Automated Mapping for Heterogeneous Multiprocessor Embedded Systems

Abhijit Davare

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-115
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-115.html

September 7, 2007

Automated Mapping for Heterogeneous Multiprocessor Embedded Systems

by

Abhijit Davare

B.S. (University of Pittsburgh) 2002
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Jan Rabaey
Professor Alper Atamtürk

Fall 2007

The dissertation of Abhijit Davare is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

Automated Mapping for Heterogeneous Multiprocessor Embedded Systems

Abhijit Davare

Abstract

Automated Mapping for Heterogeneous Multiprocessor Embedded Systems

by

Abhijit Davare

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

Increasing design complexity and time-to-market concerns have led to the increased prevalence of programmable processing elements for embedded systems. These platforms feature multiple processing elements, some of which may be customized for specific domains. Deploying applications typical of embedded system domains such as multimedia and transportation onto these platforms is difficult, not only due to the heterogeneous parallelism in the platforms, but also due to the performance constraints that typify embedded systems.

This dissertation advocates and validates a design flow that enables designers to deploy applications onto this emerging class of embedded platforms. The design flow is based on the platform-based design methodology which initially separates the modeling of the application and architectural platform. The design flow advocates transforming these models in a structured manner such that both have compatible execution models and abstraction levels. The goal of this structured model transformation is to simplify the

deployment challenge into a covering problem where portions of the application are assigned to processing elements in the architectural platform.

The focus of this work is to validate the design flow by applying it to embedded systems from the multimedia and automotive domains. The case studies explore the trade-offs inherent in modeling these systems as well as techniques for automatically solving the mapping problems. The automated techniques use mathematical programming approaches which have the flexibility to handle changes in the problem assumptions.

We observe that regardless of the domain, some aspects of the design flow such as modeling and simulation are shared between systems. Based on the insights gained in applying this design flow to the case studies, the requirements and initial implementation for Metro II – a next-generation design framework for platform-based design – are described.

Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my advisor Prof. Alberto Sangiovanni-Vincentelli for his excellent mentorship during my years at Berkeley. His emphasis on formal underpinnings for system-level design and on effective written and verbal communication of research results have not only influenced my graduate work, but will also continue to guide me in the years to come. His enthusiasm and tireless support are appreciated, and I look forward to future collaboration with him.

Prof. Jan Rabaey was the chair of my qualifying exam committee and a member of my dissertation committee. His broad knowledge of embedded systems and astute identification of industry trends has helped focus this work in the larger context of semiconductor-based systems.

Prof. Alper Atamtürk was the outside member in the dissertation committee and the instructor of two fascinating courses which I took in the Department of Industrial Engineering and Operations Research. His introduction to Mixed Integer Mathematical Programming and feedback on initial versions of the automated approach for multimedia systems were crucial to this work.

Prof. Edward Lee was the reader of my Masters report as well as a member of my qualifying exam committee. His course on models of computation as well as his extensive set of publications on dataflow models and their deployment were the most heavily leveraged for the multimedia portion of this work.

Prof. Kurt Keutzer was my temporary advisor during my first year of graduate school and the instructor who introduced me to computer-aided design and entrepreneur-

ship. Prof. Keutzer has done a great deal to bring a sense of community to the D.O.P Center.

Apart from professors, I had the distinct privilege of receiving mentorship from and collaborating with a fantastic set of industrial researchers, postdoctoral researchers, and senior graduate students. These include: Felice Balarin, Marco Di Natale, Paolo Giusto, Sri Kanajan, Alex Kondratyev, Farinaz Koushanfar, Luciano Lavagno, John Moondanos, Michael Orshansky, Roberto Passerone, Claudio Pinello, Stavros Tripakis, and Yosinori Watanabe.

The friends and colleagues whom I interacted with while at Berkeley are the most talented, hard-working, and downright most interesting group of people I've had the pleasure of knowing. These include, but are not limited to: Alvise Bonivento, Bryan Brady, Mike Case, Bryan Catanzaro, Donald Chai, Arindam Chakrabarti, Satrajit Chatterjee, Rong Chen, Xi Chen, David Chinnery, Jike Chong, Douglas Densmore, Arkadeb Ghosal, Yujia Jin, Shinjiro Kakita, Nathan Kitchen, Vinay Krishnan, Animesh Kumar, Yanmei Li, Cong Liu, Kelvin Lwin, Slobodan Matic, Mark McKelvin, Trevor Meyerowitz, Matthew Moskewicz, David Nguyen, Alessandro Pinto, William Plishker, Kaushik Ravindran, N.R. Satish, Kedar Shah, Vishal Shah, Farhana Sheikh, Sampada Sonalkar, Xuening Sun, Martin Trautmann, Gerald Wang, Guang Yang, Yang Yang, Haibo Zeng, Wei Zheng, and Qi Zhu. In particular, Qi Zhu was heavily involved in the research that is presented within this dissertation and has been a steadfast friend and collaborator during the past three years.

Jennifer Stone, Dan MacLeod, and Jontae Gray provided excellent administrative support and flawlessly processed countless reimbursements. Special thanks to Ruth Gjerde

in the EE division Graduate Student Affairs office for her friendly and patient guidance in navigating the treacherous bureaucratic channels at UC Berkeley.

Non-academic pursuits including science fiction, tennis, and racquetball provided the necessary distractions. The Recreational Sports Facility and the Berkeley Public Library played a key role in energizing my body and mind while at Berkeley.

The encouragement of my parents was crucial during the past five years. Their continued advice to focus on longer-term goals and ignore the short-term trials and tribulations provided me the perspective and support I needed.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota. This work was also supported in part by the MARCO-sponsored Gigascale Systems Research Center (GSRC) and a grant from the Consumer Electronics Group of Intel Corporation.

Chapter 1

Introduction

Embedded electronic systems are specialized to carry out specific tasks and are "embedded" in their environment. This is in contrast to personal computers or supercomputers which are general purpose and interact with users. Embedded systems are much more prevalent than their general-purpose counterparts; for instance, 98% of all microprocessors manufactured in a given year are used within embedded systems [82]. Embedded systems typically have strict performance requirements relating to issues such as latency, throughput, jitter, memory usage, and energy consumption. Due to their widespread usage and performance-critical nature, the design of embedded systems is both relevant and challenging.

Today, there is a shift toward using multiple heterogeneous programmable processing elements (PEs) for a variety of embedded systems domains including multimedia and automotive. Deploying complex embedded applications on such platforms is especially challenging since these systems must meet strict performance constraints.

This section will summarize the main characteristics of embedded applications, trends that give rise to heterogeneous parallel platforms, and the resultant design challenges. The solutions proposed in the remainder of the dissertation to tackle these problems will also be outlined.

1.1 Trends

Embedded applications are rapidly increasing in complexity at a time when design times, power consumption, and non-recurring engineering (NRE) costs are becoming critical. The confluence of these factors leads to a set of design challenges.

1.1.1 Embedded Applications

Computational requirements for embedded applications are increasing exponentially [92] [53] [83] [84]. During the past 15 years, a variety of new protocols and standards have been introduced which feature rapidly increasing computational requirements. Figure 1.1 shows some of these trends for three classes of multimedia applications: video, cellular, and wireless LAN. Code size for these applications is also increasing, reflecting the trend that application complexity is increasing along with computational requirements. This exponential trend is creating demand for the increasing number of transistors that can be integrated with scaling [104].

In the automotive domain, the trend is similar. The value of software in a vehicle is expected to increase from 4% of the overall cost in 2000 to 13% of the overall cost in 2010[50]. Currently, a high-end automobile may contain over 270 different types of functionality that

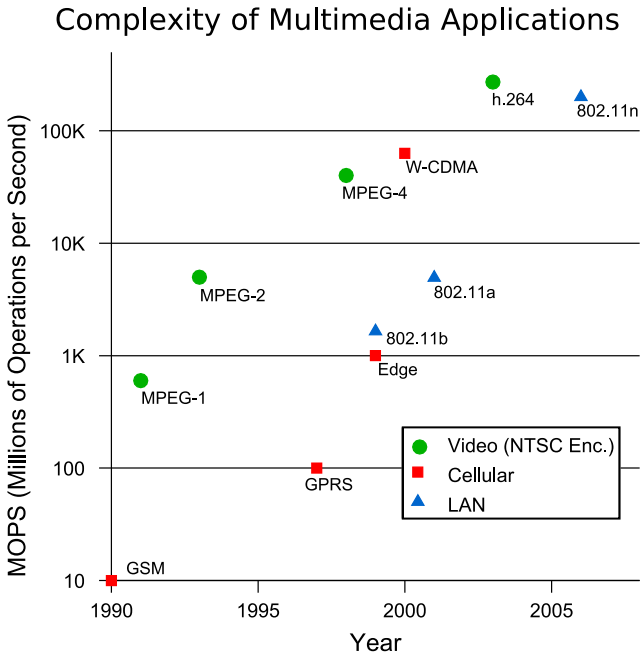

Figure 1.1: Exponentially increasing application complexity

use 65 MB of binary code. This is expected to increase to 1 GB of code by 2010.

1.1.2 Programmable Platforms

During this same time period, a variety of technical and economic reasons have made application-specific utilization of transistors (e.g. ASICs) more difficult. First, time-to-market concerns and the need for flexibility may preclude the use of application-specific hardware. Second, the non-recurring engineering (NRE) costs associated with hardware fabrication often necessitate high product volumes to recoup the initial investment. NRE costs have continued to increase in recent years, with the mask costs alone for a single chip surpassing $1 million[69]. Total design costs for designs implemented in 130 nm, 90 nm, and 65 nm technology are shown in Figure 1.2 [68]. Coupled with data indicating that the

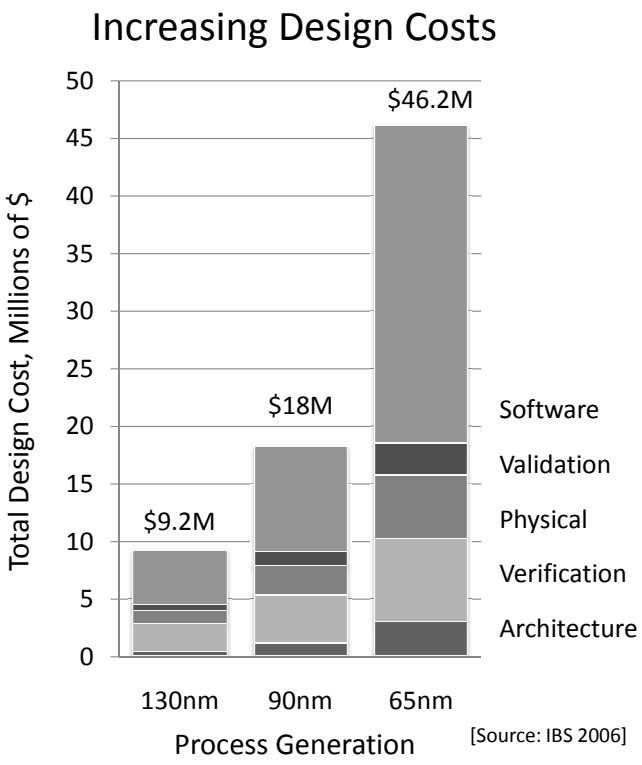

Figure 1.2: Design costs rapidly increasing for smaller process generations

average selling price for ASICs is under $10 [117] and not increasing, it becomes clear that implementing such an application-specific hardware device requires very high volumes to justify the initial expenditure.

The main strategy to overcome these trends is to manufacture programmable devices, which can be used with multiple applications, thereby increasing volumes and justifying the NRE costs. This trend is becoming clear when the number of total design starts is tallied, as shown in Figure 1.3 [68]. During the past few years, the number of unique designs implemented worldwide has slowly started to decrease.

Programmable hardware devices can be termed as architectural platforms, since they can support a wide variety of applications, but usually within a specific domain [52].

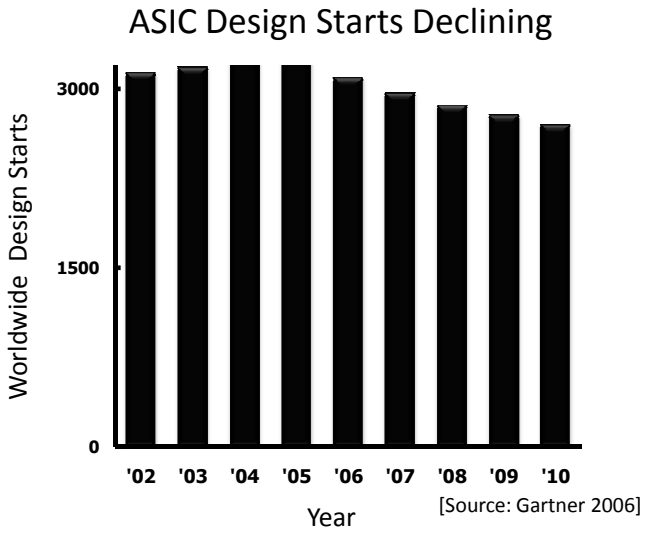

Figure 1.3: Number of worldwide design starts declining

Design cost and effort now shifts to creating the software that is deployed on these platforms for a particular application [59]. Enabling increased design productivity for such platforms is therefore important.

1.1.3 Heterogeneous Parallel Platforms

Heterogeneous multiprocessor architectural platforms are gaining prevalence for embedded systems. Greater parallelism is needed primarily due to increased energy efficiency requirements. Previously, greater performance for processing elements was achieved by increasing the clock frequency, exploiting instruction-level parallelism, introducing deeper pipelines, and carrying out speculative execution. For these techniques, the amount of performance gain achieved for a given increase in energy consumption has become smaller in recent years [93]. 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 specified application. Greater parallelism with relatively simple PEs running at lower frequencies provides increased computational capabilities with higher energy efficiency.

For embedded platforms, applications in a particular domain usually have a set of commonly used "kernels" or core computations that are carried out relatively frequently. In order to obtain high performance, PEs in the platform must have support for these kernels. Typically, parallel embedded platforms therefore contain a variety of PEs, customized for the common types of kernels found in applications for a domain [81].

1.2 Design Challenges

The design challenges for heterogeneous parallel embedded systems become apparent by examining current design practice. The current flow for heterogeneous parallel platforms is typically an ad-hoc adaptation of the uniprocessor design flow. The uniprocessor flow typically involves manual implementation of code for the processor in a low-level language such as assembly code or C followed by extensive simulation to debug and meet performance constraints [122].

The ad-hoc adaptation of this strategy for heterogeneous parallel platforms usually adds an initial partitioning step for the application that roughly divides functionality and assigns it to individual PEs. The manual implementation of code is carried out as before for each PE, and then simulation/testing is carried out on the entire system. However,

due to the many possible interleaving patterns that may occur between the different PEs, simulation/testing is much less effective at finding problems in the implementation. If performance requirements are not met, then the code may have to be repartitioned, leading to another long design iteration. The main problems with this design practice are threefold: strong binding between the application and the architectural platform, lack of application verification, and inability to explore the design space efficiently.

The first problem occurs since design capture takes place only after rough partitioning has been carried out on the functionality. If portions of the application have to be migrated to other PEs, the existing design cannot be used. Similarly, a different architectural platform will require complete re-implementation. Therefore, early binding of this type between the application and the architectural platform limits the amount of design reuse [107].

The lack of application verification is partly due to the linkage between application and architecture but also because of the lack of a formal specification. Unless structured techniques are utilized, interactions between concurrently executing pieces of code cannot be analyzed [73]. In the implementation, communication between PEs can cause serious problems in terms of synchronization issues, deadlock, and race conditions [48]. The lack of application verification means that verification must be delayed until the implementation is complete, at which point it may not be known whether the error originated in the application specification or the implementation of this specification.

The inability to effectively explore the design space stems from the long design iterations and the reliance on simulation/testing to verify correctness of the system. Under

these circumstances, each new point in the design space requires a significant amount of time and effort to produce. With this design practice, designer intuition is very important for producing a valid design. Since designer intuition cannot be relied upon, especially as new architectural platforms emerge, the lack of automated design space exploration techniques strongly limits design productivity [46].

Heterogeneity increases the programming burden as well. Code which is deployed on a PE cannot easily be migrated to another PE, especially at runtime. For embedded systems, this heterogeneous parallelism implies that parallel programming techniques from other communities may not be applicable. For instance, the supercomputing community generally deals with single-program-multiple-data types of programs on homogeneous parallel architectures [86]. For embedded systems, the platforms are more irregular, and the focus is not on average-case performance.

1.3 Overview

The work described in this dissertation tackles these design challenges. The four main contributions of this work are: a design flow that addresses modeling and mapping challenges, exploration of this design flow in the multimedia domain, customization of the flow for the automotive domain, and the distillation of requirements for a next-generation design framework based on the flow.

1.3.1 Approach

The approach taken in this work is based on the platform-based design (PBD) methodology [60]. PBD advocates a meet-in-the-middle design process, with the functional portion of the design (what the design does) constituting the top-down part and the architectural portion of the design (how this is carried out and at what cost) constituting the bottom-up part. An explicit mapping step binds these two parts together to realize the system model.

To meet the challenges of deploying applications on heterogeneous multiprocessor embedded platforms, a synthesis – or automated mapping – approach is needed. Not only does such an approach decrease design time, but it also enables a correct-by-construction approach which reduces verification effort.

Structured modeling techniques are necessary to facilitate tractable synthesis. Our flow [127] within PBD is therefore focused on structured modeling between functionality and architecture. The 4-stage flow involves both modeling and mapping. Modeling is based on the concept of common modeling domains (CMDs). CMDs are defined in terms of services. Mapping can be carried out using a variety of exact and heuristic techniques. Initially, we believe that mathematical programming [89] approaches are useful to consider since they are extensible, provide bounds on solution quality, and leverage advances in general-purpose solvers. The design flow is validated by applying it to studies from the multimedia and automotive domains. Functionality, architecture, and mapping for these systems are modeled within the METROPOLIS [8] framework.

1.3.2 Multimedia Domain

Multimedia systems deal with computation carried out on streams of data. The model of computation (MoC) [54] most often considered for multimedia systems is usually a specialization of Kahn Process Networks, where actors consume data from input streams, carry out computation, and produce data on output streams. The two case studies in this domain explore both modeling and mapping stages of the flow.

The first case study consists of deploying a JPEG encoder application onto the Intel MXP5800 architectural platform [27]. The JPEG encoder is a multimedia application whose building blocks are used in many image and video processing algorithms. The MXP5800 is an imaging processor which is highly parallel and heterogeneous. The focus for this case study is choosing the appropriate dataflow MoC and abstraction level for modeling and then carrying out manual mapping. The case study demonstrates that the appropriate choice for MoC and abstraction level can allow designers to fully exploit the capabilities of such platforms while also enabling future automation.

The second case study [29] investigates the automated [24] mapping of a motion-JPEG application onto a heterogeneous soft-core multiprocessor FPGA platform. Motion-JPEG is an extension of JPEG for video and is used in consumer electronics and video editing systems. The architectural platform consists of soft-core *u*Blaze PEs and custom hardware connected with point-to-point FIFOs on the Xilinx Virtex II FPGA platform. The automated mapping is carried out with a Mixed Integer Linear Programming approach that is shown to be both efficient and extensible.

1.3.3 Automotive Domain

Automotive applications are typically control-related and have strict timing requirements. Due to the intrinsic physical distribution of the sensors/actuators, these applications are deployed on multiple electronic control units (ECUs) connected with standardized buses. Again, both modeling and mapping aspects of the design flow are addressed in the two case studies.

The first study in this domain involves reconciling the MoC used for the functional model with the MoC exhibited by the architectural platform [125]. We demonstrate that current industry practice can lead to problems such as message loss and priority inversion due to limited buffer sizes in the bus controllers. These problems are not evident in the original functional model, making correct-by-construction deployment difficult. To remedy these problems, we modify the functional MoC such that these problems can be diagnosed prior to mapping.

The second study develops automated mapping techniques that meet worst-case end-to-end latency requirements [28]. The three stages of the automated mapping problem for this domain are allocation, priority assignment, and period assignment. For the period assignment stage in mapping, and iterative approach to assign task and message periods has been developed and applied to two case studies.

1.3.4 Design Framework

The uniqueness of the design flow presented in this dissertation is that it provides a unified way to view mapping problems from multiple domains. This allows the non-domain-

specific aspects of the design flow to be combined into a common design framework. For instance, many manual design activities such design import, simulation, and debugging [18] can be handled in a common way. Based on this flow and the lessons learned from case studies, we have re-evaluated the goals from METROPOLIS and started development on the METRO II framework [25]. METRO II aims to support heterogeneous IP import and provide a more structured means for specifying performance annotations and mapping.

Chapter 2

Background

The design flow in this work is based on Platform-based design and utilizes the Metropolis design framework to carry out modeling for the case studies in Chapters 4 and 5. Metropolis is also the basis for the Metro II framework described in Chapter 6. In this section, Platform-based design and Metropolis will be described in more detail, along with an overview of other design frameworks and approaches that have similar goals.

2.1 Platform-based Design

To deal with constantly increasing complexity, safety and security requirements, and time-to-market pressure, embedded system designers are turning to more rigorous design methods. These favor the adoption of higher levels of abstraction in system specification, correct-by-construction deployment, and reusability. The Platform-based design [59] paradigm has been proposed cope with these difficulties.

In this paradigm, a platform is designed with sufficient flexibility to support the

implementation of an entire set of products. The product design problem then involves configuring the platform, and deciding which parts of the product's functionality are to be implemented by which platform resources. Typically, designers evaluate several configurations before selecting one that meets design goals. This process is known as *design space exploration*. It requires building a series of models, one for each combination of configurations to be evaluated. Developing these models is traditionally time consuming and error prone. Therefore, it is natural to re-use them as much as possible. However, it is often hard to do so because modifying a configuration or a part of the description of a model usually requires extensive changes to models in the rest of the design.

A solution to the re-use problem is to orthogonalize concerns and keep various aspects of a design separate. There are several concerns in embedded system design that can be orthogonalized. The three main concerns are the following:

- *Functionality versus Architecture*: Functionality represents the application that the designers want the system to carry out, while the architecture represents a configuration of resources that can implement this functionality. 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. The architecture determines the performance in terms of the quantities of interest (e.g. energy, time) while the functionality determines which services are used and in what way. By allowing an architectural model to be reconfigurable or instantiated in different ways, we can easily represent a family of parameterizable architectural platforms. 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 separation between functionality and architecture is also captured in the Y-chart approach [62].

- *Behavior versus Cost*: 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.

- *Computation vs. Coordination*: The behavior of a design is often specified as a set of

concurrent processes, where each object executes a sequential program and communicates with other processes. A process may share resources with other processes, this often requires coordination among multiple processes. Coordination can be described separately from the sequential programs for the individual processes. Computational activities are usually highly design-specific while coordination schemes are usually standardized.

Specification of computation and coordination may be carried out in different ways. It is often convenient to model coordination using declarative constraints, rather than imperative programs. For instance, it is simpler to declare that two actions should be mutually exclusive, rather than write a program for a protocol that realizes the exclusion. Such declarative statements are very useful during the initial stages of the design flow, when conciseness of specification is more important than performance estimation.

2.2 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 system-level design (ESL) [3]. The METROPOLIS framework consists of a specification language – the Metamodel [113] – as well as a compiler and a set of plugins that can interface with external tools. Figure 2.1 shows the architecture of the METROPOLIS design framework. Functional, architectural, and mapping specification for the system is first captured within

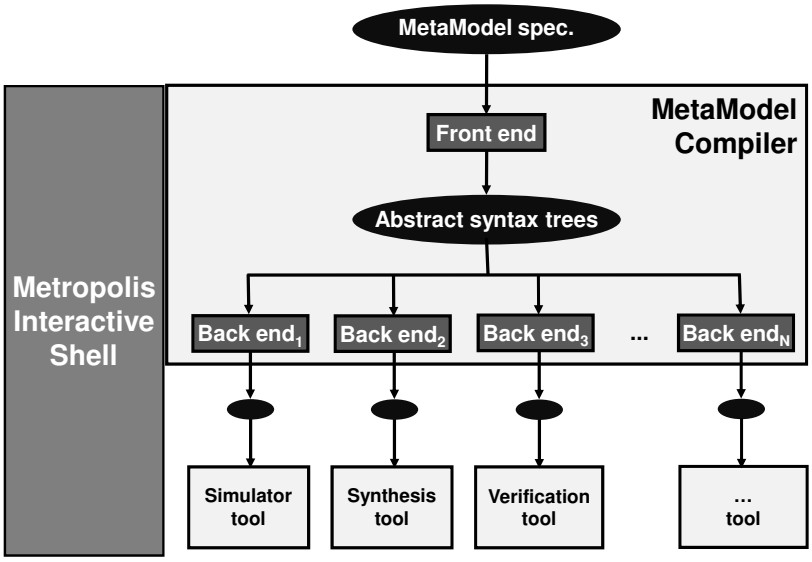

Figure 2.1: METROPOLIS Design Framework

the Metamodel language. The front end allows this specification to be compiled into a set of abstract syntax trees. Various backends, including simulation, synthesis, and verification tools can then access this information. The backends are invoked by the user using the interactive shell.

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.

2.2.1 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. Design reuse follows from the platform-based design methodology discussion in Section 2.1. In the following, we will examine the other goals in further detail. An overview of the design methodology for METROPOLIS is given in [99].

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 [113] [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. This formal underpinning can be used for different types of analysis, including refinement verification [33].

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) [110] and the Logic of Constraints (LOC) [4]. 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.

2.2.2 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: modeling the behavior – including the computation and communication – in the functional and architectural models, annotating costs to operations in the architectural model, and 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 [26].

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 medium must be used to manage the interaction between multiple processes. When an object wishes to utilize the services provided in another medium, it must communicate with that medium by using ports which have associated interfaces. The concept of ports and interfaces is widely used for design specification in frameworks like SystemC [47]. 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.

Using Quantity Managers to annotate cost

Quantity managers in METROPOLIS are similar to aspects in aspect-oriented programming [61] 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 2.2. 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 some thread

executing the beginning and end of the operation respectively. 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. *gt* refers to the global time quantity manager.

```
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 2.2: Annotating costs for operations with quantity managers

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. 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 2.3. 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 2.3: Synchronizing events to realize mapping

2.3 Related Design Frameworks

In this section, an overview of related system-level design frameworks is provided. The focus is placed on the support provided for automated mapping [46] capabilities.

The Spade [78] and Sesame [118] approaches within the Artemis [97] project focus on synthesizing Kahn Process Networks (KPN) [57] specifications in hardware/software. The most relevant optimization approach from their work utilizes an evolutionary algo-

rithm 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 [112] 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 [63] library, and does not consider deadlock.

CAKE [32] (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 each processor.

ForSyDe [103] 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 transformations do not change the behavior of the model, while design decision transformations are unrestricted. The focus of ForSyDe is on the verification

aspects of design, and they do not focus on automation.

MESH [96] 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 [85] 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 [56].

Polis [5] 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 [6]. Architectural components can only be chosen from a set of predefined components, limiting the expressiveness. METROPOLIS is the direct successor of the Polis effort.

Finally, Ptolemy II [79] 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, several hardware targets have been explored in the context of the precursor Ptolemy framework [88].

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. Thus, Metropolis is a singular framework which will allow the implementation of mapping approaches for systems from different embedded systems domains.

Chapter 3

Approach

The approach taken in this work is to customize the platform-based design methodology to support our objective: enabling automated mapping for parallel heterogeneous embedded systems. Platform-based design advocates an initial separation between functionality and architecture, and a distinct mapping step to realize the system. The main problems tackled in this design flow are twofold: choosing how to model the functionality and architecture and then mapping the two together.

The problem of mapping between arbitrary functional and architectural models can be solved with two broad strategies. The first strategy attempts to bridge the gap between dissimilar models. The algorithms and techniques applied by this strategy are usually specific to the models of computation and abstraction levels employed by the models being mapped. This strategy has the capability to produce very good results for specialized problems, but there is little applicability to a broad class of systems. As a result, when either the functional or the architectural model changes, a new approach may have to be

developed and reuse is difficult.

The second strategy initially transforms the models into a "common modeling domain" (CMD) where both the semantics and abstraction levels are compatible. The automated mapping approaches are then tailored to the common modeling domain. The advantage of this two-step approach is flexibility. Many models can be transformed into a common modeling domain and leverage the automating mapping flow.

The latter approach is adopted in this work. The first step of choosing a CMD is described in Section 3.1, the second step of automated mapping is developed in Section 3.2. The development of the design flow in the remainder of the dissertation is outlined in Section 3.3.

3.1 Common Modeling Domains

Common modeling domains ensure that the semantics and granularities of the services being mapped match. First, concerns specific to parallel embedded systems modeling will be enumerated in Section 3.1.1. Next, these concerns are integrated within the functionality-architecture separation within PBD by using the concept of a common modeling domain. Mapping is described in terms of used and provided services in Section 3.1.2 and illustrated with a small example. A discussion of the modeling tradeoffs for both functionality and architecture is given in Section 3.1.3.

3.1.1 Parallel Systems Modeling

Parallel embedded systems modeling can be carried out at different levels, depending on which aspects are explicitly captured. The modeling can be broken up into five stages, loosely based on the breakdown suggested in [111]. The stages are categorized by the modeling aspects which are made either explicit or implicit. Four aspects are considered in this categorization.

1. The first aspect is whether or not *parallelism* is made explicit in the model. If parallelism is implicit, then the model has no concept of parallel execution, and parallelism must be extracted automatically to enable it to run on multiple resources. With explicit parallelism, processes that run concurrently have already been identified.

2. *Allocation* determines if computational resources are bound to the tasks/processes in the model. If allocation is explicit, the designer must specify which processing element each process executes on.

3. *Communication* fixes the realization of messages that are sent between processes. This involves determining the encoding of each message as well as the communication resources in the platform to which they are mapped.

4. *Performance* denotes whether the relevant cost metrics can be directly calculated or not. Models with explicit performance can evaluate cost metrics analytically, whereas implicit performance metrics must be calculated via simulation. Different performance metrics are used for different systems. Predictable performance analysis is key to developing automation techniques.

Level	Parallelism	Allocation	Communication	Performance
1	Implicit	Implicit	Implicit	Implicit
2	Explicit	Implicit	Implicit	Implicit
3	Explicit	Explicit	Implicit	Implicit
4	Explicit	Explicit	Explicit	Implicit
5	Explicit	Explicit	Explicit	Explicit

Table 3.1: Modeling of Parallel Systems

In embedded systems programming, industry practice is to describe the system at level 1, in the form of untimed sequential C/C++ code. The latter stages of modeling are carried out manually. Typically, performance constraints for the system can be verified only at level 5. Since the transition from level 1 to level 5 is time-consuming and hard to verify, the design process becomes more complex.

Note that the current industrial practice is well suited for general purpose uniprocessor systems. Levels 2–4 are absent in the general purpose uniprocessor flow, and there are less stringent (perhaps only average case) performance constraints on the system as a whole. For general purpose parallel systems, levels 2–4 are present, but the constraints to verify at level 5 are still lax. This laxity is more tolerant of manual design flows. However, for parallel embedded systems, all the problems need to be addressed.

In order to remedy the problems, we need to automate the transformation of models between levels. Ideally, specification would be at level 1, and automation would enable transformation to level 5. However, this is difficult for a number of reasons. First, the transformation between levels 1 and 2 involves the automated extraction of parallelism, which is notoriously difficult. Secondly, in general, determining allocation, communication details, and analyzing performance in an automated fashion is infeasible. To tackle these

concerns, the methodology has two main restrictions. First, the transformation from level 1 to level 2 is not automated, and still left to the designer. Second, system specification at level 2 can only be carried out in a specific manner. Namely, a correspondence must exist between the services that are used by the functionality and the services provided by the architecture.

3.1.2 Services

Operationally, the functional model consists of a set of *processes* $\mathcal{P}$ that execute a sequence of *services* $\mathcal{S}_F$ provided by different *components* $\mathcal{C}$. The architectural model consists of a set of *resources* $\mathcal{R}$ which provide *services* $\mathcal{S}_A$, each of which may be *parameterized*.

Mapping assigns processes from the functionality to resources in the architecture. This is a many-to-one mapping. Whenever a process $p \in \mathcal{P}$ executes a service $s_f \in \mathcal{S}_F$, it is bound to a single resource $r \in \mathcal{R}$ invoking a service $s_a \in \mathcal{S}_A$. s_a and s_f must be identical. Verifying this requirement is beyond the scope of this work, it is assumed that the functional and architectural designers agree on the definition of services. Any parameter(s) π required for s_a are determined by the mapping based on the component $c \in \mathcal{C}$ in which the service s_f is executed. Note that service parameterization is not permitted for functional services, only for architectural ones. Also, parameters are determined statically based on the mapping, the architectural parameters cannot be determined during the execution of the functional model.

The semantics, or model of computation, depends on the set of services used by the functionality as well the sequence in which these services are used by the processes. The architecture determines the performance of the system by calculating a cost for each

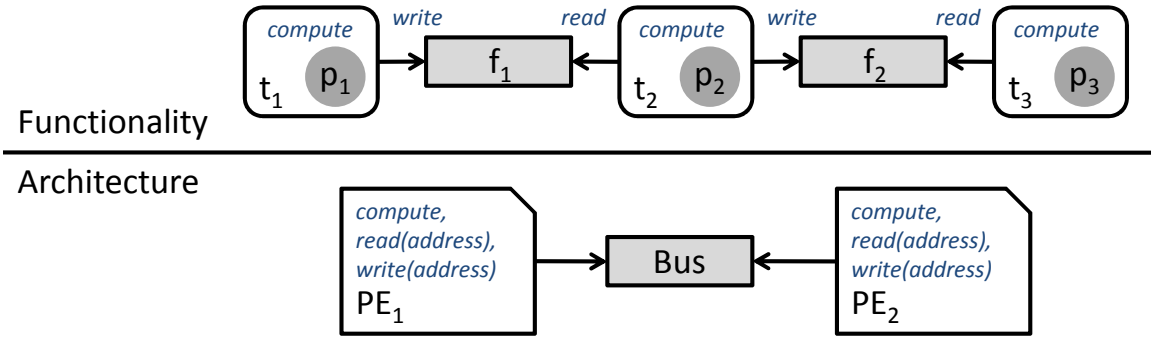

Figure 3.1: An Example of Service-based Mapping

service. Functional performance constraints or performance metrics can only be verified after mapping has been carried out, not before.

Example

A simple example, shown in Figure 3.1, illustrates this approach to mapping. The functional model consists of three processes: p_1, p_2, and p_3, as well as three tasks: t_1, t_2, and t_3, and two FIFOs: f_1 and f_2. The processes can execute up to three services each: *read*, *write*, and *compute*. *read* and *write* services are executed by the processes within the FIFO components, while the *compute* services are executed within the task components.

The architecture consists of two resources (PE_1 and PE_2) connected with a bus. Both resources provide all three services required by the processes in the functionality. The *read* and *write* services are parameterized in the architecture with an address argument. Note that the amount of data written or read is missing from these services. This is because that amount is implicit in the definition of the service. For example, if the cost of reading two items is different than twice the cost of reading a single item, a new service should be introduced to differentiate these two situations.

$< p_1,$ *compute*, $t_1 >$	$\longrightarrow$	$< PE_1,$ *compute*, $\emptyset >$
$< p_1,$ *write*, $f_1 >$	$\longrightarrow$	$< PE_1,$ *write*, 0x0$FFA >$
$< p_2,$ *compute*, $t_2 >$	$\longrightarrow$	$< PE_1,$ *compute*, $\emptyset >$
$< p_2,$ *read*, $f_1 >$	$\longrightarrow$	$< PE_1,$ *read*, 0x0$FFA >$
$< p_2,$ *write*, $f_2 >$	$\longrightarrow$	$< PE_1,$ *write*, 0x0$CC0 >$
$< p_3,$ *compute*, $t_3 >$	$\longrightarrow$	$< PE_2,$ *compute*, $\emptyset >$
$< p_3,$ *read*, $f_2 >$	$\longrightarrow$	$< PE_2,$ *read*, 0x0$FFD >$

Table 3.2: Mapping for example system

The mapping for this system is described in Table 3.2. It indicates the mapping for each used service to the associated provided service. The mapping has allocated p_1 and p_2 to PE_1, and p_3 to PE_2. The mapping has also determined the parameters for the *read* and *write* services. These parameters refer to the location in memory where the FIFO communication is mapped. Note that f_1 is realized in the local memory of PE_1, while f_2 is realized in the local memory of PE_2. Presumably, 0x0$CC0$ corresponds to the memory-mapped address of the remote memory for PE_1 while 0x0FFD is the same location in the local memory space of PE_2.

3.1.3 Tradeoffs

Functional and architectural modeling must balance the conflicting aims of accurate/efficient system performance vs. automated design space exploration. This is directly related to the granularity of services that are mapped. Service relationships can be captured in the form of a directed acyclic graph (DAG) where nodes represent services and the directed edges capture service containment. A service s_1 contains s_2 and s_3 iff s_1 can be decomposed into some sequence of invocations of s_2 and s_3. Note that since we do not deal with the definitions of the services, the construction of this graph is beyond the scope of

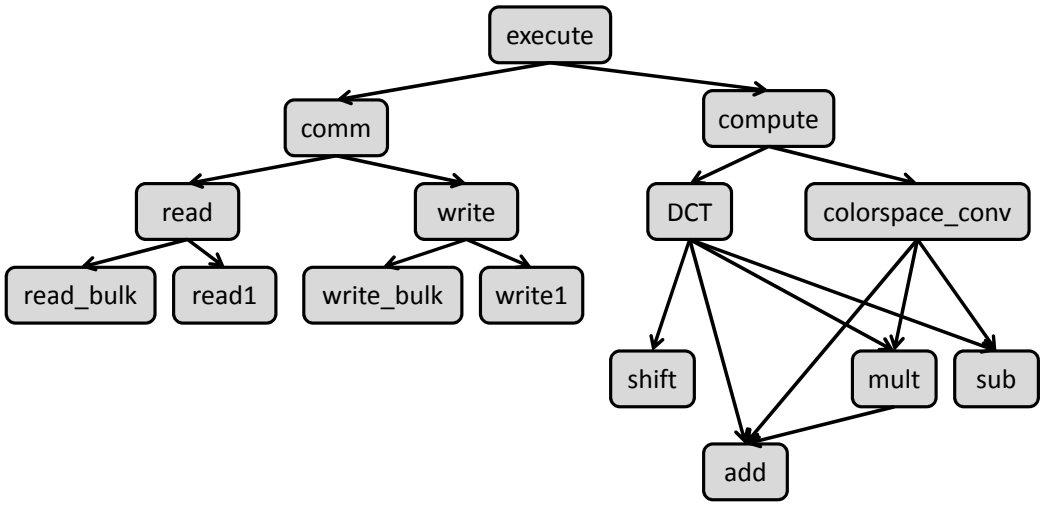

Figure 3.2: Example: Hierarchy of services

this work.

An example of a service relationship graph is given in Figure 3.2. In this example, a set of services is hierarchically arranged from most to least abstract. The more abstract services, such as *comm* and *compute*, are invoked in a much simpler pattern than the leaf services. However, the leaf services can be more accurately characterized than the higher level services.

Functional Tradeoffs

The aim of functional modeling is to capture the behavior of the application and enable both behavioral and performance verification. These aims are often contradictory. Under the definition of services given in Section 3.1.2, the behavior of the application can either be captured with a large set of focused services that are used in a complex manner or a small set of generic services that are used in a straightforward manner.

Behavioral verification relates to the way in which the atomic services are used by the processes to carry realize the behavior. Therefore, coarse granularity services are preferable; as long as the services themselves have been pre-verified. With coarse-granularity services, the usage patterns are simpler, and more likely to fall into a verifiable pattern.

Architectural Tradeoffs

The choice of the CMD also has an important impact on the architectural modeling. Architectural modeling has to be carried out in such a way that the services which define the CMD are provided by the architectural model. For instance, for a dataflow CMD, blocking read operations must be provided by the architectural model. These services are typically provided with some measure of cost, which may not be captured in the CMD. For instance, the blocking read operation may have latency and energy costs associated with it in the architectural platform. The tradeoff to be considered is that the services associated with the CMD may be fairly expensive for a given architectural platform. So, the CMD may a priori restrict the maximum achievable performance of any system that can be implemented on that architectural platform. Ideally, the architectural services should be defined at a fine granularity to enable accurate and low overhead characterization.

The other choice in architectural modeling is to expose or hide the flexibility that may be present. For instance, a blocking read operation may either be carried out using limited local memory or a larger amount of global memory with different energy and latency costs. Exposing this choice from the architectural model may require more detailed modeling and unnecessarily complicate the mapping stage.

Type	Effort	Runtime	Bounds	Portability
Deterministic Heuristics			X	X
Approximation Algorithms	X			X
Randomized Algorithms		X	X	
Mathematical Programming		X		

Figure 3.3: Automated Mapping Techniques

3.2 Mapping

Mapping is the process of associating the functional and architectural models together such that the services used by the functionality are bound to those provided by the architecture. As mentioned, mapping involves allocating functional processes to architectural resources and configuring the parameters for architectural services. The allocation of process to resources can be seen as a covering problem, where each process may be "covered" by at most one architectural resource.

Mapping must ensure that the functional performance constraints are satisfied and that the relevant metrics are optimized. For embedded systems, satisfying performance constraints typically requires worst-case analysis. Optimization requires the ability to analyze a number of points in the design space, either explicitly with simulation or implicitly with analysis. Due to the complexity of modern multiprocessor embedded systems, our approach emphasizes automated approaches to mapping.

Figure 3.3 summarizes four techniques that can be used for automated mapping: heuristics, approximation algorithms, randomized algorithms, and mathematical programming. Even though heuristics are relatively easy to create and can often provide solutions

quickly, they do not provide bounds on solution quality. More importantly, they are brittle with respect to changes in the problem assumptions. For instance, partial solutions and side constraints are typically difficult to add to most heuristics without sacrifices in effectiveness.

Approximation algorithms do provide bounds, but the analysis applied to produce these bounds is even less resilient to problem changes. Consequently, even though heuristics and approximation algorithms excel at clearly defined problems, their applicability is limited within a design flow where platform-specific constraints are needed.

Randomized algorithms, such as genetic programming and simulated annealing are flexible to changes in problem assumptions. However, they do not provide bounds on solution quality and are typically computationally expensive.

An alternative is to use mathematical programming (MP) techniques. In MP, the system is represented with parameters, decision variables, and constraints over the parameters and decision variables. An objective function is defined over the same set of variables. Generic solvers can be utilized to find the optimal solution. The complexity of finding the optimal solution depends upon the variable types as well as the form of the objective function and constraints.

These techniques are much easier to customize, since application or platform-specific constraints can be added as required. Branch-and-bound solution techniques for MPs provide lower and upper bounds on the desired cost functions at each step of the solution process. This allows us to trade off solution time and quality. The main difficulty with using MP approaches lies in finding a formulation that is sufficiently accurate to capture the behavior of the system and yet remains amenable to efficient solving.

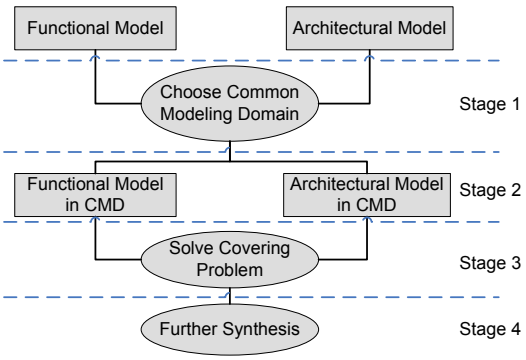

Figure 3.4: Common Modeling Domain Design flow

3.3 Development of the Design Flow

The flow is summarized in Figure 3.4. Stage 1 first finds the common modeling domain (CMD) between the functional and architectural models. Next, Stage 2 is concerned with transforming both models into the appropriate CMD. Mapping can then be formulated as a covering problem and solved in Stage 3. Further configuration of the system, i.e. assigning the architectural parameters, is carried out in Stage 4.

The focus of this research is to apply this flow to a number of heterogeneous programmable platforms from the multimedia and automotive domains. For the multimedia domain, the focus will be on stages 1, 2, and 3. For the automotive domain, the focus is on stages 1 and 4.

The reason that these tradeoffs are interesting to explore is that there is a lack of agreement on how heterogeneous parallel platforms should be programmed, evidenced by the large amount of experimentation for new MoCs for these systems. For lower abstraction levels such as hardware design, a single MoC – synchronous circuits – was sufficient for a large majority of designs, and considering such tradeoffs would have been less useful.

Also, these tradeoffs are feasible to consider due to the existence of several design frameworks that support many different MoCs. In particular, case studies from the multimedia and automotive domains using different MoCs have been modeled in the METROPOLIS [8] framework. These case studies will serve as the primary vehicles for further development of this flow.

Chapter 4

Multimedia Domain

The multimedia domain can be broadly characterized by the presence of large amounts of data streamed between different stages in the system. Data streaming applications such as audio, video, and image codecs as well as wireless communication, all characterized as multimedia systems, are predominant in many consumer electronics.

In this section, the applicability of the design flow to systems in the multimedia domain will be illustrated. The outline of this chapter is as follows: In Section 4.1, the two applications that are evaluated within this domain are described. In Section 4.2, the two architectural platforms are introduced. Based on the characteristics of the applications and the architectural platforms, relevant models of computation are explored in Section 4.3, and a suitable MoC is identified. Based on this, two case studies are described. The first case in Section 4.4 considers manual design space exploration and demonstrates that the modeling is capable of capturing the design space accurately. The second case study applies Mixed Integer Linear Programming techniques for allocation and scheduling to carry out

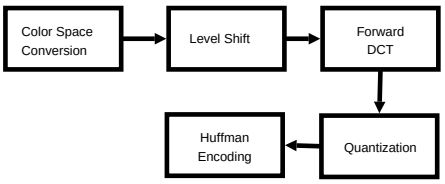

Figure 4.1: JPEG encoder block diagram

automated design space exploration.

4.1 Applications

In this work, we consider two related multimedia applications. The two applications - the JPEG image encoder and the Motion JPEG video encoder will be described in more detail within this section.

These applications are chosen since they are representative of a wide class of multimedia applications. In particular, the DCT, quantization, and Huffman blocks in both applications are utilized in several other image/video compression applications, including h.264 [121].

4.1.1 JPEG Encoder Application

The JPEG encoder [119] application is required in many types of embedded multimedia systems, from digital cameras to high-end scanners. The application compresses raw image data and emits a compressed bitstream.

A block diagram for the JPEG encoder is shown in Figure 4.1. The input for the application is an image described with raw RGB data. Each pixel is characterized with

three bytes of information: one each for the red, green, and blue components.

The first step is color space conversion, where the raw data is first converted into YCbCr format. YCbCr format consists of luminance (Y), blue chrominance (Cb), and red chrominance (Cr) information, with each of the three components being represented by a single unsigned byte. Since the human eye is more sensitive to the luminance components of an image, the chrominance components can be compressed further. This makes the YCbCr colorspace better suited for image compression than RGB.

Different variants of the JPEG algorithm can be used depending on the ratio of the chrominance to luminance information used in the compression. If all three components of the colorspace appear in the same ratio, then the mode is known as 4:4:4. If every other chrominance component is used in the horizontal dimension, the mode is known as 4:2:2. Sampling every other chrominance component in both the vertical and horizontal dimensions is referred to as 4:2:0 mode. In this work, we only utilize the baseline 4:4:4 mode.

The next step in the algorithm involves level shifting each of the component values such that they can be stored as signed bytes. The pixels are then bundled together into 8x8 blocks from top left to bottom right in scan order (row-major order). The blocks are processed independently for the following steps in the algorithm.

The subsequent step of the algorithm is a forward integer DCT transform. In this step, the 8x8 YCbCr spatial data is transformed into frequency data. Besides errors introduced through rounding, this step of the algorithm is non-lossy. Different algorithms can be used to carry out the forward DCT transform. In this work, we utilize the Chen

Wang [120] fast DCT algorithm.

The next step in the algorithm is quantization. Each component in each 8x8 block is divided by a user-supplied coefficient from a quantization table. Two separate tables are used, each with 64 coefficients: one for the luminance components and the other for the chrominance components. Quantization is the main information-losing step in the JPEG algorithm. Larger coefficients lead to lower image quality and higher compression. Standard quantization tables are provided with the JPEG standard, and they are used here.

After the division has taken place, the next step is to rearrange the component values within each 8x8 block from scan order into zig-zag order. This ordering tends to group the higher frequency components together, preferably leading to long sequences of zeros.

The first part of the Huffman encoding step is run-length compression which takes long strings of zeros and represents them in a concise intermediate form. The second stage is the actual Huffman table lookup, which translates the intermediate form into compact bit sequences. Like the quantization tables, the Huffman tables are statically specified by the user. Huffman encoding transforms the bytestream into a bitstream.

The final JPEG image file consists of header data along with the compressed bitstream. The header data includes the quantization and Huffman tables for both the chrominance and luminance components. The JPEG file interchange format (JFIF) is the standard for representing JPEG-encoded images.

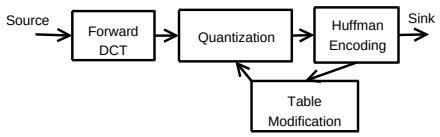

Figure 4.2: Motion JPEG Encoder

4.1.2 Motion JPEG Application

The Motion JPEG encoder application carries out video encoding without inter-frame compression. Motion JPEG encoding is commonly implemented in consumer and security cameras as well as high-resolution video editing.

The Motion JPEG encoder application is quite similar to the JPEG encoder, except that quantization and/or Huffman tables are changed adaptively between frames. The table modifications are carried out based on discrepancies between the actual and desired compression rates for the encoded video stream. Unlike the JPEG encoder, there is no standard file format for a Motion JPEG encoded stream. A block diagram of this application is shown in Figure 4.2.

In this work, only the quantization tables are modified between frames, based on a linear scaling of the quantization coefficients provided in the standard tables. The modification is based on the size of the previously compressed frame as compared to the desired size.

4.2 Architectural Platforms

Multimedia applications are deployed on a variety of platforms, and the design flow used depends strongly on the characteristics of these platforms. In this work, the focus

is on heterogeneous multicore architectures, and both of the platforms presented here fall into this category.

4.2.1 The Intel MXP5800 Platform

The Intel MXP5800 digital media processor [2] is a heterogeneous, programmable processor optimized for document and image processing applications. It implements a data-driven, shared register architecture with a 16-bit data path and a core frequency of 266 MHz. The MXP5800 provides specialized hardware to accelerate frequently repeated image processing functions along with a large number of customized programmable processing elements.

The basic MXP5800 architecture, shown in Figure 4.3, consists of eight Image Signal Processors at the top level (ISP_1 to ISP_8) connected with programmable Quad Ports (8 per ISP). Quad Ports are used for data I/O and are each essentially FIFOs of size two. They provide blocking read and write semantics which ensure that all communication is data driven. Quad Ports are statically configured by the system developer during mapping according to the data flow topology of the application. In addition to Quad Port connections, all of the ISPs are connected to DMA units and some are connected to other expansion ports. Each ISP consists of five programmable Processing Elements (PEs), instruction/data memory, 16 16-bit General Purpose Registers (GPRs) for passing data between PEs, and up to two hardware accelerators for key image processing functions. Two of the PEs are used for Data I/O: The Input PE (IPE) which is used to read data from the Quad Ports, and the Output PE (OPE) for writing data to a Quad Port. Of the remaining 3 PEs per ISP, one is for general purpose use (GPE) while two PEs have Multiply/Accumulate (MACPE)

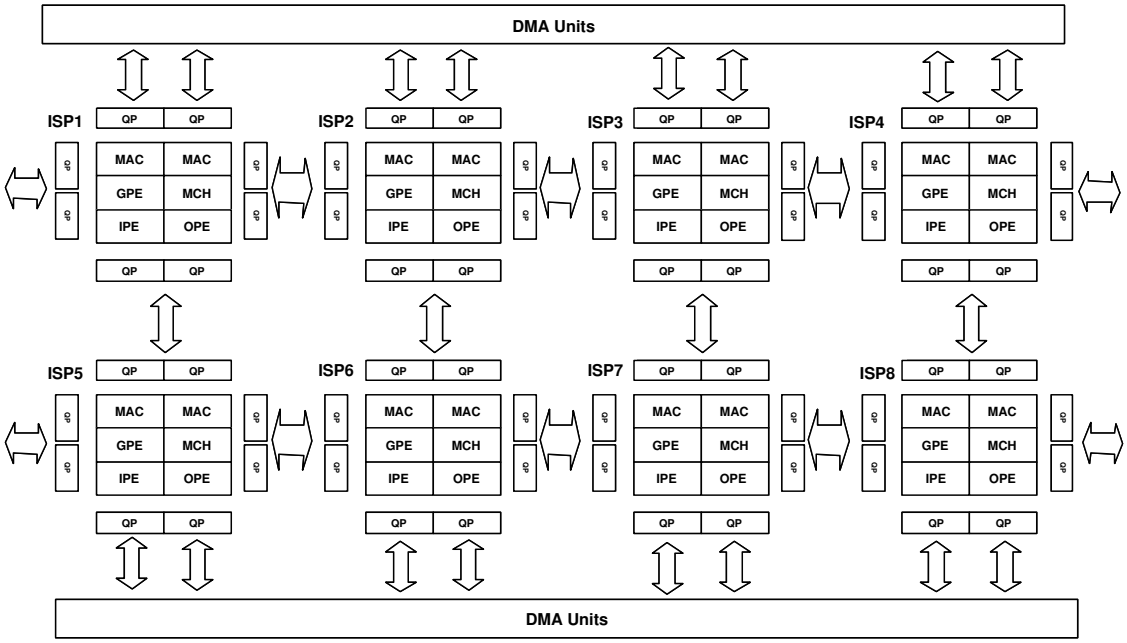

Figure 4.3: Block Diagram of MXP5800

capabilities in addition to the general purpose functionality.

Each general purpose register in an ISP has a set of 8 data valid (DV) flags - one per PE. If all the DV flags for a register are cleared, a PE may atomically write data to the register and set the DV flags for all of the destination PEs. Each of the destination PEs can clear its own flag when it reads the data. In this way, the global registers serve as single-place blocking-read, blocking-write buffers for possibly multiple writers and readers.

A Memory Control Handler (MCH) provides the interface to the SRAM data memory block within each ISP. The MCH has support for a number of different read/write modes which support variable offsets and stride lengths. Access to the MCH is provided using global registers, just as for the other PEs.

Each ISP is optimized for a particular function and the hardware accelerators in

the ISP reflect that optimization. ISP_2, ISP_5 and ISP_6 each have variable-tap and single-tap triangular filters. ISP_4 and ISP_8 contain Huffman encode/decode engines that are useful for many compression/decompression applications. ISP_3 contains G4 encode/decode blocks. ISP_7 contains 8x8 DCT/iDCT hardware. Finally, ISP_1 has an additional 16 KB of data SRAM instead of a hardware accelerator.

The major characteristic of this architecture platform is the extremely high degree of parallelism and heterogeneity. Harnessing the diverse capabilities of the PEs to realize high application performance is the main design challenge.

4.2.2 Xilinx Virtex II Pro Platform

The second architectural platform used in this work is the Xilinx [20] Virtex II FPGA. It is a platform FPGA that may be customized at the lookup table (LUT) level to implement a variety of functionality. For the purposes of this work, we will only utilize a limited subset of the flexibility offered by the platform. Specifically, hard and soft processor cores, bus and FIFO interfaces between cores, and one type of customized IP block is utilized in this work. This case study uses a 2VP30 part on Xilinx XUP board with a maximum frequency of 100 MHz.

The *u*Blaze [21] is 32-bit RISC Harvard processor which can be instantiated on the FPGA fabric. Caches and hardware support for operations such as shifting, multiplication, and division can be enabled or disabled based on designer choice. The *u*Blaze can interface with three types of interconnect. The Local Memory Bus (LMB) is used for fast access to data and instruction memory. The Fast Simplex Link (FSL) is used to interface with other *u*Blaze processors or hardware accelerators instantiated on the fabric. Finally, the On-chip

Peripheral Bus (OPB) is used to interface with other peripherals, such as timers, network controllers, and UARTs.

FSL [22] FIFO links are used in this work to carry out data transfers between multiple *u*Blaze cores, and between *u*Blazes and synthesized hardware accelerators. FSL depth can range from 1 to 8,192 entries, each of which may be 4 bytes in width. Reads and writes to the FSL FIFOs from the *u*Blaze take only a single cycle. Both blocking and non-blocking read/write access to FSLs is provided.

Hardware accelerators can be directly synthesized onto the fabric. In this work, we make use of a DCT-specific processing element with FSL interfaces created using the XPilot [17] synthesis system.

4.3 Choosing the Model of Computation

In order to apply the design flow, the first step is to choose a common modeling domain. In this section, one component of that decision is described: choosing a model of computation. The granularity of the computation services that comprise the CMD is decided later, in Sections 4.4 and 4.5.

4.3.1 Prior Work: Models of Computation

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 MoCs will be evaluated. All of the MoCs are specializations of Process Networks, where concurrently executing processes communicate with each other

using explicit messages.

Kahn Process Networks

Kahn Process Networks [57] is a well-studied MoC where concurrent processes communicate with each other through one-way point-to-point 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 [71], 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 [30].

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 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 [57]. 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.

The 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 [58]. 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 [41]. 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 [95]. 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

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 [72]. This MoC in general suffers from the same undecidability as Kahn Process Networks [15]. However, certain variants are more suitable for analysis.

Homogeneous and static dataflow [74] are two such examples. In homogeneous dataflow, each process consumes and produces the same number of data tokens on each firing, for all channels. In static dataflow, the number of tokens produced and consumed must be constant for each firing on all channels, but can vary between channels. Cyclo-

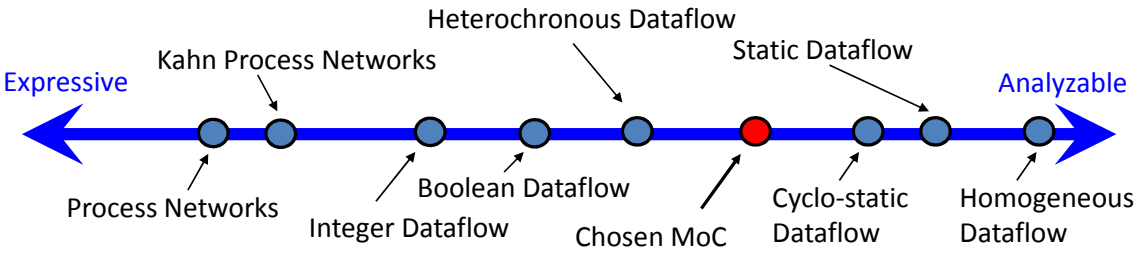

Figure 4.4: Classification of MoCs

static dataflow [11] is a slight generalization that permits the firing characteristics of each process to vary in a cycle. The main advantage of a cyclo-static dataflow model is reduced buffer size requirements. Heterochronous dataflow [43] permits firing characteristics to vary according to the current state in a finite state machine. Boolean dataflow [15] and integer dataflow [16] allow special actors where the number of tokens produced and consumed are based on Boolean and integer-valued control tokens respectively.

4.3.2 Chosen model of computation

A rough classification of these MoCs along an axis of expressiveness vs. analyzability is shown in Figure 4.4. The approximate location of the MoC that we have chosen is also shown in the diagram. Like other dataflow models, processes in the chosen MoC consume and produce tokens according to firing rules. Multiple firing rules can be specified for each process. Each process cycles between its firing rules in a fixed pattern. Therefore, this MoC is most similar to cyclo-static dataflow, but contains two extensions that allow for more concise specification.

First, only one writer is permitted per channel, but multiple reader processes are allowed. For all channels, each reader process can read each data token exactly once. Tokens

are removed from the FIFO only after all reader processes have read them once. Note that this extension allows for more succint specification, but does not change the expressiveness of the model.

Second, we allow limited forms of data-dependent communication. If a data-dependent number of tokens is to be exchanged on a channel, the sender is required to first indicate how many such tokens will be sent in a "header" token. In this way, the property of effectiveness [41] is guaranteed.

To enable support for executing multiple processes on a single processing element, this MoC has support for cooperative multitasking. Specifically, a process may only be suspended between firings. Scheduling, buffer sizing, and mapping are decidable problems for this MoC. Processes may be scheduled statically, allowing for lower overhead implementation.

An example of a process within this MoC is shown in Figure 4.5. The process reads from an input channel "input1" and writes to an output channel "output1". A variable number of tokens are first read from the input channel. Computation is then carried out on the input data. Finally, all of the processed tokens are then written to the output channel. This sequence of actions occurs within an infinite loop.

Like many specialized dataflow models, our dataflow model induces stronger constraints on the application as opposed to the architecture. In fact, many dataflow models can be supported by multiprocessor architectures that allow efficient blocking read and blocking write operations.

```
void main()
{
  int n;
  int data[5];

  while(1)
  {
    read(input1, n);
    assert(n >= 0 && n < 5);

    for(int i = 0; i < n; i++)
       read(input1, data[i]);

    // computation

    write(output1, n);

    for(int i = 0; i < n; i++)
       write(output1, data[i]);
  }
}
```

Figure 4.5: An example process within the chosen MoC

4.4 Manual Design Space Exploration: JPEG on MXP5800

Having chosen the appropriate MoC for the applications and architectures, they can be modeled at different abstraction levels and then mapped. In this case study, we deploy the JPEG encoder application onto the Intel MXP5800 architecture and demonstrate that the chosen MoC is able to accurately represent the system at a specific abstraction level. After the MoC is shown to be suitable for these types of systems we can consider automating the mapping in Section 4.5.

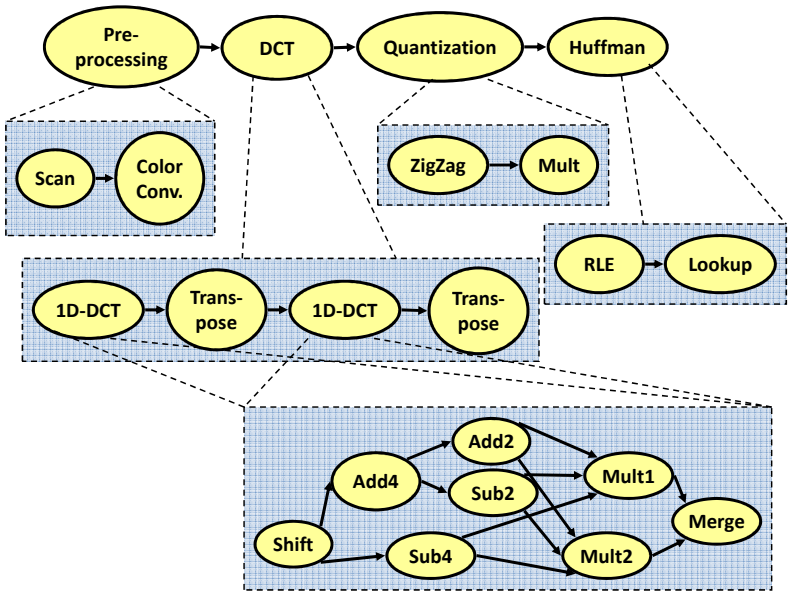

Figure 4.6: METROPOLIS model of JPEG encoder

4.4.1 JPEG Application Modeling

Starting from both a sequential C++ implementation [1] and the concurrent assembly language implementation provided in the Intel MXP5800 development kit, we assembled an architecture-independent model of the JPEG encoder in METROPOLIS expressed in our dataflow model. The model carries out a full implementation of the 4:4:4 JPEG encoder baseline algorithm and is described hierarchically at multiple abstraction levels. A total of 20 FIFO channels and 18 separate processes are used in the finest granularity representation of the application model. An overview of the METROPOLIS model is given in Figure 4.6. Characteristics of the original C++, assembly, and Metamodel designs are provided in Table 4.1. Note that the IJG model implements a superset of the functionality implemented by the METROPOLIS and ASM models.

To describe the application model in further detail, we will concentrate on the

Implementation	Language	Concurrency	Lines of Code
IJG	C++	Sequential	18,000
MXP5800 Library	ASM	Concurrent	915
METROPOLIS	MMM	Concurrent	2,695

Table 4.1: JPEG encoder models

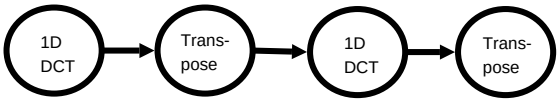

Figure 4.7: 2D-DCT block diagram

breakdown of the discrete cosine transform step in the algorithm. At the top level, the required two-dimensional DCT can be broken down into four basic steps: two one-dimensional DCT operations, each followed by a transpose operation. This is shown in Figure 4.7. Each of the 1D-DCT block reads in 8 spatial data values (corresponding to either a row or a column) and outputs 8 frequency values. The algorithm used to carry out the 1D-DCT is based on the implementation given by Chen-Wang [120].

Each one dimensional DCT operation is broken down into concurrent steps, as shown in Figure 4.8. The diagram shows that all the channels require multiple readers. This observation, along with the fact that the MXP5800 architecture supports multiple readers through data valid bits, is the reason why our MoC explicitly supports this type of communication, as described in Section 4.3.2.

4.4.2 Architecture Modeling

The MXP5800 architecture platform can be modeled in METROPOLIS by using processes, media, and quantity managers from the METROPOLIS Metamodel. A single ISP

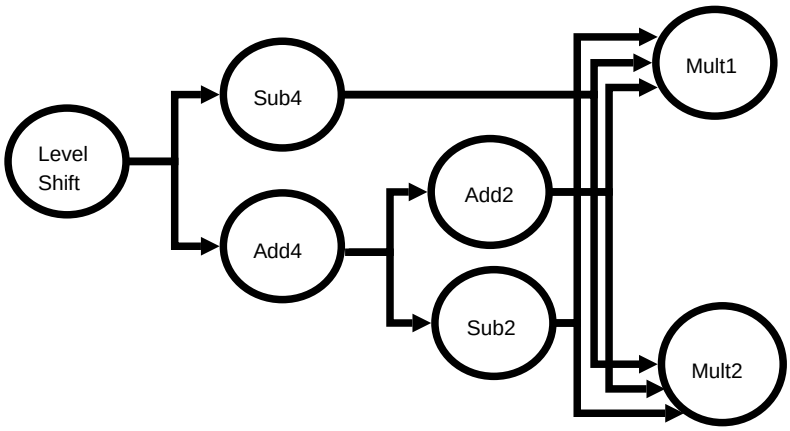

Figure 4.8: Dataflow model for 1D-DCT

is modeled as shown in Figure 4.9. The rectangles in the diagram represent tasks, the ovals represent media, while the diamonds are the quantity managers. The ISP contains the Huffman hardware accelerator, and is sufficient for implementing the JPEG encoder application. If we want to use the DCT hardware accelerator, another ISP is needed. In this case, the two ISPs will be connected through Quad Ports. Modeling various ISPs with different hardware accelerators is very similar, we use the diagram in Figure 4.9 as an example.

Each PE is modeled as a medium, which supports multiple tasks running on it. Each task is modeled as a process. These processes represent the possible functionality executed on the PE. After mapping, the behavior of each task will be restricted by the corresponding process from the application; there is a one-to-one correspondence between architecture processes and application processes.

The scheduling of multiple tasks on a single PE is carried out by the quantity manager connected to this PE. The quantity managers support static scheduling, which is

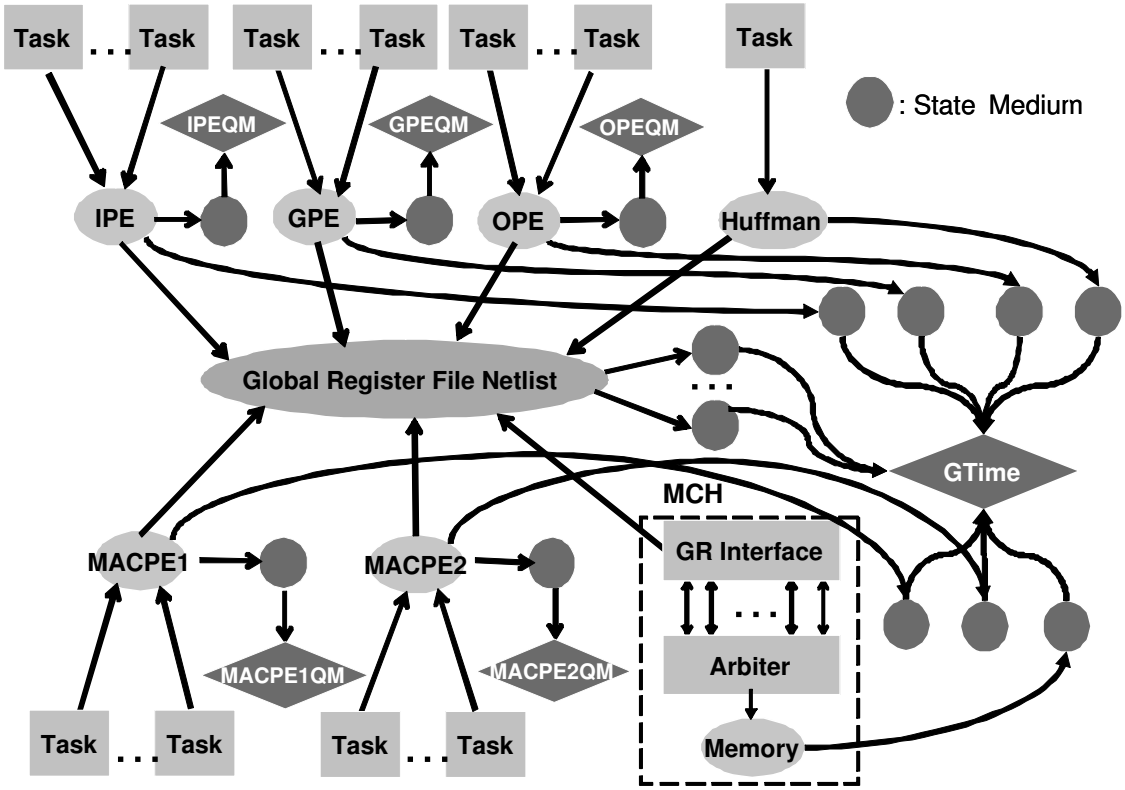

Figure 4.9: MXP5800 ISP Modeling in Metropolis

required by the MoC.

The communication between programming elements occurs through either global registers or local SRAM. The global register file is modeled as a netlist that contains 16 global registers. Each global register is modeled as a medium which implements blocking write and allows multiple simultaneous reads. These global registers can be accessed by all PEs, hardware accelerators, and the MCH. To reduce the modeling complexity, a single interface medium in the netlist represents the entire global register file and is used to communicate with the PEs, accelerators, and MCH.

The SRAM is controlled by a memory command handler(MCH). The MCH contains a global register interface (GR interface), arbiter and memory. The GR interface is used to communicate with the global register file and is modeled as a process that waits for the appropriate data valid bits to be set in the global register file. The arbiter obtains memory access requests from the GR interface through multiple FIFO channels, then uses a round-robin access scheme to select one of them to access the local memory, which is modeled as a medium.

To model running time, a global time quantity manager is used. All PEs and global registers and the local memory are connected to it. Both computation and communication costs can be modeled by sending requests to this global time quantity manager and obtaining time annotations.

4.4.3 Design Space Exploration and Results

Given the application and architectural models in Metropolis as described in Sections 4.4.1 and 4.4.2 respectively, the design space can be explored by attempting differ-

ent mappings between the application model and the architectural model. Each mapping scenario is specified in METROPOLIS with two types of information. The first is a specific set of synchronization constraints between the events in both models corresponding to the services that constitute the MoC. Along with these events – which represent the read, write, and execution services defined in our MoC – the parameters such as register location or local memory address can also be configured. The second is the set of schedules for the PEs that determine the execution order between the tasks. Both of these are relatively compact, meaning that new mapping scenarios can be created quickly and without modifying either the application or the architectural models.

The application is a total of 2,695 lines as shown previously in Table 4.1. The architectural model is 2,804 lines, while the mapping code is 633 lines. Each additional mapping scenario can be described with approximately 100 lines of additional code, and without modifying any of the other code.

To show the fidelity of our modeling methodology and mapping framework, we initially abstracted two mapping scenarios from the implementations provided in Intel MXP5800 algorithm library and carried out simulation in the METROPOLIS environment. We also tried an additional two scenarios which did not have corresponding assembly language implementations. For all of the scenarios, only the mapping of the fine granularity 1D-DCT processes was varied.

The first scenario makes use of the DCT hardware accelerator and clearly has the highest performance. The other three scenarios use various software implementations of the row-wise and column-wise 1D-DCT operations. For these three scenarios, the transpose

Process	Hardware	Balanced	OPE Emph.	OPE Heavy
Level Shift	IPE	IPE	IPE	IPE
Add4-R	DCT HW Accelerator	OPE	OPE	OPE
Sub4-R		OPE	OPE	OPE
Add2-R		IPE	OPE	OPE
Sub2-R		IPE	OPE	OPE
Mult1-R		MACPE1	MACPE1	MACPE1
Mult2-R		MACPE2	MACPE2	MACPE2
Add4-C		IPE	IPE	OPE
Sub4-C		OPE	OPE	OPE
Add2-C		IPE	IPE	OPE
Sub2-C		OPE	OPE	OPE
Mult1-C		MACPE1	MACPE1	MACPE1
Mult2-C		MACPE2	MACPE2	MACPE2

Table 4.2: Mapping assignments

process is mapped to the MCH, which natively supports this type of operation. Register mappings are taken from the Intel library implementations and consist of 1, 2, or 4 global registers per FIFO channel. The second scenario uses a balanced partitioning of the processes among the available PEs, while the third and fourth scenarios put progressively more load on the OPE. The details for all four scenarios are provided in Table 4.2. The rows indicate the 13 processes in the refined two-dimensional DCT model. The columns indicate the mappings for the four different scenarios.

For each scenario, the number of clock cycles required to encode an 8x8 sub-block of a test image was recorded through simulation in Metropolis. For the first two scenarios, implementations from the MXP5800 library are available and were compared by running the code on a development board. The results are shown in Figure 4.10. The cycle counts reported with the Metropolis simulation are approximately 1% higher than the actual implementation since we did not support the entire instruction set for the processing

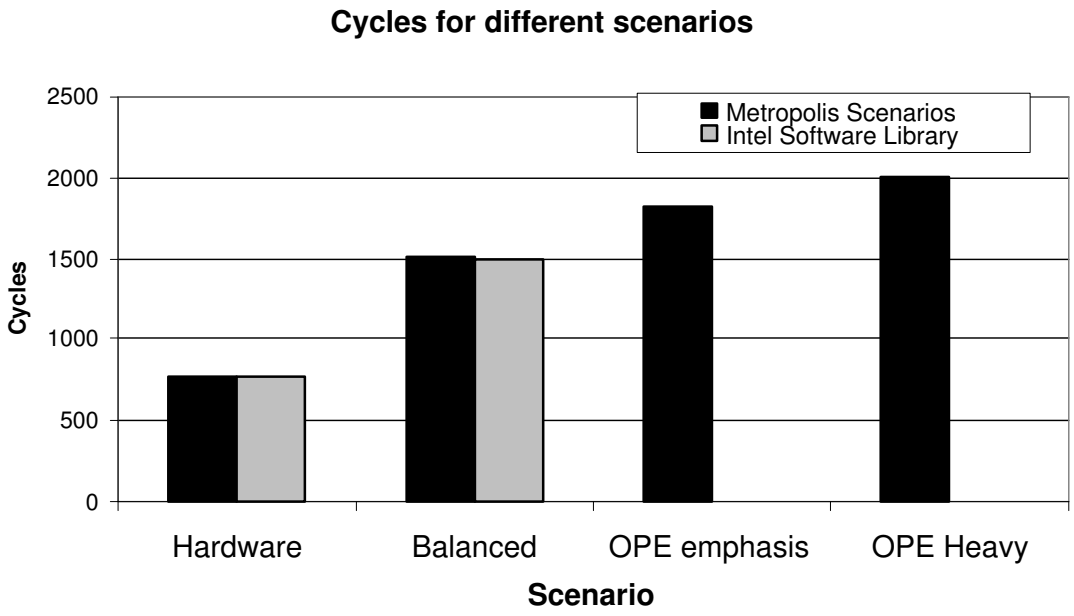

Figure 4.10: Performance Comparisons

elements. The latter two scenarios provide reasonable relative performance, but assembly implementations were not available for comparison.

As long as the granularity of the each dataflow process is small (such as for most DSP-like systems), we expect that this model will provide very accurate estimates of performance. Regardless of the computational granularity, the schedules and deadlock analysis capabilities of this MoC will still remain valid.

4.4.4 Conclusions

The main steps of the design flow exercised in this case study are choosing the model of computation, modeling the application and architecture in a common framework using this MoC, and evaluating different mappings.

If the MoC chosen captures the important characteristics of the system, as shown, then accurate performance estimates can be obtained at a fraction of the cost and much faster than with other verification methods and tools. The main tradeoff when choosing a MoC, as described in Section 4.3.2, is between expressiveness and analysis capabilities. After determining that a particular MoC captures the main characteristics of a particular class of systems, automated design space exploration techniques can be developed.

4.5 Automated Design Space Exploration: Motion-JPEG on Xilinx

The MoC chosen for this class of systems in Section 4.3.2 is amenable to analysis. In this section, the analysis capabilities are leveraged to develop an automated approach for allocation and scheduling. The Motion JPEG application and the Xilinx Virtex II FPGA platform are used in this work.

Mathematical programming techniques such as Mixed Integer Linear Programming (MILP) provide the ability to easily customize the allocation and scheduling problem to application or platform-specific peculiarities. The representation of the core problem in a MILP form has a large impact on the solution time required. In this work, we investigate a variety of such representations and propose a taxonomy for them. A promising representation is chosen with extensive computational characterization. We demonstrate that our approach can produce solutions that are competitive with manual designs.

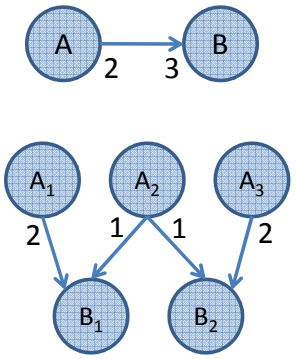

Figure 4.11: Transforming an SDF graph into a data precedence DAG

4.5.1 Problem Statement

Statically schedulable dataflow descriptions can be automatically transformed into acyclic data precedence graphs. These graphs are then used in the automated mapping procedure for multiprocessors [98]. An example of such a transformation for SDF systems is given in Figure 4.11. The graph on the top is a SDF graph where process A writes 2 tokens to the channel each time it is fired while process B reads 3 tokens from the channel each time it is fired. The associated data precedence DAG is shown in the lower part of the figure and captures the relationships between the process firings that are required to return the system to its initial state of having no tokens within the channel. The simplest such DAG requires process A to fire three times while B fires twice.

The core problem to be solved is to map the weighted directed acyclic graph (DAG) representing the application onto a set of architectural nodes. In the application DAG, nodes represent process firings while edges represent data precedence relationships. The architecture is represented with a weighted directed graph where nodes are processing elements (PEs). Edges that represent communication channels may be added explicitly to

the architecture graph if connectivity between PEs is restricted, otherwise, a fully-connected graph is assumed. The execution time for each task on each processor is fixed and given. The amount of communication between each application task is given by weights on the application graph edges. Communication cost only depends on the edge weight, not on the allocation of the source and sink processes. The objective is to allocate and schedule the tasks onto the PEs such that the completion time – or makespan – is minimized.

This work has four main contributions. First, a classification of existing MILP representations for this problem into a taxonomy. Second, based on the taxonomy, a core MILP formulation and useful customizations. Third, computational characterization and a comparison of our approach with a competing approach. Finally, a representative case study that illustrates the ability or our approach to produce competitive designs in terms of throughput and area.

4.5.2 Prior Work: Allocation and Scheduling

The scheduling problem we consider is a *generalization* of $R|prec|C_{max}$ [44] and is strongly NP-hard. R refers to the usage of multiple heterogeneous PEs with unrelated processing times, *prec* indicates that the application description includes precedence constraints, and C_{max} indicates the objective is to minimize the makespan or the maximum completion time. Unrelated processing times means that different processors may execute different tasks with arbitrary worst-case execution times.

For the special case of $R||C_{max}$ (no precedence constraints), there exist polynomial-time approximation algorithms that can guarantee a solution within a factor of 2 of the optimal [109]. No poly-time approximation algorithm exists that can provide a solution

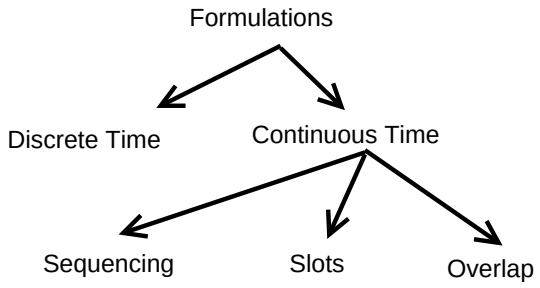

Figure 4.12: Taxonomy of MILP Approaches

for $R||C_{max}$ within 1.5 times the optimal, unless $P = NP$ [76]. If precedence constraints are added, there are no known good approximation results; an overview of related work for $R|prec|C_{max}$ is provided in [66]. A comprehensive listing of known lower and upper approximation bounds for a variety of scheduling and allocation problems can be found in [23], while an overview of heuristics is given in [67].

In [55], a MILP-based approach for mapping onto a multiprocessor FPGA platform is described. However, their approach is more specialized since it does not handle task precedence constraints nor determine scheduling.

4.5.3 MILP Taxonomy

Solution time for MILP instances is strongly affected by the representation used for the core allocation and scheduling problem. We observe that the effective encoding of task precedence relationships is key not only for approximation algorithms as mentioned in Section 4.5.2, but also for MILP representations. Along these lines, we propose a taxonomy of known MILP representations in Figure 4.12.

Discrete time approaches introduce a variable for each instant of time on each

PE. The resultant scheduling constraint requires that each such time instant be allocated to at most one task. The advantage of this method is that the formulation can easily be constrained to use only integer or binary variables. A rich variety of SAT solvers [37] can be utilized to solve these problems. However, this formulation has a significant drawback: the number of time variables introduced can quickly become very large, especially if diverse task execution times are present.

Continuous time approaches represent time with real-valued variables in the formulation. Vastly different execution times can easily be handled by these approaches, but the choice of variables and constraints used to specify a correct scheduling on each PE becomes critical in determining performance [45].

Sequencing: Variables are used to indicate sequencing to schedule tasks on PEs. These sequencing variables indicate whether a task is executed after another task on the same PE [9, 19]. This choice of variables can be viewed as a straightforward extension of the well-known formulations used in uniprocessor scheduling [91]. Typically, a large number of constraints or variables is required to enforce the scheduling requirements on each PE. Many of these constraints can be attributed to the linearization of bilinear terms [77].

Slots: This method uses explicit slots on each PE [80, 31] to which at most one task can be allocated. The start and finish time for each slot is not fixed a priori. With slots, the scheduling constraints between tasks on each PE become simpler to represent. However, since the exact number of slots on each PE is unknown, a conservative amount need to be used. As a result, this approach may suffer from variable blow-up if the typical number of tasks allocated to each PE is large.

Overlap: Variables are used to indicate temporal overlap (independent of PE assignment) in the execution of tasks [101, 108, 116]. Constraints that prevent overlap on the tasks allocated to each PE are used to enforce scheduling. Since the scheduling constraints can be expressed succinctly, this type of formulation scales well with respect to variables and constraints than the formulations in the other categories.

In this paper, we focus on continuous-time MILP formulations that use overlap variables, since this category seems the most promising for generating problems with fewer constraints and variables.

4.5.4 MILP Approach

In this section, our core formulation and customizations will be described in detail. The core formulation is based closely on the formulation presented in [116].

Core Formulation

Let $\mathbf{F}$ represent the set of tasks in the application DAG while $\mathbf{E} \subset \mathbf{F} \times \mathbf{F}$ represents the set of communication edges. The set $\mathbf{A}$ indicates the set of architectural PEs. The parameter $t \in \mathbb{R}^{\mathbf{F} \times \mathbf{A}}$ specifies the execution time of each task on each PE.

The variable $d \in \mathbb{B}^{\mathbf{F} \times \mathbf{A}}$ indicates if a task is mapped to a PE. Variables $s \in \mathbb{R}^{\mathbf{F}}$ and $f \in \mathbb{R}^{\mathbf{F}}$ indicates the start and finish times respectively for each task. $o \in \mathbb{B}^{\mathbf{F} \times \mathbf{F}}$ is a variable which is used to determine overlap in execution times between a pair of tasks.

$$min \qquad \max_{i \in \mathbf{F}} f_i \tag{4.1}$$

$$s.t. \qquad \sum_{x \in \mathbf{A}} d_{ix} = 1 \qquad \forall i \in \mathbf{F} \tag{4.2}$$

$$f_i \leq s_j \qquad \forall (i,j) \in \mathbf{E} \tag{4.3}$$

$$f_i = s_i + \sum_{x \in \mathbf{A}} (t_{ix} d_{ix}) \quad \forall i \in \mathbf{F} \tag{4.4}$$

$$f_j - s_i \leq M o_{ij} \qquad \forall i,j \in \mathbf{F}, i \neq j \tag{4.5}$$

$$o_{ij} + o_{ji} + d_{ix} + d_{jx} \leq 3 \quad \forall i,j \in \mathbf{F}, x \in \mathbf{A}, i \neq j \tag{4.6}$$

The objective function in 4.1 minimizes the maximum finish time over all tasks. This has the effect of minimizing the makespan. Constraint 4.2 ensures that each task is mapped onto exactly one PE. Constraint 4.3 requires that the precedence relationships between edges in the application DAG hold. Constraint 4.4 relates the start times and finish times of each task based on the execution time of the task on the appropriate PE. Constraint 4.5 ensures that if a task j finishes after task i begins, the corresponding variable o_{ij} is set to 1. In this constraint, M represents a large constant, which can be no less than the maximum finish time. Finally, Constraint 4.6 is a particularly elegant means of ensuring that no two tasks mapped onto the same PE may overlap in time.

In this constraint, the sum $o_{ij} + o_{ji}$ has a value of 2 iff the executions of the two tasks i and j overlap. An example is shown in Figure 4.13 where o_{ij} is set to 1 since task j finishes after task i begins. o_{ji} is set to 1 since task i finishes after task j begins. Both of these ensure that tasks i and j overlap. Therefore, they may not be allocated to the same PE.

Note that Constraint 4.6 only needs to be defined over $i,j \in \mathbf{F}$ such that neither

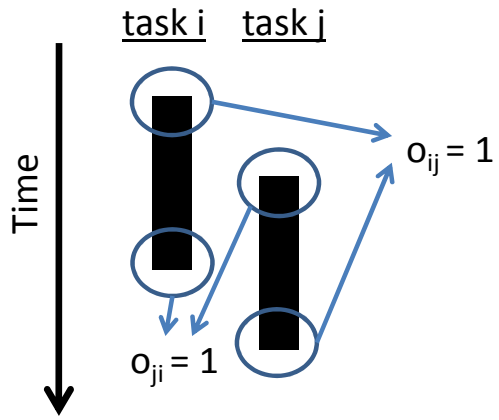

Figure 4.13: Overlap between two tasks i and j

i nor j are in each other's transitive fan-out (TFO). For all other cases, the sum $o_{ij} + o_{ji}$ must be 1.

Customizing the Formulation

The core formulation does not support communication cost, restricted architectural topologies, partially specified task allocation on the platform, and real-time requirements on portions of the application. Of these, the first three are crucial for the case study we target in Section 4.5.8.

The additional set $\mathbf{C} \subseteq \mathbf{A} \times \mathbf{A}$ represents the directed edges between PEs. The parameter $c \in \mathbb{R}^{\mathbf{E}}$ denotes the communication cost for each edge in the application DAG. The parameter $n \in \mathbb{B}^{\mathbf{F} \times \mathbf{F}}$ indicates whether two tasks are required to be mapped onto the same PE. Parameter $e \in \mathbb{B}^{\mathbf{F} \times \mathbf{A}}$ indicates if a particular task must be mapped onto a particular PE. The variables $r \in \mathbb{R}^{\mathbf{F}}$ and $w \in \mathbb{R}^{\mathbf{F}}$ represent the reading and writing time required for each task.

$$f_i = s_i + \sum_{x \in \mathbf{A}} (t_{ix} d_{ix}) + r_i + w_i \quad \forall i \in \mathbf{F} \tag{4.7}$$

$$r_i \geq \sum_{(j,i) \in \mathbf{E}} c_{ji} (d_{ix} - d_{jx}) \quad \forall i \in \mathbf{F}, x \in \mathbf{A} \tag{4.8}$$

$$w_i \geq \sum_{(i,j) \in \mathbf{E}} c_{ij} (d_{ix} - d_{jx}) \quad \forall i \in \mathbf{F}, x \in \mathbf{A} \tag{4.9}$$

$$d_{iy} + d_{jz} \leq 1 \quad \forall (i,j) \in \mathbf{E}, (y,z) \notin \mathbf{C} \tag{4.10}$$

$$d_{ix} \geq e_{ix} \quad \forall i \in \mathbf{F}, x \in \mathbf{A} \tag{4.11}$$

$$n_{ij} - 1 \leq d_{ix} - d_{jx} \leq 1 - n_{ij} \quad \forall i,j \in \mathbf{F}, x \in \mathbf{A} \tag{4.12}$$

Constraint 4.7 replaces Constraint 4.4 from the core formulation and considers the reading and writing time for each task. Constraint 4.8 charges time for reading iff the predecessor task is assigned to a different PE. Likewise, Constraint 4.9 charges the corresponding write time. Constraint 4.10 ensures that the mapping conforms to the restricted architectural topology. Constraint 4.11 is a forcing constraint that allows some allocations to be fixed. Finally, Constraint 4.12 restricts certain pairs of tasks to be allocated to the same PE. This is useful when considering applications derived from dataflow specifications, where multiple invokations or firings of the same actor may be constrained to occur on a single PE.

4.5.5 Characterizing the Formulation

In this section, we compare our formulation against the sequence-based formulation and identify characteristics of problem instances that affect the runtime of the MILP formulation.

The experimental setup involves coding the sequence-based formulation from [19]

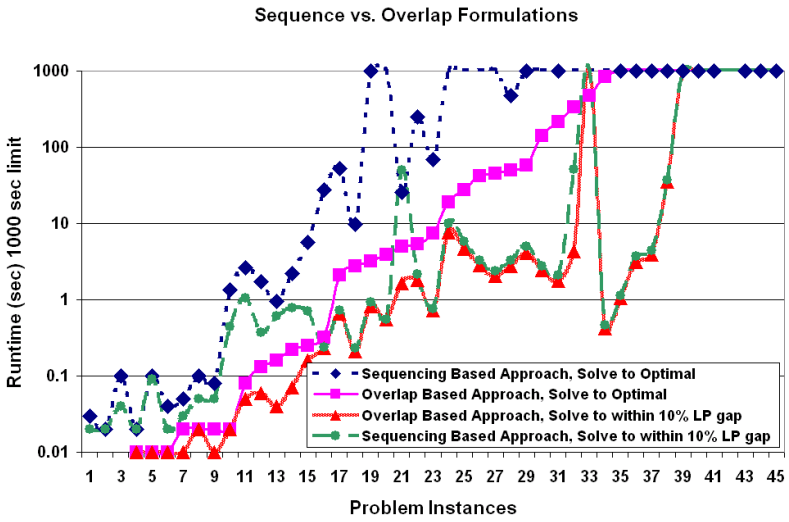

Figure 4.14: Sequence vs. Overlap Runtime

and our core formulation in AMPL [40] and evaluating them with a set of 45 test cases. The test cases were generated with the TGFF [36] tool with three random seeds. Five problem sizes, ranging from 10 to 50 tasks, were generated from each seed. Each task graph was allocated to different numbers of PEs to keep the average task/PE ratio the same. The test cases were solved using CPLEX 9.1.2 on 2.8GHz Linux machines with 2GB of memory under a time limit of 1000 seconds.

4.5.6 Comparison: Sequencing vs. Overlap

For the 45 test cases, on average, our overlap-based formulation has 30% more variables than the sequence-based formulation. However, our formulation also has 63% fewer constraints, which substantially reduces overall problem size.

For solving problems to optimality in a balanced branch-and-bound exploration of the solution space, our approach is an *order of magnitude* faster than the sequence-based

approach, as shown in Figure 4.14. LP relaxations of the problems are usually quite tight, often within 10-15% of the optimal value. This means that good lower bounds can be obtained in polynomial time for these problem instances. For instances that could not be solved to optimality within the time limit, feasible solutions within 14% of optimal were obtained on average.

If a solution within 10% of the optimal is sufficient, we can bias the branch-and-bound exploration to find feasible solutions. These results are also plotted in Figure 4.14 and show that solution time can be decreased by 1-2 orders of magnitude with biasing and a 10% optimality gap. For very few cases, feasibility biasing may increase solution time.

4.5.7 Factors Influencing Solution Time

Solution time is typically analyzed with respect to the number of tasks, the number of constraints or the number of PEs for a given problem instance. None of these factors is a good indicator of solution time for this formulation. For a problem with same number of tasks, as the number of PEs available decreases, we discover a counterintuitive trend: the number of constraints (and variables) drops, but the runtime increases.

The rising solution time for test cases with fewer PEs and constraints can be explained with three observations. First, when there are relatively fewer PEs, more unrelated tasks (tasks not in each other's TFOs) have to be sequentialized onto each PE. A formulation that relies on binary variables and big M constants to enforce non-overlapping of tasks (Constraint 4.5) has a weaker LP lower bound with more tasks/PE. Secondly, when many unrelated tasks have similar processing times, many feasible solutions have similar makespans, this prevents effective pruning of the branch and bound tree based on known

feasible solution upper bounds. Thirdly, the number of feasible permutations of task ordering explodes with more tasks/PE. If we have k unrelated tasks allocated on the same processor, many of the $k!$ permutations must be considered in the branch and bound tree. The inverse application graph has edges between unrelated nodes (those without precedence constraints). The total number of permutations increases as a function of the maximum clique (fully connected component) of the inverse application graph. Making more PEs available disperses unrelated tasks - fragmenting the cliques and improving the LP lower bound.

4.5.8 Case Study

We now turn our attention to demonstrating the applicability of our customized MILP formulation on a case study. The chosen application is the Motion JPEG encoder described in Section 4.1.2. The architectural platform we consider contains soft-core processors and processing elements on a Xilinx Virtex II Pro FPGA fabric as described in Section 4.2.2. For various manual and automated mappings, we compare the performance in terms of system throughput and area utilization. For applications derived from dataflow specifications, the makespan of the data precedence DAG of an unrolled dataflow graph is equivalent to throughput [98].

Manual Design Space Exploration

The goals in manual design space exploration are to utilize various numbers of uBlaze processors and DCT-specific PEs to maximize the throughput of the Motion-JPEG application. A nominal frame size of 96x72 is assumed for all implementations. We start

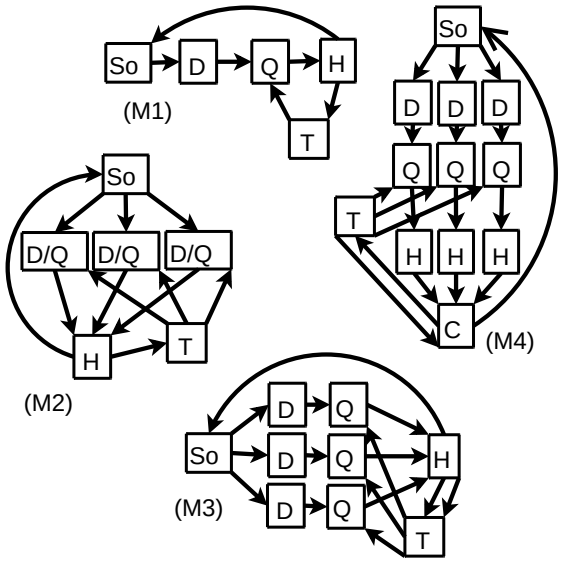

Figure 4.15: Topologies of Manual Designs

from a baseline topology where the entire application is mapped onto a single *u*Blaze processor. As additional PEs are utilized, portions of the application are migrated to these PEs to improve throughput.

The PEs in the various designs are connected with FSL queues that are accessed in blocking-read, blocking-write mode. Data is fed to and retrieved from the device with a 100 Mbps Ethernet connection to a host PC. An Ethernet MAC device is instantiated on the fabric to handle this communication. One of the *u*Blaze PEs in each design is designated as the I/O processor and connects to the Ethernet device. In addition, this *u*Blaze is connected to peripherals to allow for debugging and performance measurement. In all designs except the baseline topology, a single *u*Blaze is reserved for quantization table updates. The experimental setup is shown in Figure 4.16.

The different manual designs obtained are shown in Figure 4.15. The blocks used

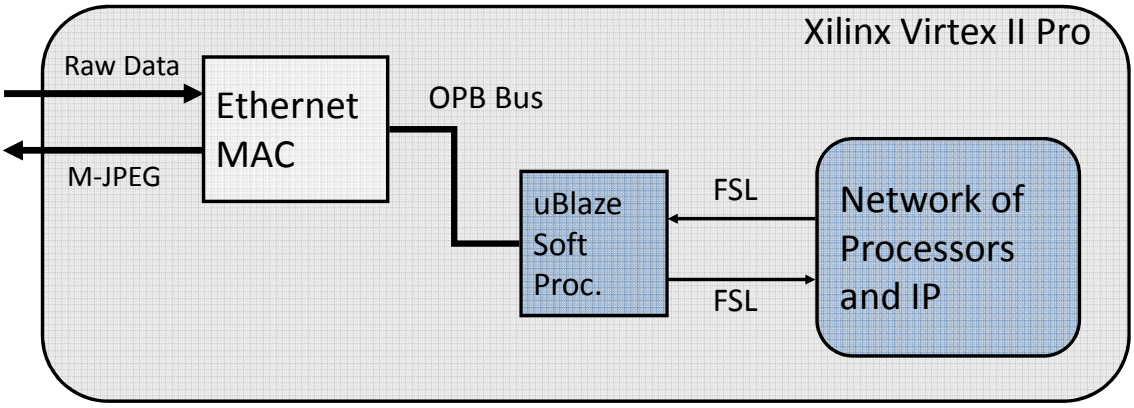

Figure 4.16: Experimental Setup

Design	***u*Blaze**	**DCT**	**fps**	**Area**
Base	1	0	26.5	21%
M1	5	0	51.1	39%
M1D	4	1	72.0	53%
M2	6	0	85.1	47%
M3	9	0	85.3	62%
M3D	6	3	85.6	94%
M4	12	0	148.8	83%

Table 4.3: Manual Designs

include the data source (So), DCT (D), quantization (Q), Huffman encoder (H), and table update (T). A combiner (C) is necessary when the Huffman block is split into 3 parts. Salient characteristics of each implementation – the number of *u*Blaze and DCT PEs used, the frames processed per second, and the area (% slices occupied on the FPGA) – are summarized in Table 4.3. Designs $M1D$ and $M3D$ are obtained from $M1$ and $M3$ respectively by substituting *u*Blazes with DCT PEs where possible.

The manual designs exploit the task-level and data-level parallelism in the application. The designs first attempt to use task-level parallelism between the different stages

Task	Cycles
Source	200
DCT	4,760
Quant	2,572
Huffman	3,442
Combiner	2,542
DCT Acc.	328

Table 4.4: Profiling Information

and the exploit the natural data-level parallelism between the three components in the color space.

Automated Design Space Exploration

Automated Design Space Exploration uses the MILP formulation developed in Section 4.5.4 to determine the scheduling and allocation for tasks in the case study. The aim is to show that the cost model in the formulation accurately captures the design space and can be used to implement competitive designs.

The first step is to create a representation for the application which identifies the maximal amount of available concurrency. Since we would like to compare against the manual implementations, we create a task representation which extracts no more concurrency than is utilized in the manual designs. This corresponds to design (M4) from Figure 4.15. We also reserve a separate *u*Blaze for the table update portion of the application, just as in the manual designs. Note that both of these restriction can be relaxed to obtain higher quality automated designs.

The next step is to characterize the application so that the task execution times (the t parameters in the formulation) can be obtained. For both the *u*Blaze processor and

the DCT accelerator, the cycle times are obtained from the timer peripheral. The cycle times for the tasks are shown in Table 4.4.

If these parameters are used in our MILP formulation, any legal solution produces a static estimate for throughput. The accuracy of this estimate is important in determining the effectiveness of an automated approach. We compare the estimates for the base design and the 6 manual designs from Section 4.4 against the actual implementation results obtained from the development board.

The makespan estimated from the formulation is, on average, within 5% of the execution time measured on the development board. Most of the predictions overestimate the makespan, since the formulation does not consider simultaneity between reads and writes on each FIFO.

With this accurate model of the design space, we can automatically evaluate a number of different solutions from the MILP formulation. Based on the characteristics of the manual designs, we picked three promising MILP solutions and implemented them on the development board. These three automated solutions (A1, A2, and A3) use 3, 5, and 8 uBlazes and were obtained with a 100s time limit on the solver. The MILP solution also confirms that design M4 is optimal given the chosen granularity of the task graph.

Both the manual and automated solutions can be plotted in terms of frame rate vs. FPGA slices consumed - which is roughly proportional to the number of uBlazes and DCT PEs used. Figure 4.17 shows this tradeoff and indicates that the automated designs do indeed result in competitive implementations that lie on the Pareto curve.

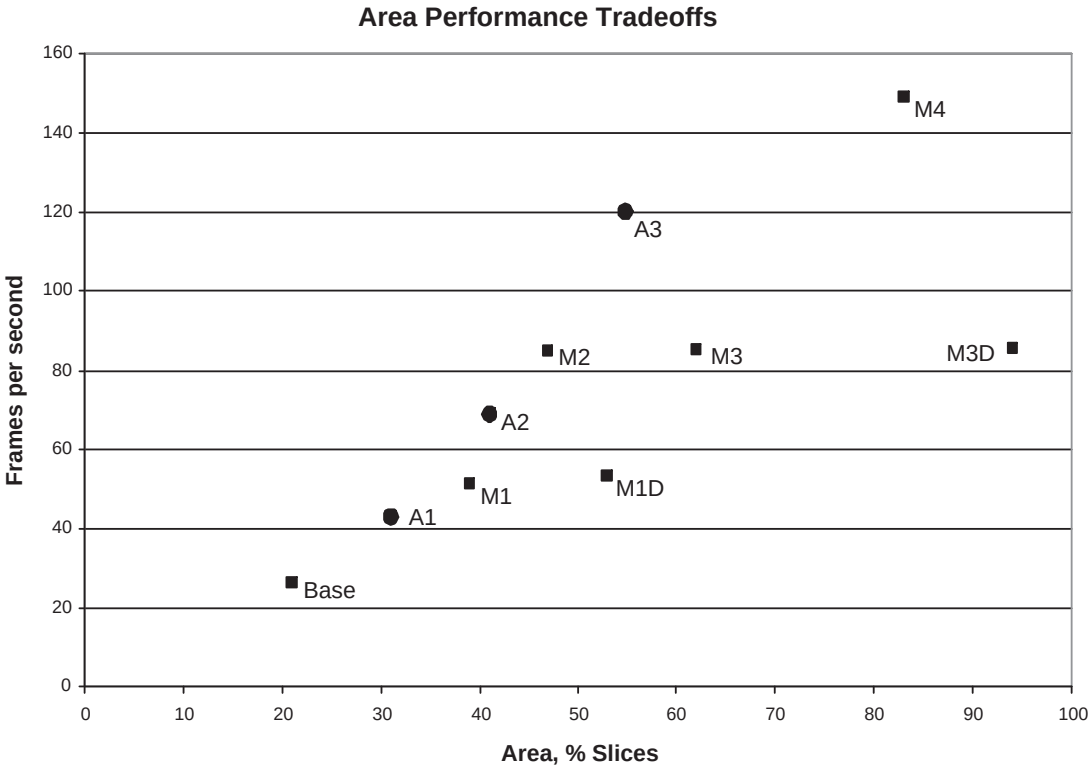

Figure 4.17: Manual vs. Automated Designs

4.5.9 Conclusions and Future Work

In this work, we considered a number of MILP approaches for solving the task allocation and scheduling problem. Based on their treatment of precedence constraints, a taxonomy of known MILP representations was proposed, and the most promising core formulation was selected. Extensions were then added to customize the formulation for our needs. With extensive computational testing, we showed that our overlap-based formulation has better solution time than a competing sequence-based formulation, and demonstrated that tight lower bounds could be obtained in polynomial time. We also identified key metrics for determining the difficulty of problem instances - demonstrating that for a given application, larger platforms actually decrease solution time. Our formulation was applied to a case study that considers the deployment of an MJPEG application on a Xilinx FPGA platform. We showed that our formulation can accurately predict the performance of the system and quickly produce solutions that are competitive with manual designs.

In the future, we plan to further extend the customizations used here to enable the targeting of more complex platforms. For instance, by handling the allocation of distributed memory. Also, we would like to consider more specialized solution techniques. In particular, the addition of constraints corresponding to critical paths during the branch-and-bound process seems promising.

4.6 Conclusions

The multimedia domain features increasingly complex applications deployed on highly heterogeneous parallel platforms. In this work, we have considered two related ap-

plications and their deployment on two separate heterogeneous multiprocessor platforms. The first step is choosing an appropriate model of computation. For the systems considered here, a statically-schedulable dataflow variant based on cyclo-static dataflow was chosen. First, in Section 4.4, it was verified that the MoC was expressive enough to succinctly capture the system and accurately characterize its performance. In Section 4.5, a Mixed Integer Linear Programming approach was developed to automate the allocation and scheduling problem for these types of systems. It applicability was demonstrated by applying it to a case study consisting of deploying a Motion-JPEG encoder onto a multiprocessor Xilinx FPGA platform.

Chapter 5

Automotive Domain

The automotive domain is characterized by distributed controls applications deployed onto heterogeneous distributed architectures. These architectures consist of electronic control units (ECUs) connected with standardized buses. The mapping problem for these systems involves allocating the processing tasks onto ECUs, the messages to buses, and configuring the execution of both. A variety of technical and business factors have contributed to making this problem especially challenging.

First, automobiles are increasingly becoming differentiated based on their electronic content, especially the type of active safety functionality they feature. It is estimated that a majority of the innovation and cost of a new vehicle now resides in the electronics [75]. Due to the large volumes, automotive electronics are price-conscious, meaning that up-front optimization effort is valuable if system costs can ultimately be reduced.

However, system-level optimization has not always been feasible in the automotive domain. Typically, hardware and software is bundled together in modules that are sold

to the system integrators - the carmakers. Recently, there has been an effort from the carmakers to de-link and standardize the hardware and software portions of the system. This is being carried out under the auspices of the AUTOSAR [51] initiative. Not only does AUTOSAR enable the carmakers to carry out system-level optimization, it also enables the suppliers to compete more effectively [106]. The de-linking of software from hardware facilitates the modeling and optimization that will be described in this section.

5.1 Applications

The embedded automotive applications considered here are active safety applications. These applications collect data from 360^o sensors around the vehicle to understand the positioning of surrounding objects and detect hazardous conditions. On hazard detection, the active safety functions attempt to inform the driver or provide control overlays to reduce the risk to the occupants of the vehicle. Most of these functions are high-level controls which drive low-level actuation loops.

Application descriptions can be viewed as directed graphs, with nodes representing function blocks and edges representing data dependencies. Data dependencies are messages that are sent between blocks. The application description is further characterized by end-to-end latency constraints along certain paths from sources to sinks. These constraints bound the execution time of certain chains of tasks to occur within the predefined time.

Two active-safety applications are considered in this work, both of which have been obtained through the collaboration with General Motors.

5.1.1 Distributed Supervisory Control Application

The application is a limited-by-wire system that implements a supervisory control layer over the steering, braking and suspension systems. The objective is to integrate active vehicle control subsystems to provide stability and comfort to the occupants. The high-level view of the functionality is shown in Figure 5.1. The supervisor plays a command augmentation role for braking, suspension and steering by using sensors to collect data from the environment. This supervisory two-tier control architecture enables a flexible and scalable design where new chassis control features can be easily added into the system by only changing the supervisory logic.

5.1.2 Experimental Vehicle

This application supports advanced distributed functions with end-to-end computations collecting data from 360° sensors. The actuators consist of the throttle, brake and steering subsystems and of advanced HMI (Human-Machine Interface) devices. A total of 92 tasks exchange 196 messages in the block diagram for the application.

End-to-end deadlines are placed over paths between 12 pairs of source-sink tasks in the application. A change of data at the input of the path must lead to a corresponding change at the path output within the specified end-to-end latency. Most of the paths follow a six-stage structure: sensor preprocessing & sensory fusion, object detection, selection of target objects in the environment, core functions, vehicle longitudinal & lateral controls with actuator arbitration & planning, and, finally, low-level loops of the actuators themselves. Most of the intermediate stages are shared among the tasks. Therefore, the blocks in the

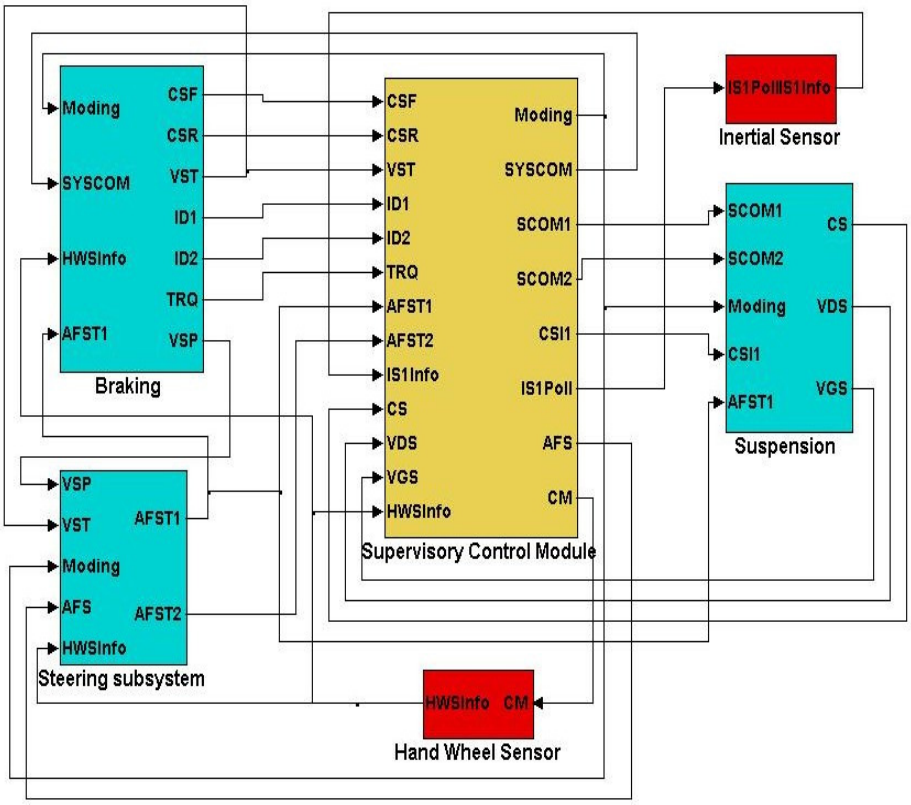

Figure 5.1: Functional Model for Distributed Supervisory Control System

application graph are quite densely connected. Despite the small number of source-sink pairs, there are 222 unique paths among them.

5.2 Architectural Platforms

Architectural platforms for the systems we consider are highly distributed. Distributed architectures supporting the execution of hard real-time applications are common not only for automotive, but also for avionics and industrial control systems. To provide design-time guarantees on timing constraints, different design and scheduling methodologies are used. For instance, avionics systems are often built based on static, time-driven schedules. Due to resource efficiency and ultimately price concerns, many automotive systems are designed based on run-time priority-based scheduling of tasks and messages. In this work, we consider architectures which feature two standards supporting this model: the OSEK operating system standard [94] and the CAN bus arbitration model [12]. OSEK OSs provide preemptive priority-based run-time task execution while CAN buses allow non-preemptive priority-based run-time message transmission.

ECUs feature CAN controllers which provide send and receive functionality. The number of buffers available in the CAN controller is strongly correlated with price, and typical CAN controllers have a limited number of send/receive buffers. This necessitates different tasks on the ECU to share CAN controller buffer space. Due to this limited buffer space and because ECUs are not synchronized with each other, message loss and duplication may occur.

Different interaction models may be implemented at the interface between any two

resource domains (such as an ECU and a bus). The simplest interaction model consists of the periodic activation with asynchronous communication, where all interacting tasks are activated periodically and communicate by means of asynchronous buffers based on non-blocking read/write semantics. Similarly, message transmission is triggered periodically and each message contains the latest values of the signals that are mapped into it.

5.3 Choosing the Model of Computation

To choose an appropriate MoC for systems in this domain, we will first consider the current design methodology. In current industrial practice, the application is typically described in Matlab/Simulink. Simulation is typically carried out to validate the application model. The architectural model is not captured explicitly during the design flow. Code generation from the application model to the architectural model is carried out using RealTime Workshop (RTW). However, RTW assumes that the generated code is deployed on uniprocessor architectures. In practice, since the architectural platform can lose and duplicate messages, the semantics of the architecture and the functionality are not compatible. Therefore, a correct-by-construction deployment is not guaranteed. The system needs to be re-validated, even if the functionality has already been tested.

In current industrial practice, system validation is aided by two techniques: overdesign and in-vehicle testing. Overdesign involves sending additional messages between tasks to compensate for expected message loss. Regardless of overdesign, in-vehicle testing is required to assure the designer that the system works as expected. Unfortunately, in-vehicle testing is expensive and occurs late in the design cycle.

Finding a suitable MoC to capture automotive systems involves a tradeoff between expressiveness and analyzability, just as in the multimedia domain. One choice is to keep the more analyzable synchronous reactive model of computation for the application, and add wrappers to the architecture to ensure compatibility. The second choice is to expose architectural non-idealities in the application model.

The first choice involves making the architecture compatible with the synchronous reactive model. This may involve adding synchronization capabilities between different ECUs in the architecture, expanding buffer sizes for the transmission, or adding retransmission capabilities to the architectural platform. These options directly increase either architectural cost or utilization, but allow better analysis.

The second choice involves making the lack of synchronization, message loss, and message duplication visible in the functional model. This "lossy" MoC does not require any wrappers to be defined for the architectural model. However, since the functional model becomes more expressive under this MoC, the analysis that can be carried out is much weaker. We explore the second MoC further in this work, since it more easily captures the current industrial situation, and leads to interesting automated mapping problems.

More specifically, the execution model considered in this work is the following. Input data (generated by a sensor, for instance) is available at one of the system's ECUs. A periodic activation signal from a local clock triggers the computation of an application task on this ECU. Local clocks on different ECUs are not synchronized. The task reads the input data, computes intermediate results, and writes them to the output buffer from where they can be read by another task or used for assembling the data content of a message.

Messages - also periodically activated - transfer the data from the output buffer on the current ECU over the bus to an input buffer on another ECU. Tasks may have multiple fan-ins and messages can be multicast. Eventually, task outputs are sent to a system output (an actuator, for instance).

In the lossy MoC, concurrent processes communicate through FIFOs. FIFOs are fixed length and may nondeterministically delete or duplicate messages. FIFO access is non-blocking. The MoC is timed, so processes are triggered periodically, each with a specific rate and a relative phase. If all FIFOs are one-place buffers and processes are triggered at the same rate and with the same phase, then the MoC is equivalent to the synchronous reactive model. Otherwise, depending on the configuration used for the rates, phase offsets, FIFO sizes, and message duplication and deletion policies, the functional model can behave as it would when mapped to an actual architectural platform.

When implementing feedback control applications in this fashion, the (quasi) periodic stream of actuator commands may be based on sensor data taken a variable number of samples in the past, depending on how the various clocks align. For this reason, the control algorithms are typically designed favoring robustness over performance. Techniques like time-stamping and sequence counters are sometimes used at the application level to compensate for variations and to improve robustness. Nonetheless, hard bounds on latency and periodicity are provided as implementation requirements.

5.4 Manual Design Space Exploration: Distributed Supervisory Control System

The lossy MoC has been used to model a distributed supervisory control system from General Motors. Implementation of the MoC and detailed architectural modeling is carried out in Metropolis. A more complete description of this case study is given in [125].

The architecture consists of 6 ECUs and includes 2 smart sensors connected over a high-speed CAN communication bus. The two sensors are the hand wheel sensor which obtains steering position and the inertial sensor for yaw rate lateral acceleration measurements. The interfaces to the body and powertrain vehicle subsystems are not modeled. The architectural model is shown in Figure 5.2.

Details of the ECU modeling are shown in Figure 5.3. Multiple software tasks can execute on the ECU. Initially, they interact with the middleware, which provides both location and access transparency. Location transparency means that the data sources and sinks remain hidden from the software tasks. For instance, communication may be mapped to the CAN bus for remote tasks or a local buffer in the case of communication between multiple tasks on the same ECU. Access transparency means that the internal representation as well as access policies for shared communication resources are also hidden from the software tasks. The RTOS model implements an OSEK-compliant priority-based preemptive scheduler. The CAN driver transfers messages between the middleware buffers and the CAN controller. The CAN controller has bus sender and receiver processes that execute concurrently with the software tasks.

A variety of options can be explored during design space exploration including:

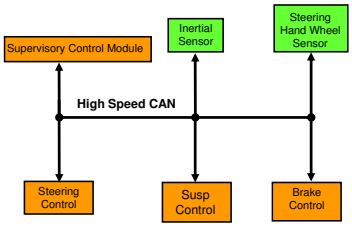

Figure 5.2: Architectural Model for Distributed Supervisory Control System

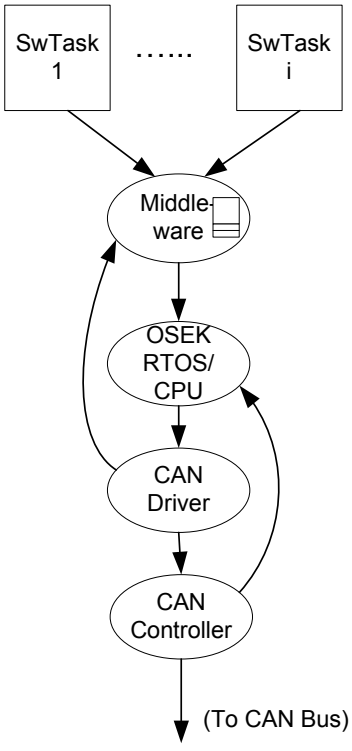

Figure 5.3: Details of ECU Modeling

- Allocation of tasks to ECUs
- Priority assignments for tasks and messages
- Packing of signals to messages
- Configuration for CAN controllers and drivers

For this system, when carrying out mapping between the functionality and the architecture, there is a one-to-one mapping between the tasks in the application and the ECUs in the architecture. For this case study, only the CAN controller and driver configuration are varied.

The objective of design space exploration is to mitigate the effects of priority inversion within the system. Priority inversion is a well known but often overlooked problem in embedded real-time systems. The basic scenario involves three or more tasks and a shared resource.

When the shared resource needs mutually exclusive access to a shared resource, it is generally accepted that a higher priority task cannot execute until the resource is "released", even if the task "occupying" the resource has lower priority: no preemption can take place. This phenomenon is not referred to as priority inversion, even though the higher priority task may need to wait for the completion of the lower priority task. The actual priority inversion comes from the presence of one or more intermediate priority tasks that do not need the shared resource and that may preempt or delay the completion of the lower priority task, which in turn delays the completion of the higher priority task. This latter phenomenon occurs between lower priority tasks and the higher priority task(s) without

any mutual exclusion constraints that would otherwise justify it.

In the models used for this case study, this inversion may occur between messages of different priority originating from the same ECU, when transmit buffers in the CAN controller are shared between them. It is fairly common to have a single transmit buffer shared by all transmitted messages from a single ECU, making this problem especially acute.

If a high priority message is queued in the middleware while a low priority message resides in the transmit buffer of the CAN controller, the high priority message will be blocked until the low priority message is successfully transmitted. Again, this is not the inversion per se, but when intermediate priority messages are transmitted on the CAN bus by other nodes, they effectively delay the transmission of the low priority message and, consequently, of the high priority message.

In the case study, described further in [125], different amounts of buffer space is used for the CAN controller transmit buffers. The system model accurately captures the effect on varying the transmit buffer sizes on the transfer times for messages of low, medium, and high priorities.

5.5 Automated Design Space Exploration: Period Assignment

The correct deployment of active-safety applications on distributed automotive architectures requires end-to-end latency deadlines to be met. This is challenging since deadlines must be enforced across a set of ECUs and buses, each of which supports multiple

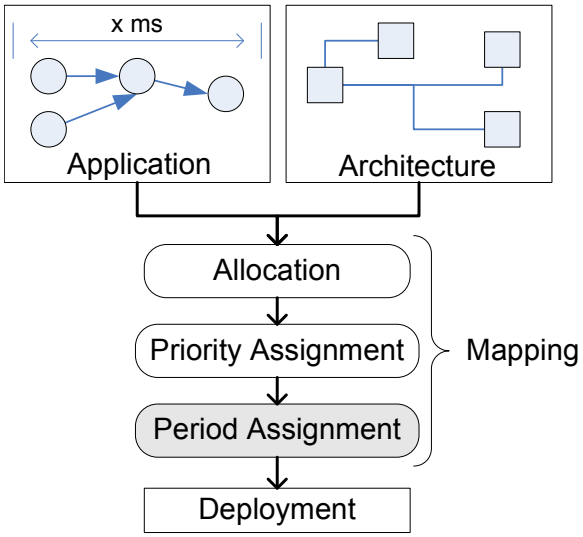

Figure 5.4: Period assignment within the overall design flow

functionality. The need for accommodating legacy tasks and messages further complicates the scenario.

In this work, we *automatically* assign task and message periods for distributed automotive systems. This is accomplished by leveraging schedulability analysis within a convex optimization framework to simultaneously assign periods and satisfy end-to-end latency constraints. Our approach is applied to an industrial case study as well as an example taken from the literature and is shown to be both effective and efficient.

5.5.1 Design Flow

The period assignment problem addressed in this work tackles a part of the larger design flow shown in Figure 3.4. Mapping deploys functional blocks to tasks and tasks to ECUs. Correspondingly, signals are mapped into local communication or messages that are exchanged over the buses. We further divide the mapping step into three stages: allocation,

priority assignment, and period assignment. Allocation is the first stage and assigns tasks to ECUs and messages to buses. Each task is allocated to a single ECU while each message is allocated to a single bus. The second stage assigns priorities to both tasks and messages. The last stage assigns periods to task and messages.

In this work, we restrict the focus to the period assignment stage. Given an allocation and priority assignment for both tasks and messages, our approach automatically assigns periods for all tasks and messages in order to satisfy the end-to-end latency requirements. The results of period assignment may trigger design iterations over the allocation and/or priority assignment stages when a feasible solution cannot be found or when the design can be further improved by changing the allocation or reassigning the priorities (for example, following the rate monotonic rule).

5.5.2 Prior Work

Both static and dynamic priority, distributed as well as centralized scheduling methods have been proposed in the past for distributed systems. Static and centralized scheduling is typical of time triggered design methodologies, like the Time-Triggered Architecture (TTA) [65] and its network protocol TTP and of implementations of synchronous reactive models, including Esterel and Lustre [10]. Also, the recent FlexRay standard [39] for high speed communication in automotive systems provides two transmission windows, one dedicated to time-driven periodic streams with static design-time assignment of transmission slots, and the other for asynchronous event-driven communication.

Priority-based scheduling is also very popular in control applications and is supported by the native CAN network arbitration protocol. The worst case transmission laten-

cies of CAN messages (with timing constraints) have been analyzed and discussed in past research work [114]. Also, the OSEK operating system standard for automotive applications supports not only priority scheduling, but also resource sharing with predictable blocking times. Priority-based scheduling of single processor systems has been thoroughly analyzed with respect to worst case response time and feasibility conditions [49].

End-to-end deadlines have been discussed in research work in the context of both single-processor as well as distributed architectures. The synthesis of task parameters (activation rates and offsets) and (partly) of task configuration itself in order to guarantee end-to-end deadlines in single processor applications is discussed in [42]. Later, the work has been tentatively extended to distributed systems [105] where a set of design patterns are applied to meet the deadlines using offset-based scheduling.

The periodic activation model with asynchronous communication can be analyzed quite easily in the worst case, because it allows the decomposition of the end-to-end schedulability problem into local problem instances, one for each resource (ECU or bus). This is not true in the case of data-driven activation models, where local schedulers have cross dependencies due to the propagation of activation signals. In this case, the problem of distributed hard real-time analysis has been first addressed by the holistic model [100] based on the propagation of the release jitter along the computation path.

While the prior work provides analysis procedures with reduced pessimism, the synthesis problem is today largely open, except for [102], where the authors discuss the use of genetic algorithms for optimizing allocation and priority assignments with respect to a number of constraints, including end-to-end deadlines and jitter.

5.5.3 Representation

The systems we consider can be represented with a weighted directed graph ($\mathcal{O}$,$\mathcal{L}$) and a set $\mathcal{R}$. $\mathcal{O}$ is the set of vertices denoting the schedulable objects (tasks and messages), $\mathcal{L}$ is the set of edges representing the flow of information (data dependencies), and $\mathcal{R}$ is a set of shared resources supporting the execution of the tasks (ECUs) and the transmission of messages (buses).

- $\mathcal{O} = \{o_1, \ldots, o_n\}$ is the set of schedulable *objects* implementing the computation and communication functions of the system. An object o_i represents either a task or a message and is characterized by two parameters: a maximum time requirement c_i and a resource R_j to which it is allocated ($o_i \rightarrow R_j$). All objects are scheduled according to their priority and a total order exists between the priorities of all objects on each resource. The object is periodically activated with a period t_i. r_i is the worst case response time of o_i, representing the largest time interval from the activation of the object to its completion in case it is a task, or its arrival at the destination in case it is a message. The response time of an object includes its own time requirement as well as the time spent waiting to gain access to the resource.

- $\mathcal{L} = \{l_1, \ldots, l_m\}$ is the set of *links*. A link $l_i = (o_h, o_k)$ connects an object o_h (the source) to object o_k (the sink). One object can be the source or sink of many links. At the end of its execution or transmission, an object delivers results (task) or its data content (message) on all outgoing links. For any link, the sink object is activated by a periodic timer and, when it executes, reads the latest signal value that was transmitted over the link.

- $\mathcal{R} = \{R_1, \ldots, R_z\}$ is the set of logical resources that can be used by the objects to carry out their computations. Resources are either ECUs or buses and are scheduled with a priority-based scheduler.

A *path* p is a finite sequence of objects ($p \in \mathcal{O}^*$) that, starting from $o_i = src(p)$, reaches $o_j = snk(p)$ with a link between every pair of adjacent objects. o_i is the path's source and o_j is the sink. Sources are activated by external events, while sinks activate actuators. Multiple paths may exist between each source-sink pair. The worst case end-to-end latency incurred when traversing a path p is denoted as ℓ_p. The *path deadline* for p, denoted by d_p, is an application requirement that may be imposed on selected paths.

The graph in Figure 5.5 can be used to explain the representation. It consists of 8 tasks and 5 messages allocated to 3 ECUs and 1 bus respectively. For each object, the time requirement is given by the value inside the object, while the priority is given by the value beside it. Three paths are present in the example; two of which have deadlines associated with them. Shaded nodes denote external events or actuators.

Object Schedulability

Object schedulability requirements in the system ensure that each task and message is processed every activation. This requirement enforces the assumption that objects are not queued for later processing. This assumption is compatible with all modes of the OSEK standard. The constraint that must be met is:

$$r_i \leq t_i \qquad \forall i \in \mathcal{O}$$

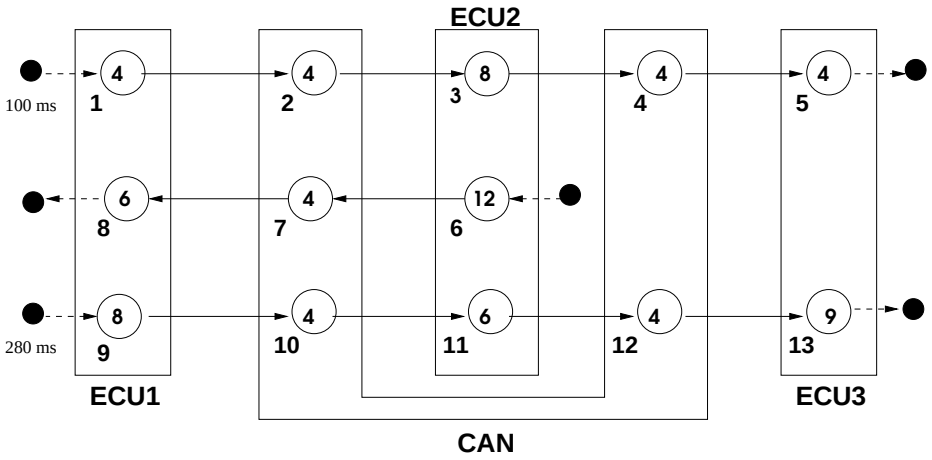

Figure 5.5: An example system graph

Resource Utilization

Resource utilization constraints place an upper bound on the fraction of time a resource may spend processing its objects. The utilization must always be less than 100%, and may be constrained further due to the designer-imposed restrictions. Resource utilization is calculated as:

$$\sum_{i:o_i \to R_j} \frac{c_i}{t_i} \leq u_j \qquad \forall R_j \in \mathcal{R}$$

To calculate the utilization on a resource R_j, we take the sum of the processing time divided by the period for all objects which are allocated to that resource ($i : o_i \to R_j$). u_j is the utilization bound for the resource and may be set to less than 1 for reasons such as future extensibility, where the ability to add additional tasks or messages to the resource late in the design cycle is important.

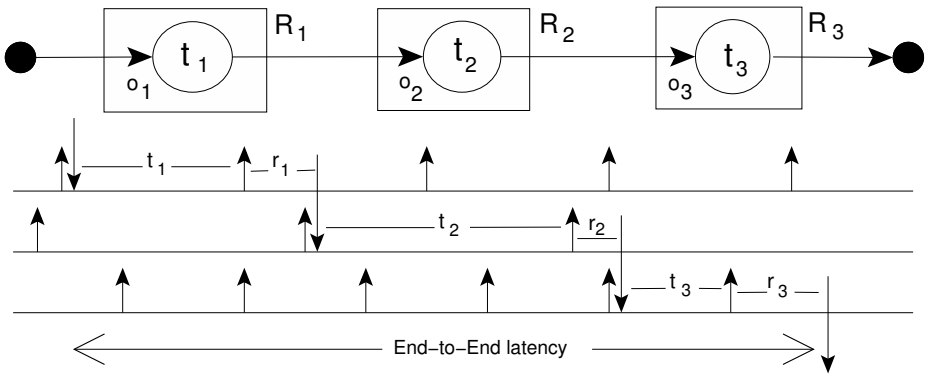

Figure 5.6: End-to-End Latency Calculation

End-to-End Latency

The worst case end-to-end latency can be computed for each path by adding the worst case response times and the periods of all the objects in the path:

$$\ell_p = \sum_{k:o_k \in p} t_k + r_k$$

In the worst case, as shown in Figure 5.6, an external event arrives immediately after the completion of the first instance of task o_1. The event data will be read by the task on its next instance and the result will be produced after its worst case response time, that is, $t_1 + r_1$ time units after the arrival of the external event. Since there is no coordination between tasks on separate resources, the situation repeats in the worst case for each link in the path. To get more precise results, the best case response time v_i of any predecessor object o_i should be subtracted from the period t_i in the previous formula. However, in most cases, including the case studies in Section 5.5.5, $v_i \ll t_i$ and v_i can be ignored.

For multiple communicating tasks with harmonic periods on the same ECU, the analysis can be less pessimistic if we assume that the designer can select the relative activa-

tion phase of all tasks. In case the sink task is activated with a relative phase with respect to the source equal to its worst case response time, then the contribution of the pair to the end-to-end latency can possibly be reduced. Let o_1 and o_2 be two tasks on the same ECU that appear (in that order) in a path with an end-to-end deadline. If $t_1 = kt_2$ is satisfied, where $k \in \mathbb{N}^+$, then t_2 is *oversampled-harmonic* with respect to t_1. Similarly, if $kt_1 = t_2$, where $k \geq 2$, then t_2 is *undersampled-harmonic* with respect to t_1. Latency analysis for these situations is developed in [90] and summarized in Table 5.1.

Condition	Path Fragment Latency
Non-local or non-harmonic	$r_1 + t_1 + r_2 + t_2$
Local oversampled-harmonic	$r_1 + t_1 + r_2$
Local undersampled-harmonic	$r_1 + r_2 + t_2$

Table 5.1: Latency over local harmonic path fragments

Response Time Analysis

The key in adjusting object periods to meet end-to-end latency constraints is determining the relationship between object periods and response times. Response time analysis is also important in the calculation of object schedulability. The response time relationships are similar, but not identical, for tasks and messages. The analysis in this section summarizes work from [49] and [114].

Task Response Times In a system with preemption and priority-based scheduling, the worst case response time r_i for a task $o_i \in T$ depends on the computation time requirement c_i for the task itself as well as the interference from higher priority tasks on the same

resource. r_i can be calculated using the following recurrence:

$$\begin{aligned} w_i(q) &= (q+1)c_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i(q)}{t_j} \right\rceil c_j \\ r_i &= \max_q \{w_i(q) - qt_i\} \\ \forall q &= 0 \ldots q^* \text{ until } r_i(q^*) \leq t_i \end{aligned} \tag{5.1}$$

Where $j \in hp(i)$ refers to the set of higher priority tasks on the same resource. The need of evaluating the first q instances inside the busy period is caused by the uncertainty about the instance which causes the worst case response time. A lower bound on the worst case response time can be obtained by restricting the calculation to the first instance ($q = 0$). This bound is tight in case tasks complete their work within a single period, i.e. $r_i \leq t_i$ for all o_i. In this case, the formula can be simplified as:

$$r_i = c_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{t_j} \right\rceil c_j \qquad \forall o_i \in \mathcal{T} \tag{5.2}$$

Note that the term $\lceil \frac{r_i}{t_j} \rceil$ indicates the maximum number of preemptions from a higher priority task j. The numerator indicates the amount of time that the task is vulnerable to preemption, whereas the denominator indicates how often the higher priority task j is activated. The ceiling function is used since this is a worst-case analysis.

Message Response Times Worst case message response times are calculated similarly to worst case task response times. The main difference is that message transmission on the CAN bus is not preemptable. Therefore, a message o_i may have to wait for blocking time b_i, which is $\max_{j \in lp(i)} c_j$ where $lp(i)$ is the set of lower priority messages that are allocated to the same bus as o_i. Likewise, the message itself is not subject to preemption from higher

priority messages during its own transmission time c_i. The response time relationship is:

$$\begin{aligned} w_i(q) &= b_i + qc_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i(q)}{t_j} \right\rceil c_j \ (w_i > 0) \\ r_i &= \max_q \{c_i + w_i(q) - qt_i\} \\ \forall q &= 0 \ldots q^* \text{ until } r_i(q^*) \leq t_i \end{aligned} \tag{5.3}$$

Again, a lower bound on r_i can be computed by only considering the first instance ($q = 0$) and the formula is simplified as:

$$r_i = c_i + b_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i - c_i}{t_j} \right\rceil c_j \qquad \forall o_i \in \mathcal{M} \tag{5.4}$$

In calculating the number of preemptions from a higher priority message j, the difference from Equation 5.2 is that a message is not vulnerable to preemption while it is being transmitted.

5.5.4 Period Optimization Approach

From the relationships given in Equations 5.2 and 5.4, it is apparent that the response time of each object is related to the periods of higher priority objects on the same resource. Intuitively, reducing the period of an object will increase the response times of other objects with lower priorities on the same resource. The end-to-end latencies of multiple paths may be affected as a result. Also, modifying object periods also affects the utilization of the resource. Lowering the period for an object increases the utilization of the resource. Finally, lower periods also makes object schedulability more difficult.

If object periods are modified individually, then achieving convergence is difficult, since any change to one period affects many others. Instead, we concentrate on mathe-

matical programming (MP) techniques, which simultaneously consider modifications to the periods of all objects.

The benefits of a MP optimization approach are particularly relevant to the period synthesis problem. First, in assigning periods, there are a large number of interdependencies between the objects on different paths. Considering one path at a time is not guaranteed to find a feasible, let alone optimal, solution. MP approaches consider all constraints simultaneously. Next, and more importantly, MP approaches can be customized with system-specific issues by simply adding additional constraints. Whereas other solution mechanisms are brittle to changes in the problem assumptions, MP approaches can adapt to different problem assumptions or partial solutions. For example, the existence of legacy tasks and messages whose periods are fixed or otherwise restricted can be handled quite easily with additional constraints.

This section is organized as follows. First, the problem is captured with a generic mathematical programming formulation. Next, two specialized forms of mathematical programming - geometric programming (GP) and mixed-integer geometric programming (MIGP) are described. The period optimization problem is defined as an MIGP and a GP approximation is developed. Approximation error is reduced by an iterative procedure.

Mathematical Programming Formulation

The period assignment problem is defined over the following sets: the objects $\mathcal{O}$, which are partitioned into messages $\mathcal{M}$ and tasks $\mathcal{T}$, the set of resources $\mathcal{R}$, and the paths with end-to-end constraints $\mathcal{P}$. All objects $o_i \in \mathcal{O}$ have associated computation time parameters c_i, lower bounds on periods n_i, and upper bounds on periods x_i. Additionally,

messages $o_i \in \mathcal{M}$ have associated blocking times b_i. Path deadlines d_p are specified for all $p \in \mathcal{P}$. u_j are the maximum permitted utilization values for all resources $R_j \in \mathcal{R}$. The main decision variables for all $o_i \in \mathcal{O}$ are the periods t_i while the response times r_i are used as helper variables.

The problem to be solved can be formulated as follows:

$$min. \quad \sum_{o_i \in \mathcal{O}} r_i \tag{5.5}$$

$$s.t. \quad \sum_{k:o_k \in p} t_k + r_k \leq d_p \qquad \forall p \in \mathcal{P} \tag{5.6}$$

$$r_i = c_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{t_j} \right\rceil c_j \qquad \forall o_i \in \mathcal{T} \tag{5.7}$$

$$r_i = c_i + b_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i - c_i}{t_j} \right\rceil c_j \quad \forall o_i \in \mathcal{M} \tag{5.8}$$

$$r_i \leq t_i \qquad \forall o_i \in \mathcal{O} \tag{5.9}$$

$$\sum_{i:o_i \to R_j} \frac{c_i}{t_i} \leq u_j \qquad \forall R_j \in \mathcal{R} \tag{5.10}$$

$$n_i \leq t_i \qquad \forall o_i \in \mathcal{O} \tag{5.11}$$

$$t_i \leq x_i \qquad \forall o_i \in \mathcal{O} \tag{5.12}$$

The objective function can be selected according to the optimization goals. 5.5 corresponds to the minimization of average response time over all objects in the system. However, a different choice related to the extensibility of the solution can also be used. For instance, minimizing the maximum resource utilization.

Constraint 5.6 ensures that the path deadlines are met. Note that the less pessimistic path latencies from Section 5.5.3 can be substituted here when possible. Constraints 5.7 and 5.8 relate the node response times to the computation times and periods, according

to Equations 5.2 and 5.4. Constraint 5.9 adheres to the assumption that response times are lower than object periods and enforces object schedulability. Resource utilization is bounded by Constraint 5.10.

Finally, even when there are no explicit end-to-end deadlines imposing a constraint on the maximum execution periods of tasks and messages, such bounds may be specified separately – especially for feedback control applications – as in Constraints 5.11 and 5.12.

Depending on system-specific situations, additional constraints may be added that relate the periods of different objects. For instance, periods for two objects o_i and o_j may be constrained to be equal, i.e. $t_i = t_j$, or with a given oversampling ($t_i = nt_j$) or undersampling ($mt_i = t_j$) ratio (where n and m are positive integer constants). A more generic requirement might be to ensure that the objects are undersampling or oversampling with some unknown integer proportionality k between the periods. For example, $t_i = kt_j$ where $k \in \mathbb{Z}^+$. If such constraints are defined over adjacent tasks on the same resource, the less conservative analysis from Section 5.5.3 can be used.

Geometric Programming

Geometric programming (GP) is a special form of convex programming [14]. GPs have polynomial time computational complexity and can be solved very efficiently by a variety of off-the-shelf solvers. After [13], a GP in standard form is:

$$\begin{aligned} \text{minimize} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \leq 1 \quad i = 1, \ldots, m \\ & g_i(x) = 1 \quad i = 1, \ldots, p \end{aligned}$$

where $x = (x_1, ..., x_n)$ is a vector of positive real-valued decision variables. f is a set of *posynomial* functions, while g is a set of *monomial* functions. A posynomial is the sum of monomials, where a monomial function m has the following form:

$$m(x) = cx_1^{a_1} x_2^{a_2} \ldots x_n^{a_n} \qquad c > 0, a_i \in \mathbb{R}$$

If x contains both integral and real-valued decision variables, the resulting problem is a mixed-integer geometric program (MIGP). Unlike GPs, MIGPs are not convex and cannot be efficiently solved.

In this work, we make use of the gpposy [64] solver to solve GPs. Solver interfacing is handled by the Yalmip [70] framework, which can overlay a branch-and-bound approach to solve MIGP problems as well.

Mixed Integer Geometric Programming Formulation

Based on the original mathematical programming formulation, we can transform it into a mixed integer geometric program with some slight changes.

$$min. \quad \sum_{o_i \in \mathcal{O}} r_i \tag{5.13}$$

$$s.t. \quad \frac{\ell_p}{d_p} \leq 1 \qquad \forall p \in \mathcal{P} \tag{5.14}$$

$$\frac{c_i + \sum_{j \in hp(i)} z_{ij} c_j}{r_i} \leq 1 \qquad \forall o_i \in \mathcal{T} \tag{5.15}$$

$$\frac{c_i + b_i + \sum_{j \in hp(i)} z_{ij} c_j}{r_i} \leq 1 \quad \forall o_i \in \mathcal{M} \tag{5.16}$$

$$\frac{r_i}{t_i} \leq 1 \qquad \forall o_i \in \mathcal{O} \tag{5.17}$$

$$\sum_{i:o_i \to R_j} \frac{c_i}{t_i \times u_j} \leq 1 \qquad \forall R_j \in \mathcal{R} \tag{5.18}$$

$$\frac{n_i}{t_i} \leq 1 \qquad \frac{t_i}{x_i} \leq 1 \qquad \forall o_i \in \mathcal{O} \tag{5.19}$$

$$\frac{r_i}{t_j \times z_{ij}} \leq 1 \qquad \forall o_i \in \mathcal{T} \tag{5.20}$$

$$\frac{r_i}{t_j \times z_{ij} + c_i} \leq 1 \qquad \forall o_i \in \mathcal{M} \tag{5.21}$$

Constraints 5.14, 5.17, 5.18, and 5.19 are simple reformulations of their counterparts from the original formulation. z_{ij} is a new set of integer variables which captures the number of preemptions from a higher priority object j on a lower priority object i on the same resource. Note that the integrality of these variables forces the formulation to be a MIGP. Constraints 5.20 and 5.21 determine the values of these variables.

To enable this formulation to be compatible with the standard form of MIGP, we need to carry out a simple change of variables. This change of variables replaces the term $r_i - c_i$ with a new variable r'_i for all messages ($\forall o_i \in \mathcal{M}$).

Approximation

Since MIGP problems are very difficult to solve, we approximate the MIGP period optimization problem with a GP formulation. In order to cast the problem into a GP form, the interference variables z_{ij} are relaxed to real-valued variables and parameters $0 \leq \alpha_{ij} \leq 1$ are added to them. For clarity, let the approximated response time variables be s_i; then, Constraints 5.20 and 5.21 from the MIGP become:

$$\frac{s_i}{t_j(z_{ij}+\alpha_{ij})} \leq 1 \quad \forall o_i \in \mathcal{T} \tag{5.22}$$

$$\frac{s_i}{t_j(z_{ij}+\alpha_{ij})+c_i} \leq 1 \quad \forall o_i \in \mathcal{M} \tag{5.23}$$

Thus, the GP approximation consists of the objective function 5.13 with s_i in place of r_i, Constraints 5.14) – 5.19 (also with s_i in place of r_i) and Constraints 5.22 and 5.23.

If the values of all α_{ij} are 1, then the approximation is always conservative, i.e. $s_i \geq r_i$. If some $\alpha_{ij} < 1$, no such guarantees can be made. Clearly, the accuracy of the approximation depends upon the α parameters that are used.

Fixed integer harmonicity constraints can be handled directly within the GP formulation, whereas variable integer harmonicity constraints require a branch-and-bound approach with much higher complexity. However, if the number of such constraints is small, the impact on overall runtime is not prohibitive. The Yalmip framework used to solve the MPs can handle such Mixed Integer Geometric Programs without additional modifications.

Reducing Approximation Error

The α parameters in the GP formulation represent the degree of conservatism used for the approximation of the response times. Setting all $\alpha_{ij} = 1$ is a safe, but pessimistic approximation that may produce an infeasible problem instance. In this section, an iterative procedure is presented to find α parameters that preserve feasibility with reduced conservatism.

Given some set of α parameters, if the GP is feasible, optimal t_i values from the GP solution can be obtained. We can obtain the r_i values by substituting these t_i values into Constraints 5.2 and 5.4. For all $o_i \in \mathcal{O}$, let e_i represent the relative error between the estimated and actual response times, i.e. $e_i = \frac{s_i - r_i}{r_i}$. If all $e_i \geq 0$, then the optimal GP solution results in a feasible solution to the exact problem, while if all $e_i = 0$, then the GP solution is not only feasible, but optimal. If some $e_i < 0$, then the GP has underestimated some response times and Constraints 5.14 or 5.17 in the exact problem may have been violated.

An iterative procedure can be used to assign the α parameters. A new GP problem is solved during each iteration, and the e_i values are used to recalculate the α parameters for the subsequent iteration. The procedure is summarized in Algorithm 1.

The input parameter to the procedure is f, which represents the maximum permissible estimation error. At initialization, all α_{ij} are conservatively assigned to 1. Inside the loop, the GP problem is solved and the estimated response times and assigned periods are obtained. If the problem is infeasible, then all α values are scaled, and a new GP problem is solved during the next iteration. If the GP problem in the current iteration is feasible,

Input Parameter = f ; /* acceptable error bound */

forall $o_i \in \mathcal{O}$ **do**
  $\alpha_{ij} = 1$;
  ; /* conservative initialization */

while *true* **do**
  (s, t) = GP(α) ; /* solve the GP */
  if *GP is infeasible* **then**
    forall $o_i \in \mathcal{O}$ **do**
      $\alpha_{ij} = \frac{1}{2}\alpha_{ij}$; /* reduce α values */
  else
    $vior = 0$;
    $viol = 0$;
    forall $o_i \in \mathcal{O}$ **do**
      calculate r_i using fixpoint ;
      $e_i = \frac{s_i - r_i}{r_i}$; /* relative approximation error */
      if $r_i > t_i$ **then**
        $vior = vior + 1$; /* schedulability constraint violation */
      $\alpha_{ij} = \alpha_i - e_i$ tcc*α values for next iteration ensure $0 \leq \alpha_{ij} \leq 1$;
    forall $p \in \mathcal{P}$ **do**
      if $\ell_p > d_p$ **then**
        $viol = viol + 1$; /* path constraint violation */
    if $viol = 0 \wedge vior = 0 \wedge (\forall o_i \in \mathcal{O}, \max(|e_i|) < f)$ **then**
      exit ;

Algorithm 1: Iterative Period Assignment Procedure

then the exact response times are calculated with Constraints 5.2 and 5.4. The relative error e_i and possible violations to Constraint 5.17 can then be calculated. Next, α_{ij} values are adjusted based on e_i, and are saturated either at 0 or 1 if necessary. After all exact response times have been calculated, violations to path constraints 5.14 can be checked. If none of the constraints have been violated, and if the maximum absolute estimation error is lower than the limit for all objects, the procedure terminates, otherwise the next iteration is executed with the modified α values. An iteration limit may also be specified.

5.5.5 Case Studies

The period optimization approach is validated in this section with two case studies. The first is an experimental vehicle system that incorporates advanced active safety features, as described in Section 5.1.2. The second case study is a fault tolerant distributed system taken from [115].

5.5.6 Active Safety Vehicle

The architecture consists of 29 ECUs connected with 4 CAN buses, with speeds ranging from 25kb/s to 500kb/s. Worst case execution time estimates have been obtained for all tasks. Message length and bus speed is used to calculate the maximum transmission time for all CAN messages. Each ECU is allocated from 1 to 22 tasks and each CAN bus is allocated from 14 to 105 messages. The system graph contains a total of 604 links.

The deadlines are set at 300 ms for 9 of the 12 source-sink pairs, at 200 ms for two pairs, and at 100 ms for one pair. For 9 pairs of local tasks over 2 ECUs, harmonicity constraints with fixed integer constants are present. Some task and message rates are bounded

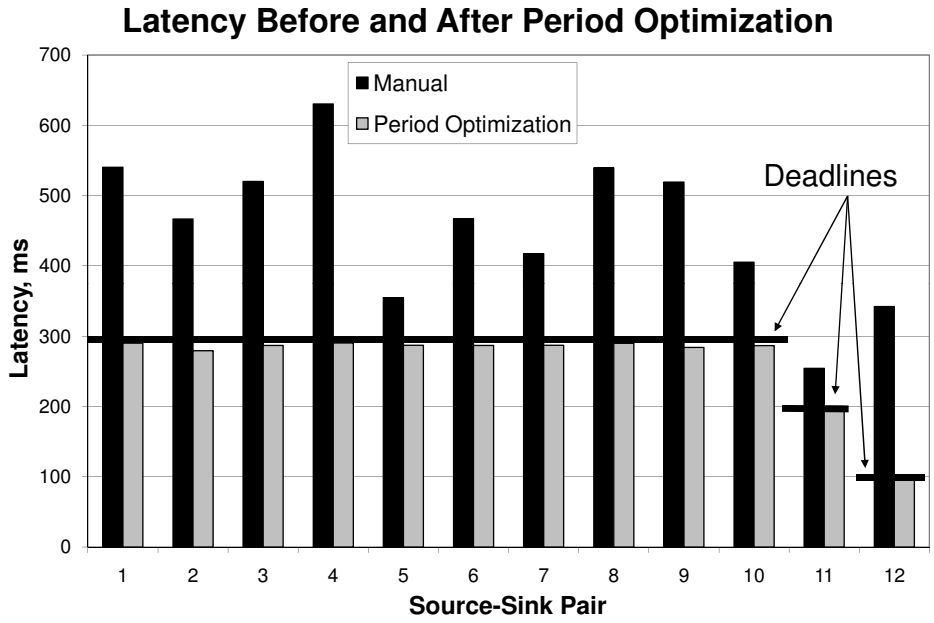

Figure 5.7: Period optimization meets all deadlines

explicitly, due to controller requirements and maximum sampling rates from sensors. To provide for future extensibility and a safety margin, maximum utilization parameters u_i from (5.18) are set at 70% for all ECUs and buses.

The system configuration used is a snapshot from an early study of the possible architecture configurations, in which the periods of task and messages had not been finalized. The preliminary manual estimates are based on designer intuition. These initial period assignments, in the worst case, do not meet any of the deadlines as shown in Figure 5.7.

Starting with all the α parameters equal to 1, we perform a GP optimization. The results of this optimization are also shown in Figure 5.7. All 222 paths between the 12 source-sink pairs meet their deadlines. The GP problem takes 24 seconds to solve on a 1.6 GHz Pentium M processor with 768 MB of RAM. The GP period assignments are quite different from the manual ones; the average period increases by 90%.

To determine the effectiveness of the iterative procedure, we can track the reduction in $\max(|e_i|)$, $\forall o_i \in O$ across several iterations. The results are shown in Figure 5.8. 15 iterations of Algorithm 1 are shown on the x-axis. The y-axis (with a logarithmic scale) shows the *maximum* absolute estimation error for the response time estimate used within the GP formulation. The *average* estimation error, not shown, drops from 6.98% to 0.009% during these same 15 iterations. Overall, the maximum estimation error is reduced by a factor of 102, while the average estimation error decreases by a factor of 780. The discrepancy between the approximated ($\sum_{o_i \in \mathcal{O}} s_i$) and actual ($\sum_{o_i \in \mathcal{O}} r_i$) objective values drops from 27.1% during the first iteration to 0.0045% during the final iteration.

Since the runtime per iteration is independent of the α values, the total solver time for 15 iterations is 6 minutes. Even though the α values are reduced below 1, (5.14) and (5.17) from the eaxct problem are not violated during any of the 15 iterations.

Finally, we can relax the 9 harmonicity constraints from fixed integer constants to integer variables. This changes the problem from a GP to a Mixed Integer GP. The *bnb* solver within Yalmip applies a branch-and-bound procedure to find the solution, and the solution time increases to 227 seconds per iteration.

Fault-tolerant Distributed System

This system is based on the example given in [115] and contains task replicas allocated to different ECUs for fault tolerance. The system consists of 43 tasks and 36 messages deployed onto an architecture with 8 ECUs and a single bus. The bus is assumed to run at 250kb/s. Initial period assignments for tasks are taken from the example, while initial message periods are assumed to be equal to the source task periods. Task and message

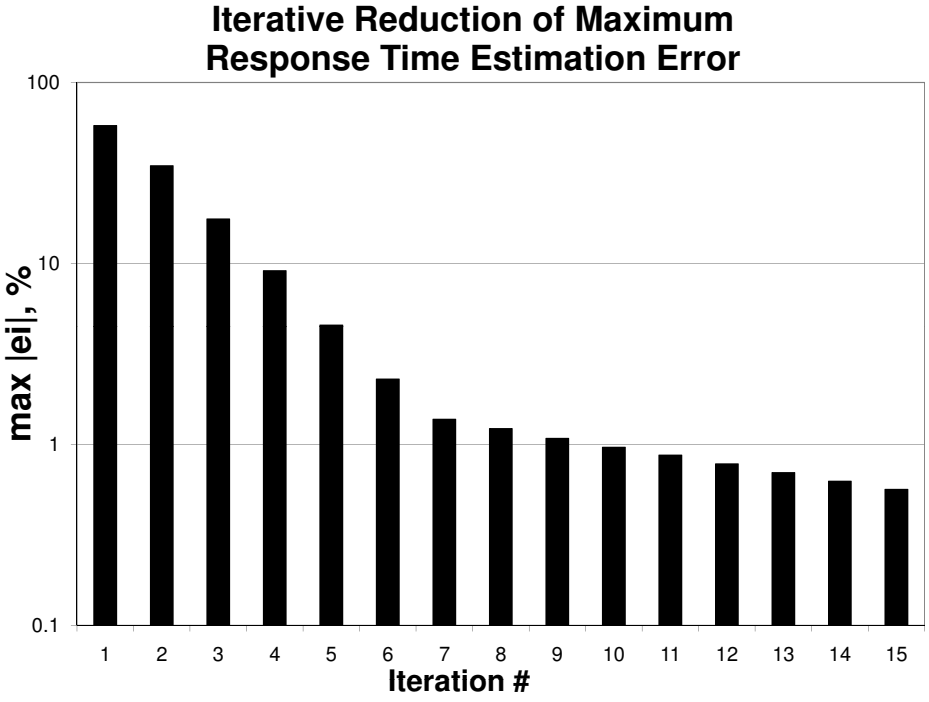

Figure 5.8: Iterative reduction in maximum estimation error

priorities are assigned using the rate monotonic rule. The initial end-to-end latencies for six paths in the system are noted.

The experiments for this system are concerned not just with meeting end-to-end delay constraints, but with reducing the path latencies as much as possible while meeting resource utilization bounds. Utilization bounds are set at 70% for each of the 9 resources, and deadlines for the six paths are set to their initial latencies.

First, we attempt to minimize average path latency on the six paths by modifying the objective function. After 15 iterations, each of which takes 1.25 seconds, the average path latency is reduced by 45%. The average utilization for the 8 ECUs is increased from 56% to 61% while the bus utilization is reduced from 74% to 52%. Next, we carry out six more experiments where we minimize each of the individual path latencies separately. The latencies for each of the six paths can be decreased an additional 17% to 63%, for a total reduction ranging between 55% and 70% from the initial latencies.

These experiments demonstrate that it is possible to customize the approach for a modified flow where the designer is interested in minimizing specific path latencies. Even without modifying allocations or priority assignments, period assignment alone is capable of significantly affecting end-to-end latencies in the system.

5.5.7 Conclusions

The continuing proliferation of distributed automotive functionality and architectures complicates the mapping process for these systems. This work provides an optimization procedure that automates the period assignment stage within mapping. First, by leveraging schedulability analysis, we develop an MIGP formulation that is applicable for

systems with run-time priority-based scheduling. Next, the MIGP formulation is approximated by a GP formulation and the approximation error between the two formulations is reduced with an iterative procedure. The approach has been applied to two case studies and shown to be efficient, accurate, and extensible. In the future, this work will be integrated with the earlier mapping stages of the design flow shown in Figure 5.4 in order to carry out joint allocation, priority assignment and period assignment. We are also considering synthesizing hybrid data-driven and periodic activation models [126] for such systems.

5.6 Conclusions

The automotive domain features active safety applications deployed on distributed architectures. In this work, a model of computation has been chosen for this class of systems and verified with respect to its accuracy at capturing the design space. The model of computation exposes the architectural non-idealities, allowing for more efficient implementations, at the cost of reduced functional verification capabilities. The automated mapping problem for such systems involves carrying out allocation, priority assignment, and period assignment. An approach based on geometric programming has been developed to automate the period assignment portion of the mapping.

Chapter 6

Design Framework

In this chapter, lessons learned from the METROPOLIS framework will be applied to drive future requirements for the METRO II framework. The case studies that have been carried out are from the multimedia [27] and automotive [125] domains, as described in Chapters 4 and 5, have used METROPOLIS for modeling. Additional case studies including [35] and [34] have also utilized this framework.

While validating the core ideas of our approach, these case studies also revealed some limitations. In this section, these limitations will be addressed by describing the proposal for the next-generation METRO II framework. One of these goals is supporting automated design space exploration with a service-based approach, more compatible with the design flow presented in Chapter 3.

6.1 Limitations of Metropolis

The case studies have uncovered three main limitations of the METROPOLIS framework: design import, handling of quantity managers, and the description of mapping.

The generality of the MetaModel language [113] leads to difficulties both for users and framework developers. For users, learning a new language is usually difficult, especially if the execution semantics are complex. From the framework developer's point-of-view, a new language requires vast amounts of support. Designing compilers, debuggers, and simulators for a new language is quite time-consuming.

The two-phase execution semantics of the MetaModel language requires that interactions with quantities must be explicitly represented. So, the simplifying assumptions made in domain-specific languages cannot be made for the MetaModel. Moreover, since quantity annotation requests are interwoven with the behavioral code, the request-making statements cannot be encapsulated into separate libraries, and the specification task is therefore complicated.

The MetaModel allows both denotational and imperative specification. Mapping is specified as synchronization constraints between events from the functional and architectural models. The two main limitations relate to the granularity at which mapping is specified as well as the restrictions on variables associated with the mapped events. First, mapping can only be specified with event-level synchronization constraints. Since there is no built-in mechanism to agglomerate events, this must be implemented by the system designer within the mapping code. The ability to export events in a structured manner would address this limitation. Second, arbitrary local variables in the scope of events can be used within the

constraints. This is an encapsulation failure and results in designs that are difficult to debug or reuse.

By focusing on the key value-added features of METROPOLIS, and addressing these limitations, we plan to make METRO II an IP-integration framework with enhanced support for PBD activities.

6.2 Metro II Features

The three main features that form the basis of the second-generation METRO II framework are based on the limitations of METROPOLIS described in Section 6.1. The three features are:

1. *Heterogeneous IP Import.* IP providers create models using domain specific languages and tools. Requiring a singular form of design entry in a system-level environment requires complex translation of the original specification into the new language while making sure that semantics are preserved. If different designs or different components within the same design can have different semantics, heterogeneity has to be supported by the new environment.

2. *Behavior-Performance Orthogonalization.* For design frameworks that support multiple abstraction levels, different implementations of the same basic functionality may have the same behavioral representation but different costs. For instance, different processors will be abstracted into the same programmable component. What distinguishes them is the performance vs. cost trade-off. Moreover, not all metrics are considered or optimized simultaneously. It should be possible to introduce perfor-

mance metrics during the design process, as the design proceeds from specification to implementation.

3. *Mapping Specification.* Mapping relates the functional and architectural models to realize the system model. Specification of this mapping must be carried out such that there is minimal modification to the functional and architectural models themselves. In addition, the mapping specification must be compatible with design flow presented in Chapter 3 to facilitate automated design space exploration.

The remainder of this section describes these three requirements in more detail.

6.2.1 Heterogeneous IP Import

Heterogeneous IP import shapes the nature of METRO II to be primarily an integration environment. There are two main challenges that have to be addressed: wrapping and interconnecting IP.

First, IPs can be described in different languages and can have different semantics that can be tightly related to a particular simulator. Importing the IP entails providing a way of exposing the IP interface. The user must have the necessary aids to define wrappers that mediate between the IP and the framework such that the behavior can be exposed in an unambiguous way.

Secondly, wrapped components have to be interconnected. Even if the interfaces are exposed in a unified way, interconnecting them is not usually a straightforward process. Data and the flow of control between IP blocks must be exposed in such a way that the framework has sufficient visibility.

6.2.2 Behavior-Performance Orthogonalization

The specification of what a component does should be independent of *how long it takes* or *how much power it consumes* to carry out a task. This is the reason why we introduce dedicated components, called *annotators* to annotate *quantities* to events.

A distinction has to be made between quantities used just to track the value of a specific metric of interest and quantities whose value is used for synchronization. For instance, time is used to synchronize actions and it is not merely a number that is computed based on the state evolution of the system. For quantities that influence the evolution of the system, special components, called *schedulers* are provided by the glue language. Schedulers are used to arbitrate shared resources.

The separation of schedulers from annotators allows for simpler specification and provides a cleaner separation between behavior and performance. As a result, instead of two-phase execution as in METROPOLIS, the execution semantics become three-phase.

6.2.3 Mapping Specification

Following the PBD approach, we want to keep functionality and architecture separate. The implementation of the functionality on the architecture is achieved in the mapping step. In order to explore several different implementations with minimal effort, the design environment needs to provide a fast and efficient way of mapping without modifying either the functional or the architectural models. The main problems to tackle are related to the specification of mapping itself and the execution semantics of the two models.

Mapping specification needs to be specified in such a way that it can easily be mod-

ified in order to facilitate design space exploration without touching either the functional or architectural models. To accomplish these goals while remaining compatible with the design flow advocated in this dissertation, mapping needs to be able to easily manipulate references to services.

The execution semantics of mapping in METROPOLIS relied on special "mapping" processes that were instantiated in the architecture. These mapping processes were completely nondeterministic in their usage of architectural services. The mapping specification was a one-to-one association between these processes and the functional processes. The functionality and the architecture were then executed concurrently with synchronization constraints present between them. Arbitrary variables in the scope of the synchronized events are allowed to be referenced by the mapping.

In the METRO II framework, we would like to reduce the complexity by not requiring special processes in the architecture only for mapping purposes and sequentially executing functionality and architecture if possible. Also, acess to variables in mapping needs to be strictly regulated in order to maintain IP encapsulation.

6.3 Metro II Execution Semantics

Like METROPOLIS, the semantics of the METRO II framework will be centered around the connection and coordination of components.

The key concept underlying METRO II is an *event*. An event is a tuple $< p, T, V >$ where p is a process, T is a tag set, and V is a set of associated values. An event denotes an action taken by a process (p). Events may be associated with annotations (T) and

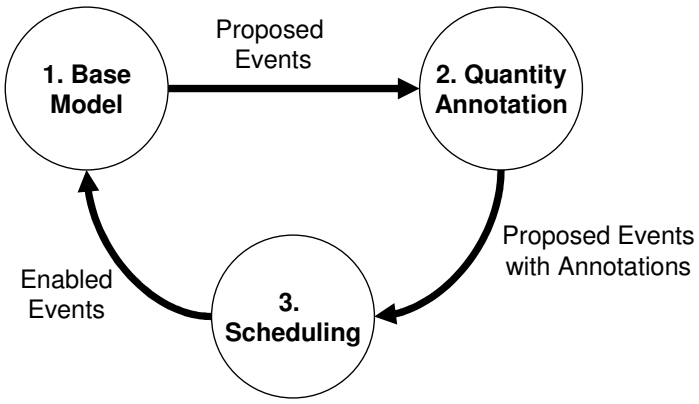

Figure 6.1: Three Phase Execution in METRO II

state (V). Annotations correspond to quantities in the design, such as time or power. State includes variables that are in the scope of an event.

Based on the treatment of events, the design is partitioned into three phases of execution. In the first phase, processes propose possible events, the second phase associates tags with the proposed events, and the third phase allows a subset of the proposed events to execute. Figure 6.1 summarizes these execution semantics.

1. **Base Model Execution.** The base model consists of concurrently executing processes that may block only after proposing events or by waiting for other processes. A process may atomically propose multiple events – this represents non-determinism in the system. After all processes in the base model are blocked, the design shifts to the second phase. The execution of processes between blocking points is beyond the control of the framework.

2. **Quantity Annotation.** In the second phase, each of the proposed events is annotated with various quantities of interest. For instance, a proposed event may be

annotated with local and global time tags. New events may not be proposed during this phase of execution.

3. **Scheduling.** In the scheduling phase, a subset of the proposed events are enabled and permitted to execute, while the remainder are blocked. At most one event per process is permitted to execute. Once again, new events may not be proposed during this stage. Scheduling may be based on the resolution of declarative constraints or on imperative code.

6.3.1 Mapping

The execution semantics of mapping involves executing mapped architectural services before their functional counterparts. When a mapped method is invoked by a functional process, the begin event of that method is initially proposed, and a phase change is permitted to occur. If this event is enabled, then the architectural service executes first, immediately followed by the invoked functional method. After this, the end event of that method is proposed, with a subsequent phase change. Both the functional method and the architectural service are executed by the functional process; there are no special mapping processes. Additionally, both the functional method and the architectural service may block internally while waiting for other processes.

The functional method is parameterized with arguments and has a return type. The architectural service is also parameterized, but the return value is not used. The correspondence between the architectural service parameters and the functional service parameters is specified at compile-time. Note that this represents a superset of the capabilities

required for the automated mapping flow described in Section 3.1.2.

6.4 Metro II Building Blocks

To simplify the designer's task of specifying models that conform to the three-phase semantics described in Section 6.3, different types of objects are defined in Metro II. First, components - the primary object for imperative specification - are described. Then, the different types of ports and connections in Metro II are described. After this, the specialized Metro II objects - constraints, mappers, annotators, and schedulers - are covered.

6.4.1 Components

A *component* is an object which encapsulates imperative code in a design, either functional or architectural. Components interface with other components via zero or more ports. There are two types of components: *atomic components* and *composite components*. An atomic component is a block specified in some language and is viewed by the framework as a black box with only its interface information exposed. A composite component is a group of one or more objects as well as any connections between them.

An atomic component with zero ports is shown in Figure 6.2. The IP encapsulated by the component is interfaced by means of a *wrapper*, which translates and exposes the appropriate events and interfaces from the IP.

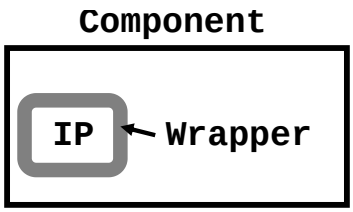

Figure 6.2: Atomic Component

6.4.2 Ports

There are two types of ports that components may have: coordination and view ports. Coordination ports are used for two-way interaction with other components by using events. View ports, on the other hand, may only expose internal events to the outside.

A *coordination* port is used to interact with other components. Each coordination port is associated with a set of methods. A *method* is a sequence of events, with a unique begin/end event pair. Variables in the scope of the begin event are method arguments. Variables in the scope of the end event are return values.

By setting constraints between events associated with coordination ports of different components, the execution of these components can be coordinated. Coordination ports are divided into three types based on the type of interaction: rendezvous ports, required ports, and provided ports.

Rendezvous Ports

Rendezvous ports can only be connected to other rendezvous ports. They are used to synchronize methods from different components. A connection between rendezvous ports implies that the begin events of all methods occur simultaneously (same valuations for all

tags) and the end events of all methods occur simultaneously as well.

The execution semantics of rendezvous ports is as follows. All components with connected rendezvous ports independently propose their respective begin events. These proposed events are allowed to occur if and only if all other begin events have also been proposed, otherwise they are blocked. Similarly, after executing the methods, all components independently propose end events and wait for all other end events to be proposed. Depending on the specifics of the connection, values in the scope of the begin/end events may be checked for equality or transferred between the components. Semantically, a rendezvous port may be viewed as two barriers.

Required Ports

Required ports are used by components to request methods that are implemented in other components. Connections are made only between a required port and a provided port.

For required ports, a component proposes a begin event and associates values with the proposed event that represent the arguments of the method being requested. When the proposed event is executed, control transfers to the component at the other end of the connection, which owns the provided port. The component waits for the end event to be executed and obtains the return values from the method. The method is executed in the same process as the caller.

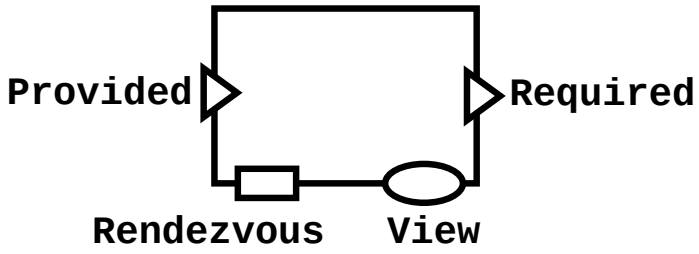

Figure 6.3: Component with 4 ports

Provided Ports

Provided ports are used by components to provide methods to other components. As stated before, connections are permitted only between a required port and a provided port. For provided ports, no separate process exists in the component to carry out the provided method. Instead, the component inherits the process from the caller component and executes the events in the provided method using that process. After the method has been executed, the process proposes the end event.

View Ports

A *view port* exposes some of a component's internal events to the outside world. These events are read-only, i.e., they cannot be blocked by outside world. View ports cannot be connected to other ports.

A component with required, provided, rendezvous, and view ports is shown in Figure 6.3.

6.4.3 Connections

Connections between coordination ports are the primary means of component interaction. One-to-one port connections are allowed between a required port and a provided port, and between a pair of rendezvous ports. Rendezvous and provided ports do not need to be connected, but each required port must be connected to a corresponding provided port.

6.4.4 Constraints and Assertions

Constraints are used to specify the design via declarative means (as opposed to imperative specification which is used in components). Assertions are used to check whether the rest of the design conforms to given requirements. Both constraints and assertions are described in terms of events: their execution, the values associated with them, and their tags. The events referenced by constraints or assertions must be exposed by means of coordination or view ports. Depending on the logic used to describe them, constraints can be enforced either by the base model or the scheduling phases of execution. Similarly, assertions may also be checked by monitors either in the base model or in the scheduling phase.

6.4.5 Mappers

Mappers relate functional methods to architectural services. The most common usage of mappers is to transform or add values to the parameters of architectural methods. For instance, a functional method may have two arguments, while the architectural service has a third argument which the functionality is unaware of. The mapper object bridges the

two together and provides the third argument.

6.4.6 Annotators and Schedulers

In METROPOLIS, the scheduling of events along with performance annotation was carried out with a special component called a quantity manager. It is difficult to have a general mechanism to handle both scenarios since different design styles are used specify both. In METRO II, these two aspects will be separated by using *annotators* and *schedulers*.

Annotators are objects that write tags to events. Each tag is determined in terms of the event, the event's values, and any parameters supplied to the annotator. Only static parameters are permitted for annotators, which may not have their own state.

Schedulers are objects that can disable proposed events based on their scheduling policy. After the annotation phase has completed, the scheduling phase begins. Based on the scheduler's local state, the proposed events, and their values and tags, scheduling occurs which can lead to the disabling of some proposed events.

6.5 Implementation

An initial implementation of the METRO II framework has been carried out in SystemC 2.2. The framework has been tested under Linux, Solaris, and cygwin.

The infrastructure is summarized in Figure 6.4. The *sc*_event and *sc*_module classes from SystemC are leveraged directly to derive the corresponding *m*2_event and *m*2_component classes. A method is characterized by begin and end events. Multiple methods are wrapped together into an interface, which is associated with ports. Components

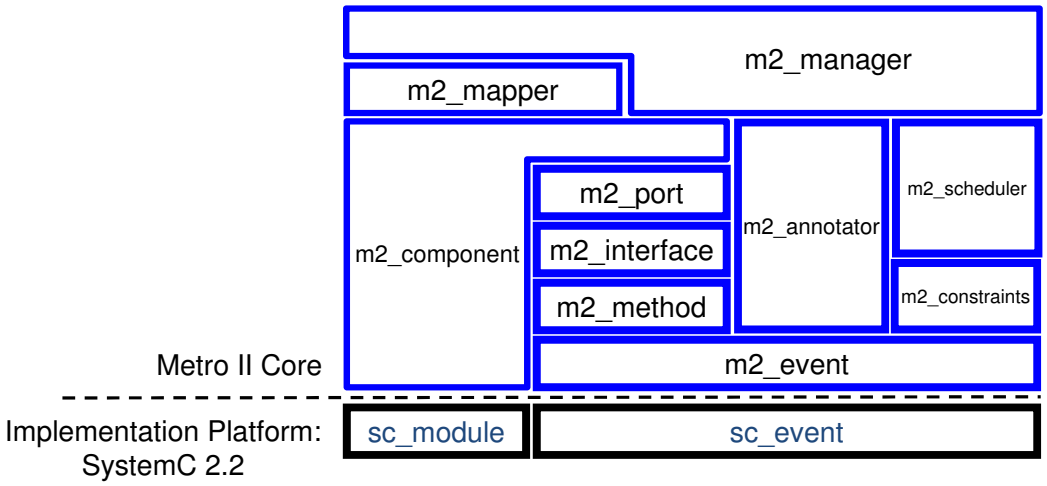

Figure 6.4: Implementation of Metro II

contain possibly multiple ports. Mappers are a special type of component which translate arguments between functional methods and architectural services. Annotators directly annotate events, while constraints are defined over them. Schedulers enable certain events after carrying out constraint resolution. The manager coordinates the execution of objects in all three execution phases.

6.6 Example: h.264 Functional Model

The h.264 [121] decoder application is the current video codec used in both HD-DVD and BluRay. It features more complex inter-frame prediction and entropy coding capabilities than previous codecs such as MPEG-4 and MPEG-2, allowing for higher compression with fewer visual artifacts. However, these techniques greatly increase the computational requirements, as shown previously in Figure 1.1.

The functional model in Metro II is based on a concurrent SystemC implementa-

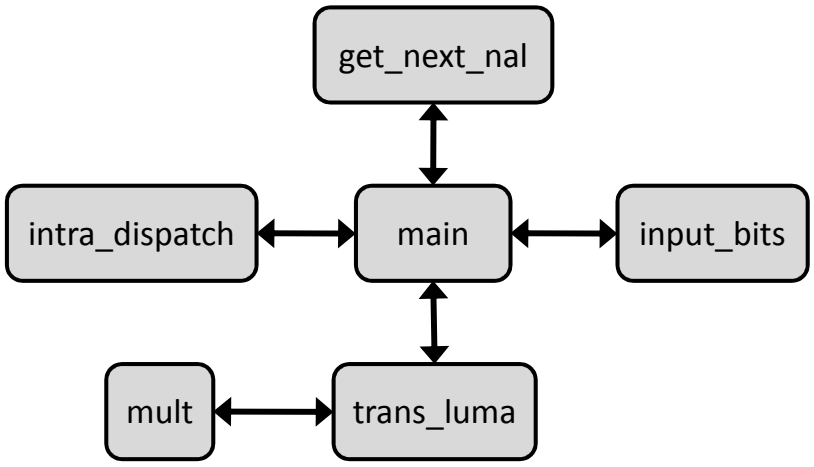

Figure 6.5: h.264 functional model

tion [124] [123]. The initial C source was obtained from [38]. The block diagram for both the SystemC and METRO II models are shown in Figure 6.5. The model consists of six concurrently executing processes, each in their own module. The processes transfer data by means of rendezvous channels (zero-place buffers). The *main* module reads the encoded h.264 stream from a file via the *input_bits* module and then utilizes the other processes to carry out different aspects of the decoding. The decoded stream is either displayed in a separate window or written to a raw file.

The aim of this example is to gauge the difficulty of importing a SystemC model into the current implementation of METRO II. To import the model, *sc_module* declarations need to be changed to *m2_component* declarations, port type declarations need to be modified, interface declarations for the ports must be changed, and SystemC blocking *wait* constructs need to be changed to their METRO II counterparts. This results in the begin and end events of each rendezvous action to be exposed to the framework and used for phase-changes. Less than 40 lines of code need to be modified from the 3,750 lines that

constitute the SystemC model.

6.7 Conclusions

The Platform-based design methodology imposes a number of requirements on system-level design frameworks. METROPOLIS represents the first attempt at such a framework. To address the limitations of METROPOLIS, in this paper we identified three main features that must be enhanced and described how the next generation METRO II framework will support them. The aim is to develop a framework that supports the import of heterogeneous IP, facilitates behavior-performance orthogonalization, and eases design space exploration. This is achieved by building an integration framework based on events with three separate phases of execution.

We are currently implementing the mapping semantics of the METRO II implementation, and developing further case studies to exercise its capabilities.

Chapter 7

Conclusions and Future Directions

The design flow espoused in this dissertation enables automated mapping for heterogeneous multiprocessor embedded systems. In this chapter, the main learnings as well as directions for future work will be presented.

7.1 Reflections

The design flow proposed in this dissertation has been applied to representative systems from the multimedia and automotive domains. Both of these areas have experienced a recent proliferation of heterogeneous multiprocessor platforms without a corresponding changes to the design flow.

The proposed approach is based on modeling, where the functionality and architecture are captured separately and mapped together in a later step. Models are transformed such that they either provide or use a common set of services. The usage of the services by the functionality dictates the MoC, whereas the cost of providing the services is captured

by the architectural model. Both the usage pattern of the services as well as the cost estimation offered by the architectural platform are crucial to developing automated mapping approaches.

In this work, automated mapping is carried out using mathematical programming techniques, which, above all, provide the flexibility of adding platform or application-specific constraints. The main challenge is ensuring that the computational complexity of mathematical programming techniques does not become prohibitive. A particularly promising technique is to approximate the exact optimization problem with another mathematical program which can be more easily solved. This technique, illustrated for the automotive domain, provides most of the benefits of exact mathematical programming techniques except bounds on solution quality.

In the multimedia domain, the architectural platforms typically natively provide the data-driven execution services required by functionality. Therefore, the concern is determining how services will be used by the functional models. In this work, a suitably expressive MoC is chosen for these systems. The key is allowing task scheduling to remain static, resulting in a simpler automation problem and lower-overhead implementations.

In the automotive domain, the use of services by functionality is relatively simpler. The challenge lies in estimating the performance impact of the services. In this work, we leverage the robustness of the applications and expose architectural non-idealities at the functional level. On the architectural side, we develop worst-case performance analysis techniques that allow us to efficiently automate the mapping problem when end-to-end latencies are the primary metric.

Finally, these case studies (and others) illustrate that many aspects of embedded multiprocessor systems modeling can be shared across different domains. Several shortcomings of the METROPOLIS design framework have been identified and solutions proposed. The proposed METRO II framework builds on SystemC and supports IP import, behavior-performance separation, and cleaner mapping specification.

7.2 Future Work

Besides future work already outlined for the multimedia and automotive domains in Chapters 4 and 5, an additional direction for developing the approach is detailed in this section. It involves leveraging the changing nature of architectural platforms to modify the simulation and automated mapping tools themselves.

Not only are embedded architectural platforms becoming more parallel, but the general-purpose platforms used to carry out the automated mapping are becoming more parallel as well. This means that the algorithms used to carry out the automated mapping need to scale with the increasing number of processors as well. Inherently sequential algorithms that enjoy favorable runtimes today will see their relative advantage diminish in the future. Simulation and optimization algorithms that can be partitioned into fairly independent units of work will fare better.

For simulation, the key is to simulate concurrent portions of the system using separate processes. This is, for instance, supported by the execution model of SystemC. However, the execution semantics of SystemC requires that the kernel execute in an interleaving manner with the processes in the model [87], reducing the possible speedup on

multiple processors. Unfortunately, the multi-phase nature of METRO II only exacerbates these issues.

For optimization, the challenge is to concurrently explore different part of the design space. Any approach can be modified to take advantage of concurrent execution resources, but randomized algorithms such as simulated annealing and mathematical programming approaches such as integer programming are particularly well-suited to handle this challenge. Even though these may not necessarily provide the best uniprocessor performance, they may exhibit better scaling.

It is indeed fortunate that parallel embedded platforms are becoming prevalent after general-purpose parallel platforms become commonplace. This affords designers the capability of exploiting parallelism on host platforms to design parallel target systems.

Bibliography

[1] Independent JPEG group, http://www.ijg.org.

[2] Intel MXP5800 Digital Media Processor Product Brief, Intel Corporation, 2004.

[3] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification*. Morgan-Kaufmann, 2007.

[4] Felice Balarin, Jerry Burch, Luciano Lavagno, Yosinori Watanabe, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of HLDVT'01*, page 129. IEEE Computer Society, 2001.

[5] 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.

[6] Felice Balarin, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the Design Automation Conference*, 1996.

[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] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, April 2003.

[9] Armin Bender. MILP based task mapping for heterogeneous multiprocessor system. In *Proceedings of EURO-DAC*, september 1996.

[10] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91, January 2003.

[11] Greet Bilsen, Marc Engels, Rudy Lauwereins, and J.A. Peperstraete. Cyclo-Static Data Flow. In *Proc. ICASSP'95*, volume 5, page 3255, Detroit, USA, 1995.

[12] Robert Bosch. CAN specification, version 2.0. Stuttgart, 1991.

[13] Stephen P. Boyd, Seung Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Optimization and Engineering*, 2006.

[14] Stephen P. Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[15] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using*

the Token Flow Model. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[16] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer- valued control streams. In *Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers*, november 1994.

[17] Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. xPilot: A platform-based behavioral synthesis system. In *SRC TechCon'05*, November 2005.

[18] Xi Chen, Abhijit Davare, Harry Hsieh, Alberto Sangiovanni-Vincentelli, and Yosinori Watanabe. Simulation based deadlock analysis for system level designs. In *Design Automation Conference*, June 2005.

[19] Pablo E. Coll, Celso E. Ribeiro, and Cid C. De Souza. Multiprocessor scheduling under precedence constraints: Polyhedral results. Technical Report 752, Opt. Online, October 12, 2003.

[20] Xilinx Corporation. http://www.xilinx.com.

[21] Xilinx Corporation. Microblaze processor reference guide.

[22] Xilinx Corporation. Fast simplex link (fsl) bus (v2.00a), December 2005.

[23] Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski, and Gerhard Woeginger. A compendium of NP optimization problems, 20 March 2000.

[24] Abhijit Davare, Jike Chong, Qi Zhu, Douglas Densmore, and Alberto Sangiovanni-Vincentelli. Classification, customization, and characterization: Using milp for task allocation and scheduling. Technical Report UCB/EECS-2006-166, EECS Department, UC Berkeley, Dec. 11 2006.

[25] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A next-generation design framework for platform-based design. In *Design and Verification Conference (DVCON07)*, February 2007.

[26] 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.

[27] Abhijit Davare, Qi Zhu, John Moondanos, and Alberto Sangiovanni-Vincentelli. Jpeg encoding on the intel mxp5800: A platform-based design case study. In *3rd Workshop on Embedded Systems for Real-time Multimedia*, Sep. 2005.

[28] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *DAC*, pages 278–283. IEEE, 2007.

[29] Abhijit Davare, Qi Zhu, and Alberto L. Sangiovanni-Vincentelli. A platform-based design flow for kahn process networks. Technical Report UCB/EECS-2006-30, EECS Department, University of California, Berkeley, Mar 2006.

[30] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[31] Tatjana Davidovic, Leo Liberti, Nelson Maculan, and Nena Mladenovic. Mathematical programming-based approach to scheduling of communicating tasks. Technical report, GERAD, December 15, 2004.

[32] 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.

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

[34] Douglas Densmore, Adam Donlin, and Alberto Sangiovanni-Vincentelli. Fpga architecture characterization for system level performance analysis. In *Design Automation and Test Europe 2006*. DATE, March 2006.

[35] Douglas Densmore, Sanjay Rekhi, and Alberto Sangiovanni-Vincentelli. Microarchitecture development via metropolis successive platform refinement. In *Design Automation and Test in Europe (DATE)*, February 2004.

[36] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101, 1998.

[37] Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *International Con-*

ference on Theory and Applications of Satisfiability Testing (SAT), LNCS, volume 6, 2003.

[38] Martin Fiedler and Robert Baumgartl. Implementation of a basic H.264/AVC decoder. Technical report, Chemnitz University of Technology, June 2004.

[39] Flexray. Protocol specification v2.1 rev. a. available at http://www.flexray.com, 2006.

[40] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL – A Modeling Language for Mathematical Programming.* The Scientific Press, South San Francisco, 1993.

[41] 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.

[42] Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. on Software Engineering*, 21(7):579–592, July 1995.

[43] 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.

[44] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Mathematics*, 5:287–326, 1979.

[45] Martin Grajcar and Werner Grass. Improved constraints for multiprocessor system scheduling. In *DATE*, page 1096. IEEE Computer Society, 2002.

[46] 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.

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

[48] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[49] M. Gonzalez Harbour, M. Klein, and J. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1), January 1994.

[50] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In Giorgio C. Buttazzo, editor, *EMSOFT*, pages 203–210. ACM, 2004.

[51] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Mat/e, K. Nishikawa, and T. Scharnhorst. Automotive open system architecture an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. In *Proceedings of Convergence 2004*, October 2004.

[52] Jörg Henkel. Closing the soC design gap. *IEEE Computer*, 36(9):119–121, 2003.

[53] Michael Horowitz, Anthony Joch, Faouzi Kossentini, and Antti Hallapuro. H.264/AVC baseline profile decoder complexity analysis. *IEEE Trans. Circuits Syst. Video Techn*, 13(7):704–716, 2003.

[54] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings - Computers and Digital Techniques*, 152(2):114–129, 2005.

[55] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems. In *Proceedings of CODES+ISSS '05*, pages 273–278, Jersey City, NJ, USA, 2005. ACM Press.

[56] Yujia Jin, Nadathur Rajagopalan Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for fpga-based soft multiprocessor systems. In *Proceedings of the 2005 International Conference on Hardware/Software Codesign and System Synthesis (CODES-05)*, pages 273–278, September 2005.

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

[58] 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.

[59] 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.

[60] 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.

[61] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda andCristina 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.

[62] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. A methodology to design programmable embedded systems - the Y-chart approach. volume 2268 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2002.

[63] 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.

[64] Kwangmoo Koh, Seungjean Kim, Almir Mutapcic, and Stephen Boyd. gpposy: A matlab solver for geometric programs in posynomial form. Technical report, Stanford University, May 2006.

[65] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[66] V. S. Anil Kumar, Madhav V. Marathe, Srinivasan Parthasarathy, and Aravind Srinivasan. Scheduling on unrelated machines under tree-like precedence constraints. In

Proceedings of APPROX-RANDOM 2005, volume 3624 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2005.

[67] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[68] Oh-Hyun Kwon. Perspective of the future semiconductor industry: Challenges and solutions. In *Keynote Address at the 44th Design Automation Conference*, June 2007.

[69] David Lammers. Shift to 65 nm has its costs. *EE Times*, July 11, 2005.

[70] J. Löfberg. Yalmip : A toolbox for modeling and optimization in MATLAB. In *Proc. of the CACSD Conference*, Taipei, 2004.

[71] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pages 1217–29, December 1998.

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

[73] Edward A. Lee. The problem with threads. *Computer: IEEE Computer*, 39, 2006.

[74] Edward A. Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[75] Gabriel Leen and Donal Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1):88–93, 2002.

[76] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *28th Annual Symposium on Foundations of CS*, pages 217–224, Los Angeles, California, 12–14 October 1987. IEEE.

[77] Leo Liberti. Compact linearization for bilinear mixed-integer problems. Technical Report 1124, Opt. Online, May 6, 2005.

[78] 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.

[79] 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.

[80] Nelson Maculan, Stella C. S. Porto, Celso Carneiro, Ribeiro Cid, and Carvalho Souza. A new formulation for scheduling unrelated processors under precedence constraints, April 28 1997.

[81] Philippe Magarshack and Pierre G. Paulin. System-on-chip beyond the nanometer wall. In *Proceedings of the Design Automation Conference*, pages 419–424, June 2003.

[82] Anthony Massa and Michael Barr. *Programming Embedded Systems*, chapter 1. O'Reilly Publishers, October 2006.

[83] K. Masselos, S. Blionas, and T. Rautio. Reconfigurability requirements of wireless

communication systems. In *IEEE Workshop on Heterogeneous Reconfigurable Systems on Chip*, 2002.

[84] Paul Master. Worldphone challenges designers. *EE Times*, September 25, 2001.

[85] A. Mihal and K. Keutzer. *Mapping Concurrent Applications onto Architectural Platforms*, chapter 3, pages 39–59. Kluwer Academic Publishers, 2003.

[86] MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Portland, OR, November 1993. IEEE CS Press.

[87] W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of systemc, 2001.

[88] Praveen K. Murthy. Multiprocessor DSP code synthesis in Ptolemy. Technical Report ERL-93-66, University of California, Berkeley.

[89] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming*. McGraw-Hill, January 1996.

[90] Marco Di Natale, Paolo Giusto, Sri Kanajan, Claudio Pinello, and Patrick Popp. Architecture exploration for time-critical and cost-sensitive distributed systems. In *Proceedings of the SAE Conference*, 2007.

[91] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[92] Anders Nilsson, Eric Tell, and Dake Liu. An accelerator architecture for programmable

multi-standard baseband processors. In *Proceedings of Wireless Networks and Emerging Technologies (WNET)*, 2004.

[93] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, September 2005.

[94] OSEK. OS version 2.2.3 specification. Available at http://www.osek-vdx.org, 2006.

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

[96] 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.

[97] 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.

[98] José Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs, May 30 1995.

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

[100] Traian Pop, Petru Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *10th International Sympo-*

sium on Hardware/Software Codesign (CODES 2002), pages 187–192, Estes Park, Colorado, USA, May 6-8 2002.

[101] Shiv Prakash and Alice C. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *J. Parallel Distrib. Comput.*, 16(4):338–351, 1992, December.

[102] Razvan Racu, Marek Jersak, and Rolf Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proceedings of the 11th Real Time and Embedded Technology and Applications Symposium*, pages 160–169, San Francisco (CA), U.S.A., March 2005.

[103] 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.

[104] Ronald A. Rohrer. DAC, moore's law still drive EDA. *IEEE Design & Test of Computers*, 20(3):99–100, 2003.

[105] M. Saksena and S. Hong. Resource conscious design of distributed real-time systems – an end-to-end approach. In *Proc. IEEE Int'l Conf on Engineering of Complex Computer Systems*, 1996.

[106] Alberto L. Sangiovanni-Vincentelli. Quo vadis sld: Reasoning about trends and challenges of system-level design. In *Proceedings of the IEEE*, volume 95, pages 467–506, march 2007.

[107] Jack Shandle and Grant Martin. Making embedded software reusable for SoCs. *EE Times*, March 1, 2002.

[108] Nagaraj Shenoy, Prithviraj Banerjee, and Alok N. Choudhary. A system-level synthesis algorithm with guaranteed solution quality. In *DATE*, page 417. IEEE Computer Society, 2000.

[109] David B. Shmoys and Éva Tardos. Scheduling unrelated machines with costs. In *SODA*, pages 448–454, 1993.

[110] 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.

[111] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[112] 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.

[113] 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.

[114] Ken Tindell, Alan Burns, and A. J. Wellings. Calculating controller area network (can) message response times. *Control Eng. Practice*, 3(8):1163–1169, 1995.

[115] Ken Tindell, Alan Burns, and Andy J. Wellings. Allocating hard real-time tasks: An NP-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[116] Mark F. Tompkins. Optimization techniques for task allocation and scheduling in distributed multi-agent operations. Master's thesis, MIT, June 2003.

[117] Jim Turley. *The Essential Guide to Semiconductors*, chapter 5. Prentice Hall Publishers, December 2002.

[118] A. van Halderen, S. Polstra, A. Pimentel, and L. Hertzberger. Sesame: Simulation of embedded system architectures for multi-level exploration. In *In Proc. of the conference of the Advanced School for Computing and Imaging (ASCI)*, pages 99–106, may 2001.

[119] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *j-CACM*, 34(4):30–44, April 1991.

[120] Z. Wang. Fast algorithms for the discrete w transform and for the discrete fourier transform. In *IEEE Transactions on Acoustics, Speech, & Signal Processing*, volume ASSP-32, pages 803 – 816, August 1984.

[121] T. Wiegand, G.J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[122] Wayne Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann, 2005.

[123] Lochi Yu, Samar Abdi, and Daniel D. Gajski. H.264 tlm in systemc for point-to-point bus platform, January 2007.

[124] Lochi Yu, Samar Abdi, and Daniel D. Gajski. Transaction level platform modeling in systemc for multi-processor designs. Technical report, UC Irvine, January 2007. This report is to help GSRC core members understand the semantics of the H.264 TLMs in SystemC that are available in the SW section.

[125] Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli, Sampada Sonalkar, Sri Kanajan, and Claudio Pinello. Design space exploration of automotive platforms in metropolis. In *Proceedings of the Society of Automotive Engineers Congress*, April 2006.

[126] Wei Zheng, Marco Di Natale, Claudio Pinello, Paolo Giusto, and Alberto Sangiovanni-Vincentelli. Synthesis of task and message activation models in real-time distributed automotive systems. In *Proc. of Design Automation and Test, Europe*, 2007.

[127] Qi Zhu, Abhijit Davare, and Alberto Sangiovanni-Vincentelli. A semantic-driven synthesis flow for platform-based design. In *Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'06)*, July 2006.