploy on these highly concurrent platforms.

Today, many companies are releasing heterogeneous multiprocessor architectural platforms for the embedded systems domain. Intel [24] has a number of platforms in the multimedia processors area such as the MXP5400 and the MXP5800 which are prime examples. The MXP5800 consists of 40 concurrent processing elements, which are of eight different types. The IXP2400 is one of Intel's network processors which also contains the same flavor of parallelism. Xilinx [53] is an FPGA maker which allows the embedding of both hard and soft cores within its devices. The interconnect structure and the capabilities of each core can be configured in many different ways. These are the types of architectural platforms we are interested in targeting in this work.

3 The Metropolis Design Framework

The Metropolis Design Framework is an embodiment of the Platform-based design methodology. The methodology is flexible enough to be applied to many different types of design problems, but the Metropolis framework is focused on electronic design at the system level. The Metropolis framework consists of a specification language – the Metamodel [52] – as well as a compiler and a set of plugins that can interface with external tools.

In this section, we will cover the goals of the Metropolis framework and how they manifest themselves in the infrastructure and the Metropolis Metamodel Specification language.

3.1 Goals of the Framework

The Metropolis framework is geared toward attacking common electronic design problems that occur at the system level. With that aim in mind, the major goals of the framework are three-fold: Facilitating design reuse to enable the design of large systems, preserving analysis capabilities by capturing the design specification with formal semantics, and enabling declarative statements in

the specification to formally capture capture constraints, assertions, and performance annotations. In the following, we will examine each of these goals in further detail. A good overview of carrying of the design methodology in Metropolis is given in [46].

3.1.1 Design reuse and orthogonalization

As designs become more complex and more heterogeneous, effective design reuse becomes crucial. Creating each design anew in order to ensure very high performance is becoming an increasingly expensive proposition. The size of design teams and the time-to-market is continuing to decrease, so effectively managing previously created IP and integrating it into new designs is the only solution. This becomes especially important when the IP is heterogeneous and developed by different design groups.

In the consumer electronics domain, the trend is toward increasingly customized products which typically have small sales volumes. This fragmentation of the market implies that the incremental cost of developing a new product must remain small. However, due to the economics of fabrication, creating small batches of new hardware cannot be the solution. Most of the product differentiation has to come from software or the configuration of reconfigurable hardware.

These factors motivate the orthogonalization between functionality and architecture which underlies the Platform-based design methodology. The functional portion of the design exercises *services*, which can be provided by different architectural models – or platforms – with different costs. A particular mapping of a functional model with an architectural model corresponds to a system model. By allowing an architectural model to be reconfigurable or instantiated in different ways, we can easily represent a family of parameterizable architectural models. Then, the mapping must also choose an appropriate platform instance from the choices available.

Since the only interaction between the architectural and functional models takes place due to the mapping of services together, once these are agreed upon, separate groups of developers can code, debug, and maintain the functional and architectural models.

The second type of orthogonalization emphasized in Metropolis is the orthogonalization between computation and communication. Computational activities are usually highly design-specific while communication schemes are usually standardized. With multiprocessor and distributed architectures becoming more common in the embedded systems world, the impact of communication on overall system performance is also quite large.

Finally, behavior vs. cost is the last type of orthogonalization emphasized by the Metropolis framework. Behavior reflects the services offered by the component, while cost represents the expense of providing these services. Cost can be defined in terms of time, power, chip area, or any other quantity of interest. This orthogonalization allows the framework to easily support the usage of “virtual” components and facilitates back-annotation to accurately model cost-metrics. Virtual components are architectural resources that do not reflect existing physical designs (hardware/software). A designer can configure and utilize virtual components in a system, and dictate the final parameters as constraints for implementation once she is assured that the component can be successfully used. Even if an architectural component is available and its behavior known, its

performance can be obtained at various levels of accuracy. A separation between behavior and cost allows this component to be used even if accurate numbers are not available. For instance, a synchronous bus component can be used without knowing the exact number of cycles taken for a transfer. An estimate can be used and system evaluation can proceed. Once cycle-accurate numbers become available, they can be substituted without requiring additional changes to the system.

3.1.2 Preserving Analysis Capabilities

A major complication with large, heterogeneous systems lies in the task of verification. Ensuring that the system performs according to the specification can easily take a majority of the total design time. In an attempt to remedy these problems, the Metropolis design framework stresses the usage of formally defined models of computation for modeling. The specification language used in Metropolis – the Metropolis Metamodel [52] [7] – has formally defined semantics that allow the expression of many different models of computation. Each statement in this language has a formal representation in the form of action automata. One important type of verification that is often encountered within the design process is that of refinement verification [15].

3.1.3 Event interactions and annotations

One of the unique aspects of the Metropolis framework is the support for both operational code and declarative statements in the specification. Most other system-level design environments only allow operational code as the specification mechanism. Supporting declarative statements allows the designer to succinctly specify behavior or assertions in the design. This is especially important in the initial phases of the design process, when the designer may be more interested in specifying *what* properties the components of the design need to have, rather than *how* those properties will be manifested in an implementation.

Currently in the Metropolis framework, support is provided for declarative statements in two different logics, Linear Temporal Logic (LTL) [49] and the Logic of Constraints (LOC) [5]. Both of these logics allow for statements to be made about event instances in the design. Event instances are generated whenever a thread of control in the design executes any action. Event instances may be annotated with quantities, which may represent a diverse set of indices, from access to a shared resource to energy or time.

LTL statements can be used to specify mutual exclusion constraints and synchronization between events. LOC may be used to make statements about quantity annotations on events. The Metropolis framework is unique in the respect that it allows design specification in the form of operational code as well as declarative statements over relevant events from the operational code.

3.2 Design activities within the Metropolis framework

After having described the goals of the Metropolis framework, we turn to the design activities that are important for the task of implementing functional applications on architectural platforms.

Specifically, we look at three major issues. First, in Section 3.2.1, we look at modeling the behavior – including the computation and communication – in the functional and architectural models. In Section 3.2.2, we describe the method of annotating costs to operations in the architectural model. Finally, in Section 3.2.3, we explore the mechanism by which the functional and architectural models are associated together. An overview of these design activities within the context of a simple case study is provided in [12].

3.2.1 Describing the Functional and Architectural Models with Processes and Media

Processes are objects that possess their own threads of control. Media are passive objects that provide services to processes and other media. A process cannot connect to other processes directly. Instead, an intermediate media must be used to manage the interaction between multiple processes. When an object wishes to utilize the services provided in another media, it must communicate with that media by using ports which have associated interfaces. The concept of ports and interfaces is widely used for design specification in frameworks like SystemC [23]. Media implement certain interfaces which then become services provided to processes or other media. For instance, a media may implement read and write services which can be used by processes.

Processes in the architectural model may represent tasks which are executing on architectural media such as processors. By themselves, these processes do not carry out any useful work, they just execute a nondeterministic sequence of operations. In this sense, the set of possible behaviors in the architectural model encompasses all legal traces of operations.

3.2.2 Using Quantity Managers to annotate cost

Quantity managers in Metropolis are similar to aspects in aspect-oriented programming [28] languages. Quantity managers can be used to assign costs to operations in the architectural platform. The cost can be in terms of any useful quantity, such as time, power, or access to a shared resources. The quantity manager collects all requests for annotation and determines which requests are to be satisfied and which ones need to be blocked.

An example of this type of annotation is shown in Figure 2. In this example, the *L_Exec* operation in a media is annotated with cost of *EXEC_CYCLE* cycles. First, the relevant events *beginEvent* and *endEvent* are identified which correspond to the execution of the beginning and end of the operation respectively by some thread. The first event is annotated with the current time according to the quantity manager whereas the second event is annotated with the time of the first event plus the cycle time.

3.2.3 Mapping with synchronization constraints

Synchronization statements are used to intersect the behaviors of the functional and architectural models by constraining events of interest from each to occur simultaneously. Along with simultaneity, we can also control the values of specific variables that are in the scope of these events.

```
L_Exec {@
    event beginEvent = beg(getThread(), L_Exec);
    event endEvent = end(getThread(), L_Exec);
    {$
      beg { gt.RelativeReq(beginEvent, 0); }
      end {
       currentTime = gt.getQuantity(beginEvent, LAST);
       gt.AbsReq(endEvent, currentTime + EXEC_CYCLE);
      }
    $}
    ... // code for this operation
@}
```

Figure 2: Annotating costs for operations with quantity managers

This type of synchronization can be used to restrict the behavior of architectural processes to follow that of the functional processes to which they are mapped.

An example of a function that would emit these synchronization constraints is shown in Figure 3. In this example, the function takes as arguments the two processes that are to be mapped together – a functional process and an architectural process. First, the beginning of the read operation is identified for both processes and recorded as the events *e1* and *e2*. The two events are synchronized together and two variables in the scope of these events are constrained to be equal. In this example, the number of items read by the functional process is constrained to be the same as the number of items read by the architectural task. By changing the arguments to this function, various mappings can be realized and evaluated relatively easily.

```
void mapPair(process f, process a) {

    event e1 = beg(f, f.read);
    event e2 = beg(a, a.read);
    ltl synch(e1, e2: numItems@e1 == numItems@e2);

    event e3 = end(f, f.read);
    event e4 = end(a, a.read);
    ltl synch(e3, e4);

    ... // similar code for write() and exec() services
}
```

Figure 3: Synchronizing events to realize mapping

4 Prior Work

In this section, an overview is provided of existing system-level design frameworks. The focus is placed on the support provided for automated design space exploration [22] activities. Prior work that is directly related to specific steps in the design flow will be described directly in Section 6.

First, we will consider some frameworks that are primarily focused on data streaming applications and either implicitly or explicitly utilize the Kahn Process Networks model of computation. The Spade [35] and Sesame [18] approaches within the Artemis [45] project focus on synthesizing KPN specifications in hardware/software. These approaches are limited to the process networks model of computation. However, deadlock issues are not considered. The most relevant optimization approach from their work utilizes an evolutionary algorithm to minimize a multi-objective non-convex cost function. This cost function takes into account power and latency metrics from the architectural model. The optimization problem is solved using a randomized approach based on evolutionary algorithms.

The Compaan/Laura [51] approach uses Matlab specifications to synthesize KPN models, which are then implemented on a specific architectural platform as hardware and software. The architecture platform consists of a general purpose processor along with an FPGA, which communicate via a set of memory banks. Software runs on the general purpose processor, while the hardware is synthesized into VHDL blocks which are realized in the FPGA. The partition between hardware and software occurs relatively early in the design flow and is based on workload analysis. The types of optimizations that are carried out automatically relate to loop analysis. The software implementation makes use of the YAPI [30] library, and does not consider deadlock.

CAKE [14] (Computer Architecture for a Killer Experience) is a project affiliated with Philips research that attempts to realize multimedia applications by using the YAPI libraries. Their focus is mainly on homogeneous "tiled" multiprocessor architectures. The automated design space exploration approach they describe is divided into two steps, where the first step partitions the processes and the second step schedules them on a single processor.

For all of the KPN-based frameworks described above, none consider the problem of deadlock detection and resolution even though all eventually implement functional models on architectural platforms with finite memory resources. One of the aims in this work is to formulate deadlock detection and resolution strategies in the context of a system-level design environment, and to determine the impact of this additional overhead. Specifically, we can look at the impact on system latency for industrial multimedia applications. Next, we can look at some design frameworks that are more generic, and not focused on streaming applications.

ForSyDe [47] focuses on formal design transformations that enable design refinement. This allows the designer to start with an abstract definition of the design and proceed toward implementation. At each step, there are two types of transformations can be made. Semantic preserving do not change the behavior of the model, while design decisions are unrestricted. The focus of ForSyDe is on the verification aspects of design, and they do not focus on automation.

MESH [44] is a design framework that separates the design into three parts: the application layer, the physical layer and the scheduling layer. These three layers are roughly equivalent to the

functional model, the architectural model, and mapping within the Platform-based design methodology.

Mescal [1] is an environment for developing software for customized processors. The main domain of concentration is network processors, which can be considered a specialized type of multimedia applications. Past work has been carried out on customizing instruction sets of processors according to the application. Recently, the investigation of FPGAs as an implementation fabric and automated allocation techniques have also been explored.

Polis [6] is a design environment which was one of the first to allow for function-architecture separation. Designs in this framework are based on the communicating finite-state machines model of computation. Architectural components can only be chosen from a set of predefined components, limiting the expressiveness.

Finally, Ptolemy [37] is a meta-modeling framework which focuses on simulation and the interaction between different models of computation. While the focus is not on function-architecture separation and mapping, a domain does exist for the distributed simulation of process networks. Ptolemy does consider the problem of deadlock detection and resolution in great detail in its implementation of the process networks domain.

Compared to these other design environments, the Metropolis framework is unique in the respect that it provides meta-modeling capabilities and is geared toward function-architecture mapping. Meta-modeling capabilities allow reasoning about designs which are expressed using different models of computation. The function-architecture mapping is a natural method by which these diverse models come together. This motivates a method of reasoning about mapping between different models of computation. This topic is developed in Section 5.

5 Mapping with Common Semantic Domains

The problem of mapping between a functional model and an architectural model can be solved with two broad strategies. The first strategy attempts to reason about the aspects of each in a common semantic domain. As long as a reasonable semantic domain can be found, a good mapping can usually be found by solving a covering problem. The second strategy instead attempts to bridge the gap between dissimilar semantic domains. Due to the lack of a common domain to carry out reasoning, the algorithms and techniques applied by this second strategy are usually specific to the semantic domains being bridged and the abstraction levels of the models within these domains. This strategy has the capability to produce very good results for specialized problems, but there is little applicability to a broad class of problems.

We adopt the former approach in this work. Note that this concept of finding a mechanism to reason about the architectural and functional aspects of multimedia systems is related to the work in [29]. Our contribution is the generalization of this concept in the framework of common semantic domains.

First, in Section 5.1, we will describe the idea behind semantic domains and look at some prior work that adopts this philosophy. In Section 5.2, we will consider the task of finding a common

semantic domain given two models, and how we can reason about the mapping between them. The technology mapping problem for combinational circuits is easily cast into this approach, and we present it in Section 5.3 as an example. Finally, in Section 5.4, we determine a common semantic domain for the problem at hand – associating KPN applications with heterogeneous multiprocessor architectures.

5.1 Semantic Domains

Intuitively, in order to reason about the functional and architectural models, we must be able to talk about both in a common language. The common semantic domain represents this language.

More formally, a semantic domain consists of a set of concepts. These concepts are a result of adding some set of constraints to the usage or composition of the primitive semantic concepts. A model can be described in this semantic domain if we can reason about the behavior of the model by only considering the restricted concepts in this semantic domain. A model can be described in multiple semantic domains. For each semantic domain, there may be some type of analysis that is well-suited for that domain. Depending on the type of analysis that we need to carry out on a model, describing it in a particular semantic domain may be advantageous.

Let's define the super-domain as the least restrictive semantic domain which contains just the set of primitive semantic concepts, with no additional constraints on their usage or composition. According to this definition, we can reason about all models by simply describing them in the super-domain. However, this type of description may not be desirable. For instance, the primitive concepts contained in the super-domain may be defined with such fine granularity that decomposing the model into these concepts may be untenable. Also, carrying out useful analysis on such primitive concepts may not be possible.

We can now construct a lattice of semantic domains referred to as the domain lattice. In this lattice, the super-domain occupies the uppermost position. All other semantic domains lie below the super-domain and consist of different types of constraints on the primitive concepts that form the super-domain. In effect, the super-domain allows the expression of the most general set of behaviors. Other semantic domains restrict the set of possible behaviors that can be expressed by adding constraints.

A semantic domain S is said to be the ancestor of another semantic domain T if the set of behaviors that can be expressed by T is a subset of the behaviors expressed by S. Graphically, S then lies above T in the lattice. According to this definition, the super-domain is the common ancestor of all other semantic domains. A graphical view of this lattice is shown in Figure 4. In this Figure, domains A, B, and C are all incomparable semantic domains. Domain D inherits the constraints on the usage and composition of the primitive semantic concepts from Domains A and B.

An example of a useful super-domain is the Tagged Signal Model [33]. The primitive concepts in this super-domain include values, tags, events, signals, processes, and connections. The Tagged Signal Model is generic enough to encompass a wide range of models of computation, including continuous time, discrete event, process networks, hybrid systems and many others. Each of these

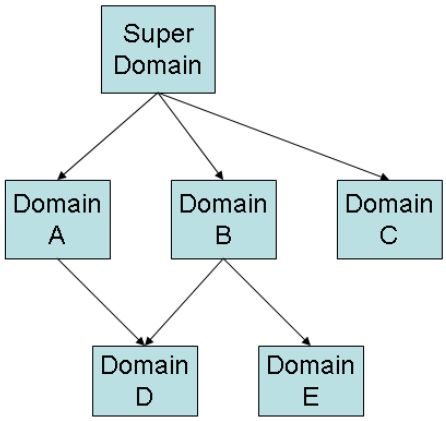

Figure 4: Lattice of Semantic Domains

other models of computation is a semantic domain in our terminology, with a particular restricted usage of the primitive elements permitted.

Note that semantic domains are not restricted to only involve the well-known models of computation in the literature. Models of computation are traditionally identified based on how naturally they can be used to model designs. Our goals with semantic domains are somewhat broader. We are concerned not with modeling, but only with a particular type of analysis – that which is necessary for design space exploration. Therefore, the semantic domains we identify may be "unnatural" from a modeling standpoint, but be perfectly suitable for carrying out the limited type of analysis that we are considering.

5.2 Finding a Common Semantic Domain

To carry out joint analysis on multiple models, we require that these models be described in a common semantic domain. Finding the common semantic domain and determining the type of analysis that needs to be carried out are the two major problems that need to be solved. In this Section, we concentrate on explaining the former problem.

Given an architectural model described in a semantic domain X and a functional model described in a semantic domain Y, we can proceed to find the common semantic domain between the two. First, let A_1 be the set of ancestors of X and A_2 be the set of ancestors of Y. A common semantic domain can now be chosen from the set of common ancestors (i.e. $A_1 \cap A_2$) of X and Y.

There is usually a tradeoff to consider when choosing a particular common semantic domain from the set of common ancestor domains. At one extreme, the super-domain can always serve as the common semantic domain. Since it is the most generic common ancestor, the behavior of the functional and architectural models can always be expressed with the primitive concepts contained in the super-domain. However, this description of the models may be too generic for our needs, and the types of analysis we can carry out may be quite weak. On the other hand, we can consider the other extreme. This involves choosing the common semantic domain to be the least upper bound between X and Y. If this choice is made, then stronger analysis techniques may be available.

The type of analysis that we are focusing on here attempts to determine the optimal mapping between a functional model and an architectural model. The nature of this analysis will be different depending on the semantic domain at which it is applied.

5.3 Example: Combinational circuits

Combinational circuits are an example where the concept of a common semantic domain has greatly aided the mapping between a combinational circuit and a particular gate library. In this section, we will describe the well-known technology mapping problem in terms of our CSD formulation. The aim is to solidify the concepts introduced by associating them with a well-known example.

Let the super-domain for this problem consist of the Tagged Signal Model. Since this super-domain can be used to represent many practical models of computation, we can represent the continuous time domain as a refinement of this domain. This semantic domain corresponds to the constraints that signals contain events which have real-valued tags representing time and arbitrary values. Again, we can restrict the expressiveness of the continuous time domain by only considering time-varying Boolean values. In this new domain, events are further restricted to have only Boolean values.

Next, we can remove the concept of time completely, and obtain an untimed Boolean semantic domain with arbitrary processes. In this domain, all tags may be constrained to be same. The procedure continues with the identification of a restricted legal set of processes, those that correspond to $+$ and $*$ Boolean operations. After this marathon procedure of adding further constraints, we have obtained the semantic domain that is typically used to reason about combinational circuits.

At this semantic domain, we can take a slight detour and describe a particular type of analysis that can be carried out on a model: manipulation. At this domain, the combinational circuit model is described as a set of Boolean expressions. We can manipulate these Boolean expressions with certain cost functions in mind. For instance, if we take the number of literals as a rough indication of area, we can apply kerneling and factoring operations to manipulate the representation of the model in this domain. The most important requirement is that this manipulation not change the behavior of the model. Since we can express the behavior of the model in this domain, we can ensure that this is indeed the case.

At this point, we can continue our downward stroll in the lattice to arrive at NAND2 circuits, which represent the common semantic domain which is eventually adopted for carrying out mapping. In this domain, the models are represented as a netlist of two-input NAND gates. Briefly continuing beyond this, we obtain the two incomparable domains, of which the NAND2 domain is a common ancestor, that represent the domains of the functional model (combinational circuit specification) and the architectural model (the gate library). The gates that are used in the specification may not correspond to those defined in the gate library, but the behavior of both can be represented in the NAND2 domain.

Returning to the NAND2 domain, the mapping between the functional and architectural models can be formulated as a covering problem. The constraints for this covering problem ensure that

each NAND2 gate in the functional model is covered by an element of the architectural model (also represented as NAND2 gates in this domain). The objective function for the covering problem tries to reduce the overall cost for the mapping. The metrics for the cost of each element of the architecture are specified independently. For instance, each NAND2 representation of a gate may have an associated metric indicating area.

5.4 Common semantic domain for Kahn Process Networks

To choose a common semantic domain for KPN applications and architectural platforms, we have to examine the key aspects of the composite behavior that we wish to reason about. Both models contain a notion of concurrency. The functional model has Kahn processes, while each processor in the architecture supports a set of concurrent tasks.

We also need to reason about the interaction between the concurrent processes. In the functional model, this takes place in a specialized way, with blocking read and non-blocking write actions on FIFOs being the only mechanism of interaction. In the architectural model, these can be modeled as compositions of more primitive architectural behaviors. For instance, blocking read behaviors can be decomposed into accesses to shared memory and semaphore primitives. In effect, we would like to restrict the set of behaviors that can be expressed by our architectural model. If this is done, then semantic domains lower in the lattice will be able to capture the behavior that we would like to reason about. The motivation for this is that the types of analysis techniques that we would like to apply are only available in lower domains of the lattice.

The common semantic domain we choose for this design flow contains the following elements: execution actions taken by a single thread, blocking read actions to FIFOs, and non-blocking write actions also to FIFOs. The read and write actions to any given FIFO are guaranteed to obey mutual exclusion constraints to avoid race conditions. In effect, we have chosen the common semantic domain that is roughly equivalent to the KPN model of computation. Figure 5 shows a graphical view of this domain. In effect, the services used by the functional processes (F_1, F_2, F_3, F_4) are provided by the set of architectural processes (F_1, F_2, F_3, F_4, F_5). The cost functions, which are not shown in this Figure, are used to choose a particular "covering" of the functional processes by the architectural processes. This gives only a rough view of the mapping problem, it will be specialized further in Section 6.

6 Design Flow

After we have expressed the functional and architectural models in a common semantic domain, we can carry out the analysis necessary for determining a mapping between these two models. In this Section, an automated design flow will be described that is applicable to the common semantic domain of Kahn Process Networks. The design flow associates the functional and architectural models together in such a way that the cost function of interest (typically energy or latency) is optimized. This design flow is divided into four distinct steps.

Implementing this design flow as a single step by solving a single optimization problem would

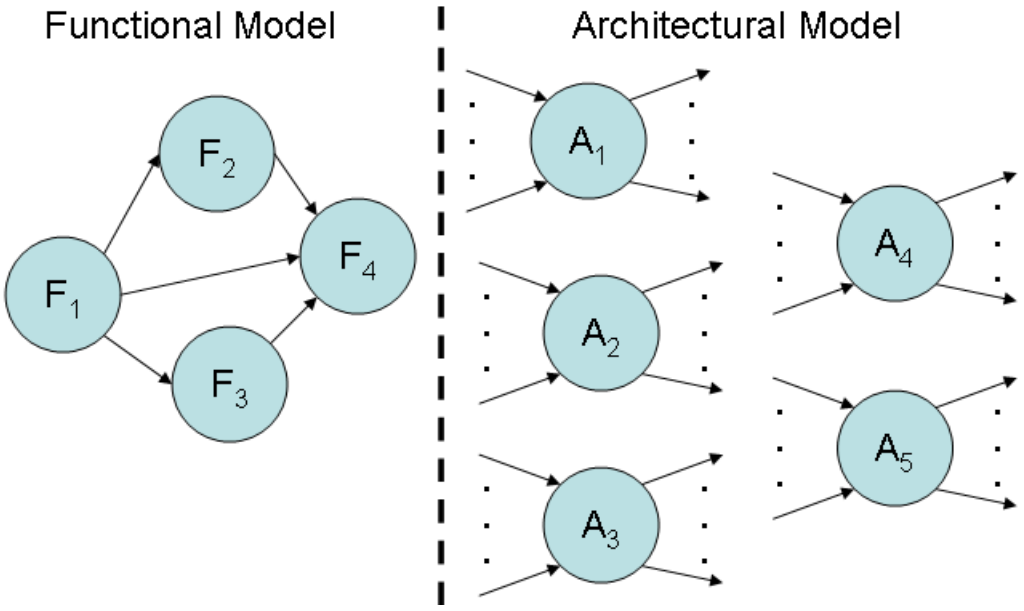

Figure 5: The common semantic domain decomposes the architectural and functional models into a common set of primitives

possibly produce the best quality implementation. However, this direct approach suffers from at least three major problems. First, due to the many different aspects of the mapping that this unified problem would need to take into account, it would be quite large and very complex. Second, an amalgamated formulation would leave little opportunity for designer intervention. Designers would be unable to easily customize the flow or incrementally restrict the design space, both of which may be desirable in practice. Finally, a single-step optimization problem would necessarily be customized for this particular design problem. Optimization approaches from related areas could not be used and relying on prior work would be difficult.

The alternative flow that we propose is divided into several clearly defined steps. Optimality of the final solution may be lost because of the decoupling between these steps, but it does result in a set of tractable problems that are amenable to designer intervention. The designer can iterate over a specific step or set of steps in the flow until she is satisfied with the results. Perhaps most importantly, the problems at each step in the design flow can leverage prior work and utilize well-studied approaches. A summary of the design flow is presented in Figure 6.

In this Section, we will describe the four major steps of the design flow in more detail. We will develop algorithms and heuristics that can be utilized to automate each step. Section 6.1 describes the initial step of manipulating the architectural and functional models in an effort to make them more compatible according to certain metrics. Section 6.2 explains the most important step of the design flow – the allocation of computational and communication resources from the architectural model to entities in the functional model. In Section 6.3, the heuristics used to initially allocate resources to the communications channels will be described. In Section 6.4, we develop system-specific algorithms to detect and resolve artificial deadlock situations at runtime. Finally, in Section 6.5, a brief overview is provided of the support for this design flow within the Metropolis framework.

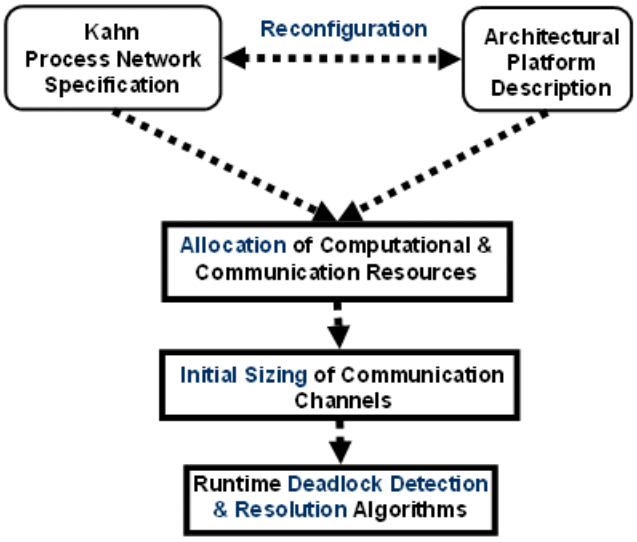

Figure 6: Overview of Design Flow

6.1 Step 1: Reconfiguration of functional and architectural models

In preparation for mapping the functional model with an architectural model, we may like to manipulate these models in order to optimize for certain design metrics. This step is akin to the kerneling and factoring operations described in Section 5.3. Approaches for manipulating the architectural and functional models will be described in Sections 6.1.1 and 6.1.2 respectively. It is important to note that this first stage of the design flow is meant to provide a starting point. Incremental improvements to this initial configuration can be achieved by iterating between the first and second steps in the design flow, although we do not explore these iteration activities further here.

6.1.1 Architectural Reconfiguration

If the architectural model represents a set of architectural platform instances, architectural reconfiguration attempts to choose a particular platform instance according to the desired metrics. If the architectural model does not have any parameters that need to be set, then architectural reconfiguration is not possible. The architectural reconfiguration step restricts the freedom encoded into the architectural model by fixing some of the parameters that need to be fixed. For instance, the architectural platform may allow a choice in the number of processors that are utilized. If latency is the metric of interest, then increasing the number of processors is likely beneficial. In the common semantic domain we are considering, this corresponds to an increased capability for concurrent execution. On the other hand, if power is the main metric, then increasing the number of processors might not be the best choice. Similarly, tradeoffs can be explored for the number of buses in the architectural model and other parameters.

Architectural reconfiguration can be carried out with a rough idea of the functional model that will later be mapped to it. This type of information is typically utilized in workload analysis, which attempts to choose an architecture which is well-suited for a class of functional models that will be

implemented on it.

6.1.2 Functional Reconfiguration

Functional reconfiguration manipulates the representation of the functional model such that it becomes better suited for association with architectural models according to certain metrics. The constraint on functional reconfiguration is that it should not affect the behavior of the functional model, only the representation. This implies that the types of functional reconfiguration that can be carried out are limited to the types of analysis that can be carried out in the semantic domain.

Here, we describe a simple type of functional reconfiguration that can be performed within this domain – Kahn process clustering. The constraint for this reconfiguration is to preserve the original behavior. The cost metric we are trying to manipulate is the amount of concurrency in the functional model. Too little concurrency may not be desirable since it may not completely make use of the architectural capabilities. Too much concurrency may not be desirable since it may require too much overhead to eventually manage in the architecture, which may increase the overall latency. Prior work along these lines has been described in [50], although the algorithm developed here is unique.

Clustering attempts to reduce the concurrency in the functional model such that a particular concurrency constraint from the architecture is met. Suppose the architectural model provides a maximum number of concurrent tasks that can be effectively handled. For instance, an architectural platform instance with 3 processing elements with a minimalist task scheduler on each element may only be able to support a maximum of 10 concurrent tasks. In this case, if the original functional model contains more than 10 processes, we cannot simply carry out a one-to-one mapping between functional processes and architectural tasks.

Clustering reduces the concurrency in the functional model by merging together functional processes. Assume that a set of original processes is clustered to produce a single merged process. The behavior of the merged process appears indistinguishable to the rest of the system from the behavior of the set of original processes. This means that the merged process has the same number of input and output channels as the set of original processes (modulo the channels that are subsumed in the merging).

To determine which processes must be merged, we need to get an idea of the communication requirements between processes in the original functional model. Since we assume that, in general, the internals of a process cannot be analyzed, we must rely on simulation-based profiling to determine the processes to be merged. The information we need to obtain from these processes relates to the relative amount of communication between them. Since functional models in KPN are deterministic and not affected by scheduling, cost or any other effect of the architectural platform, this information can be obtained directly from the functional model.

As a heuristic, we can merge together any two processes that have a high degree of communication between them and minimal communication with other processes. This pairwise merging can be carried out iteratively until the constraint from the architectural model is satisfied. The implementation of a merged process stitches together the implementations of the two original processes

along with wrapper code to handle the newly internalized communication. Of course, this heuristic is order-dependent and will not give the optimal results, but it has polynomial-time complexity.

In general, this type of functional process merging is very flexible. In fact, techniques like quasi-static scheduling [38] can be applied to stitch together code from multiple processes. This is done in an effort to reduce the reliance on dynamic schedulers from the architectural platform.

However, it is important to note that this particular type of clustering makes certain assumptions. For instance, we assume that the internalized communication that occurs as a result of clustering can be supported by the data memory resources available to a single process. In general, this may not be true for the communication between arbitrary processes in a KPN.

6.2 Step 2: Allocation of Computational and Communication Resources

The allocation step is perhaps the most important in the design flow since it is the first in which the functional and architectural models are explicitly associated with each other. The input to this step is an architectural model and a functional model, both represented in the common semantic domain of KPN. The output is a system where the computational and communication services in the functional model are associated with counterparts from the architectural model. Allocation is carried out by solving a covering problem which ensure that all of the services being used in the functional model are covered by the services from the architectural platform.

For the specific KPN design flow being described, the functional model determines how the services are used, while the architectural model determines the cost of providing these services. The semantics of the KPN model of computation allows us to make this type of distinction since the cost of carrying out the basic communication and computation services does not influence the behavior of the system.

As a reminder, in this flow, the common semantic domain consists of user-specified computational services along with read and write services. For the allocation problem, the architectural platform provides these services, while the functional specification utilizes the same services. The quantities of interest in the architecture such as time or power do not constrain the covering problem, but serve as parameters to the cost function. To enable a good quality allocation according to requisite metrics, we need to characterize the cost of the services provided by the architectural platform.

We can view the architectural platform as a set of architectural resources connected together in a specific configuration. Each architectural resource provides a certain set of services and itself may utilize other services. For instance, consider a CPU that provides read, write, and execution services. To provide the read and write services, the CPU utilizes the transfer services of a bus component.

At the architectural component level, if we are able to specify the cost of providing services and the dependencies between the utilized services and the provided services in terms of the required metric (time, power, etc), then we can compose these costs together to obtain the cost of services at the top-level architectural model. In Section 6.2.2, we will briefly focus on this decomposition. First, however, we will evaluate some previous work that has been carried out in this area.

6.2.1 Prior Work

Static allocation of resources is one of the most common types of tasks that any mapping problem needs to consider. Often, this static allocation needs to be carried out from estimates of the dynamic behavior of the system. Many of the KPN-specific system-level design frameworks described in Section 4. The approach described here differs in the sense that it deals with a generic notion of services and cost and can therefore be customized to the specific allocation problem being considered.

Statically capturing metrics of interest from simulation or other types of analysis is crucial. A overview of metrics that are considering important from a system-level design perspective is provided in [48]. The approach developed here only focuses on one metric – intensity – and costs given in relation to this intensity but can easily be extended to consider additional metrics.

6.2.2 Determining Service Cost

In general, we can view the architectural platform as an interconnection of components. Each component provides a certain set of services and may utilize some set of services. During the execution of the system, architectural services are utilized - driven by the requests of the functional processes. One of the aims of the allocation step is to estimate the cost of utilizing these services statically. In this section, we will describe a simple procedure that accomplishes this aim.

Let each component in the architecture provide a set of n services. Assume each of the n services provided by the component is annotated with an *intensity* variable. Define the intensity as a static estimate of the frequency by which the services are utilized in the system at runtime. Practically, the intensity may indicate the number of read actions carried out in a single time unit. Let these intensity values be represented by the variables $i_1, i_2, ..., i_n$.

The component incurs a certain cost for providing each service. Assume that this cost is specified by the component in terms of the intensities of the provided services. Let these costs be given by the functions $c_1(i_1, ..., i_n), ..., c_n(i_1, ..., i_n)$. Also, the component utilizes certain services in order to carry out the services that it provides. Let there be m such services which are utilized by the component. The component must specify the intensity of service usage as well. This will be done in terms of the intensity of services that are provided. So, for the m services utilized by the component, we obtain a set of intensities $f_1(i_1, ..., i_n), ..., f_m(i_1, ..., i_n)$. An architectural component that provides two services, uses one service, and is annotated with generic c and f functions in this manner is shown in Figure 7.

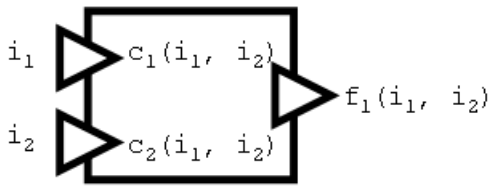

Figure 7: Service cost and output intensities annotated for a simple architectural component

If each component in the architectural platform provides the costs and output intensities as a function of input intensities, we can use the connectivity of the components to compose the overall cost functions for the services provided by the architectural platform. This composition depends on the type of metric being calculated.

Let's consider the example shown in Figure 8. In this example, all four architectural components in the platform have cost functions and output intensities specified in terms of the input intensities. The cost of the service utilized with intensity i_4 is simply i_4, since this service is not propagated further. The service with intensity i_3 is propagated to the next component, so its cost will be the sum of two terms: $(i_3 + i_4) + (i_1 + i_2 + 0.5i_3)$. Likewise, the second service has a total cost of $(4i_2) + (2i_1 + 2i_2)$. The first service is more complex, since it is propagated to two separate architectural components. If the cost being calculated is in terms of additive quantity like energy, then the cost required will be the sum along all paths in the transitive fan out: $(2i_1 + i_2) + (1.2i_1) + (2i_1 + 2i_2)$. On the other hand, if the quantity is non-additive, then the expression assumes a different form. For instance, if the costs reflect the time taken to provide services, then we have to find the sum of costs along the longest sequential path. In this case, the cost for the first service will be $(2i_1 + i_2) + max((1.2i_1), (2i_1 + 2i_2))$. To develop the optimization problem given the cost functions, in the sequel, we will concentrate on time as the quantity of interest.

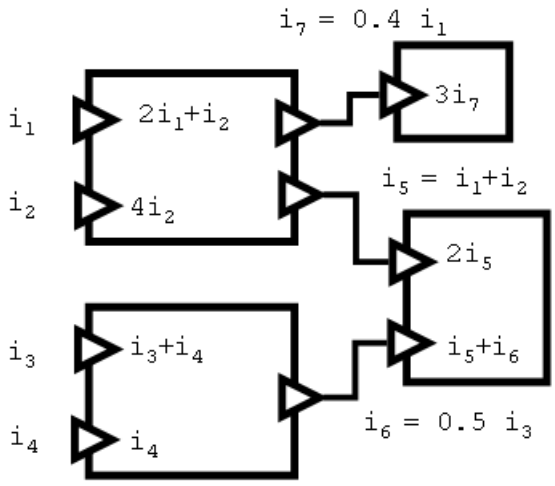

Figure 8: An architectural platform instance where all four components have been annotated with service costs

6.2.3 Formulating the optimization problem

Once the cost of providing each service has been determined for the architectural platform, we can formulate and solve an optimization problem to obtain a solution to the allocation problem. For this design flow, the optimization will determine the allocation of processes in the functional model to processes in the architectural model. To avoid complications, we restrict our analysis to one-to-one mappings between functional and architectural processes. The input for this optimization problem

is two sets of processes: the set U of users from the functional model and the set of P of providers from the architectural model. Due to the assumption of one-to-one mapping, it is required that $|U| \leq |P|$.

Each element in the set of users and the set of providers has an associated set of services. For instance, a user process from the functional model has an associated set of execution, read, and write services that it needs to utilize. Although the formulation of the optimization is greatly simplified if the set of used services is the same as the set of provided services, we consider a slightly more complex situation here. In this formulation, we allow one-to-many mappings between used services and provided services. These mappings are allowed to be dependent on the variables of the optimization problem (i.e. they cannot be merely parameters of the optimization). This type of relationship between these services is influenced by the allocation of users to producers. The introduction of this complexity is motivated by the usage of the communication services by the functional processes. In the functional model, there is no differentiation between a read to local memory and a read to global memory, since the functional model is independent of any such architectural artifacts. On the other hand, this type of differentiation is important to take into account in the architectural model, since the costs are very different. Therefore, the set of used services chosen remains execution, read, and write while is the set of provided services is expanded to contain execution, local read, local write, global read, and global write. The allocation procedure is tasked with the responsibility of relating the used read and write services with their local and global counterparts from the set of provided services. Note that the option of considering global communication even when local communication is possible might be necessary if local buffer memory is insufficient. However, information relating to memory and buffer space is deferred to the latter two steps in the design flow.

Returning to the formulation, we can define two additional sets: the set $USERV$ of used services and the set $PSERV$ or provided services. The one-to-many constraint on these sets manifests itself as a requirement that $|USERV| \leq |PSERV|$. For each member $SPLIT$ of the set $USERV$ that may be mapped into multiple provided services, there are two members in the set of $PSERV$: $LOCAL$ and $GLOBAL$.

In order to determine how the usage of a read service translates into a global or local read, we have to reason about the connectivity between the users and the providers. This necessitates the insertion of additional information into the formulation. For instance, all read and write actions from the user processes originate from user-to-user communication. On the architectural side, global reads are differentiated from local reads based on the two users to which this communication corresponds. We restrict the exploration to always choose the local communication option if this is available. If both users are mapped onto the same processor, then the read actions will be assigned onto local reads, whereas if the two users are on separate processors, then the reads will have to go through the bus and will become global reads. Therefore, a set C of components is introduced into the formulation, where a component corresponds to a processor.

After setting up the initial sets used in the optimization problem, we can examine the parameters that represent the functional and architectural models in the formulation. The parameters are summarized in Table 1. The *service_weight* parameter determines the weighting of each provided services in the overall cost function. The *bonding* and *user_intensity* intensity parameters are obtained from the functional model and indicate the amount of grouping between functional

Parameter Name	Domain	Range	Bounds
service_weight	$PSERV$	**R**	≥ 0
bonding	$U \times U$	**R**	$\geq 0, \leq 1$
user_intensity	$U \times USERV$	**Z**	≥ 0
provider_to_component	$P \times C$	**Z**	$\geq 0, \leq 1$
cost_coefficient	$P \times PSERV \times P \times PSERV$	**Z**	≥ 0

Table 1: Parameters for the Optimization Problem

processes and the intensity of service usage respectively. The *provider_to_component* parameter is obtained from the architectural model and represents the assignment of tasks to processors. Finally, the *cost_coefficient* parameters provide an encoding of the top-level cost functions (as described in Section 6.2.2) for each provided service as a linear combination of the costs of all provided services. In order to ensure the tractability of the optimization problem, we restrict the cost functions to be linear functions of the intensities. Since the architectural platform does not have fan-out from provided services to used services (there is only one bus), the complications related to calculating latency in the presence of fan out are avoided.

Variable Name	Domain	Range	Bounds
assign	$U \times P$	**Z**	$\geq 0, \leq 1$
provider_intensity	$P \times PSERV$	**R**	≥ 0
users_to_users	$U \times U$	**Z**	$\geq 0, \leq 1$
users_to_components	$U \times C$	**Z**	$\geq 0, \leq 1$
used_to_provided	$U \times USERV \times PSERV$	**R**	$\geq 0, \leq 1$

Table 2: Variables for the Optimization Problem

The variables for the optimization problem are summarized in Table 2. The main job of the optimization problem is to assign each user to a provider. This is reflected in the *assign* variable. It is important to note that the *assign* variable is the main variable in the optimization problem and the other variables are simply derived variables that make the constraints simpler. The *provider_intensity* variable determines the intensity of usage for each provided service of each provider. The *users_to_users* variable has a value of 1 iff the two users are allocated to the same component. In order to determine this information, the helper variable *users_to_components* reflects the assignment of users to components, equivalently, functional processes to architectural processors. Finally, *used_to_provided* determines the breakdown of the one-to-many mapping for the read and write services.

Now, the constraints can be formulated in terms of the given variables and parameters. First, we need to ensure that each user is assigned exactly one user and that each provider is assigned at most one user.

$$\forall p \in P, \sum_{u \in U} assign[u,p] <= 1$$
$$\forall u \in U, \sum_{p \in P} assign[u,p] = 1$$

Next, the *users_to_components* variable is assigned to help assign the *users_to_users* variable. The *users_to_users* variable is only bounded from below since the objective function will tend to decrease the value of this variable.

$$\forall u \in U, c \in C, users\_to\_components[u,c] = \sum_{p \in P} assign[u,p] * provider\_to\_component[p,c]$$
$$\forall u \in U, w \in U, c \in C, users\_to\_users[u,w] \leq 1 - users\_to\_components[u,c] + users\_to\_components[w,c]$$

The *used_to_provided* variable is set depending on the value of the *bonding* parameter and the *users_to_users* variable. The local and global read and write intensities are set in the first two constraints below. The third constraint ensures that all the common services are assigned correctly.

$$\forall u \in U, a \in SPLIT, b \in LOCAL, used\_to\_provided[u,a,b] = \sum_{x \in U} bonding[u,x] * users\_to\_users[u,x]$$
$$\forall u \in U, a \in SPLIT, b \in GLOBAL, used\_to\_provided[u,a,b] = 1 - \sum_{x \in USERS} bonding[u,x] * users\_to\_users[u,x]$$
$$\forall u \in U, a \in USERV \cap PSERV, used\_to\_provided[u,a,a] = 1$$

Finally, the last constraint ensures that the usage intensity values are correctly assigned to the provided intensity values according to the *assign* variable. The well-known "big M" technique is used here to maintain linearity of the constraint. M simply represents a very large constant.

$$\forall p \in P, s \in PSERV, u \in U, provider\_intensity[p,s] \geq (1 - assign[u,p]) * -M + \sum_{a \in USERV} used\_to\_provided[u,a,s] * user\_intensity[u,a]$$

Based on constraints explained above, the objective function tries to minimize the cost of the provided services. The product of the *cost_coefficient* parameter, the *provider_intensity* variable, and the *service_weight* parameter is utilized to determine the overall cost. The second term in the objective guarantees that the lower bound constraint on the *users_to_users* variable is sufficient.

$$minimize:$$
$$\sum_{p \in P, s \in PSERV} \sum_{a \in P, b \in PSERV} (provider\_intensity[a,b] * cost\_coefficient[p,s,a,b] * service\_weight[s]) - \sum_{u \in U, w \in U} users\_to\_users[u,w]$$

Experiments relating to the complexity of using this model and its efficacy will be described in Section 7. For these experiments, the formulation is specified using the GNU MathProg [36] language, which is a subset of the popular AMPL [41] language.

6.3 Step 3: Assigning Initial Channel Sizes

After the computational and communication resources have been assigned in the allocation, we now turn to the final type of association that needs to be carried out between the functional and architectural models: that of allocating the memory or buffer resources. This is manifested by choosing the initial size of the communication channels in the functional model and associating them with memory components in the architectural model.

This type of allocation has the potential to change the behavior of the system. Specifically, artificial deadlock situations may develop if the channel sizes are not chosen appropriately. Unfortunately, determining the channel sizes statically is undecidable in general for the KPN model of computation. This fundamental undecidability is why the allocation of memory resources has been partitioned into two steps in the design flow. In the current step, the channel sizes are assigned statically. In the subsequent step in the design flow, a runtime manager process is deployed on the system in an attempt to ensure that artificial deadlock situations do not affect the overall behavior.

The initial sizing of a communication channel influences the degree of coupling between the reader and writer processes and also affects the likelihood of deadlock. If the channel size is large, then it can effectively serve as a buffer between the reading and writing actions of the associated processes. If it is small, then the frequent interleaving between the read and write actions needs to occur, slowing the progress of both processes.

The input to this step in the design is an allocated system where the computational and communication actions in the functional model have been associated with an architectural model. All reads and writes to the channels in this allocated system inherit the temporal characteristics of the architectural platform. With the allocation decisions made in the previous step, we also obtain constraints on the amount of memory that can be allocated to subsets of channels in the system. For instance, if five channels in the system are assigned to a architectural memory component with a size of 100 words, then the sum of initial sizes of these channels must be less than 100. Therefore, the major task in this step of the design flow is to partition the available memory between the channels in the system.

The sizing decisions taken in this step may need to be modified if deadlock situations develop. Under these circumstances, additional memory may be allocated to or removed from certain channels in the system. This is described in detail in the final step of the design flow in Section 6.4.

6.3.1 Prior Work

The prior work in the initial sizing of communication channels comes from the NoC and distributed systems communities. Sizing of buffers based on simulation is explored in [11]. Sizing based on formal analysis of processes is given in [31]. Finally, [3] develops sizing in the context of high level synthesis.

6.3.2 Approach

The approach we use utilizes the temporal information that is available to us after the allocation step in the design flow. Specifically, we can profile the production and consumption characteristics of the allocated system on a set of representative traces. To run this initial simulation, we size each channel such that it hypothetically uses the maximum memory allowed by the memory constraints. For instance, if we know that channels 1, 2, and 3 together need to fit within a memory block of 30 words, then each channel will be hypothetically sized to 28 words (the minimum size of a channel is 1 word).

Two pieces of data are recorded from each channel in the system during this hypothetical-size simulation. First, the probability histogram for the number of tokens in the channel is recorded. For this histogram, the x-axis has integer values that range from zero to the maximum hypothetical size of the channel, while the bar associated with each of these integer values indicates the probability that the channel had the corresponding number of tokens during the course of the simulation. The second piece of data recorded during this simulation is amount of data actually transferred through the channel during the simulation. The data transfer amount is useful to track since it represents the relative impact of the channel to the latency of the system. Channels which have a high throughput of data impact the latency of the system more when processes block on these channels.

The data from these hypothetical initial simulations is used to weight the channels in the sizing process. Channels which are judged to be more critical are allocated larger amounts of memory during the partitioning process. Criticality is based on the relative comparison between the data transfer amounts and the histograms obtained for each channel.

The initial sizing approach is only a heuristic to statically partition available system memory between the channels in the system. This initial partitioning may need to be modified if artificial deadlock develops in the system. This is handled in the subsequent step of the design flow.

6.4 Step 4: Runtime Deadlock Detection and Resolution

In general, the problem of determining bounds on channel sizes for KPNs so as to prevent artificial deadlock is undecidable. In this step, we will investigate heuristic runtime algorithms that monitor the state of channels in the system, and respond to different types of deadlock situations.

From the literature, we know that only directed or undirected cycles in a KPN can cause artificial deadlock to occur [19]. If all of the processes in the system are blocked, and at least one is blocked on a write operation due to insufficient space, then we know that a global deadlock situation has occurred. On the other hand, if processes in a directed or undirected cycle are blocked, and at least one of them is write-blocked, this is termed as local deadlock. In either of these situations, the deadlock can only be resolved by allocating extra memory to a communication channel.

However, which channel needs to be allocated memory, and the amount of memory to allocate such that the deadlock is resolved with the least amount of extra memory is undecidable in general. As with previous steps, we can tailor this decision based on the structure and profiling of the system.

Practically, we have not seen any evidence that any of the KPN-specific system-level design environments mentioned in Section 4 implement any form of deadlock detection and resolution. From a theoretical point of view, this is not sufficient, since static sizing is not enough given a general KPN specification. However, practically, the applications are used with these types of environment may not require this step. For instance, moving to more restricted forms of specification such as static dataflow would remove the need for this step from the design flow, but would also constrain expressiveness.

6.4.1 Prior Work

Parks [42] is the first to provide a strategy for dealing with artificial deadlock. Parks' algorithm involves waiting for all processes in the system to become deadlocked, and then increasing the size of the smallest channel in the system until at least one process becomes unblocked. Geilen and Basten [19] consider artificial local deadlock and attempt to explain why it may be useful to detect and correct this to guarantee a correct execution of the KPN. There has been fairly little work on coming up with other heuristics for deadlock resolution. Suggestions described in [8] lay out strategies based on reclaiming previously allocated memory. We develop these strategies further in this step of the design flow.

6.4.2 Deadlock Detection

The aims for deadlock detection algorithms are two-fold: to find all instances of deadlock as soon as possible after they occur and to avoid false positives. The first aim is important since the occurrence of a deadlock usually prevents the system from producing further output and adds latency. The second aim prevents the waste of scarce memory resources. Also, the deadlock detection procedure itself should not add excessive latency to the system.

There are many instances in the literature of distributed deadlock detection algorithms [17]. These algorithms usually assume a polling type of mechanism by which processes in the system can check the status of resources that are requested. In order to adapt this mechanism to Kahn processes, we would have to place wrappers around each process that would be able to check FIFO status. Synthesizing a system scheduler [16] is another approach that can be taken. This approach may be effective, but may not be suitable for all applications.

We take a different approach which relies on a separate process in the system that detects and resolves deadlock situations. In the sequel, this will be referred to as the manager process. The input to this step in the design flow is an allocated system with initial sizes for the channels. The output is a deadlock manager process which is customized for the system being managed.

The manager can serve either as a distributor of shared resources or as an observer that influences the rest of the system only when necessary. The first role requires that all requests for reading or writing be "approved" by the manager before they can proceed. The manager can keep track of the resource contention graph of the system and take action when deadlock is imminent. The benefit of this approach is that deadlock situations can be detected immediately. The downside is

that the overall performance of the system is severely compromised by the bottleneck through the manager process. In effect, the parallelism of the system is severely reduced.

In the second role, the manager occasionally probes the channels in the system to determine if a deadlock has occurred. This implies that a deadlock may not be immediately detected. However, in the absence of a deadlock, the parallelism of the system is not restricted. The maximum amount of time required to detect a deadlock can still be bounded based on the scheduling of the processes in the system.

In our approach, we adopt the former role for the manager process. The behavior of the manager is to periodically monitor the status of each of the channels in the system. A channel can be one of three states: normal, read-blocked, or write-blocked. If the manager finds a cycle of channels that are blocked, and at least one of these channels is write-blocked, then an artificial deadlock has been found. Alternatively, to detect global deadlock situations, the manager can wait until all processes in the system are blocked, and at least one is write-blocked. Once a deadlock situation is detected, the manager invokes the deadlock resolution heuristic described in Section 6.4.3.

More specifically, to detect deadlock, the deadlock manager at runtime maintains a list of channel entries. There is a single channel entry for each channel in the system. Each channel entry also carries the ID of the reader and writer processes for the channel. Process IDs are assumed to be unique in the system. The channel entry has a status field which the manager can update. Every time it is run, the manager process reads the status of each channel in the system. The status field is marked as normal, read-blocked or write-blocked for each channel. A write-blocked channel is a full channel to which its writer has attempted to write, therefore blocking the writer process. Similarly, a read-blocked channel is an empty channel from which the reader has attempted to read, blocking the reader process. A normal channel is a channel which is not in either of the prior two situations.

Once the status of all the channels in the system has been obtained, the manager carries out a depth-first search (DFS) procedure starting at each write-blocked channel to detect local deadlock. The DFS search only considers the read or write blocked channels, it ignores all normal channels. If the DFS procedure ever returns to the base channel, then a cycle of blocked processes has been found. All write-blocked channels within this cycle (including the base channel) are marked with a tag that indicates that they are candidates for deadlock resolution.

If only global deadlock is being detected, then the procedure is much simpler. The manager simply determines that all the channels in the system are blocked, and at least one channel is write-blocked. Then, all write-blocked channels are similarly tagged for deadlock resolution.

6.4.3 Deadlock Resolution

The challenge is deadlock resolution lies in the fact that system memory is typically a scarce resource. A “good” deadlock resolution approach allocates available memory to channels in such a way that the deadlock is resolved with a minimum amount of extra memory. According to this metric, determining an optimal allocation strategy is undecidable. In this section, we investigate heuristics that attempt to develop system-specific allocation strategies that work well in practice.

Deadlock resolution can only be accomplished by allocating additional memory to channels in the system at runtime. After the deadlock manager has detected a deadlock situation (either global or local), the next step is to allocate additional memory to the tagged channels in order to allow the blocked processes to proceed.

Most prior work on artificial deadlock resolution in the literature adopts a very simple approach. Typically, the smallest write-blocked FIFO involved in a deadlock is allocated additional memory until the deadlock is resolved. This is the approach adopted by [2] and [40].

A key component of the approach that we develop here is the concept of *reclaiming* previously allocated memory. This means that if the manager previously allocated a certain amount of extra memory to a particular channel due to a deadlock situation, in subsequent deadlock situations, this extra memory (if it is not in use) can be reclaimed by the manager. The newly reclaimed memory can then be allocated to other channels that may require it. A possible downside of reclaiming previously allocated memory is that resolved deadlocks may crop up again.

More specifically, the resolution algorithm in the deadlock manager first obtains a list of all the tagged channels in the system. The tagged channels represent write-blocked channels that are involved in either local or global deadlock situations. The deadlock manager controls access to a pool of shared memory in the system. To resolve the deadlock, the deadlock allocates some of the memory in the pool to a tagged channel. If no more memory remains in the pool, the manager attempts to obtain previously allocated memory from the pool which is no longer being used. If tagged channels exist, the pool is completely allocated, and no previously allocated memory can be reclaimed, then the manager has no recourse but to raise an exception. Ultimately, this is the situation we would like to prevent with heuristics.

To decide which channel to allocate memory and the amount of extra memory to allocate, we rely on additional information in the list of channel entries that the deadlock manager maintains. In addition to a reader ID, a writer ID, and a status field, the channel entries are assigned another parameter. This parameter is an integer which corresponds to the worthiness of that channel to receive additional memory, and is initially derived from the initial sizing information obtained in the previous section. In particular, the worthiness initially is:

$$worthiness_i = (maxsize - size_i) * throughput_i$$

In this assignment, $maxsize$ is the largest size of any channel in the system, while the size and throughput of the current channel are given by $size_i$ and $throughput_i$ respectively. When a deadlock is detected, the channel with the highest worthiness value is allocated additional memory equal to a fixed fraction of the remaining memory in the pool. The worthiness value of the channel is reduced since the size of the channel is increases.

If no more memory remains in the pool, memory is collected until a predefined amount is once again gathered. The memory is recovered from channels that have been previously allocated memory and whose worthiness metrics are the lowest. Intuitively, this avoids thrashing, where the same channels are allocated and deallocated memory repeatedly. Of course, the deadlock resolution algorithm described here is merely a heuristic, and in the worst case, when no more memory remains in the pool, and no memory can be recovered, the manager raises a flag and the

system must be reset.

6.5 Integration with the Metropolis framework

The algorithms and approaches described for each step of the design have been implemented within the context of the Metropolis design framework. In this Section, the mechanics behind this integration will be described.

First, a library was created to support the KPN model of computation and to allow the types of operations on channels that are needed in the design flow. The components of this library include media describing the channels that are template-based to handle different data types. Generic processes are also defined within this library, so that it becomes relatively easy to subclass this generic process and implement real functionality. An implementation of the deadlock manager process is also implemented within the library. The deadlock manager is initialized with a structural representation of the functional model, so that it can carry out the DFS algorithms and allocate/deallocate memory as required.

An integral part of this library is the interfaces that are defined for communication between the processes, channels, and the deadlock manager. The processes can communicate with the channels using the standard read() and write() functions. The deadlock manager can query the state of a channel, allocate memory to it, and deallocate unused memory that was previously allocated. Careful use of mutual exclusion is made in the implementation of the channels to avoid unnecessary deadlock or race conditions.

Along with components of the functional model, the primitives required for the architecture have also been implemented or adapted from existing designs. In particular, non-deterministic tasks, processor media, a bus medium, memory media, first-come-first-server (FCFS) schedulers, and a GlobalTime quantity manager have all been utilized for the architectural modeling. The interaction between the processors and the GlobalTime quantity manager is carried out in such a way that it is relatively easy to change the costs associated with the services. Also, the entire architectural model is easy to reconfigure with regards to the number of processors that are instantiated, the number of tasks on each processor, and the size of the memory components.

7 Experimental Results

In this section, we will apply the algorithms developed in Section 6 to an industrial set-top box system. The experiments concentrate on the second and third steps in the design flow: allocation and initial sizing. The application for this system is a Picture-in-Picture component used in digital television. A block diagram of this application is shown in Figure 9. The architectural platform is a heterogeneous multiprocessor architecture of the type shown in Figure 10.

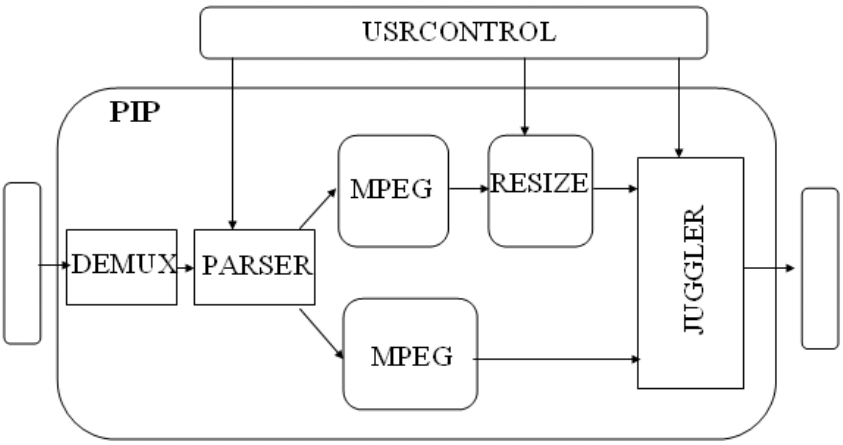

Figure 9: Picture-in-Picture Application

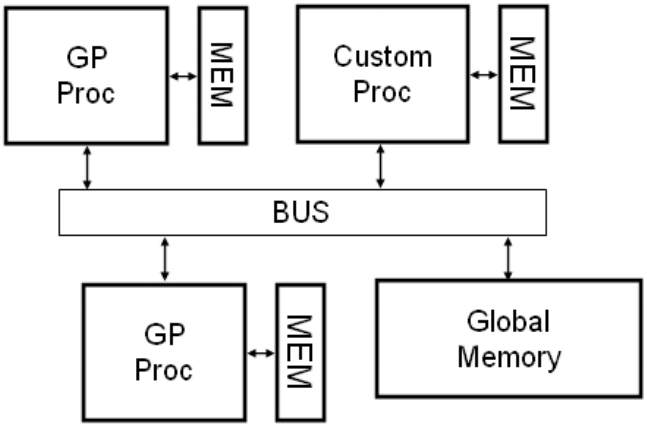

Figure 10: Architectural Platform

7.1 PiP Application

The PiP application takes as input two MPEG streams and outputs a single MPEG stream. Frames from the first input MPEG streams are resized and placed within a window. These frames are then inserted into a specific portion of the frames from the second input stream. In the set of experiments described here, we choose to focus on the core of the PiP application, which involves the horizontal and vertical resizing of frames from the first input stream. The resize component was modeled in Metropolis with 26 concurrent processes and 57 channels, not including the deadlock manager. Block diagrams of the horizontal and vertical resize blocks, which together constitute the resize component, are shown in Figures 11 and 12. The source and sink processes are inserted into the functional model to model the data production and consumption characteristics of the rest of the application.

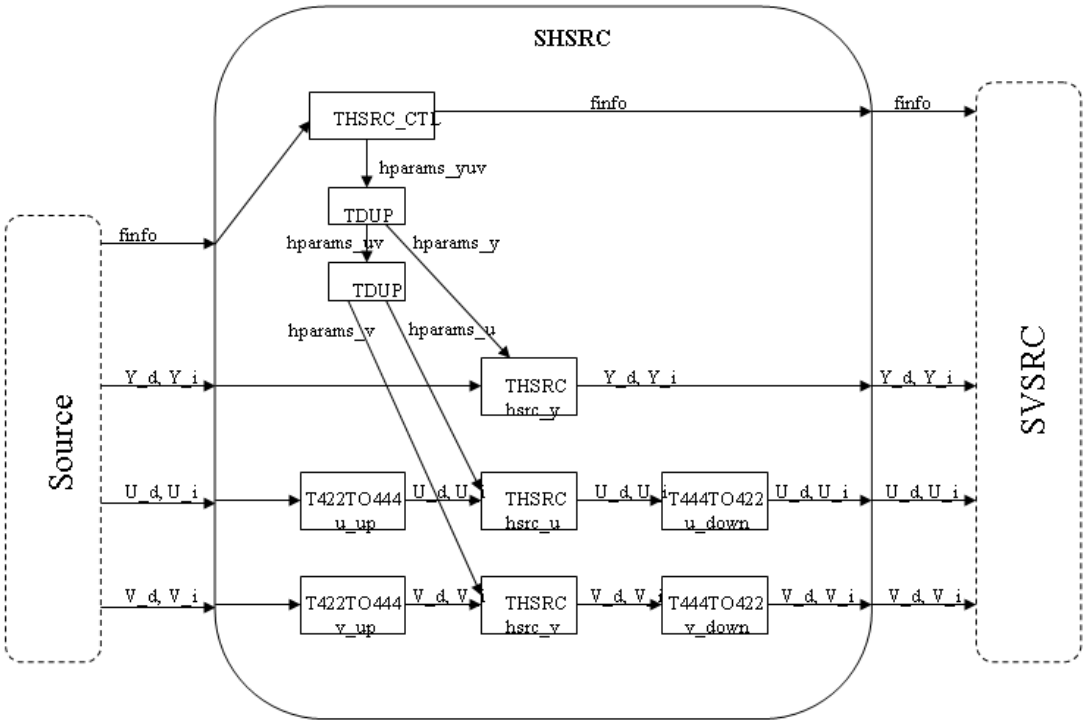

Figure 11: Horizontal Resize Component

7.2 Heterogeneous Multiprocessor Architectural Platform

The architectural platform used in the experiments consists of a set of processors, local and global memories and a system bus. The platform can be configured to have a variable number of processors connected to the global bus. Each processor can support multiple tasks, and grants access to these tasks in a first-come-first-serve manner. The size of the local memory in a processor and the system memory connected to the bus can be parameterized.

Each processor provides a set of services to the tasks that run on it. These services are: global read, global write, local read, local write, and execute. The execute service represents an abstract view of an instruction or block of code being executed on the processor. The read/write services are associated with either the system memory or the local memory of the processor. These read/write services can be parameterized with the number of the words that are to be transferred. The cost incurred by each processor and memory for providing the above services can also easily be changed.

For the experiments presented here, the specific configuration of the architecture has four processors, each with their own local memories. The processors are all connected to a single global bus component, which has a slave global memory block connected to it. The cost incurred by each processor to provide each service, in cycles, in presented in Table 3.

There are 26 tasks instantiated on this architecture, they are statically partitioned as shown in Table 4.

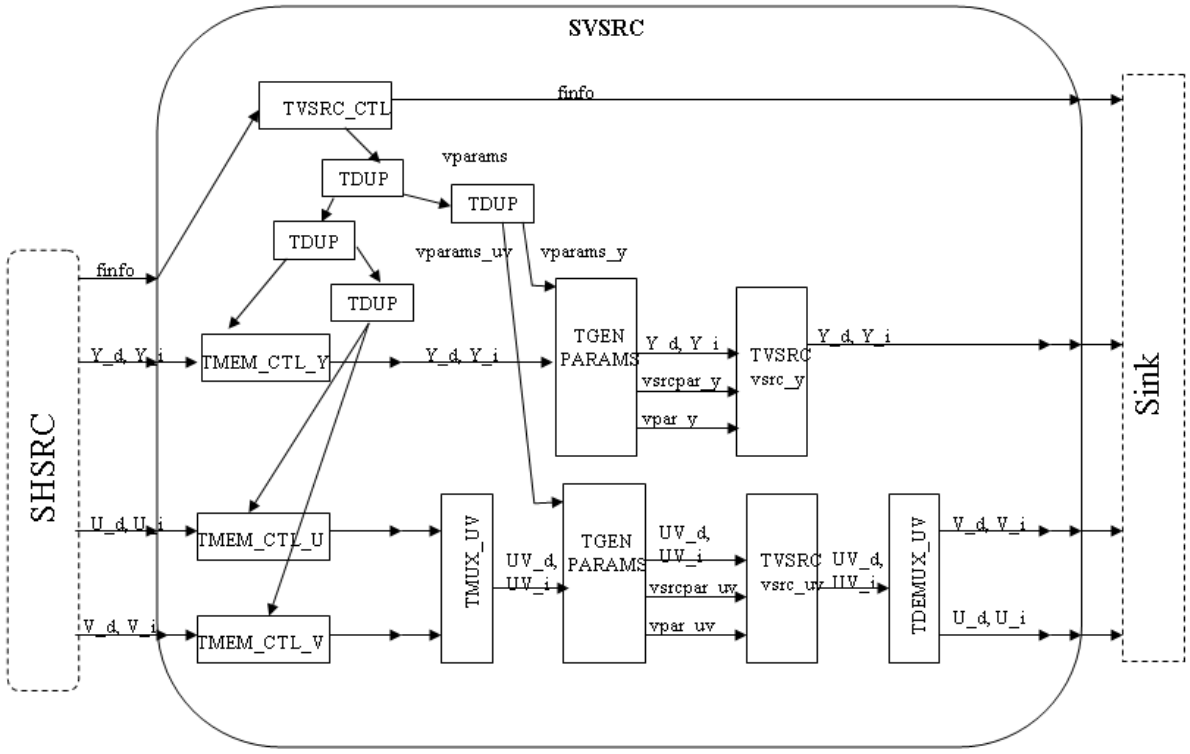

Figure 12: Vertical Resize Component

Services	Proc. #1	Proc. #2	Proc. #3	Proc. #4
execute	3	3	5	5
local read	2	3	2	3
global read	3	4	3	4
local read	6	6	6	6
local write	8	8	8	8

Table 3: Service Cost on each Processor

7.3 Allocation Experiments

The first set of experiments carried out on the PiP system explores the effectiveness of the allocation algorithm described in Section 6.2. The aim here is to test the predictive capabilities of the formulation. Specifically, fidelity is the measure by which the success of the allocation optimization can be evaluated. Fidelity is the correspondence between the total orders assigned to the cost of feasible allocations by the objective function and actual system simulation. In other words, if the objective function predicts that an allocation a has lower cost than another allocation b, then the cost of a obtained via system simulation should also be better than the cost b.

Simplified cost functions for the architectural services are used for these experiments, where the cost of utilizing a particular service is independent of the intensities of other services on the same component. This simplification makes the formulation much simpler, but sacrifices some of the expressiveness that can potentially make the formulation more accurate. The costs are taken

Processor	Tasks
1	1–6
2	7–12
3	13–19
4	20–26

Table 4: Assignment of Tasks to Processors

directly from Figure 3. The formulation as given in the Appendix is automatically translated to the MPS file format and given as input to IBM's OSL solver. All of the optimizations described here are run on a dual 3.0 GHz linux machine with 4 Gb of memory. The initial sizes of the communication channels were made high enough such that they effectively simulate infinite channel memory. In other words, no write-blocking takes place for any channel for all the experiments here. Effects of reducing the memory size will be explored in Section 7.4.

In the first experiment, the objective function is modified to only minimize the inter-component communication in the system and not consider the latency cost at all. This modified objective function can be thought of solving just the min-cut problem for the allocation. The best feasible result returned by the ILP solver after 15 minutes of execution is recorded and referred to as Solution 1. Due to the handicapping of the objective function, Solution 1 should exhibit worse results than the other experiments. The second, third, and fourth experiments consider the full objective function and attempt to ascertain that the objective function tracks with the actual system performance. Solution 2 is obtained by considering the first feasible solution found by the ILP solver on the problem instance, whereas solution 3 considers the best solution found after 15 minutes of execution, while Solution 4 is the best found after 30 minutes. These points in the design space are summarized in Table 5.

Soln. No.	Obj. funct.	Cutoff Time, minutes
1	no service cost	15
2	full	first feasible
3	full	15
4	full	30

Table 5: Measuring the Fidelity of the Objective Function

Based on these four solutions from the optimization problem, synchronization constraints were automatically generated in Metropolis to reflect the allocations. The PiP system was then simulated for each solution, and the latency in cycles to process five frames of image data was recorded. The results are shown in Figure 13. As expected, the first solution has the worst result since the objective function does not take into account the service cost. The remaining solutions achieve better results depending on the amount of time that the solver was given. This shows, that at least for this particular case study, the formulation of the allocation optimization problem is able to estimate system performance.

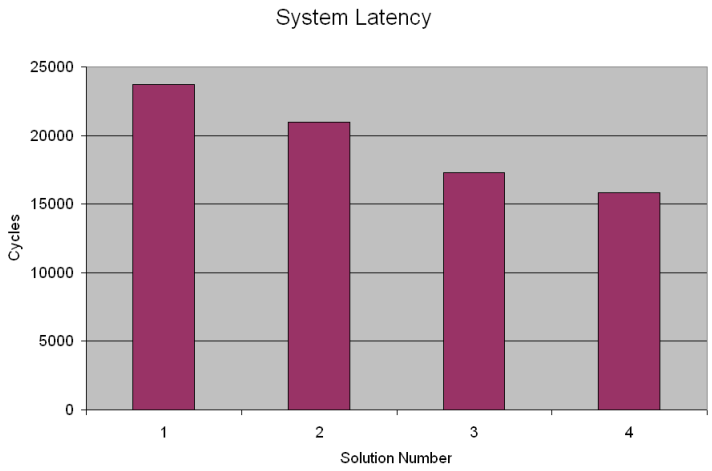

Figure 13: System latency scales well with allocation objective function

7.4 Initial Sizing Experiments

The objective in the initial sizing experiments is to take the solution with the best overall latency from the allocation experiments and progressively tighten the available memory. Then, initial sizes for all channels are assigned according to two separate algorithms. The first algorithm evenly divides the available memory while the second algorithm is the heuristic described in the third step of the design flow. The hypothesis is that as the channel sizes are reduced, the amount of interleaving between reader and writer processes will increase and add to the overall latency of the system.

In the experiments carried out here, the size of the global memory component in the architecture is reduced from its initial value, while all the local memory components remain at the same size. To fit within the available memory, the sizes of some of the 29 channels that have been allocated to the global memory component must be reduced. The first algorithm divides memory evenly between the available channels, so all 29 channels will be affected by a small decrease in the available global memory. The second algorithm only reduces the size of a channel when necessary. Table 6 shows the number of channels in the system which have their bounds reduced as a result of the heuristic sizing algorithm.

No.	Global Memory Size	Channels Affected
1	700	0
2	600	6
3	500	14
4	400	20
5	300	25
6	200	29
7	100	29

Table 6: Number of channels affected by second sizing algorithm

The effect of both algorithms on the overall latency incurred in resizing five frames of image

data is summarized in Figure 14. The results show that regardless of the algorithm applied, there is a huge impact on system latency. This confirms the hypothesis that initial sizing is a very important part of the mapping problem. For relatively large amounts of available memory (¿ 450 words), both algorithms perform equally well, this is not shown in the Figure. When the amount of available memory ranges between 20% and 60% of the initial memory, the heuristic algorithms performs better than the equal sizing algorithm, resulting in system latencies which are approximately 30% lower. As the available memory falls below 20% of the original, both algorithms peform badly, as expected. For this region, the heuristic does worse than the equal sizing algorithm. This is explained by the fact that the heuristic is "trained" based on data obtained from large initial sizes of the channels. The predictive capabilities of the heuristic break down as the reduction in memory becomes larger. This suggests that the heuristic can be enhanced by incorporating input data obtained from a variety of channel sizes.

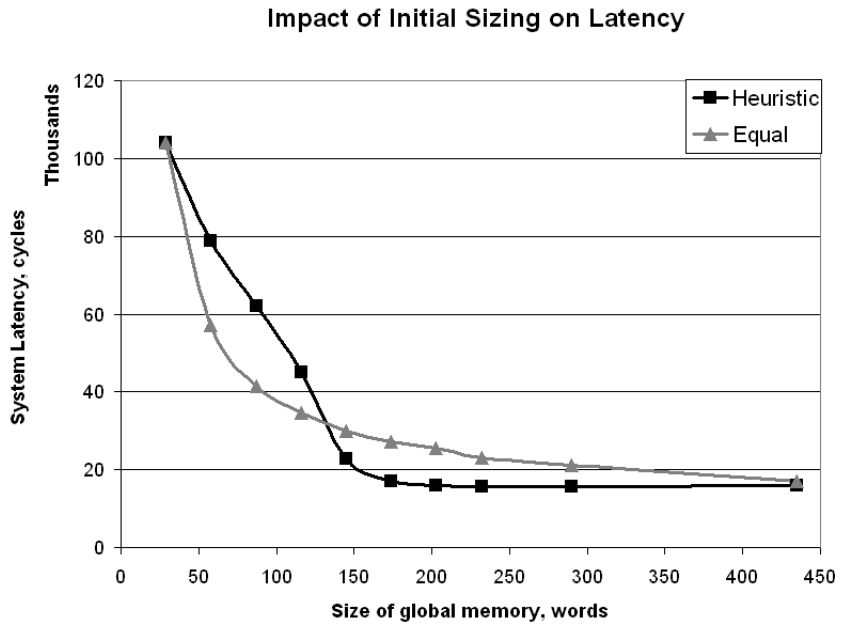

Figure 14: System latency is greatly impacted by channel size

8 Conclusions

Mapping problems are commonly encountered during system-level design. Common practice both in industry and academia involves developing specialized algorithms for each problem. There is much similarity in the algorithms which are eventually developed, but the lack of a common framework for reasoning about these problems prevents the reuse of these approaches. We have presented an initial framework for reasoning about many different types of mapping problems using the concept of a common semantic domain. Even though the task of finding a common semantic domain given a class of functional and architectural models is very difficult, reasoning about the properties of this domain helps determine the types of optimizations that can be applied.

In this work, the major contribution is a design flow for implementing Kahn Process Networks specifications within the Platform-based design methodology. This design flow has clearly been separated into a number of steps. For each step in the design flow, we have outlined algorithms and approaches to tackle the corresponding optimization problems. Experiments have been carried out on a case study in the Metropolis design framework to test the validity of some of these approaches.

9 Future Work

In the future, we would like to apply our design flow to additional case studies, to better evaluate its effectiveness. We would also like to more fully develop the concept of a common semantic domain to allow for automated mapping. We believe that some of the ideas from this design flow can be carried over to develop similar flows for other models of computation.

For functional modeling, it is relatively easier to capture the behavior of interest if we rely on well-known models of computation. Capturing architectural behavior is somewhat more complicated. Since the architectural specification is usually captured bottom-up from characterization of existing physical platforms, it is not usually clear which aspects of the behavior need to be taken into account. This work assumed that an architectural model was provided which offered a certain set of services at a certain cost. The assumption was made that the cost was accurately captured by the cost functions provided. However, further work is required to validate this assumption. Specifically, we would like to correlate our architectural models better with the existing architectural platforms.

One of the benefits of the four-step design flow that was mentioned was the ability to build on prior work from different research communities. For the first step of the design flow, formulating the clustering problem with a clear cost function and building a clustering tree – or dendrogram – in the manner described in [13] is interesting to pursue. Also, a formal description of the sufficient conditions for ensuring that clustering is legal is required as well. In terms of the architectural reconfiguration stage, linking the possible configurations of the architectural model to performance data obtained from real designs (e.g. FPGAs) is a related topic.

For the second step of the design flow, we would like to explore heuristic approaches to solving the covering problem. Covering problems are a well-studied topic, and several heuristics have been developed that have applicability to the types of problems we consider. Two such areas are negative thinking [21] and cutting planes [39]. Also, enhancing the formulations to deal with statistical distributions of parameters is also an interesting area of future research since the type of data utilized during this step is a static estimate of dynamic behavior. Aspects such as correlation between service usage may be interesting to track for some kinds of cost functions.

For the third step, the results have shown the benefits and the weaknesses of the simple heuristic algorithm that was developed. Enhancing this algorithm to deal with estimates of behavior under tight memory constraints is necessary. It might be worthwhile to obtain additional information as input for this algorithm such as correlation between producers and consumers or a finer granularity view of the processes themselves (perhaps in the form of a CDFG).

For the fourth step, testing needs to be carried out to determine the effectiveness of the memory reclamation strategy described. Even if actual designs do not require a deadlock manager, it may still be useful to consider a manager in conjunction with the initial sizing step. For instance, simulations can be run with sample traces in which the deadlock manager allocates and deallocates memory as needed. Once the memory has stabilized for channels, the simulation can be stopped and the sizes of the channels used as the initial sizes for future simulations. After running a sequence of these simulations, it may be possible to converge to the set of initial sizes that will provide the best system performance for an implementation.

Finally, a broad area of future research deals with the idea of common semantic domains for solving mapping problems. Specifically, we are interested in applying the same concepts developed here to distributed systems and control dominated systems. After reasonable common semantic domains have been found for several types of mapping problems, we can then begin to extract the commonalities in the mapping problems and more effectively consider the reuse of algorithms and heuristics.

References

[1] K. Keutzer A. Mihal. Mapping Concurrent Applications onto Architectural Platforms. pages 39–59. Kluwer Academic Publishers, 2003.

[2] G.E. Allen, B.L. Evans, and D.C. Schanbacher. Real-time sonar beamforming on a unix workstation using process networks and posix threads. In *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems & Computers, Vol.2, Iss.*, pages 1725–1729, 1998.

[3] Tod Amon and Gaetano Borriello. Sizing Synchronization Queues: A Case Study in Higher Level Synthesis. In *Proceedings of the 28th conference on ACM/IEEE design automation*, pages 690–693. ACM Press, 1991.

[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, April 2003.

[5] Felice Balarin, Jerry Burch, Luciano Lavagno, Yosinori Watanabe, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*, page 129. IEEE Computer Society, 2001.

[6] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software Co-design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[7] Felice Balarin, Luciano Lavagno, and et al. Concurrent Execution Semantics and Sequential Simulation Algorithms for the Metropolis Metamodel. *Proc. 10th Int'l Symp. Hardware/Software Codesign*, pages 13–18, 2002.

[8] T. Basten and J. Hoogerbrugge. Efficient execution of process networks. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Proc. of Communicating Process Architectures 2001*, pages 1–14. IOS Press, 2001.

[9] B. Bhattacharya and S. S. Bhattacharyya. Parameterized modeling and scheduling of dataflow graphs. Technical Report UMIACS-TR-99-73, Institute for Advanced Computer Studies, University of Maryland at College Park, December 1999. Also Computer Science Technical Report CS-TR-4083.

[10] Joseph Tobin Buck. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical Report ERL-93-69, 1993.

[11] Vikas Chandra, Anthony Xu, Herman Schmit, and Larry Pileggi. An Interconnect Channel Design Methodology for High Performance Integrated Circuits. In *Proceedings of the conference on Design, automation and test in Europe*, page 21138. IEEE Computer Society, 2004.

[12] Abhijit Davare, Douglas Densmore, Vishal Shah, and Haibo Zeng. Simple case study in metropolis. Technical Report UCB.ERL 04/37, University of California, Berkeley, September 2004.

[13] Abhijit Davare, Kelvin Lwin, Alex Kondratyev, and Alberto Sangiovanni-Vincentelli. The best of both worlds: The efficient asynchronous implementation of synchronous specifications. In *Design Automation Conference (DAC)*. ACM/IEEE, June 2004.

[14] E. A. de Kock. Multiprocessor Mapping of Process Networks: a JPEG Decoding Case Study. In *Proceedings of the 15th international symposium on System Synthesis*, pages 68–73. ACM Press, 2002.

[15] Doug Densmore. Metropolis Architecture Refinement Styles and Methodology. Technical Report UCB/ERL M04/36, University of California, Berkeley, CA 94720, September 14, 2004.

[16] Javed Dulloo and Philippe Marquet. Design of a real-time scheduler for kahn process networks on multiprocessor systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2004.

[17] Ahmed K. Elmagarmid. A Survey of Distributed Deadlock Detection Algorithms. *SIGMOD Rec.*, 15(3):37–45, 1986.

[18] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 182–187. ACM Press, 2003.

[19] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming*, 2003.

[20] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999. Research report UCB/ERL M97/57.

[21] E. Goldberg, L. P. Carloni, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Negative thinking in branch-and-bound: the case of unate covering. *IEEE Transactions on Computer-Aided Design*, 19(3):281–294, March 2000.

[22] Matthias Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integration, the VLSI Journal, Elsevier*, 38(2):131–183, December 2004.

[23] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[24] Intel. http://www.intel.com.

[25] G. Kahn. The Semantics of a Simple language for Parallel Programming. In *Proceedings of IFIP Congress*, pages 471–475. North Holland Publishing Company, 1974.

[26] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of IFIP Congress*, pages 993–998. North Holland Publishing Company, 1977.

[27] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[28] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[29] Bart Kienhuis and Ed F. Deprettere. Modeling stream-based applications using the sbf model of computation. *J. VLSI Signal Process. Syst.*, 34(3):291–300, 2003.

[30] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modeling for Signal Processing Systems. *Proceedings of the 37th Design Automation Conference*, 2000.

[31] Tilman Kolks, Bill Lin, and Hugo de Man. Sizing and Verification of Communication Buffers for Communicating Processes. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 660–664. IEEE Computer Society Press, 1993.

[32] E.A. Lee and T.M. Parks. Dataflow Process Networks. In *Proceedings of the IEEE, vol.83, no.5*, pages 773 – 801, May 1995.

[33] E.A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, Dec. 1998.

[34] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[35] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System Level Design With Spade: An m-jpeg Case Study. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 31–38. IEEE Press, 2001.

[36] GLPK: GNU Linear Programming Kit. http://www.gnu.org/software/glpk/glpk.html.

[37] Xiaojun Liu, Yuhong Xiong, and Edward A. Lee. The Ptolemy II Framework for Visual Languages. In *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 50. IEEE Computer Society, 2001.

[38] Y. Watanabe M. Sgroi, L. Lavagno and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proceedings of Design Automation Conference, DAC '99*, June 1999.

[39] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Appl. Math.*, 123(1-3):397–446, 2002.

[40] F.S. Mendes, M.S.M. Silva, F.A.B. Silva, E.P. Lopes, E.P.L. Aude, H. Serdeira, J.T.C. Silveira, and M.F. Martins. Cpnsim: A computational process network simulator. In *2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2003), v.1.*, pages 234–239, 2003.

[41] AMPL: A Modeling Language for Mathematical Programming. http://www.ampl.com.

[42] T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.

[43] T. Parks, J. Pino, and E. Lee. A comparison of synchronous and cyclostatic dataflow, 1995.

[44] J. Paul and D. Thomas. A Layered, Codesign Virtual Machine Approach to Modeling Computer Systems. In *Proceedings of the conference on Design, automation and test in Europe*, page 522. IEEE Computer Society, 2002.

[45] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring Embedded-Systems Architectures With Artemis. *Computer*, 34(11):57–63, 2001.

[46] Alessandro Pinto. Metropolis Design Guidelines. Technical Report UCB/ERL M04/40, University of California, Berkeley, CA 94720, September 14, 2004.

[47] Tarvo Raudvere, Ingo Sander, Ashish Kumar Singh, and Axel Jantsch. Verification of Design Decisions in ForSyDe. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 176–181. ACM Press, 2003.

[48] Donatella Sciuto, Fabio Salice, Luigi Pomante, and William Fornaciari. Metrics for Design Space Exploration of Heterogeneous Multiprocessor Embedded Systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 55–60. ACM Press, 2002.

[49] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 159–168. ACM Press, 1982.

[50] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 7–12. ACM Press, 2002.

[51] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart, and Ed Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of the conference on Design, automation and test in Europe*, page 10340. IEEE Computer Society, 2004.

[52] The Metropolis Project Team. The Metropolis Meta Model Version 0.4. Technical Report UCB/ERL M04/38, University of California, Berkeley, CA 94720, September 14, 2004.

[53] Xilinx. http://www.xilinx.com.