ABSTRACT

Title of dissertation: PROFILE- AND INSTRUMENTATION-DRIVEN METHODS FOR EMBEDDED SIGNAL PROCESSING

Ilya Chukhman, Doctor of Philosophy, 2015

Directed by: Professor Shuvra S. Bhattacharyya (Chair/Advisor),
Dr. Peter Petrov (Co-advisor),
Department of Electrical and Computer Engineering,
and Institute of Advanced Computer Studies

Modern embedded systems for digital signal processing (DSP) run increasingly sophisticated applications that require expansive performance resources, while simultaneously requiring better power utilization to prolong battery-life. Achieving such conflicting objectives requires innovative software/hardware design space exploration spanning a wide-array of techniques and technologies that offer trade-offs among performance, cost, power utilization, and overall system design complexity. To save on non-recurring engineering (NRE) costs and in order to meet shorter time-to-market requirements, designers are increasingly using an iterative design cycle and adopting model-based computer-aided design (CAD) tools to facilitate analysis, debugging, profiling, and design optimization.

In this dissertation, we present several profile- and instrumentation-based techniques that facilitate design and maintenance of embedded signal processing systems:

1. We propose and develop a novel, translation lookaside buffer (TLB) preloading technique. This technique, called context-aware TLB preloading (CTP), uses a

synergistic relationship between the (1) compiler for application specific analysis of a task's context, and (2) operating system (OS), for run-time introspection of the context and efficient identification of TLB entries for current and future usage. CTP works by (1) identifying application hotspots using compiler-enabled (or manual) profiling, and (2) exploiting well-understood memory access patterns, typical in signal processing applications, to preload the TLB at context switch time. The benefits of CTP in eliminating inter-task TLB interference and preemptively allocating TLB entries during context-switch are demonstrated through extensive experimental results with signal processing kernels.

2. We develop an instrumentation-driven approach to facilitate the conversion of legacy systems, not designed as dataflow-based applications, to dataflow semantics by automatically identifying the behavior of the core actors as instances of well-known dataflow models. This enables the application of powerful dataflow-based analysis and optimization methods to systems to which these methods have previously been unavailable. We introduce a generic method for instrumenting dataflow graphs that can be used to profile and analyze actors, and we use this instrumentation facility to instrument legacy designs being converted and then automatically detect the dataflow models of the core functions. We also present an iterative actor partitioning process that can be used to partition complex actors into simpler entities that are more prone to analysis. We demonstrate the utility of our proposed new instrumentation-driven dataflow approach with several DSP-based case studies.

3. We extend the instrumentation technique discussed in (2) to introduce a novel tool for model-based design validation called dataflow validation framework (DVF). DVF addresses the problem of ensuring consistency between (1) dataflow properties that are declared or otherwise assumed as part of dataflow-based application models, and (2) the dataflow behavior that is exhibited by implementations that are derived from the models. The ability of DVF to identify disparities between an application's formal dataflow representation and its implementation is demonstrated through several signal processing application development case studies.

PROFILE- AND INSTRUMENTATION- DRIVEN METHODS FOR EMBEDDED SIGNAL PROCESSING

by

Ilya Chukhman

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Dr. Peter Petrov, Co-Advisor
Professor Gang Qu
Professor Steven Tretter
Professor Rance Cleaveland, Dean's Representative

© Copyright by
Ilya Chukhman
2015

Dedication

To my friend Samarth Gupta

Acknowledgments

I would like to express my sincere gratitude to Prof. Shuvra Bhattacharyya, my advisor for the last 4 years, for his mentoring, guidance, and encouragement. I am especially grateful to him for introducing me to the interesting signal processing problems being pursued in the DSPCAD group, sponsoring my travels to conferences, and enabling me to become a better writer through his rigorous review of my writing and helpful suggestions. He gave me the freedom to develop and pursue my own ideas while providing insights and direction to shape those ideas into useful research. Through all of this and much more, Prof. Bhattacharyya has shown me how to be an exemplary advisor to PhD students.

I also want to thank my co-advisor, Dr. Peter Petrov, who guided my research for the first 3 years in the PhD program. He introduced me to interesting research problems in embedded systems, worked very closely with me in generating and analyzing results, and gave me first exposure to the art of writing scientific papers. I will look back fondly on my Friday evenings spent in Dr. Petrov's office discussing research results late into the evening.

I am also very thankful to the members of my PhD dissertation committee – Prof. Gang Qu, Prof. Steven Tretter, and Prof. Rance Cleaveland for agreeing to serve on the committee, reviewing this thesis, and providing valuable feedback.

I further wish to express appreciation to Dr. William Plishker and Dr. Chung-Ching for helping me get started in the DSPCAD group and introducing me to the concepts, tools, and methodologies used in the group.

It has been a pleasure working with many excellent members of the DSPCAD group, including Dr. Hsiang-Huang Wu, Dr. Inkeun Cho, Dr. Lai-Huei Wang, Dr. George Zaki, Dr. Zheng Zhou, Kishan Sudusinghe, Shuoxin Lin, Scott Kim, Yang Jiao, Haifa Ben Salem, Yanzhou Liu, Lin Li, and Kyunghun Lee. Thank you for the many fruitful discussions and collaborations that you have made possible.

I would like to thank Chacha (Rajeev), Chachi (Renu), and Eena Gupta, my Indian family. They were there to share the joy and celebrate my happiest moments, and their unequivocal love and support enabled me to persevere through the most difficult times during my PhD studies.

My two sisters, Yuliya and Inessa, were always there to support me. Their love and understanding made the trials and tribulations of the long path much more tolerable. For that, I am very grateful.

I would like to thank my parents for demonstrating the necessity of hard work, entrenching the value of education, and instilling the confidence needed to pursue my goals. Their love and support continues to drive me to achieve things I never thought would be possible.

Lastly, but most importantly, I would like to thank my wife, Zhanna. She has supported me throughout my studies, made numerous sacrifices to allow me to achieve my goals, and exhibited incredible patience in allowing me to explain my research. Her unwavering love and confidence served as the bedrock during my PhD studies.

The research underlying this thesis was supported in part by the U.S. Air Force Office of Scientific Research, and U.S. National Science Foundation.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AADL	Analysis & design language
ADEC	ADPCM decoder
ADPCM	Adaptive differential pulse-code modulation
AENC	ADPCM encoder
ASIC	Application specific integrated circuit
ASR	Automatic speech recognition
AT	Acoustic tracking
BDF	Boolean dataflow
BMP	Bitmap
CAD	Computer-aided design
CAM	Content-addressable memory
CFDF	Core functional dataflow
CMS	Compact muon solenoid
CSDF	Cyclo-static dataflow
CTP	Context-aware TLB preloading
DBD	Dataflow-based design
DDD	Data dependent dataflow
DFF	Dataflow fault-tolerant
DFI	Dataflow instrumentation
DFT	Discrete fourier transform
DICE	Dspcad integrative command line environment
DMCS	DVF monitoring code segments
DSP	Digital signal processing
DTW	Dynamic time warping
DVF	Dataflow validation framework
EIDF	Enable-invoke dataflow
EJ	Extended jacobian
ELS	Extended live set
EPF	ELS preload function
FDCT	Forward discrete cosine transform
FFT	Fast fourier transform
FIFO	First-in, first-out
FPGA	Field programmable gate array
FSM	Finite state machine
GPP	General purpose processor
GPU	Graphics processing unit
GST	Generalized schedule tree
HCBP	Hardware-counter based profiling
HSDF	Homogeneous synchronous dataflow
IAS	Instrumentation-augmented scheduler
IBV	Instrumentation-based validation
IDCT	Inverse discrete-cosine transform
IDF	Integer-controlled dataflow
IR	Intermediate representation

ISA	Instruction set architecture
KPN	Kahn process network
LDA	Linear discriminant analysis
LHC	Large hadron collider
LIDE	Lightweight dataflow environment
LPC	Linear-prediction coding
MAC	Multiply-and-add
MARTE	Modeling and analysis of real-time and embedded systems
MBD	Model based design
MFCC	Mel Frequency Cepstral Coefficient
MMUL	Matrix multiplication
MMU	Memory management unit
MoC	Model of Computation
MPPA	Multiprocessor profiling architecture
MPSoC	Multiprocessor system-on-chip
MSM	Most specialized model
MUT	Module under test
NRE	Non-recurring engineering
OS	Operating system
PA	Physical address
PDSP	Programmable DSP
PE	Processing element
PGO	Profile guided optimization
PTE	Page table entries
RVC	Reconfigurable video coding
SDF	Synchronous dataflow
SIT	Software instrumentation tool
SNR	Signal-to-noise
SoCDMMU	System-on-a-chip dynamic memory management unit
SOR	Successive over-relaxation
TLB	Translation lookaside buffer
TRI	Lower TRIangular transformation
UML	Unified modeling language
VA	Virtual address
VLIW	Very-large instruction word
VM	Virtual memory
V & V	Verification and validation
WCET	Worst-case execution time

Chapter 1: Introduction

1.1 Overview

Modern embedded systems for digital signal processing (DSP) run increasingly sophisticated applications that require expansive performance resources, while simultaneously requiring better power utilization to prolong battery-life. Achieving such conflicting objectives requires innovative design trade-offs across the entire system design space.

Software/hardware design space exploration can span a wide array of techniques and technologies that offer trade-offs among performance, cost, power utilization, and overall system design complexity. Performance and correctness evaluation during every stage of the design process is essential in mitigating possible cascading problems [1]. However, evaluation has become significantly more complex, along with the systems themselves. As a result, designers have created various tools to facilitate analysis, debugging, and profiling. Although such tools vary in their design, many rely on instrumentation to achieve their objectives.

Instrumentation in this context is frequently achieved by statically or dynamically adding to a system extra components whose sole purpose is to enable analysis and evaluation [2]. The added instrumentation can then be used to monitor the system for the purposes of profiling, performance analysis, and optimization; error detection, and debugging; and quality assurance, and testing.

Intrusiveness is an important metric used to measure the impact of the added instrumentation on the original system. For example, extra code can increase the execution time of the program being evaluated or affect the cache behavior by altering the order of the memory locations being accessed. Such intrusive behavior may not be appropriate for certain types of analysis and requires the designer to ensure that the added instrumentation does not perturb the behavior of the system being evaluated to the extent where the analysis becomes inaccurate.

Profiling a system's performance enables the designer to identify performance bottlenecks. A variety of profiling tools have been created that manually (or automatically) instrument a system by inserting facilities that record and output performance information. For example, *prof*, a popular UNIX profiling tool, inserts extra instructions during compilation that then extract run-time timing information as the program executes [3]. By identifying computationally intensive portions of code (or hotspots), the designer can focus on optimizing the parts of the system that will contribute most to the overall system performance.

The increasing number of components and features being incorporated into current embedded systems often results in having multiple teams working on different aspects of the design. Ensuring that the overall system works correctly becomes more difficult as the number of teams and the size of the system increase. Thus, in addition to performance evaluation, designers are increasingly using computer-aided design (CAD) tools to facilitate verifying and validating (V&V) the final implementation.

Existing CAD tools offer numerous opportunities to optimize and configure a signal processing system to meet the design goals. Most tools, however, are specific to a particular language, compiler, OS, and/or processor. Thus, while these tools can be used to design and optimize systems, the non-recurring engineering (NRE) costs associated with such a traditional design approach make it increasingly expensive to maintain and upgrade the system. NRE refers to one-time costs to research, develop, design, and test a new product. In signal processing systems, NRE costs can include the cost to research available platforms, the cost of the analysis to create a design that meets the requirements, and the cost to implement signal processing applications on the selected platform. The specificities of a traditional design result in the designer having to go through much of the development process for a new iteration of the system and limits the ability to leverage previous development investments.

To amortize development costs, designers are increasingly using a model based design (MBD) approach, which allows a designer to decompose a complex system into simpler sub-functions. Refining the original algorithm to a formal model enables strong analysis and optimization that can identify various forms of parallelism, derive efficient schedules for computational tasks, and allocate resources effectively. In addition, many existing tools facilitate re-targeting an application to a different platform by automatically generating platform-specific code.

1.2 Contribution

In this dissertation, we develop profile- and instrumentation-driven techniques to improve the design of signal processing applications on embedded systems. We first focus on a traditional design challenge by developing a new compiler-assisted profile-based technique to improve the performance of one of the critical operations in embedded systems, virtual address translation.

In the next two parts of the dissertation, we investigate the MBD process and develop several techniques to accelerate dataflow based design (DBD), a specific type of MBD. In the second part of the dissertation, we develop a dataflow instrumentation technique and use it with traditional profiling tools to facilitate converting of legacy designs to DBD semantics. Finally, we enhance the instrumentation technique to create a framework that can be used to validate dataflow properties in DBD applications.

1.2.1 TLB Interference Reduction in Multi-tasked Systems

Rapid system responsiveness and execution time predictability are of significant importance for a large class of real-time embedded systems. Multitasking leads to interference in the shared processor resources such as caches and translation lookaside buffers (TLBs). Such interference in turn results in not only deteriorated performance, but more importantly for some applications, highly sub-optimal worst-case execution time (WCET) estimates due to the unpredictability of interference. In our first contribution, we develop a methodology for task-aware D - TLB *interference reduction* and *preloading* through an *application-specific task's state introspection* at context-switch time for embedded multi-tasking.

We address the problem of precisely identifying and loading into the TLB the set of memory mappings that will be used by a computational task immediately after it is allocated to the CPU and before it is preempted again. This problem is impossible to solve through compiler-only or OS-only techniques. The compiler has static knowledge about a program, including which registers are used for array indices, array sizes, and memory access patterns. However, the compiler lacks run-time knowledge, such as current location in the code, and information about the platform. This type of run-time information is available to the operating system. Thus, combining the static information, available to the compiler, with the run-time information, available to the operating system, enables deriving the set of memory mappings that will be needed by a process after preemption. Our contribution addresses this problem through a synergistic cooperation between (1) the compiler, for an application-specific analysis of the task's context; and (2) the OS, for a run-time introspection of the context and an efficient identification of TLB entries of current (live) and "near-future" usage.

Signal processing programs often contain large numbers of operations involving vectors and matrices that are implemented using loops. Such linear algebra operations exhibit well-understood memory access patterns and are highly amenable to compiler optimizations. Optimizing these computationally intensive sections of code (or hotspots) can be achieved with profile guided optimizations (PGO) [4]. In our approach, we first identify hotspots using profiling and then optimize them with our proposed new TLB preloading methodology. The set of extensive experimental results with signal processing kernels demonstrates the effectiveness of the proposed technique in eliminating inter-task TLB interference and preemptively allocating TLB entries during context-switch.

1.2.2 Model Detection and Actor Partitioning

Dataflow modeling offers a myriad of tools to improve optimization and analysis of signal processing applications, and is often used by designers to help design, implement, and maintain systems on chip for signal processing [5]. However, maintaining and upgrading legacy systems that were not originally designed using dataflow methods can be challenging. Designers often convert legacy code to dataflow graphs by hand, a process that can be difficult and time consuming.

We develop a method to facilitate this conversion process by automatically detecting the dataflow models of the core functions from bodies of legacy code. First, we introduce a method for instrumenting dataflow graphs that can be used to profile, measure various statistics, and extract run-time information. Second, we use this instrumentation technique to demonstrate a method that facilitates the conversion of legacy code to dataflow-based implementations. This method operates by automatically detecting the dataflow model of the core functions being converted. Third, we present an iterative actor partitioning process that can be used to partition complex actors into simpler subfunctions that are more prone to analysis techniques. We demonstrate the utility of the proposed approach on several signal processing applications.

1.2.3 Validation of Dataflow Applications

Dataflow based designs enable a designer to take advantage of dataflow properties to effectively tune the system in connection with functionality and different performance metrics. However, a disparity in the specification of dataflow properties and the final implementation can lead to incorrect behavior that is difficult to detect. This motivates the problem of ensuring consistency between dataflow properties that are declared or otherwise assumed as part of dataflow-based application models, and the dataflow behavior that is exhibited by implementations that are derived from the models.

We address this problem by introducing a novel dataflow validation framework (DVF) that is able to identify disparities between an application's formal dataflow representation and its implementation. We demonstrate the utility of our DVF through design and implementation case studies involving an automatic speech recognition application, a JPEG encoder, and an acoustic tracking application.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. In the next chapter, we provide background on virtual address translation and TLB operation, formal specifications, dataflow models, and verification. In chapter 3, we describe a compiler-OS synergistic TLB preloading technique that enables significant reduction of TLB misses in signal processing applications. We describe in chapter 4 a dataflow instrumentation technique that is useful to obtain trace information and apply it to facilitate the process of converting legacy designs to DBD by detecting instances of well-known dataflow models.

In chapter 5, we introduce a dataflow validation framework that can identify disparities between an application's formal dataflow representation and its implementation. Finally, conclusions and directions for future work are presented in chapter 6.

Chapter 2: Background

In this chapter, we provide background information on multiple topics that form the basis of this dissertation. First, we discuss several profile- and instrumentation-based CAD tools. We then describe embedded system design strategies, followed by an introduction to dataflow modeling concepts and notation that will be used in chapter 4 and chapter 5. We then introduce several other formal specifications that, like dataflow, can be used to describe a system. Lastly, we provide the basics of virtual address translation and discuss the importance of the TLB, which are relevant in the next chapter.

2.1 Instrumentation and Profiling

As discussed in chapter 1, instrumentation and profiling tools are used by designers to facilitate analysis, debugging, and performance evaluation. In this section, we describe several profile- and instrumentation-based techniques used by popular CAD tools.

Software instrumentation tools are usually classified based on how and when instrumentation code is inserted inside a program. Static instrumentation tools insert code before, during, or after compilation, but before program execution. Source-to-source transformations add instrumentation code directly to the program's source. This technique can enable measuring parallelism [6], evaluating the complexity of signal processing algorithms [7], and conducting performance estimation of SoC designs [8].

One of the most popular static instrumentation tools (`prof` on Unix and `gprof` on Linux) enables profiling of programs. It works by adding instrumentation code to the program, as it is being compiled, that extracts performance information when the program executes [3]. When the instrumented program executes, the added monitoring routines collect and output execution times of different functions, which can then be used in an analysis to determine which portions of the program consume most of the processing cycles.

Several tools have been created that add instrumentation to object code, a process that is more easily extensible to different compilers and languages than source- and compiler-based techniques. This technique is used by the IBM Rational Purify tool to identify memory leaks and access errors [9].

In addition to static instrumentation, several modern tools have begun to use dynamic instrumentation. Dynamic instrumentation is inserted while the program is running, and is inserted between the executing program and the operating system or host machine. For example, both Pin [10] and DynInst [11] use Linux `ptrace`, a part of the Linux kernel, to monitor the execution of every instruction according to the instrumentation code. DynamoRio uses a dynamic instrumentation technique called library interposing in which an instrumentation library is interposed between the program and the actual library, allowing examination and monitoring of library function calls [12].

The last class of dynamic instrumentation tools use a just-in-time-compilation technique. Such frameworks, including Valgrind, intercept program elements that are about to execute and recompile the application code with the inserted instrumentation [13].

In addition to the analysis tools described above, several FPGA and SoC profiling tools have been created that help evaluate system level design decisions, such as the mapping of applications to different SoC architectures. For example, Wang, Li, and Zhao propose a novel system design approach that uses profiling to map embedded media streaming applications to heterogeneous SoC systems [14]. Profiling enables identifying critical sections that benefit when mapped to higher performing components, thereby increasing overall system performance.

The authors of [15] present a memory-trace profiling tool that enables performance and memory access analysis of embedded systems. The tool is demonstrated on a hardware/software design for H264/AVC video decoding. The authors demonstrate how the tool can be used to evaluate the performance of system components with different design options (e.g., different cache configurations). Such analysis can then aid with scheduling and mapping of functionality onto the available platforms.

There have been several FPGA-based techniques that enable profiling SoC designs. Unlike the software tools described above, profiling FPGAs can be done with the use of hardware-counter based profiling (HCBP) tools [16]. HCBP tools utilize on-chip hardware counters to monitor specific events that occur during the run-time of an application. These counters can then be queried to conduct performance analysis. Chen et al. introduce a multiprocessor profiling architecture (MPPA) for profiling MPSoC embedded systems [17]. Using MPPA, designers can collect low level events of the target system by automatically adding extra profiling hardware to the design.

2.2 Embedded System Design Strategies

Software/hardware design space exploration can span a wide-array of techniques and technologies that offer trade-offs among performance, cost, power utilization, and overall system design complexity. A common design methodology entails optimizing individual layers within a system. While this approach can achieve exceptional results, it rarely translates to future iterations of the design. Thus, practitioners are increasingly using MBD methods to design complex systems, since MBD facilitates correctness verification and can be used to automatically generate implementations for new platforms [18, 19]. In this section, we first describe different design decisions and optimization techniques that can be applied in the various layers of a system. We then describe MBD methodologies and their benefits in Section 2.3.

At the highest layer of abstraction, problems are addressed in algorithmic terms. Over the years, numerous innovations at the algorithmic layer of signal processing have greatly improved the output quality and simultaneously, reduced the computational requirements. Examples of such innovations include the Fast Fourier transform (FFT); multi-rate signal processing; numerous filtering algorithms; and adaptive algorithms, such as linear predictive coding (LPC) and Kalman filtering [20]. Instead of relying on features of a given programming language, signal processing practitioners may apply mathematical tools to significantly reduce the quantity of computations necessary to solve a given problem without sacrificing quality.

At the next layer of abstraction are languages used to implement signal processing algorithms. Whether it is hardware design languages, such as Verilog and VHDL, which are used to design application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs), or whether it is software programming languages, such as assembly, C, and C++, which are used to program general purpose processors (GPPs), graphics processing units (GPUs), and programmable digital signal processors (PDSPs), it is important to choose the appropriate language to solve the given problem. Programs were initially written in assembly language, where a simple program could involve thousands of lines of assembly code. The performance of initial compilers, which convert high-level programs into assembly language, was considered inferior to human-generated assembly programs. However, as the complexity of problems and the resulting length of assembly programs increased, high-level languages became increasingly preferred by designers. Also, compiler technology has undergone a significant transformation within the last 40 years, which has resulted in compilers that can produce output that is similar in quality to that of hand-crafted assembly language code.

The complexity of modern signal processing systems has required the use of operating systems (OS) to help manage multiple parallel jobs and interface with various peripherals and sensors. As a result, modern signal processing applications typically run within the context of an OS, thereby allowing the programmer to rely on the OS to provide the interface to features such as virtual memory (VM), scheduling, and process management, while simultaneously abstracting away the complexity required to enable these features.

Selecting the appropriate computing platform to satisfy a set of system requirements remains a challenge in the design of signal processing systems. Over the years, designers have created numerous platforms that utilize advanced micro-architecture techniques including, pipelining, caching, out-of-order execution, and branch-prediction, to significantly improve performance at the micro-architecture layer.

At the instruction set architecture (ISA) layer, several data-level parallelism enhancements, such as SIMD and vectorization, have resulted in significant throughput improvements in signal processing applications. In addition, several PDSPs offer very-large instruction word (VLIW) platforms that provide instruction-level parallelism with the aid of compilers. However, while active research to improve micro-architectures continues to be pursued at commercial companies [21], a system designer can rarely modify the micro-architecture on an existing platform. Instead, signal processing practitioners can design custom hardware with FPGAs and ASICs for scenarios when the performance offered by the available programmable processors is insufficient to meet design constraints.

2.3 Dataflow Modeling

MBD approaches involve decomposing the original algorithm to formal models, which are more amenable to analysis. Many existing tools and languages use MBD approaches to facilitate iterative designs by enabling re-targeting of applications to different platforms (e.g., see Ptolemy [22], CAL [23], and DIF [24]). Dataflow modeling is a type of MBD methodology that is commonly used with signal processing systems. In this section, we describe dataflow modeling and supported formal dataflow models.

2.3.1 Formal Description

A dataflow graph G is an ordered pair (V, E), where V is a set of vertices (or actors), and E is a set of directed edges. A directed edge $e = (v1, v2) \in E$ is an ordered pair of a source vertex $v_1 \in V$ and a sink vertex $v_2 \in V$. Actors represent computational modules, while edges represents first-in, first-out (FIFO) communication links between actors. The complexity of the actors depends on the application, where actors may represent simple arithmetic operations, such as multiply-and-add (MAC), or more complex operations such as a JPEG encoder.

When an actor v executes or *fires*, it consumes zero or more tokens from each input edge and produces zero or more tokens on each output edge. For each edge e, the numbers of data values produced by the source actor of e and consumed by the sink actor of e are defined as the *production rate* (denoted by $prd(e)$) and *consumption rate* (denoted by $cns(e)$), respectively. Given an edge e, we denote its source actor as $src(e)$ and its sink actor as $snk(e)$. The set of ports for actor a is defined as $port(a) = port_{in}(a) \cup port_{out}(a)$, where $port_{in}(a) = \{e | e \in E \wedge snk(e) = a\}$ is the set of input ports, and $port_{out}(a) = \{e | e \in E \wedge src(e) = a\}$ is the set of output ports for actor a.

In this thesis, the “dataflow rate” symbols $prd(e)$ and $cns(e)$ can be viewed as annotations that are available to provide known or derived information about the numbers of tokens that are produced and consumed during individual actor invocations. In general, the mathematical form of $prd(e)$ and $cns(e)$ depend on the enclosing actor, the associated actor port, the underlying dataflow model of computation, and how much information is known or has been derived about the associated actor and port. For example, as explained below, synchronous dataflow actors have dataflow rates that are expressed in the form of constant integers, while cyclo-static dataflow actors have dataflow rates that are expressed as integer vectors. For some dynamic dataflow models, little may be known apriori about $prd(e)$ and $cns(e)$; in this case, the associated annotations would contain correspondingly little information. For example, $prd(e) \in \{0, 1\}$ may indicate that the number of tokens produced on e during a given firing may be 0 or 1, and that this number may vary from one firing to the next. Thus, our use of $cns(e)$ and $prd(e)$ is an abuse of notation, where the actual form of the represented rates depends on context that includes the actor, port, and overall design scenario. With this abuse of notation, we avoid cluttering our notation with multiple symbols that have same general meaning (dataflow rate characterization) in the new kinds of DBD analysis methods that we develop in this thesis.

An actor's dataflow behavior is a function of the dataflow model to which the actor conforms [5]. Actors exhibiting static behavior have fixed consumption and production rates, while in general production and consumption rates can vary across distinct firings of the same actor. In the next subsection, we will describe the dataflow models considered in this dissertation and describe their consumption and production behavior.

2.3.2 Dataflow Model Comparison

Many dataflow models of computation have been developed for both actors and graphs to enable realization of a wide variety of applications and design techniques in DBD environments [5]. Some of the associated actor models are highly restrictive and can be used to infer powerful system-wide properties, which help, for example, to optimize scheduling and memory management. Other actor models offer flexibility that enables their use in a variety of applications not conducive to restrictive models; however, the same flexibility makes it more difficult to reason about these models and analyzing them becomes much more complex (e.g., see [25]).

The different dataflow models of computation ("dataflow MoCs" or "dataflow models") that have been created over the years are not all easily related to one another. As a result, it is not always possible to compare dataflow models in order to, for example, determine whether or not one model A is more restrictive than a different model B. Here, by "more restrictive", we mean intuitively that the class of computations that can be represented by A is a proper subset of the class that can represented by B. However, there are useful groups of models that can be compared in such a way — i.e., that can be compared in terms of this restrictivity notion. We refer to such a group (subset) of dataflow models as a *comparable* group. In this dissertation, we limit our discussion to comparable groups, and demonstrate our methods on a *comparable* group that is supported in the lightweight dataflow environment (LIDE) [26], which is a specific DBD tool that we have employed to validate and experiment with many of the ideas explored in this thesis. Extending these methods to groups that are not comparable is a useful direction for future investigation.

Figure 2.1 shows three classes of dataflow models within the universe of dataflow models that are currently supported in LIDE: data independent, control-based, and mode-based. As illustrated in Figure 2.1, data independent models are most restrictive and mode-based models are least restrictive. Outer classes in Figure 2.1 generalize the inner classes, so that for example, control-based models also contain data independent actors, and mode-based models also contain control-based actors.

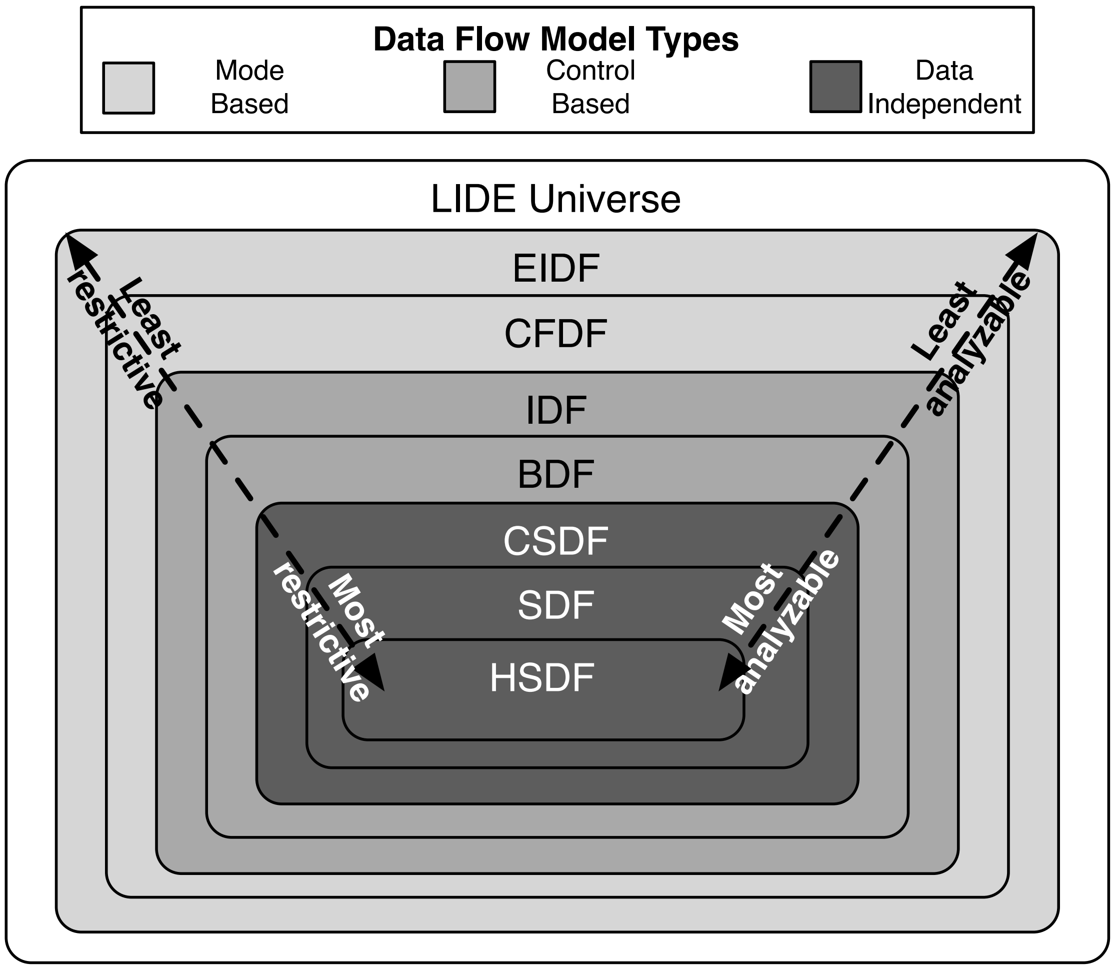


Figure 2.1: A classification of dataflow models supported in the LIDE framework

In homogeneous synchronous dataflow (HSDF), all consumption and production rates are restricted to be equal to unity [27]. Thus, an actor is an HSDF actor if every input port consumes exactly one token per firing and every output port produces exactly one token. More formally, the two conditions in Equation 2.1 have to be satisfied:

$$\begin{aligned} \forall e \in port_{in}(a) &: cns(e) = 1, \\ \forall e \in port_{out}(a) &: prd(e) = 1. \end{aligned} \tag{2.1}$$

A more general model is synchronous dataflow (SDF), where the consumption and production rates of actor ports must be constant (positive integer) valued [27] (i.e., they cannot vary as a function of data or state). The consumption and production behavior for the SDF model is defined as:

$$\begin{aligned} \forall e \in \mathit{port}_{\mathit{in}}(a) : \mathit{cns}(e) = v \text{ with } v \in \mathbb{N}^{+}, \\ \forall e \in \mathit{port}_{\mathit{out}}(a) : \mathit{prd}(e) = v \text{ with } v \in \mathbb{N}^{+}, \end{aligned} \tag{2.2}$$

where $\mathbb{N}^{+} = \{1, 2, \ldots\}$.

The cyclo-static dataflow (CSDF) model introduces the concept of actor *phases*, where in each phase an actor conforms to an "extended SDF" model (extended in the sense that zero-valued production and consumption rates are also allowed) [28]. In addition, the phases cycle through a periodic sequence, so that on each actor port, one can observe a periodic pattern of token consumption or production. CSDF behavior can be formally described by Equation 2.3:

$$\begin{aligned} \forall e \in \mathit{port}_{\mathit{in}}(a) : \mathit{cns}(e) = v^{\omega} \text{ with } v \in Z_{nn}, \\ \forall e \in \mathit{port}_{\mathit{out}}(a) : \mathit{prd}(e) = v^{\omega} \text{ with } v \in Z_{nn}, \end{aligned} \tag{2.3}$$

where v^{ω} is a periodic sequence with $||v|| > 0$, and Z_{nn} is the set of non-negative integers.

If for each actor port, the dataflow rates are the same in each phase (i.e., $||v|| = 1$), and they are all positive valued, then the CSDF actor conforms to the more restrictive SDF model. Similarly, if the consumption and production rates of an SDF actor are all equal to one, then the actor conforms to the more restrictive HSDF model. The HSDF, SDF and CSDF models have the property of being data-independent in the sense that actor consumption and production rates are not related to values of the data inputs.

The next two classes of models depicted in Figure 2.1 encapsulate what we refer to as the data-dependent models. In a control-based data-dependent model, a dataflow graph can contain one or more data-dependent dataflow (DDD) actors. A DDD actor is one in which one or more inputs (data values consumed) or the actor state (or both inputs and state) determine(s) how much data is consumed and produced by a given actor firing. Boolean dataflow (BDF) is an example of a control-based model. In BDF, Boolean-valued input tokens on designated ports are used to determine the production and consumption rates for ports where the rates are data-dependent [25].

For example, when a control input is `TRUE`, the actor could consume tokens from one data port, while it consumes tokens from a different data port when the control input is `FALSE`. Compared to data-independent dataflow MoCs, it is more difficult to reason about and analyze BDF graphs; however, useful quasi-static scheduling and analysis techniques have been developed that exploit the control-token-based dynamic dataflow structure of BDF (e.g., see [25]).

Integer-controlled dataflow (IDF) is a natural generalization of BDF where the inputs used to control data-dependent actors are integer-valued [18, 29], and correspondingly, the variations in consumption and production rates for individual ports can span integer numbers of different values. For example, IDF can be used to represent a generic multiplexer, where a control input selects data from a specific graph instance among N possible instances, each of which can be structured based on a different dataflow model.

The most general dataflow model that we consider in this work is the enable-invoke dataflow (EIDF) model [30]. In EIDF, an actor is specified in terms of a set of modes, such that in each mode the production and consumption rates must be constant, non-negative integer values. Intuitively then, in each mode, the actor can be viewed as an extended SDF actor. However, different modes of an actor can have different production and consumption rates, and dynamic dataflow behavior can be achieved in this way. This form of dynamic dataflow is distinguished from the control-based data-dependent class introduced previously by the decomposition of actor operation into distinct modes, and the elimination of the requirement that there be any specific control ports through which dynamic dataflow behavior is formulated. For dataflow models that employ decomposition into distinct modes, where production and consumption rates need not be constant or periodic across distinct modes, we introduce the third (most general) class of models, the mode-based models, represented in Figure 2.1.

In EIDF, each actor has two associated functions, called the *enable* function, and the *invoke* function. The process of implementing an EIDF actor includes providing implementations for these two functions. The enable function is a Boolean-valued function that returns `TRUE` if the actor has sufficient data on its input edges, and sufficient empty spaces on its output edges to execute a single firing based on the current state of the FIFOs in the enclosing dataflow graph. The invoke function carries out a single firing of the actor in the current actor mode and returns the set M of possible *next modes*. This set M gives the set of possible modes in which the actor can be invoked under normal operation (i.e., unless the actor or graph is somehow reset or reconfigured through external control). The process of selecting a specific next mode to execute among multiple possible next modes (i.e., when M contains multiple elements) is not prescribed by the EIDF MoC; this selection process is left up to the implementation.

The provision in EIDF for multi-element sets of valid next modes allows for non-determinism, as in its next firing, an actor can be invoked in any mode within the next mode set. A more restrictive, deterministic form of EIDF is the core functional dataflow (CFDF) model [30]. CFDF enforces that the set of next modes must always have exactly one element — i.e., $|N(m)| = 1$. Note that CFDF modes are different from CSDF phases in that the selection of the next mode in CFDF can be data dependent.

In summary, a group of dataflow models in a given DBD environment can cover a wide spectrum of trade-offs between expressive power and formal analysis potential. This is demonstrated by the spectrum of comparable models illustrated in Figure 2.1. Model detection helps designers identify the most restrictive dataflow model a given actor conforms too, thereby helping to identify the most powerful sets of analysis and optimization methods that can be applied to subsystems that contain the actor.

2.3.3 Tools

To implement and experiment with our proposed DBD methodologies, we have employed the DSPCAD Integrative Command Line Environment (DICE) [31], which is a framework for facilitating efficient management of design and software projects. DICE defines platform- and language-agnostic conventions for describing and organizing tests, and uses shell scripts and programs written in high-level languages to run and analyze these tests.

To create a generic method for instrumenting dataflow graphs, we used a DBD framework called the Lightweight Dataflow Environment (LIDE) [32], which is supported by DICE. This framework supports dynamic dataflow applications with the CFDF semantic model. From its foundation in CFDF semantics, LIDE enables dynamic behavior through structured application descriptions, making it an effective platform to instrument dataflow graphs, and prototype techniques for automated dataflow model detection and validation.

The application programming interface (API) in LIDE for defining actors includes the enable and invoke functions that are fundamental to CFDF semantics. Thus, each actor implementation in LIDE has an associated enable function, which returns a Boolean value, and an associated invoke function, which carries out an actor firing, and returns an integer index that identifies the next mode for the actor.

2.4 Formal Specification

From a designer's perspective, a formal system specification aims to define the expected behavior precisely while abstracting away the implementation details. Using such a specification, designers can evaluate system-level trade-offs without worrying about the underlying details. In addition, the correct behavior of the final implementation can still be verified by comparing the observed behavior to the expected behavior described in the original specification.

Many different specification formalisms have been introduced over the years that vary in the strictness of the semantics and applicability to various problems. For example, finite state machines (FSMs) have been used extensively to specify sequential logic by decomposing behavior into possible states and encoding the transitions between states using events [33].

Petri nets are commonly used to specify concurrent behavior, such as in distributed systems, due to their non-deterministic execution [34]. A Petri net consists of *places* and *transitions* that are connected by *arcs*, which are used to transfer data, called tokens. A Petri net may fire when there are sufficient numbers of tokens on all input arcs. When multiple transitions are be enabled at the same time, any one of them may fire, leading to non-deterministic execution.

Several high-level modeling languages and tools have been used to carry out system-level analysis, verification and validation (V & V), and architectural exploration. Modeling and Analysis of Real-Time and Embedded systems (MARTE) is an extension to the Unified Modeling Language (UML) that was released in 2009. MARTE strives to achieve interoperability between development tools to provide a common framework for specification, design, and verification [35]. Another example is the Architecture Analysis & Design Language (AADL) which has been used in automotive and avionics fields to model software and hardware architecture of embedded systems. It enables architecture exploration, system property checking, and timing analysis [36].

Using such specification facilitates the verification and testing processes [37]. For example, the authors in [38] describe the formal specification and verification of a parallel DSP chip that has 64 processors. They abstract away the implementation details by specifying the expected behavior from the programmer's perspective and then use an assume-guarantee method with a model checker called MOCHA to do compositional verification.

Dataflow-based specification is used to describe MPEG reconfigurable video coding (RVC) to abstract away implementation specific complexities [39]. In [40], the authors propose using CAL to specify the system, and then perform functional verification using the OpenDF environment [41].

2.5 Virtual Address Translation and TLB

In this section, we provide a brief overview of virtual memory and address translation, including the use of a TLB. For a comprehensive coverage of the topic, see [42–44].

Virtual memory is the process of mapping virtual address (VA) space, as seen by a program, to physical address (PA) space corresponding to RAM, which serves as a cache for the program's memory residing on disk. Memory is often segmented in fixed size segments called pages, and a page table is used to track the mapping between virtual and physical pages. Doing a translation from VA to a PA requires traversing the page table and was traditionally done by the OS. Some CPUs include a hardware-based a memory management unit (MMU) that automatically traverses page tables to perform VA to PA translation. Because the process of converting a VA to a PA is expensive (i.e., for every virtual memory access there are multiple physical memory accesses), most modern systems cache recent translations in a TLB.

A TLB-based VA translation is depicted in Figure 2.2, where a translation is first attempted using the TLB. If the page table entry (PTE) is present (TLB hit), the frame number is retrieved and the PA is formed. If the desired PTE is not present in the TLB (TLB miss), the traditional translation is done by indexing the page table to retrieve the PTE, which is then used to form the PA.

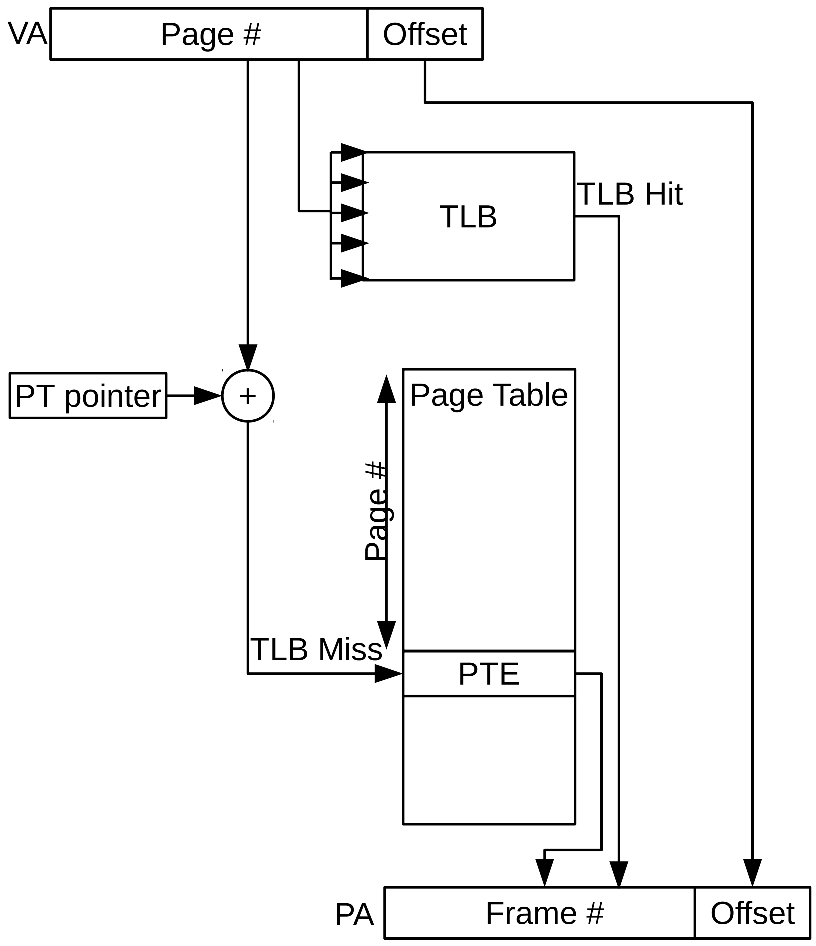


Figure 2.2: The virtual address translation is first attempted using the TLB. If the PTE is present in the TLB, the frame number is retrieved and the PA is formed. If the desired PTE is not present in the TLB, the traditional translation is done by indexing the page table.

Modern CPUs implement the TLB to be highly-associative and perform a parallel search using content-addressable memory (CAM) cells. In addition, TLBs are usually separated into I - TLB, used for instruction accesses, and D - TLB, used for data accesses.

A TLB translation occurs every memory access resulting in energy consumption of up to 12% of total processor power [45].

A TLB hit takes less than 1 cycle, while a TLB miss penalty can be 10s of cycles [43]. The miss penalty becomes significantly more expensive (on the order of milliseconds) if the translation results in a page fault. As a result, improving overall system performance often entails maximizing TLB hit rate.

Most operating systems maintain separate page tables for each process. This can result in some of the TLB entries becoming invalid after a context switch. One way to deal with this is to flush the TLB after every context switch. However, this results in having an empty TLB, resulting in a TLB miss for initial memory accesses. Another strategy is to add process-specific identifiers to each TLB entry to help each process identify its entries, and thus effectively share the TLB [46]. However, as the number of processes operating concurrently increases, the effectiveness of the TLB decreases due to interference among the processes [47, 48].

Due to these challenges, optimizing TLB performance has remained an active area of research for the last 20 years, and the TLB remains a fixture in modern processors due to its beneficial effect on the performance of VA translation. In chapter 3, we address the TLB sharing problem by proposing a novel OS/compiler synergistic technique that enables preloading the TLB at the start of a context switch with the needed PTEs.

2.6 Summary

In this chapter, we have provided general background on embedded system design strategies and model-based design. We have described the use of TLBs to improve virtual address translation and introduced several profile- and instrumentation-based CAD tools. Finally, we have provided background on formal specification and dataflow modeling.

Chapter 3: Context-aware TLB Preloading for Interference Reduction in Embedded Multi-tasked Systems

Signal processing programs often contain multiple operations involving vectors and matrices that are implemented using loops. Such linear algebra operations exhibit well-understood memory access patterns and are highly amenable to compiler optimizations. Optimizing these computationally intensive sections of code (or hotspots) can be achieved with profile guided optimizations (PGO) [4]. In this chapter we present a TLB preloading methodology that works by first identifying application hotspots using profiling, and then optimizing them with a TLB preloading methodology.

The set of extensive experimental results with signal processing kernels demonstrate the effectiveness of the proposed technique in eliminating the inter-task TLB interference and preemptively allocating TLB entries during context-switch. Material in this chapter was published in partial, preliminary form in [49].

3.1 Introduction

Modern systems require multiple tasks to be operating concurrently. A cell phone, for example, might be used to concurrently playback audio and manipulate discrete data streams for SMS messaging. Due to size, cost, and power constraints, it is infeasible to have a dedicated processor for each task. Instead, multitasking is used where a set of tasks share the underlying hardware. To achieve multitasking in a way transparent to the tasks, process state such as the PC and register file needs to be preserved at context switch. Resources large in size that cannot be saved, such as the translation lookaside buffer (TLB) and caches, are shared among the different processes. As a result, inter-task interference can degrade performance.

A TLB is a cache used to enable fast virtual address translation. The TLB stores recently used page table entries (PTE) thereby greatly reducing the need for page table walks and improving performance of virtual memory systems. To reduce task self-interference, TLBs employ highly associative structure with a relatively small number of entries in order to minimize the impact on system power [42]. While appropriate in some designs, reducing high-cost TLB misses using expensive hardware, such as hardware-based page table support, is not feasible for low-power embedded systems.

In multiprocessing systems, a TLB is either flushed during a context switch, resulting in significant performance degradation undesired for high-end embedded systems, or a process-specific identifier is added to each TLB entry to help each process identify its entries, and thus effectively share the TLB [46]. As the number of processes operating concurrently increases, the effectiveness of the TLB decreases due to interference among the processes [47, 48].

Another very important consequence of increased TLB contention in multiprocessing systems is the complication of *Worst-Case Execution Time* (WCET) analysis [48, 50, 51]. Many embedded system applications have real-time constraints where the application must meet a deadline to ensure correctness. WCET analysis is used to establish an upper bound on execution time which is then used to schedule tasks on the processor such that all the deadlines are met. In single task systems, WCET analysis, although complex, is tractable [52, 53]. Predicting whether a memory access will find its memory mapping in the TLB in the presence of inter-task interference becomes extremely difficult, if not impossible. Consequently, extremely conservative assumptions must be followed resulting in overly pessimistic execution time estimates causing underutilization of the processor.

A simple solution to decrease contention is to increase TLB size, yet that degrades access time and significantly increases power requirements. Another approach to alleviate TLB interference is for a process to lock its PTEs in the TLB, thus preventing other processes from evicting its TLB entries when preempted. This approach is ineffective when the working set is large and can cause very high contention / starvation if each process locks all of its entries. An alternative approach is to treat the TLB as part of a process state and to save its TLB entries during a context switch. The next time the scheduler allocates the process to the CPU, the OS would restore the saved TLB entries. For a large working set, saving and restoring all TLB entries becomes infeasible, since each process would need to traverse the entire TLB table at every context switch.

Instead of saving all the TLB entries, a method is needed to find the set of *live* TLB entries, or more specifically, the set of live virtual page numbers (VPNs) for the task at the moment it is switched for execution. A *live* entry is an entry that is currently in the TLB, and that will be used in the subsequent execution time-slice of the task. Clearly, there is no need to save entries in the TLB that will not be referenced by the process in the future. Furthermore, at context-switch time, the utilization of application-specific information will enable the identification of memory mappings that are *not yet live* but will be used by the task in the *near future*. We refer to the combined set of live VPNs plus VPNs to be used in the subsequent time-slice (but not yet used) as the *Extended Live Set (ELS)*. The ELS is defined at context-switch time – its determination is possible only through the cooperation of compiler and OS.

We propose a *Context-aware TLB Preloading (CTP)* methodology, which not only reduces D-TLB interference in multitasked embedded workloads, but also further leverages compiler-generated application information to preload TLB entries that will be used in the near future. Fundamentally, CTP attempts to determine the ELS at context-switch time, and as such, to rapidly preload the set of live memory mappings and mappings that are not yet active but are found to be needed in the near future. CTP leads to a more predictable TLB hit/miss behavior, since the group of tasks sharing the TLB are guaranteed to have their most recently used PTEs in the TLB after preemption. A more predictable TLB behavior alleviates the difficulty of WCET analysis for multitasked systems leading to a more realistic WCET and a better processor utilization.

The rest of this chapter is organized as follows. In the next section we provide a brief overview or prior research and then provide a high-level overview of CTP in Section 3.3. An analysis of TLB contention is presented in Section 3.4. In Section 3.5, we provide a detailed explanation of the context-aware TLB prefetching technique and discuss the required OS and compiler support. The experimental setup and results are presented in Section 3.6. Section 3.7 provides a summary of our contribution.

3.2 Related Work

TLB organizations and management policies have been the active focus of research due to the fundamental role of virtual memory management in modern systems. The interference effects on TLBs and caches caused by context switches were demonstrated in [47, 48]. An in-depth study of the TLB characteristics on the performance of modern applications was performed in [54]. One of the conclusions is that software-managed techniques for TLB prefetching may be quite rewarding if implemented efficiently.

Several architectural techniques have been proposed to minimize translation overhead. For example, [55] introduce a TLB architecture to dynamically support up to two pages per entry with a banked fully-associative structure. Such an organization benefits applications where larger pages can be used to minimize the translation overhead. In [56], a special MMU has been proposed and evaluated that supports dynamic memory allocation and deallocations for a chip multiprocessor. The *System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU)* allocates and grants, in deterministic time, portions of the global on-chip memory.

Preserving useful TLB entries between context switches has been an active area of research. In [57], TLB references are directed to a special set of registers capturing several recently accessed TLB entries. Due to the small size of the register set compared to the VPN footprint, the compiler needs to transform the code appropriately. A recency-based TLB prefetching mechanism have been proposed in [58]. The technique maintains a stack of page references and prefetches the page next to the one referenced. The distance-prefetching TLB methodology was introduced in [59]. The authors also compare and evaluate a set of TLB prefetching techniques. An compiler-based approach that introduces predictable form of paging through the use of page-in and page-out points for a single task was described in [51]. A TLB tagging method is presented in [50] that refrains from flushing the hypervisor TLB entries during a virtual machine's context switch.

3.3 CTP Overview

Even finding only even the set of live TLB entries is impossible through a compiler-only or an OS-only technique. The compiler has access to static information about the program, but lacks run-time knowledge needed to extract the actual values of the current VPNs from the task's context at the time when a preemption occurs. The OS, on the other hand, has run-time information about the task's state but lacks static information about the program needed to interpret the context, e.g. which registers or stack frames can be used and in what ways to extract the ELS. CTP uses a synergistic cooperation between the OS and the compiler where our approach combines the compiler's knowledge of static information about the program with OS's knowledge about the task's context to allow the OS to prefetch ELS.

A high-level overview of CTP is shown in Figure 3.1 where task *T1* is initially executing on the CPU and is about to be preempted with task *T2*. In the proposed approach, the compiler has synthesized special routines for *T1* and *T2*. In fact, such routines are generated for each frequently executed loop / function in *T1* and *T2*. We refer to this routine as the *ELS Preload Function (EPF)*. At its core, the EPF employs the compiler's knowledge about registers and stack frame usage to *rapidly identify the set of live VPNs* used by the task at that moment by effectively obtaining all the pointer values of relevance to the task. Moreover, the pointer values obtained from the context combined with the knowledge of access patterns (e.g. strides) and array dimensions are used to determine with very high accuracy the future VPNs to be accessed by the task.

This compile-time step corresponds to phase (0) in Figure 3.1. The EPFs associated with each task are registered with the OS by using a specially provided API just prior to executing the corresponding loop/function in the task. At run-time, the OS invokes the EPF of the next task when the task is loaded for execution, after preempting the previous task. During the next step in the example, the OS preempts *T1* and saves its state, corresponding to (1) and (2). After selecting the next task to execute, the OS would restore that task's state, shown by (3). After *T2*'s state is restored, the OS would call the compiler-generated function for *T2*, *EPF(T2)*, which would use the recently restored *T2*'s state as input to generate a list of VPNs needed during the upcoming processing slice, as shown by steps (4) and (5) in the Figure 3.1. This list of VPNs represents a highly accurate estimate of the Extended Live Set of TLB entries for that context-switch operation. The OS would then issue a block PTE prefetch for the list of VPNs, as shown by (6). When *T2* begins execution, the TLB would contain a set of PTEs needed by *T2* during its current processing slice.

3.4 TLB Contention

A process executing on a dedicated CPU incurs very few TLB misses. This is especially true for many embedded applications which are comprised of several kernels

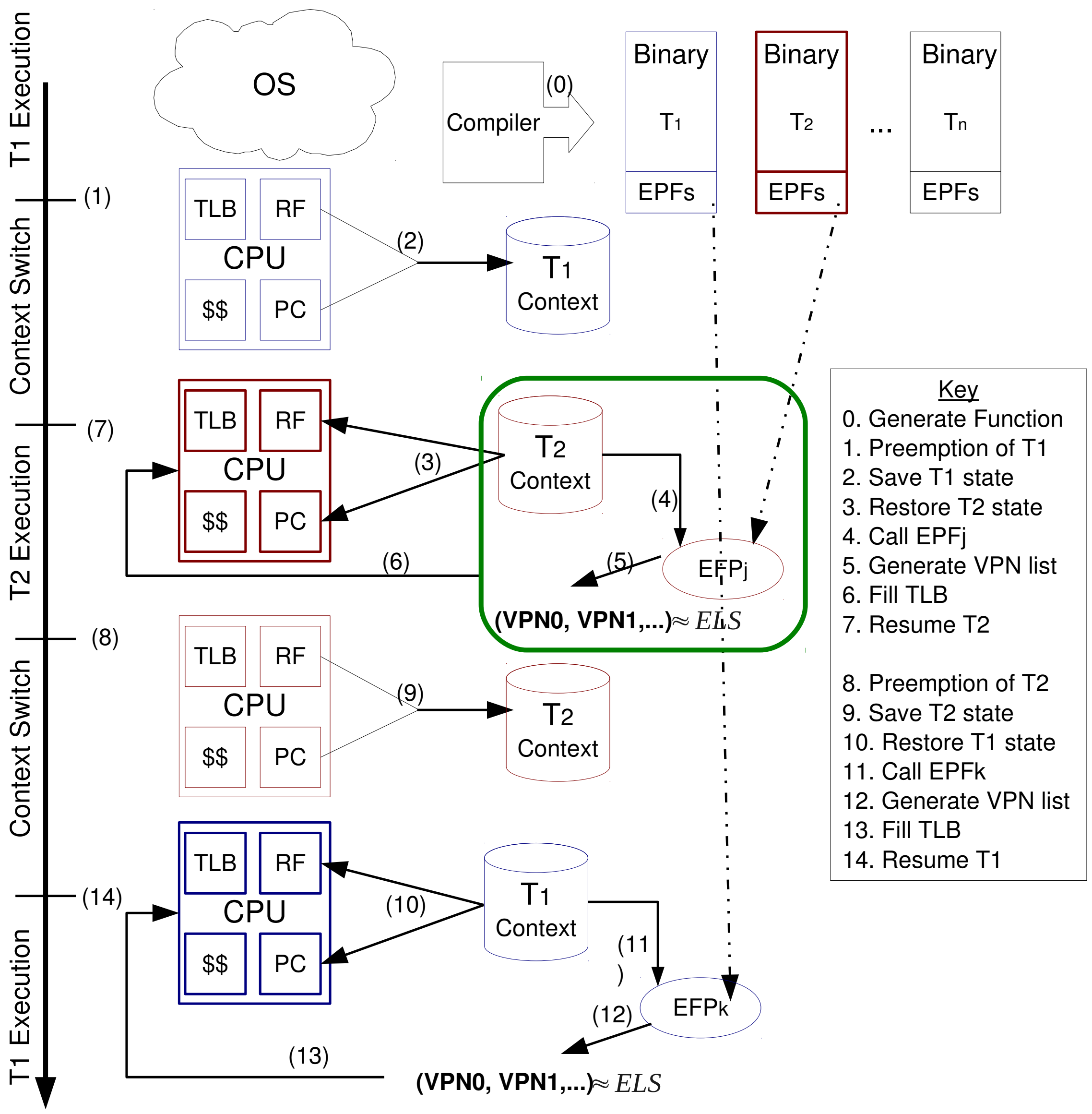


Figure 3.1: High level overview of CTP

that process the incoming data. On the other hand, when multiple tasks are executed by sharing the TLB, the number of expensive TLB misses can increase significantly. Inter-task interference in the TLB is the major culprit for such an increase.

To evaluate the effects of multitasking on the shared TLB and the resulting increase in misses, we have conducted a set of experiments. We have used a set of known embedded kernels to form multitasking benchmarks. The kernels considered for this work include ADPCM encoder (AENC) and decoder (ADEC), extended jacobian (EJ), forward discrete cosine transform (FDCT), fast Fourier transform (FFT), LU matrix factorization (LU), matrix multiplication (MMUL), successive over-relaxation (SOR), and lower triangular (TRI) transformation. A group of kernels running concurrently was defined as a benchmark; the 5-kernel, 6-kernel, and 7-kernel benchmarks used in our evaluation are defined in Table 3.1.

Table 3.1: Benchmarks used for evaluation

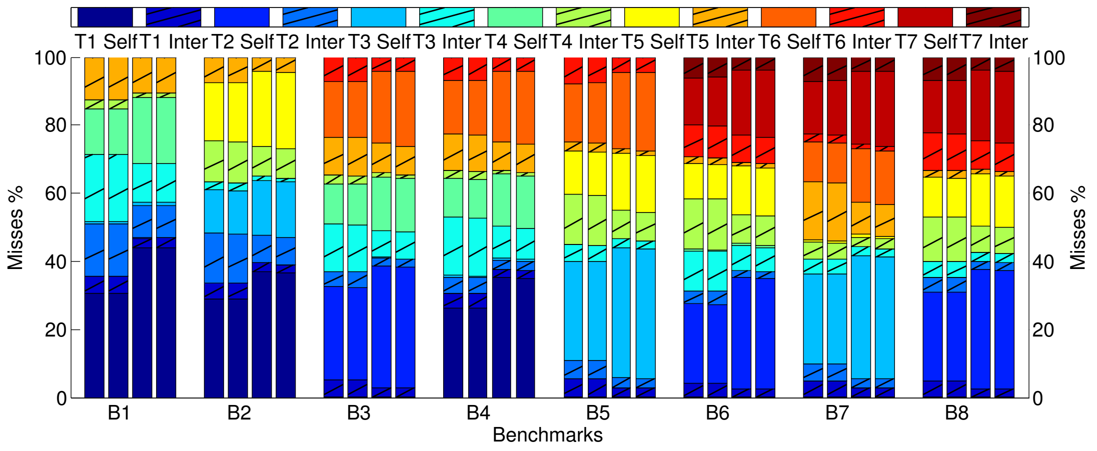
	B1	B2	B3	B4	B5	B6	B7	B8
Task 1	EJ	EJ	AENC	EJ	ADEC	ADEC	ADEC	AENC
Task 2	FFT	FFT	EJ	FDCT	AENC	EJ	AENC	EJ
Task 3	LU	MMUL	FFT	LU	EJ	FFT	EJ	FDCT
Task 4	MMUL	SOR	MMUL	MMUL	FFT	LU	FDCT	FFT
Task 5	SOR	TRI	SOR	SOR	MMUL	MMUL	LU	MMUL
Task 6	–	–	TRI	TRI	TRI	SOR	MMUL	SOR
Task 7	–	–	–	–	–	TRI	TRI	TRI

TLB sizes of 32 and 128 entries were used, where for each size, 4-way and 16-way set associative settings were applied, allowing 4 possible TLB size / associativity configurations. In order to evaluate the impact of interference, we have instantiated (for simulation purposes only) a local TLB used exclusively by each kernel in the benchmark, and a global TLB shared by the kernels in the benchmark. Each VA translation occurs in the task's local TLB, as well as in the global TLB shared between the different tasks. This setup allows distinction between self-interference (i.e., TLB misses occurring in both global and local TLBs) and inter-task interference (i.e., misses that occur only in the global TLB). All the TLBs used an LRU replacement policy. Each simulation lasted 200 million instructions. For kernels shorter than 200 million instructions, the execution was restarted until 200 million instructions were executed. Time slices of 200K and 500K[1]were used, where the tasks defined for each benchmark were switched in a round-robin order every time slice.

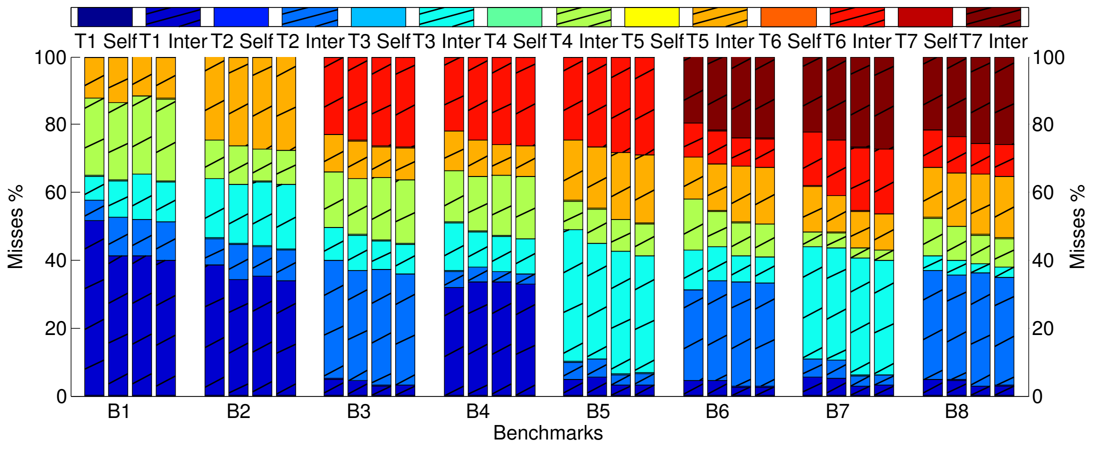
Figure 3.2 contains miss-breakdown for 32 and 128 entry TLBs. The first and second columns in each group corresponds to 200k time slice results, while columns three and four correspond to 500k time slice results. In addition, the first and third columns correspond to TLBs that are 16-way associative, while columns two and four correspond to TLBs that have associativity of 4. The local misses correspond to TLB misses resulting from self interference (i.e., a self interference miss is a miss in both the local and global TLB) and are shown by solid color sections of each bar. Global misses are the result of inter-task interference when a given kernel executes concurrently with other kernels (i.e., a global miss is a hit in the local TLB but a miss in the global TLB) and are shown by striped sections of each bar.

The results for 32 entry TLB simulations show 37% of the TLB misses were the result of inter-task interference. The other 63% were the result of self-interference, but

[1]For a low-power embedded processor with a clock speed of 100 MHz and a CPI of 1.9 [60], this would correspond to a process time slice of 4ms and 10ms respectively.



(a) 32 entry TLBs



(b) 128 entry TLBs

Figure 3.2: TLB misses can be separated into self and inter-task interference misses. The analysis of 32 entry and 128 entry TLBs are shown in subfigures (a) and (b), respectively. The first and second columns in each group corresponds to 200k time slice results; columns three and four correspond to 500k time slice results; the first and third columns correspond to TLBs that are 16-way associative; columns two and four correspond to TLBs that have associativity of 4.

after analyzing the results further one can see that the EJ and TRI kernels are responsible for a large portion of those misses. EJ and TRI both have a large working set and a 32 entry TLB is not large enough to accommodate it.

As expected, prolonging the processing time slice from 200K instructions to 500K instructions increases the proportion of self-interference misses for EJ and TRI, since by executing for a longer time, each task processes more data and requiring more page table translations.

Increasing the TLB size to 128 entries alleviates self-interference for each task. Both EJ and TRI are now able to fit their entries in the TLB without significant conflicts. As can be seen from the large portion of striped bars, however, inter-task interference contributes on average to 99% of total TLB misses. The actual number of misses, however, decreases significantly since a large TLB reduces the number of conflicts.

The total number of misses incurred by each benchmark is shown in Table 3.2 and Table 3.3. As expected benchmarks using the smaller sized, 32-entry, have a greater number of misses for each configuration than that 128-entry TLBs. Due to the small TLB size, the number of TLB misses stays approximately the same when increasing associativity from 4 to 16 for 32 entry TLBs. The miss-breakdown composition also stays approximately the same.

For the 128 entry TLBs, increased associativity leads to fewer misses on benchmarks where the working sets are better able to utilize TLBs as in B1, B2, B3, B4, which are all 5 and 6 task benchmarks. On the other hand, in benchmarks having many parallel tasks, as in B6, B7, and B8, increasing associativity allows tasks with large working sets to thrash the TLB by evicting other task's entries without reducing its own TLB miss rate. For those benchmarks, a larger associativity results in fewer non-thrashed TLB lines. These results are representative of the general multitask TLB interference and is the problem that we attempt to address by identifying and preloading the ELS.

Table 3.2: Number of Misses for 32-entry TLB (in thousands)

#Inst x Asc	B1	B2	B3	B4	B5	B6	B7	B8
200K x 16	124.0	131.0	138.1	144.2	129.5	163.5	142.7	145.0
200K x 4	124.0	131.6	138.6	144.7	130.0	163.9	143.1	145.5
500k x 16	86.2	102.2	105.0	107.5	98.9	115.7	104.2	107.9
500k x 4	86.1	103.2	106.0	108.4	99.9	116.6	105.1	108.9

Table 3.3: Number of Misses for 128-entry TLB (in thousands)

#Inst x Asc	B1	B2	B3	B4	B5	B6	B7	B8
200K x 16	73.3	98.9	111.4	121.0	98.7	155.2	117.1	123.3
200K x 4	80.6	104.4	111.8	115.5	101.1	135.8	113.8	119.2
500k x 16	77.4	93.9	96.8	99.0	90.4	107.3	95.7	99.7
500k x 4	72.3	90.3	93.1	97.6	86.1	106.2	93.6	96.1

3.5 Context-Aware TLB Preloading

To enable preservation of the TLB state between context switches, we propose using CTP, a synergistic cooperation between the OS and compiler, to predict and preload PTEs needed in the future. CTP would be primarily applied to application hotspots. A hotspot, like FFT and FDCT, is a frequently executed part of a program responsible for data processing. Hotspots can be identified by doing profiling, either manually, using tools like `gprof`, or automatically, with the use of a compiler technique like PGO. Since most of the processing time is spent in hotspots, no preloading would occur between hotspots (i.e., the system would operate as if CTP was not present).

3.5.1 Identifying the Extended Live Set

Finding the Extended Live Set (ELS) of live memory mappings is impossible through compiler-only or OS-only techniques. The compiler can extract the static information about the program, e.g., the register set used as array indeces, the stack frame or global memory locations allocated that points to the data, the sizes of arrays, loop access patterns (affine index functions). An example of algorithmic knowledge is the memory access pattern. However, in addition to this static information, run-time information is required as well, i.e., the place in the code where the context switch occurred, and the specific values of the pointers (in registers or memory locations) which the compiler lacks. The OS, on the other hand, has this run-time knowledge but lacks static information about the program being executed. Combining the compiler's knowledge of static information with the OS's knowledge of the task's current context, the proposed technique identifies the pages that will be needed during the next process slice.

In the proposed CTP technique, this synergism between the compiler and the OS is achieved through a specially generated *ELS Preload Function (EPF)* by the compiler, which is invoked at context-switch time (when the task is scheduled for execution) by the OS with arguments comprising the task's state, the anticipated duration of the time-slice given to the task, and the TLB parameters. EPF will estimate in a highly accurate manner the task's ELS for that moment in execution.

Since the CTP methodology is applied on the application hot-spots of phases in an independent manner, an EPF will be generated at compile-time for all the major application loops. Prior to commencing the execution of such a major loop, the EPF for that loop is registered with the OS and from that moment on, the OS will invoke that EPF at context-switch time for that task.

The structure of the EPF, a detailed example of which is given in subsection 3.5.3, fundamentally includes a simple set of instructions, which by reading the appropriate registers and stack frame location, obtain the current value of the data set pointers (array locations) used in that loop. Since the EPF is generated by the compiler, it is trivial to identify all the valid VPNs accessed by the loop. CTP uses the loops' access-pattern strides to estimate in a highly accurate manner the set of VPNs that are not pointed to by the loop but will be in the near future. It is noteworthy, that the access pattern here may be much more complex than a simple stride as used in many general-purpose TLB prefetching techniques. Fundamentally, the loop-specific EPF, when invoked by the OS, will rapidly identify a set of VPNs that approximate very accurately the task's ELS for that specific run-time moment.

The compiler generates EPF using the static information known about a program at compile time and includes it in the compiled program binary as one or more ELS preload functions. At context switch, the OS calls the EPF, passing it the knowledge of the underlying hardware and program state to determine which PTE entries should be preloaded for the task to be run in the upcoming time slice.

3.5.2 Compiler and Operating System Support

Recent advances in compiler technology have enabled better program understanding and have resulted in sophisticated code analysis. Compilers can already easily rearrange code, as in loop tiling, to enable better cache utilization and exploit fast scratchpad memories. To do so, the compiler determines memory access patterns based on the way the data arrays are indexed. Our approach similarly uses data reuse analysis from the compiler to identify the current live set and access strides, which are functions of loop indeces, to estimate ELS.

As was shown in Figure 3.1, a compiler supporting CTP would create one or more EPFs as part of the binary. The number of EPFs would depend on the number of hotspots present. One EPF would be generated if the program consists of a single loop nest as in MMUL. If a program, however, is comprised of several functions, then several EPFs would be generated. For irregular code or code that depends on run-time conditions (e.g., conditional branches and other control conditions that cannot be predicted by the compiler), no EPF would be generated.

In addition to generating the EPFs, the compiler would insert system calls before and after each computational segment to be used to associate the code segment with a specific EPF. During execution, the application would issue a system call when reaching a certain segment, informing the OS to use the corresponding EPF during a context switch. The system call at the end of a code segment would notify the OS to no longer use an EPF during context switches for that application.

An EPF would require as input the length of a process time slice (T) and context state of the process. The time slice length would be used to determine how many pages will be needed during the next processing time slice. For example, if T is 100K cycles, each loop takes on average 20 cycles, and a new page is needed every 1000 iterations, then 5 (i.e.,$\frac{100K}{1000*20}$) new PTEs would need to be prefetched as part of the extended live set. The output of the EPF would be a list of VPNs that will be needed for the next processing slice that will be derived from the array pointers.

Each preloading routine will derive a list of pages and corresponding VPNs needed for the next processing slice from the VAs contained in the array pointers, as well as the indexing stride (e.g., an array indexed as *A[8*i]* would need the next page sooner than an array indexed as *A[i]*). The length of the processing time slice, array sizes, TLB configuration, loop length and algorithmic information extracted by the compiler will determine the number of TLB entries to be preloaded. Array size information needs to be known to define array boundaries; thus, enabling correct preloading from the start of the array when reaching its end.

TLB configuration is an important parameter that helps avoid aggressive preloading that can evict useful TLB entries. An example of such a situation can occur in a 64-entry 4-way set associative TLB where the access pattern for a given array is every 16^{th} page. In such a situation, the fifth TLB prefetch based on that array will cause an eviction of the first preloaded entry. Thus, the EPF will prioritize PTEs such that those needed soonest would receive highest priority.

The required operating system support is shown in Figure 3.3. The operating system would operate normal, without CTF, until it reaches a part of the program for which a CTF exists. At that time, a system call would be made to setup the use of a CTF for the ensuing context switches. In Figure 3.3, the arrow from OS to EPFs corresponds to the system call which would be made when entering a section of the program for which an EPF exists. Next time that task is scheduled to the CPU during a context switch, the OS would call the EPF passing it the processing time slice length (T in Figure 3.3) as input. The EPF would use the set of array pointers to compute the list of VPNs needed to be preloaded. After receiving a list of VPNS, the OS can then do a block page table lookup filling the TLB with PTEs needed by the process. A one time block page table lookup avoids expensive system calls necessary for most TLB misses.

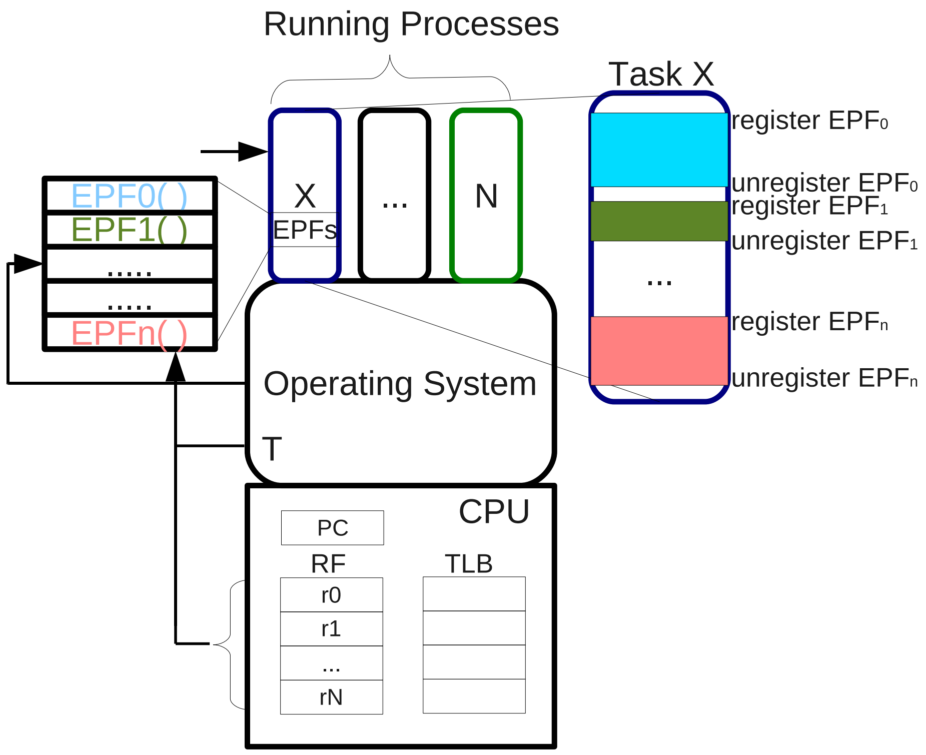


Figure 3.3: Required Operating Systems Modifications

3.5.3 CTP Example – Matrix Multiplication

To show how CTP works on an actual program, consider the MMUL kernel pseudocode shown in 3.5.3. Line 2 contains array size information which the compiler can use to compute array boundaries. The 3 loops on lines $3-9$ multiply the rows of **A** by the columns of **B** and store the result in **C**. The assembly corresponding to lines $7-9$

is shown in Figure 3.5. During the compilation process, registers $r10$, $r5$, and $r4$ are allocated for use as address indeces for arrays **A**, **B**, and **C** respectively.

As can be seen on line 6 of 3.5.3, array **A** is traversed using the two outer loops with i and j; thus, once a certain row of **A** has been used, it is not used again. On the other hand, the array **B** is traversed using the inner two loops utilizing j and k, thus when i, corresponding to the row of **A**, increases, array **B** is traversed from the beginning. Array **C** is traversed using i and k from the outer and inner loops respectively. Each column of **C** is traversed N consecutive times, but once the algorithm reaches column n, the data from columns $n - 1$ is not used. These observations can be verified by looking at lines 8 and 11 of Figure 3.5 where the registers allocated to **B** and **C** are is incremented by 4.

```
1: $N \Leftarrow 128$
2: $int\ A[N][N], B[N][N], C[N][N];$
3: for $i = 0$ to $N$ do
4:     for $j = 0$ to $N$ do
5:         for $k = 0$ to $N$ do
6:             $C[i][k] \Leftarrow C[i][k] + A[i][j] * B[j][k]$
7:         end for
8:     end for
9: end for
```

Figure 3.4: Matrix Multiplication

Using array indeces i, j, k along with page boundaries, the compiler is able to extract the pages currently used and the pages that will be needed in the future for each array. However, the compiler only has static information about the program and cannot predict what values of i, j, and k will be when a context switch occurs. Without that run-time knowledge, the compiler by itself cannot determine which PTE entries should be saved from the TLB.

```
 1:  $L25:
 2:  lw $3,0($10)      //r10 holds A[]
 3:  lw $2,0($5)       //r5 holds B[]
 4:  mmult $3,$2
 5:  mflo $3
 6:  lw $2,0($4)       //r4 holds C[]
 7:  addu $6,$6,1
 8:  addu $5,$5,4
 9:  addu $2,$2,$3
10:  sw $2,0($4)
11:  addu $4,$4,4
12:  slt $2,$6,256
13:  bne $2,$0,$L25
```

Figure 3.5: Assembly corresponding to inner loop of MMUL

Combining the compiler-generated static information with OS's knowledge of the run-time state we are able to identify some of the live TLB entries simply from knowing the registers the compiler allocated to index into the three arrays. The array pointers are usually stored in registers but can also be in the stack frame, the location of which is also known in the EPF. Restoring the VPNs obtained from those registers at the start of a process slice would enable the operating system to do a group TLB prefetch.

Since MMUL has one nested loop, only one EPF would be generated. To make this example more concrete, several assumptions were made about the architecture of the system. Each page size was assumed to be 4096 bytes. Arrays A,B, and C were assumed to be each 16 pages respectively. The inner loop on lines $5-7$ is 10 instructions corresponding to 12 cycles when the CPI is 1.2. A context switch occurs every 200K cycles. Finally, lines $4-8$ were assumed to take approximately a processing slice (200K cycles).

The compiler can extract most of these parameters from doing static analysis of the code. Being given the page size information by the OS, the compiler can the compute required space for each array.

Using this information and the values of i, j, and k, the preloading routine can compute exactly which PTEs will be needed during the next processing cycle. For example, a process slice starting when $i = 1$, $j = 0$, and $k = 0$ would require the pages $C[1][*]$, $A[1][*]$, $B[0][*]$, $B[8][*]$, $B[16][*]$, ..., $B[120][*]$

A sample EPF for MMUL is shown in Figure 3.6, where the VPNs obtained from $r10$, $r5$, and $r4$, corresponding to the current pages of arrays **A**, **B**, and **C**, respectively, are added to the VPN set in lines $1-3$. In addition, 15 predicted VPNs obtained from $r5$ corresponding to array **B** are added to the VPN set in lines $5-7$. The number of predicted pages ($N_{predict}$) is a function of the length of the process time slice (T_{slice}) and number of instructions needed to complete a given code segment (N_{instr}). N_{instr} is determined during compiler time, and T_{slice} is provided by the OS (this relationship is shown on line 4). Since the access stride is 1, corresponding to 4 bytes, the 15 future VPNs can be obtained by simply adding a multiple of 0x1000 to the current address of **B** being held in $r5$.

After receiving the list of VPNs, the OS would do a block page table lookup eliminating some potential TLB misses. TLB misses can still occur as a result of self-interference, when the preloading routine cannot predict VPNs or when the context switch occurs during a section of code for which no EPF exists.

```
1: $VPN[0] \Leftarrow VPN(r10)$ ;                         // current VPN for A[]
2: $VPN[1] \Leftarrow VPN(r4)$ ;                          // current VPN for C[]
3: $VPN[2] \Leftarrow VPN(r5)$ ;                          // current VPN for B[]
4: $N_{predict} \Leftarrow \frac{16*T_{slice}}{N_{inst}}$
5: for $i = 1$ to $N_{predict} - 1$ ;           // $N_{predict}$ is 16, predict 15 pages
   do
6:     $VPN[i+2] \Leftarrow VPN(r5 + 0x1000 * i)$ ;       // future VPNs for B[]
7: end for
8: Function $VPN(r)$ return $(r\&0xFFFFF000) >> 12$
```

Figure 3.6: MMUL EPF

3.6 Evaluation

3.6.1 Experimental Setup

To evaluate the effectiveness of the proposed TLB preloading technique, we performed an extensive simulation-based study using various TLB configurations. We have evaluated two EPF structures: a naive implementation that uses values of current pointers from each task to preload what they are pointing to, resulting in a subset of the *live* set; and an approximation of execution-aware policy that identifies the future VPNs based on the access pattern.

The approximation of ELS was implemented by prefetching the subset of the *live* set – the VPN of the current page was extracted from the pointer to each array and was prefetched during a context switch; and 4 additional VPNs for each array that were derived from the stride of each array. This was achieved by prefetching the VPNs of the current and the ensuing pages extracted from the pointers to each array (e.g, if the pointer for a given array has address 0x1000, the *extended live* set would contain VPNs extracted from [0x1000, 0x2000, 0x3000, 0x4000, 0x5000], where the page size is 0x1000).

SimpleScalar [61] was used to generate memory traces for each task. In addition to the trace information, we also extracted execution progress of each task. The custom traces were then used as input to a TLB simulator which could be configured in standard, *live* set preloading, and ELS preloading modes.

3.6.2 Analysis of Extended Live Set Results

Extending the *live* set by predicting future VPNs that will be needed, offers more opportunities to reduce the TLB miss rate. On the other hand, TLB miss rate can increase if the valid TLB entries are evicted when filling the TLB with translations for VPNs that will not be needed.

The ELS preservation results are shown in Figure 3.7. The miss-rate improves by more than 20% for all of the configurations. Each group of bars corresponds to overall miss improvement of each benchmark in different configurations. The first two bars in each group show results for 200K process time slice while the last two bars in each group show the 500k process time slice configuration results. In addition, bars one and three in each group show results for 16-way set associative TLBs while bars two and four show 4-way set associative TLB configuration results. The technique is most effective when process time slices are short, since a short slice allows numerous opportunities to preload TLB entries.

For 32-entry TLBs, increased associativity provides more tolerance for mispredicting VPNs and results in a slightly higher improvement in the overall miss rates as compared to lower associativity configurations. The larger 128 entry, 16-way associative

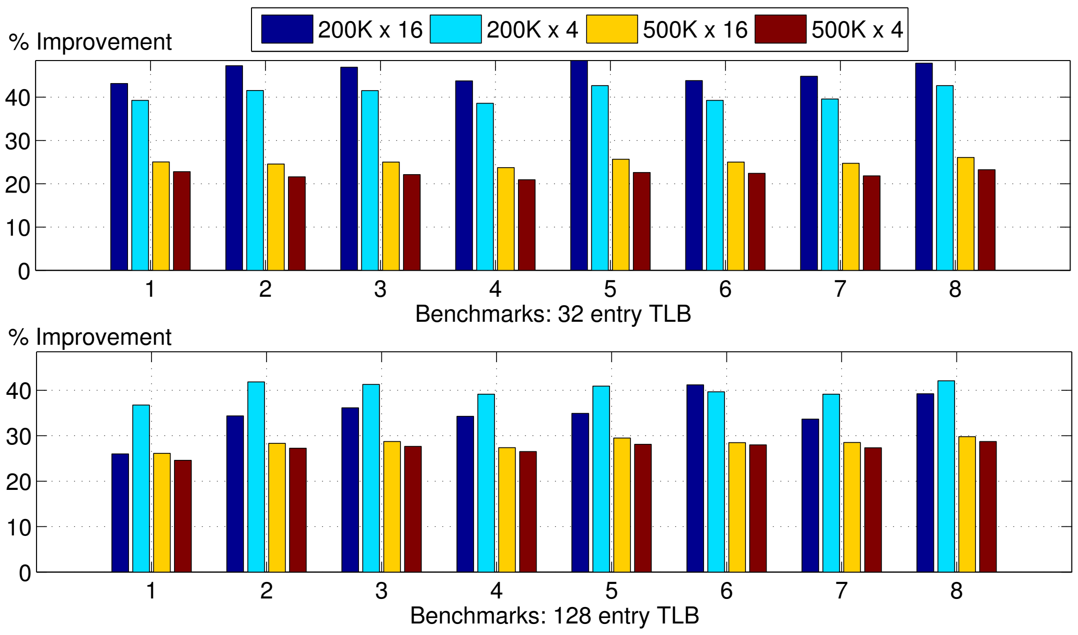
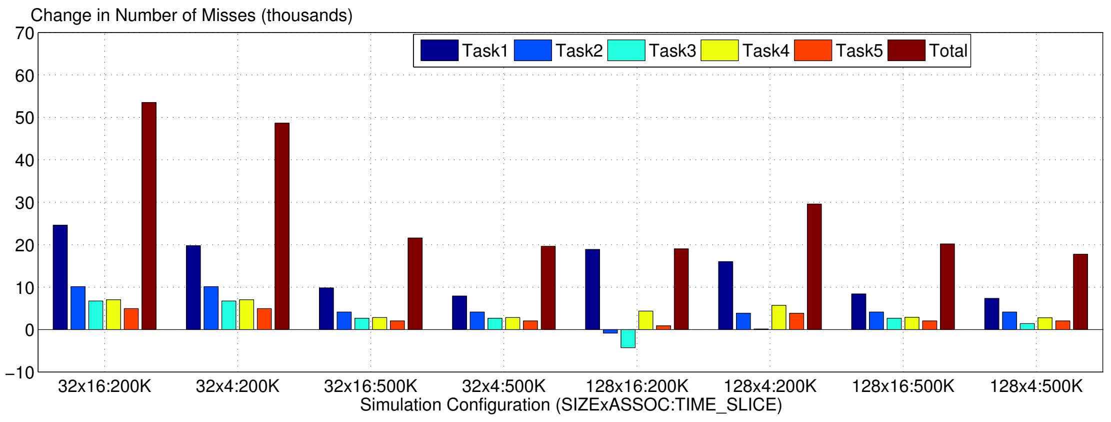


Figure 3.7: ELS Preloading - Overall Miss Improvement

TLBs are already more tolerable to TLB contention, thus preloading TLB entries offers a smaller improvement for those configurations as compared to 32-entry TLBs.

As described above, the implemented approximation of EPF preloads always the next 4 VPNs, regardless of whether they will be needed during the present time slice, and it is not ideal in most situations. Conversely, using a compiler-generated EPF as outlined in Section 3.5 would enable a more accurate preloading and thus, would achieve a higher miss rate reduction.

We selected B1 and B8 as examples to analyze the effects of CTP on the TLB performance of individual task within the benchmark. Figure 3.8 plots the breakdown of misses for B1, a 5 task benchmark, in the top plot and for B8, a 7 task benchmark, in the bottom half of the figure. For each benchmark, a breakdown of misses incurred by each task is shown across the different simulation settings. In each configuration group, the last bar shows the total change in number of misses for the entire benchmark. Positive values correspond to the reduction in the number of misses while negative values correspond to an increase in the number of misses. Notice that even though the total improvement is positive, individual tasks within a benchmark can suffer negative performance changes. This phenomenon can occur when a task with a large working set, like MMUL, has substantial miss reduction but at the same time evicts some useful entries during prefetching that belong to other tasks where the evicted-entries would not have been evicted otherwise. This explains the increased number of misses for FFT and LU tasks within B1 in 128x16:200K configuration.



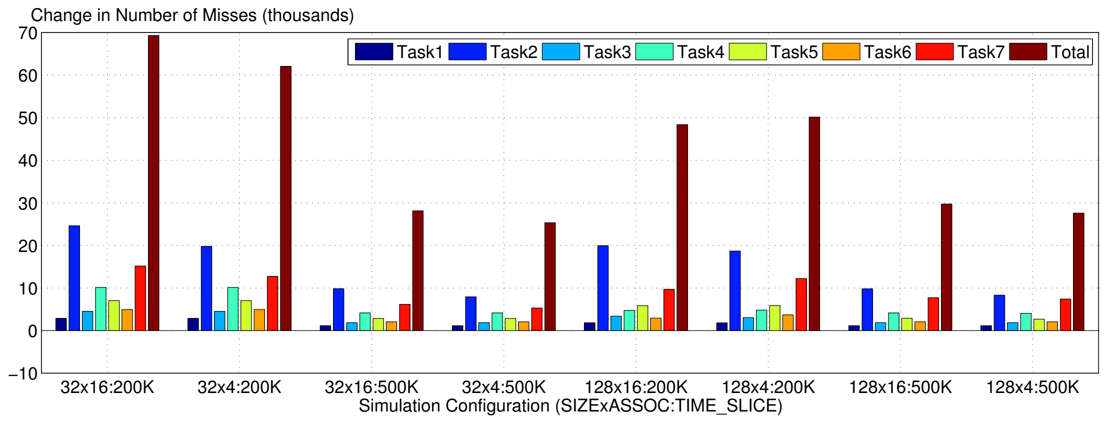


Figure 3.8: Miss change breakdown with ELS preloading for B1 on top, B8 on bottom

3.7 Summary

In this chapter, we have presented context-aware TLB preloading (CTP), a method to reduce TLB misses in multitasked workloads. CTP requires profiling the application to identify hotspots, which can be done manually using a tool such as `gprof` or achieved by using profile guided optimization (PGO) compiler technique. When executing a given hotspot, CTP utilizes a synergistic relationship between the OS and the compiler to inspect the process state and preload the *extended live* set (ELS) - the set of memory mappings that will be required during the upcoming process time slice. The implementation of CTP preloads the current and 4 "near-future" VPNs extracted from registers and memory locations used for array pointers at the start of every process time-slice. The experimental results showed up to 48% reduction in overall TLB miss rates which in turn results in a more accurate estimation of worst-case execution time (WCET).

Instead of using profiling to improve performance, as we have done in this chapter, in the next chapters we will examine instrumentation-driven techniques that enable analysis of dataflow applications. In chapter 4, we will use a traditional profiling technique with a novel instrumentation framework to facilitate converting legacy designs to DBD semantics. Then in chapter 5, we enhance the instrumentation technique to create a framework that can be used to validate dataflow properties in DBD applications.

Chapter 4: Instrumentation-driven Model Detection and Actor Partitioning for Dataflow Graphs

In this chapter, we continue to use profiling and combine it with a generic dataflow instrumentation technique to facilitate converting of legacy designs to DBD semantics. First, we introduce a generic method for instrumenting dataflow graphs that can be used to profile, measure various statistics, and extract run-time information. Second, we use this instrumentation technique to demonstrate a method that facilitates the conversion of legacy code to dataflow-based implementations. This method operates by automatically detecting the dataflow model of the core functions being converted. Third, we present an iterative actor partitioning process that can be used to partition complex actors into simpler sub-functions that are more prone to analysis techniques. We demonstrate the utility of the proposed approach on several signal processing applications. Material in chapter was published in a preliminary from in [62, 63].

4.1 Introduction

Modern digital signal processing (DSP) systems run sophisticated algorithms on high-performance platforms based on field programmable gate arrays (FPGAs), programmable digital signal processors (PDSPs), and multiprocessor system-on-chip (MPSoC) devices.

As a result, designing these systems is a complex process prone to inefficiencies and mistakes.

Formal specifications define the expected system's behavior while abstracting away the implementation details. This allows the designer to evaluate system-level trade-offs without worrying about the underlying details. The correct behavior of the final implementation can then be verified by comparing the observed behavior to the expected behavior of the original specification.

Many different specification formalisms have been introduced over the years and range in strictness of semantics and applicability to various problems. Examples of such formalisms include finite state machines (FSMs) [33], Petri nets [34], Kahn Process Networks (KPNs) [64] and synchronous dataflow graphs [27].

The techniques for verifying the correct behavior for systems designed using specification formalisms vary in both complexity and correctness guarantees. At one extreme, formal methods are impractical for large or complex systems but can guarantee correctness when applied (e.g., FSMs). At the other extreme, unit testing can be used to test a variety of systems for correctness, but offer no guarantees in finding all the defects. The quality of the unit test depends on the code coverage provided by the inputs used to exercise the system.

The complexity of modern systems and the shrinking of the transistor's feature size increase the likelihood that the verification techniques will be unable to guarantee a defect free system. The manufacturing variability in the intricate fabrication process often results in defective devices. In addition, the increases in complexity of the software that runs on the fabricated hardware make it challenging for modern software tools to guarantee correct operations. As a result, a variety of fault-tolerance strategies are employed to combat the potential hardware and software defects.

Design tools, including dataflow modeling, are often used to help with the design process. Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models have been developed for dataflow-based design (DBD). DBD allows a designer to decompose a complex system into simpler sub-functions (actors) that are connected to form a graph. A variety of dataflow modeling tools can then be used to verify correctness of the graph and optimize the entire system (e.g, see [25, 27, 30, 65]).

When employing DBD techniques, it is useful for a designer to find a match between his actors and one of the well-studied models, such HSDF, SDF, CSDF, or BDF, described in Section 2.3. When such a match is found, one can systematically exploit specialized characteristics of actors that conform to the models, and take advantage of more effective, model-specific methods for analysis and optimization. For example, if a dataflow model match cannot be found, a less efficient, generic scheduler and more conservative memory allocation may need to be employed.

Economic factors necessitate reuse of existing designs with periodic upgrades to keep up with technological advances while saving on the non-recurring engineering costs associated with new designs. For example, the Large Hadron Collider (LHC) used for high energy physics experiments is planned to undergo a periodic series of large technology upgrades to allow for new experiments and the expansion of existing experiments [66]. Having a dataflow representation of such a system can alleviate this upgrade process by facilitating correctness verification, and in some cases enabling the use of automatically generated implementations for the new hardware [18,19]. DSP systems that are not designed using DBD, including legacy systems, are more difficult to upgrade, since implementation details can lead to errors that are hard to detect. For this reason, deriving dataflow graphs for these systems is beneficial and is increasingly done even though converting existing DSP code to dataflow graphs can be difficult and time consuming (e.g., see [5]).

In this chapter, we introduce a method to facilitate this conversion process by automatically detecting specialized dataflow models that can be used to represent key functions being converted. By taking more generally described actors and automatically identifying them as instances of specific dataflow models, we enable the application of model-specific analysis and optimization techniques, which are often much more powerful than general purpose techniques [67]. We further demonstrate an iterative model partitioning process that can be used to help decompose complex actors, whose behavior does match specific dataflow models, into simpler actors that are more prone to analysis techniques.

To accomplish these goals, we propose a generic instrumentation framework that enables extracting actor-specific as well as system-wide state. This flexible instrumentation framework enables targeting only the desired part of the dataflow graph to debug graph components and extract various statistics.

As with unit testing, we rely on significant coverage of the behaviors of an actor instead of requiring a formal solution to analyze the code of the actor itself. While this requires good tests to exercise all of the behaviors that would occur during running the application in a real world environment, our approach need not understand the language or build process of the target-specific actor, which makes it applicable to a wide variety of

DSP design scenarios. Designers are free to focus on the correct, efficient implementation of the actor, while the proposed model-detection design instruments the dataflow graph to generate trace information during each test. The trace information is then automatically processed to infer the dataflow model. Although our current implementation detects static and control-based dynamic dataflow models – in particular, CSDF, HSDF, SDF, BDF, and IDF – the design is extensible to detect other models as well.

We demonstrate that correct dataflow models can be extracted with minimal overhead for different components of the triggering system in the LHC. The performance of the model detection algorithm is not related to the complexity of the actor, but rather is a function of the trace file length. However, in order to achieve appropriate coverage, analysis of complex actors may result in longer trace files. Our proposed approach for detection of data independent models has a run time complexity of $O(kn \log n)$, where n is the length (number of actor firings) of the trace file, and k is the number of actor ports. The complexity for detection of data dependent dataflow models increases to $O(ckn \log n)$, where c is the number of different control token values detected. This is a for a specific form of dynamic dataflow behavior, such as that exhibited by the BDF and IDF models of computation, in which dynamic token and production rates are determined as functions of tokens on selected ports. These selected ports are referred to as control ports, and the tokens that they carry are referred to as control tokens. Careful selection of inputs for maximum coverage helps to maximize the accuracy of dataflow model detection while minimizing the run time cost.

4.2 Related Work

Finding a match between the actors within a design and one of the well studied dataflow models enables the use of model-specific analysis and optimization methods. More restrictive models generally offer stronger analysis and optimization techniques. As a result, designers often try to find a match to the most restrictive model, which can still model the actor's behavior.

For example, [68] present a method to classify general dataflow actors into known models of computation (MoCs). The approach uses formal analysis of the SystemC FSM describing the actor to identify the actor as SDF or CSDF. While this formal analysis based approach can definitively identify dataflow models, it requires the actor to be represented by an FSM, which is not always possible, and furthermore, this approach is language specific. Such language-specific approaches provide the designer convenient methods to test individual functions, but lack the ability to provide arbitrary insight into the state of a system as a whole. Our proposed method is complementary to this approach – for example, our method can be used to provide post processing for instrumentation-driven detection of model properties that are not detected using the formal methods applied in [68].

In an effort to improve the performance of reconfigurable video coding (RVC) programs, [69] propose classifying dynamic actors defined as part of the RVC standard as instances of more restrictive MoCs. Their approach relies on converting general RVC-CAL actors into an abstract form, which can then be analyzed systematically. In their follow-on work, [70] improve the accuracy of their classification algorithm by adding an additional transformation by converting RVC-CAL actors to an intermediate representation (IR) and then analyze the IR using abstract representation. They further enhance the ability to detect time-dependent behavior by utilizing a satisfiability solving library (SMT-LIB). Although the authors claim that their approach is not CAL specific, significant effort would be required to apply such model detection to actors represented in a different language. In addition, the results in [70] show that such a classification method works best for small actors that are more amenable to analysis. Our trace based approach relies on availability of effective unit tests that exercise all of the behaviors that would occur when running the application in a real world environment, and is equally applicable for large complex actors.

Aspects of our instrumentation method are related to the actor programming model in Ptolemy [71]. We provide a detailed discussion on this relationship in Section 4.4.

While we apply DICE as the underlying unit testing framework for the implementation and experiments reported on in this chapter, our model detection methodology is not dependent on DICE or any specific kind of test suite implementation approach. Our methodology can readily be adapted to work with other kinds of unit testing frameworks (e.g., see [72] and reference in [37]).

Our work is also useful when working with system descriptions in general dataflow programming environments, such as CAL [73], by allowing tools to automatically detect specialized dataflow models that can be used to help streamline later stages of the design flow. However, unlike automated tools that have been developed for CAL and related frameworks (e.g., see [74]), the methods that we propose in this chapter are trace-driven, and hence do not depend on any specific DBD language or intermediate representation

4.3 Dataflow Graph Instrumentation

Dataflow graph instrumentation provides a modular and flexible approach for extracting run-time information, which can be used to help debug incorrect behavior, measure performance, or see how different forms of execution state evolve as a graph executes. The enable-invoke interface provided by the LIDE framework is a convenient mechanism to build upon for instrumenting dataflow graphs. As described by [30], the `enable` and `invoke` functions correspond to testing for sufficient input data, and executing a single firing (invocation) for a given actor, respectively.

An example of an EIDF graph along with a simple form of scheduler, called the *canonical scheduler*, for EIDF is shown in Figure 4.1. The canonical scheduler is usually not efficient for implementation purposes, but for simulation and testing processes, such as those relevant to model detection, it is useful as a simple, generally-applicable scheduling method that can easily be applied for instrumentation purposes. The instrumentation approach described below can easily be extended to work with other scheduling mechanisms.

When an actor is executed by the canonical scheduler, the `enable` function for the actor is called. This function, which is a basic actor primitive in the EIDF model, returns `TRUE` if and only if the actor has a sufficient number of tokens on each input edge to allow for a complete firing of the actor in its next mode. If the enable function returns `TRUE`, then the actor is executed using the invoke function; otherwise, the `enable` function of the next scheduled actor is called. The canonical scheduler applies this two-phase process (a call to `enable` followed by a conditional call to `invoke`) on a given sequence of graph actors.

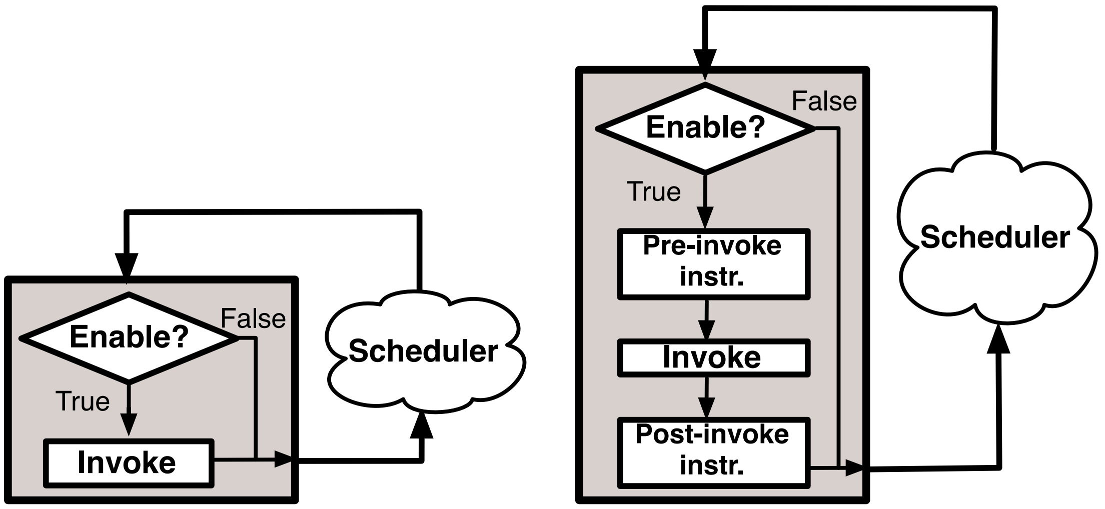


Figure 4.1: Using the enable-invoke interface to instrument dataflow graphs: a) an illustration an EIDF graph that is executed by the canonical EIDF scheduler; b) An instrumented dataflow graph.

To support model detection and related applications of dataflow graph instrumentation in a structured way, we propose "instrumentation extensions" just prior to and just after execution of the `invoke` function, as illustrated in Figure 4.1. By inserting appropriate forms of instrumentation before and after an actor fires, developers can expose powerful insight into the actor's state, and patterns or useful statistics that can be derived to characterize the progression of actor execution state over time. Such an approach enables the developer to precisely capture relevant changes in a graph state caused by the firing of an actor, which is a powerful technique that can be used for debugging purposes, as well as for understanding characteristics of actor and subsystem operation.

Based on the returned value of the `enable` function, we can precisely determine if a given actor will fire, and we can insert *pre-invoke instrumentation* (`pre_ins`), as shown in Figure 4.1. After the invoke function finishes, the *post-invoke instrumentation* (post_ins) can execute to complete instrumentation of the actor associated with its most recent firing. For example, by observing the populations of the input FIFOs before and after an actor fires, one is able to compute the consumption rate. Similarly, the actor execution time can be obtained by recording the clock during pre_ins and comparing it to the clock obtained in post_ins, after the actor finishes executing.

For relatively simple forms of instrumentation and coarse-grain actors, the proposed instrumentation approach adds minimal overhead to the scheduler. Furthermore, in the simulation/testing time context where we use instrumentation for model detection, significant overheads can often be tolerated (compared to actual run time overhead in an implementation). In addition to executing the enable and invoke functions for each actor in the given actor ordering, our *instrumentation augmented scheduler* (*IAS*) executes pre_ins and post_ins functions associated with the actors of interest (i.e., for each actor appearance in the schedule that the designer wishes to instrument). The designer determines the exact behavior of the pre_ins and post_ins functions such that only the state and statistics of interest are examined. By having full control of instrumentation functions, and allowing definition and use of arbitrary state within these functions, the designer can instrument the actor a specific number of times (e.g., the first time the actor fires, every time the actor fires, every other time the actor fires, or every time the actor is in a specific mode).

4.4 Comparison to Ptolemy's Prefire and Postfire

Aspects of our instrumentation method are related to the actor programming model in Ptolemy [71]. Similarly to the `pre_ins` and `post_ins` constructs, Ptolemy's AtomicActor API contains `prefire` and `postfire` methods, which execute before and after an actor fires (see [75]). However, there are some important differences:

- `Prefire` / `postfire` are part of the actor implementation.

- `Pre-invoke` / `post-invoke` are separate from actor implementation. Instead, they can be viewed as part of the scheduler implementation, although their modularity allows the same library of `pre-invoke` / `post-invoke` methods to be used in different schedulers. A related point of distinction is that execution of `pre-invoke` and `post-invoke` can easily be limited to selected parts of a schedule — for example, for a schedule $S = (A, B, C)$, `pre-invoke` could be executed before A, `post-invoke` could be executed after C, and all other `pre-invoke` / `post-invoke` executions for these actors could be bypassed. This allows the designer to obtain information at the boundaries of selected subschedules while avoiding the overhead of executing `pre-invoke` / `post-invoke` at the level of every actor.

- `Prefire` / `postfire` cannot be used to obtain the kind of consumption/production information that we seek, since tokens may arrive during the multiple `fire()` invocations. This distinction arises because one dataflow firing (i.e., discrete unit of dataflow actor execution, as discussed in [76]) may occur through multiple invocations of the `fire()` method.

- `Prefire` / `postfire` can access the actor's state information, while `pre-invoke` / `post-invoke` cannot.

- `Pre-invoke` / `post-invoke` can access various forms of data associated with schedule execution, as provided by the dataflow instrumentation (DFI) context.

To summarize, in contrast to Ptolemy's `prefire` / `postfire` methods, the `pre-invoke` and `post-invoke` methods were specifically designed to instrument dataflow graphs, and are therefore extensible to accomplish a variety of tasks, including collecting trace data necessary for model detection, that are relevant to dataflow graph instrumentation.

4.5 Model Detection Notation

In this section, we introduce the formal notation that is used to precisely define the problem that this chapter attempts to address. We then describe an approach to solve the problem in the Section 4.6, which is the core contribution of this chapter. As discussed in Section 2.3, a dataflow graph G is an ordered pair (V, E), where V is a set of vertices (*actors*), and E is a set of directed edges. Actors represent computations while edges represent communication links between them.

We define:

$$\mathcal{A} = \{a_1, a_2, \ldots, a_n\} \tag{4.1}$$

as the set of all actors of interest in a given DBD scenario (e.g., a DSP system design project or group of related projects). For example, these could be the set of all actors that are available across all of the actor libraries accessible to the design team. In the same design scenario, suppose that

$$\mathcal{M} = \{m_1, m_2, \ldots, m_z\} \tag{4.2}$$

is a group of comparable models that make up the "universe" of available models (analogous to how the LIDE universe is depicted in Figure 2.1). Furthermore, assume that the m_is are ordered in increasing generality (m_a is more restrictive compared to m_b whenever $a < b$). Intuitively, $\mathcal{M}$ is the set of available dataflow models of computation in the given design scenario, and we assume that the models in $\mathcal{M}$ are comparable, as discussed in Section 2.3. In conjunction with the notion that $\mathcal{M}$ is the model universe, we assume that each actor in $\mathcal{A}$ conforms to at least one of the models in $\mathcal{M}$.

We define the *actor set* $\mathcal{A}_k \subset \mathcal{A}$ of each model m_k as the set of all actors in $\mathcal{A}$ that conform to model m_k. It is important to note that some actors can be represented by multiple models, which means that $\mathcal{A}_k \cap \mathcal{A}_l$ can be nonempty for $k \neq l$. In fact, since $\mathcal{M}$ is assumed to be ordered in terms of increasing generality, we will have $\mathcal{A}_k \subset \mathcal{A}_l$ for $k < l$.

We define $\mathcal{R}(a)$ as the set of all models in M that actor a conforms to, and we define the most specialized model (MSM) for an actor a as:

$$\text{MSM}(a) = \min\{i \mid m_i \in \mathcal{R}(a)\}. \tag{4.3}$$

That is, $\mathrm{MSM}(a)$ is the most specialized model in the model universe to which a conforms. For a given actor, the model detection problem can then be defined as: given an actor $a \in \mathcal{A}$, determine $\mathrm{MSM}(a)$.

For example, consider the LIDE universe of Figure 2.1. We can represent this model universe as

$$\mathcal{M} = \{\mathrm{HSDF}, \mathrm{SDF}, \mathrm{CSDF}, \mathrm{BDF}, \mathrm{IDF}, \mathrm{CFDF}, \mathrm{EIDF}\}, \tag{4.4}$$

and given an actor a, the model detection problem amounts to determining which of these seven models is the most specialized model that a conforms to.

Note that we have assumed that each actor conforms to at least one m_i only for simplicity and conciseness. The formulation in this section can easily be adapted to handle actors in $\mathcal{A}$ that do not belong to any of the models in the universe (e.g., because of bugs in the implementation or documentation of the "misfit" actors). In such cases, the model detection problem formulation can be extended to allow for the additional "output" value of $\perp$, which represents that the given actor does not conform to any of the models in the universe.

4.6 Model Detection Process

Our proposed model detection methodology is illustrated as the iterative process shown in Figure 4.2. The given legacy code is converted to a generic LIDE-compatible dataflow format in the first stage. The dataflow instrumentation methodology discussed in the Dataflow Graph Instrumentation section is then used to analyze the LIDE-compatible component and determine whether its behavior matches one of the recognized dataflow models (i.e., one of the models from the universe of supported models). If such a match is not found, the original legacy code can be partitioned into sub-functions. Each of the sub-functions is then made LIDE-compatible and model detection is performed again. This iterative process can continue until a dataflow model is found for the each of the sub-functions or until no further partitioning can be made. In our implementation, all of the steps in the model detection process are performed in conjunction with DICE features for unit testing.

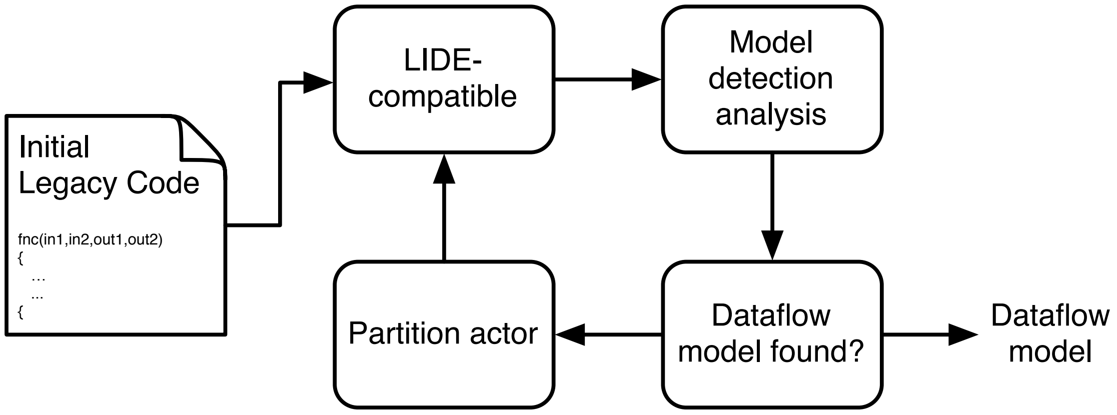


Figure 4.2: Iterative model detection process

4.6.1 Transformation of Legacy Code to LIDE-compatible Format

Figure 4.3 shows the steps in converting generic code to an LIDE-compatible format. The initial transformation step entails adding an LIDE-supported FIFO for each input and output port. In the next step, the `invoke` and `enable` functions required for LIDE compatibility are created. In the example of Figure 4.3, the `invoke` function is set to `fnc`, such that when the `invoke` function for this block is called, `fnc` would execute. The `enable` function is created to return `TRUE` when the input FIFOs have enough tokens to fire and `FALSE` otherwise. Our approach to validating correctness of this transformation relies on the availability of a collection of unit tests that can be used to populate the input buffers with an appropriate quantity of tokens, such that all of the input tokens are used after some number of invocations of the `invoke` function.

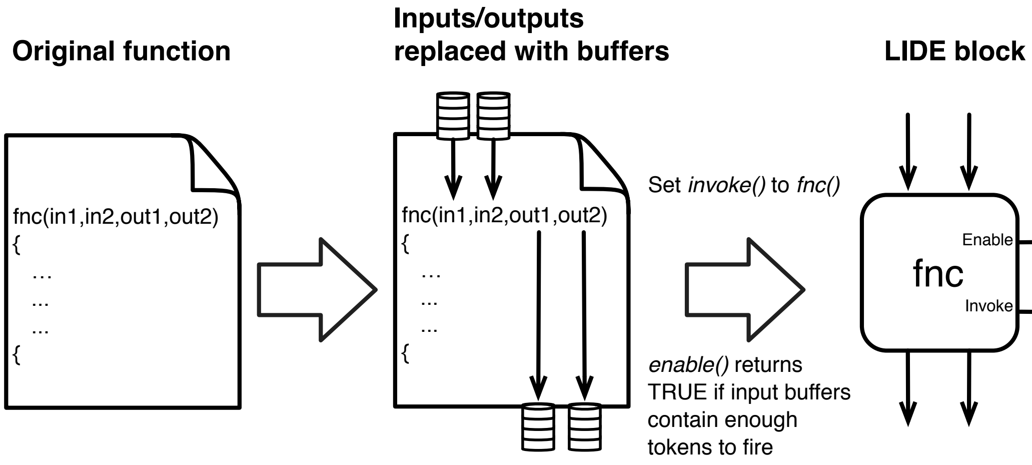

Original function
Inputs/outputs replaced with buffers
LIDE block
fnc(in1,in2,out1,out2)
{
...
...
...
{
fnc(in1,in2,out1,out2)
{
...
...
...
{
Set *invoke()* to *fnc()*
enable() returns TRUE if input buffers contain enough tokens to fire
fnc
Enable
Invoke

Figure 4.3: Converting legacy code to an LIDE-compatible dataflow block involves adding input/output buffers and creating `enable` and `invoke` functions.

The LIDE-compatible block created with this transformation conforms to the generic EIDF model. Doing further analysis to determine whether the LIDE-compatible block conforms to a more restrictive model, such as SDF or CSDF, can enable the use of stronger analysis and optimization techniques than those available for EIDF models. This further analysis step is done utilizing a unit test framework.

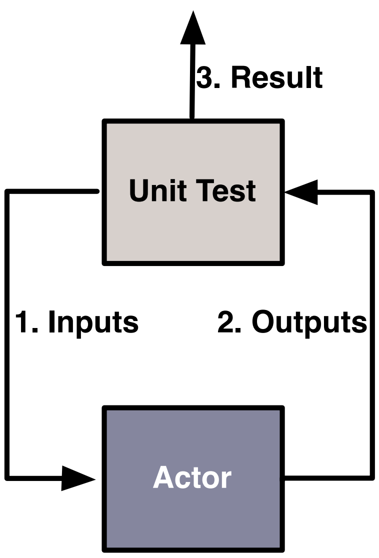
4.6.2 Reappropriation of Units Tests for Model Detection

A typical unit test is depicted in 4.4(a), where test inputs are fed to the module under test (MUT), which in our context is the intermediate dataflow actor being tested, and the outputs of the MUT are saved in the output file. After all the inputs have been processed, the “outputs” file is compared to the expected outputs. The unit test is considered `PASSED` if the expected outputs match the generated outputs, otherwise, the test is considered `FAILED`.

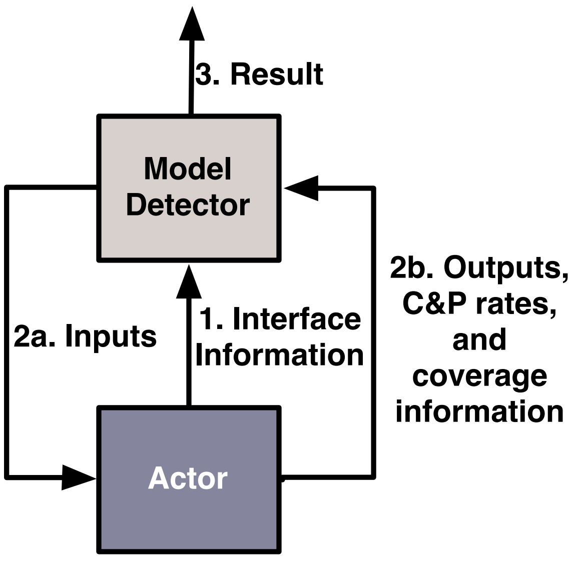
By enhancing an actor's unit test with dataflow graph instrumentation, as introduced in Section 4.3, the designer can glean key properties needed to determine the MSM for the actor. The general design of such an *enhanced unit test* is shown in 4.4(b), where the first step is to provide the actor's interface information to the `pre_ins` and `post_ins` functions, denoted by the *model detector* block. As the unit test executes by feeding inputs to the actor (shown in Step $2a$), the instrumentation functions monitor the state of the actor and extract the consumption and production information (denoted by C&P rates in the figure), as well as the coverage information, as shown in Step $2b$. Using the knowledge about inputs, outputs, and the consumption and productions rates, the model detector can test this data for certain dataflow properties, which in turn can be used to determine the MSM. The result of the enhanced unit test is no longer a PASS/FAIL criterion, but is instead the hypothesized MSM (*detected MSM*) of the actor. The accuracy of this hypothesis is generally as good as the coverage of the associated test suite, and can be improved as the test suite evolves, just as the designer's confidence in functional correctness can be improved.

4.6.3 Model Detection Algorithm

The model-detector block is depicted in Figure 4.5. The input to the block is interface information of the MUT, and the output is the detected MSM of the MUT. The model detector instruments the MUT and extracts runtime information, including the con-



(a) Generic unit test.



(b) Enhanced unit test.

Figure 4.4: The generic unit test can be enhanced to capture the actor's state information used by our model detection algorithm.

sumption and production values after each firing, as well as the coverage information. As discussed in the Section 4.1, coverage knowledge indicates how much of the MUT's typical behavior has been covered, and in some instances, enables generating inputs to exercise new code paths. The hypothesis-generator block cycles through the supported dataflow models and provides the expected pattern for a given dataflow model to the hypothesis-tester block. The hypothesis-tester block uses pattern matching functions to test whether the actor outputs conform to the expected pattern of the MSM hypothesis. If the hypothesis-tester finds a given hypothesis to be `TRUE`, then model detection is complete and that dataflow model is the detected MSM for the MUT. However, low coverage values may lead to false findings of the MSM. For those cases, it is recommended to generate new inputs to more fully exercise the MUT and rerun the model detection test.

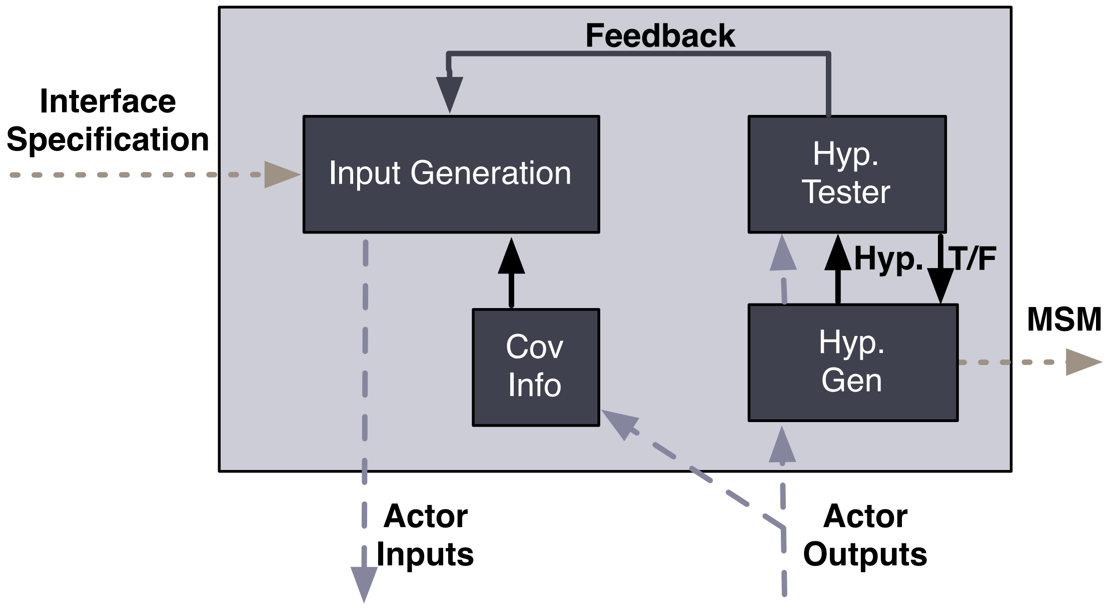


Figure 4.5: Our model detector uses the actor interface information to generate inputs to exercise the actor. The actor outputs are used by the hypothesis generator and tester components to determine the MSM relative to the enclosing dataflow model universe.

The pseudo-code specification of the `hypothesis_generator` function is shown in Figure 4.6. This function inputs the instrumentation-obtained trace information for all the ports and outputs the detected MSM of the actor, as defined by Equation 4.3 in Section 4.5. The `models` array contains all the models being tested, sorted from most restrictive to least restrictive. On lines 6-12, a data independent dataflow model is detected for each port by testing whether the observed consumption/production rates for that port are consistent with the dataflow model being analyzed. If a data independent model was found for each port, the detected MSM for the actor can be found by selecting the *least restrictive* model of all the ports (as shown on lines 13-15).

Tests for data dependent models (shown on lines 16-22) are performed when the trace data does not conform to any data independent models. The behavior of a data dependent model conforms to a static model for fixed values of the control port (since we restrict our detection of DDD models to control-based ones). Thus, data dependent models are detected by testing whether one of the ports is acting as the control. The MSM is found to be one of the data dependent values when a control port is identified for which the behavior of the actor conforms to a specific data dependent model. In cases when the actor does not conform to either, the data independent or data dependent models, the MSM of EIDF is returned since it is the least restrictive model of LIDE (see subsection 4.6.1).

A pseudo-code specification of the `hypothesis_tester` function is shown in Figure 4.7. This function tests whether the inputted instrumentation obtained trace information conforms to the specific model being tested. Our initial implementation can detect HSDF, SDF, CSDF, BDF, and IDF models. The code testing for HSDF (shown on lines 1-5) and SDF (shown on lines 7-11) each take $O(n)$ time, where n is the length of `data`.

A CSDF model is defined by consumption/production rates that repeat in a consistent pattern. We utilize the `findreps` algorithm introduced in [77] to find the positions of all repetitions in data in $O(n \log n)$ time, where n is the length of `data`. Next, `dist_reps` is computed by taking the difference between consecutive elements of the `repetitions` array in $O(n)$ time. Finally, we determine that the test for CSDF is `TRUE` if the separation between all the patterns is the same, and the patterns span the length of `data`. The entire CSDF test (shown on lines 12-21) takes $O(n \log n)$ time. As discussed in the Section 2.3, a BDF model contains a Boolean control port. If the designated control port carries non-Boolean values (lines 23-25), or more generally if there are more than two distinct values observed on the port, then the trace information does not match the BDF model. Note that the key properties and techniques of BDF hold when there are two

```
0:  hypothesis_generator(instr_data)
1:      //models ranked from most to least restrictive
2:      data_independent_models = {HSDF, SDF, CSDF}
3:      data_dependent_models = {BDF, IDF}
4:      models = {data_independent_models, data_dependent_models, EIDF}
5:      //detect data_independent_models
6:      for p in ports
7:          detected_models[p] = NONE
8:          port_data = instr_data[p]
9:          for m in data_independent_models
10:             if hypothesis_tester(m, port_data)
11:                 detected_models[p] = m
12:                 break
13:     if detected_models != NONE
14:         msm = max(detected_models)
15:         return msm
16:     //detect data dependent models
17:     for p in ports
18:         //test if p is control port
19:         for m in data_dependent_models
20:             if hypothesis_tester(m, instr_data,p)
21:                 detected_models = m
22:                 return msm = m;
23:     return EIDF
```

Figure 4.6: Pseudocode for the hypothesis generator block.

distinct values `x` and `y` — the use of `TRUE` and `FALSE` as the actual values is not critical, as the observed values can be mapped arbitrarily into `TRUE` and `FALSE` (`x` = `TRUE`, `y` = `FALSE` or vice versa) for any subsequent analysis and transformation related to the BDF control signals.

Next, a check is performed to verify that all non-control ports abide to some static model for `TRUE` values of the control token (lines 26-29). A similar check is made for `FALSE` values of the control token (lines 30-33). Finally, if all non-control ports exhibit data independent static behavior for a fixed value of the control token, the actor is determined to be BDF (lines 34-36). Each of the two calls to `check_static_models` takes $O(kn \log n)$ time, where n is the length of `data` and k is the number of non-control ports, making the total time to detect a BDF model be $O(2kn \log n) = O(kn \log n)$.

The test for adherence to the IDF model is an extension of the BDF test. Since, the control port can have more values than in a BDF actor, an extra loop is used to iterate over all of the detected values of the control port (lines 44-50). If for all the control tokens, the actor conforms to data independent, static models, then the actor is determined to be IDF (line 51). The total time to check all of the ports is $O(ckn \log n)$, where c is the number of different values of the token in the control port, n is the length of `data`, and k is the number of ports.

4.6.4 Partitioning of an Actor

The granularity (complexity) of actors used in a dataflow-based application specification is associated with important performance trade-offs (e.g., see [78]). Optimizing actor granularity for dataflow actors with respect to specific implementation criteria remains a challenging research problem. Using fine-grained partitioning of an application

```
0:  hypothesis_tester(model, data, ctrl=0)
1:      //all consumption/production values have to be 1
2:      if model == HSDF
3:          if data == 1
4:              return TRUE
5:          return FALSE
6:      //all consumption/production values have to be k
7:      if model == SDF
8:          k = data[1]
9:          if data == k
10:             return TRUE
11:         return FALSE
12:     if model == CSDF
13:         //findreps returns positions of all repetitions in data
14:         repetitions = findreps (data)
15:         dist_reps = diff(repetitions)
16:         //distance between reps has to be the same AND
17:         //patterns needs to span the entire space
18:         if length(unique(dist_reps)) == 1 &&
19:             repetitions[end] + dist_reps[1] > length(data)
20:                 return TRUE
21:             return FALSE
22:     if model == BDF
23:         //control token can only have TRUE/FALSE value
24:         if data[ctrl][ : ] != {TRUE, FALSE}
25:             return FALSE
26:         //for a given ctrl (TRUE), all data ports must conform to a static model
27:         indx = find(data[ctrl][ : ]== TRUE)
28:         //check all non-control ports
29:         rt1 = check_static_models(data[0:ctrl,ctrl+1:end][ indx])
30:         //for a given ctrl (FALSE), all data ports must conform to a static model
31:         indx = find(data[control][ : ]== FALSE)
32:         //check all non-control ports
33:         rt2 = check_static_models(data[0:ctrl,ctrl+1:end][ indx])
34:         if rt1 == rt2
35:             return TRUE
36:         return FALSE
37:     if model == IDF
38:         //control token can have C different values
39:         [C, ctrl_vals] = unique(data[ctrl][ : ]
40:         //for a given ctrl, all data ports must conform to a static model
41:         indx = find(data[ctrl][ : ]== ctrl_vals[1])
42:         //check all non-control ports
43:         rt1 = check_static_models(data[0:ctrl,ctrl+1:end][ indx])
44:         for c in ctrl_vals[2:end]
45:             //for a given ctrl, all data ports must conform to a static model
46:             indx2 = find(data[ctrl][ : ]== ctrl_vals[1])
47:             //check all non-control ports
48:             rt2 = check_static_models(data[0:ctrl,ctrl+1:end][ indx])
49:             if rt1 != rt2
50:                 return FALSE
51:         return TRUE
```

Figure 4.7: Pseudocode for the hypothesis tester block.

can result in high communication cost between multiple levels of actors in a dataflow graph. Conversely, a coarse-grained partitioning of an application may result in having the actors exhibit dynamic behavior, requiring a run-time scheduler, and not being able to take advantage of parallelism offered by finer partitioning schemes.

We propose using the model detection technique to help aid the portioning of complex functions by providing feedback regarding the detected dataflow model. The designer can apply the model detection algorithm to the LIDE actor converted from the legacy function, and observe the detected MSM. If the behavior of the actor conforms to one of the well-understood dataflow models, then further partitioning may not be necessary.

For cases in which the model detection algorithm finds EIDF as the MSM for the initial legacy function, it may be possible to partition the original code into sub-functions such that the individual sub-functions conforms to more restrictive dataflow models. The difficulty of the partitioning process depends on the regularity of the implementation and as a result varies on a case-by-case basis.

As a case study, we converted a JPEG encoder code to be LIDE-compatible. The initial model detection analysis was unable to find a MSM more restrictive than EIDF for the actor. This was largely due to the data dependent behavior of the JPEG encoder, where its consumption and production rates are functions of the size of the input image. However, partitioning the JPEG encoder into subcomponents and performing model detection on these individual parts resulted in detection of several HSDF-based actors. These and other results are discussed in more detail in the Section 4.7.

4.7 Model Detection Evaluation

We used the DICE testing framework to implement dataflow model detection of high energy physics actors that are part of the Trigger system of the Compact Muon Solenoid (CMS) detector of the Large Hadron Collider (LHC) at CERN [79]. Following recommended design practices, unit tests have been created for the core components of the CMS Level-1 Trigger [66] and added to our DICE-based test suite and LIDE-based design framework for these high energy physics design components. We modified these existing unit tests by augmenting the scheduler to instrument the MUT, as discussed in Section 4.6.

The *jet reconstruction* component attempts to identify a group of particles using a sensor grid. The actor has one input port, which consumes an 8x8 grid, where each entry represents a sensor. This results in the consumption of 64 tokens each time the actor executes. The jet reconstruction actor also has two output ports, which correspond to the total energy detected, and a Boolean value indicating if a jet has been identified. The actor produces one token on each of the two output ports.

An example of the trace file containing state information produced by the jet reconstruction actor is shown in Table 4.1. Notice that the consumption values are negative while the production values are positive (i.e., these values are listed in terms of the changes in the associated FIFO populations). The actor has one mode and two output ports conforming to the HSDF model, and one input port conforming to an SDF model. Since SDF is less restrictive than HSDF, the jet reconstruction actor is determined from this analysis to conform to the SDF model.

Table 4.1: Instrumentation results for a jet reconstruction actor.

Mode	*In[0]*	*Out[0]*	*Out[1]*
1	-64	1	1
1	-64	1	1
1	-64	1	1
1	-64	1	1
Model Detected	SDF	HSDF	HSDF

The dataflow model detection results for the CMS actors are summarized in Table 4.2. Having confidence that all of the actors in the system have static dataflow models, the designer can utilize an aggressive scheduler. Combining this dataflow model information with inter-dependency information, which can also be obtained from the dataflow graph, the developer can strategically partition the system to maximize parallelism. Finally, an efficient buffer size for each dataflow graph edge can be derived from the consumption and production rates extracted by the instrumentation code, as well as from knowing that each actor conforms to a static dataflow model (e.g., see [5]).

In addition to the CMS actors, we have tested our improved model detection process on a variety of communication and signal processing actors (see Table 4.3) from the actor libraries within LIDE [26]. This set of actors includes auto-correlator, block adder, coefficient computation, cross-correlator, fast switch, inner product, JPEG encoder, ma-

Table 4.2: Detected dataflow models of various CMS actors.

Actor	*#Inputs*	*#Outputs*	*Cons./Prod. Rates*	*Model Detected*
Jet Reconstruction	1	2	64/1	SDF
Cluster Threshold	12	12	1/1	HSDF
Cluster Compute	12	6	1/1	HSDF
Cluster Isolation	1	2	64/8	SDF

trix inversion, noise adder, and switch, among others. The actors ranged in complexity from less than 50 lines of code, for some of the simple functions, to more than 1000 lines of code, for more complex computations.

The trace information for a *block adder* conforming to a CSDF model is shown in Table 4.4. The block adder has two input ports and one output port. Unlike the previous examples where all the actors had static behavior and only one phase, the block adder cycles through three phases. The actor consumes one token from input port *In[0]* when in phase 2; it consumes one token from input port *In[1]* when in phase 3; and it produces one token to output port Out[0] when in phase 1. This results in all of the ports having repeating firing patterns: {*-1,0,0*} for *In[0]*, {*0,-1,0*} for *In[1]* and {*0,0,1*} for *Out[0]*, indicative of a CSDF dataflow model, which is what the model detector determined. Again, negative values indicate consumption, while positive values indicate production.

The model detection algorithm was able to correctly identify the MSM for 22 out of 23 actors in Table 4.3. Initially, due to a limited unit-test, the model detection algorithm incorrectly classified one of the BDF actors as CSDF. This was caused by the enhanced unit-test cycling the inputs to the control port and resulted in the trace information containing a cycle on every port, reminiscent of CSDF behavior. After refining the inputs to

Table 4.3: Detected dataflow model results for 23 actors that provide various communication and signal processing functions.

Model Detected	*Number of Actors*
HSDF	6
SDF	7
CSDF	2
BDF	2
EIDF	6

Table 4.4: Instrumentation results for a block adder.

Mode	*In[0]*	*In[1]*	*Out[0]*
2	-1	0	0
3	0	-1	0
1	0	0	1
2	-1	0	0
3	0	-1	0
1	0	0	1
2	-1	0	0
3	0	-1	0
1	0	0	1
2	-1	0	0
3	0	-1	0
1	0	0	1
Model Detected	CSDF	CSDF	CSDF

the MUT, the model detection algorithm correctly identified the dataflow model as BDF. This misclassification and our associated rectification of it demonstrate concretely the importance of test suite rigor in the model detection process.

The one actor for which the model detection algorithm was not able to identify a dataflow model more restrictive than EIDF was the JPEG encoder, the implementation of which is derived from [80]. The trace generated when testing this actor was insufficient to determine a dataflow model more restrictive than the generic EIDF. Since the JPEG encoder was composed of more than 1000 lines of code, it was a good candidate for repartitioning – i.e., for decomposition into a network of finer granularity actors.

By manual analysis and manipulation of the code, we partitioned the JPEG encoder into 5 separate actors and applied our model detection algorithm to each one. The results in Table 4.5 show the original JPEG encoder on the first row and the detected dataflow models for the partitioned actors on the 5 remaining rows. The model detection algorithm was able to find HSDF as the MSM for 3 out of the 5 actors. Further analysis revealed that the BMP Reader actor consumes the entire BMP file with one firing and produces the entire data portion of the BMP file on its output port. This results in very limited trace information, insufficient for the model detection algorithm to identify the actor as being anything other than EIDF-based. Investigating the JPEG Prep actor showed that during the first invocation, it consumes the BMP data token from the BMP Reader. On ensuing invocations, the actor outputs a fixed block of pixels until the entire BMP image has been processed. The model detection algorithm was not able to detect an MSM other than EIDF for the JPEG Prep actor, since its behavior is dependent on the size of the BMP image, and since our current model detection system detects only control-based data dependent actors.

Our results on the CMS Detector demonstrate the utility of our model detection approaches on a complex and important application. The partitioning of the complex JPEG actor into simpler actors demonstrates the efficacy of using the proposed approach to explore trade-offs involving actor granularity. The low run time complexity of our

Table 4.5: Detected dataflow models of JPEG encoder actors.

Actor	*#Inputs*	*#Outputs*	*Cons./Prod. Rates*	*Model Detected*
JPEG Encoder	1	1	1/1	EIDF
BMP Reader	1	1	1/1	EIDF
DCT	1	1	1/1	HSDF
Huffman Encoder	3	0*	1/0	HSDF
JPEG Prep	2	2	1/6	EIDF
Quantizer	1	2	1/1	HSDF

*writes the data to a file

model detection techniques (see the subsection 4.6.3) enhances this utility, and facilitates high confidence detection from large traces associated with coverage-intensive test suites.

4.8 Summary

A common problem of modern, high-performance system-on-chip designs is the need for frequent upgrades to keep up with current technology or evolving application requirements. Designers can convert legacy code to dataflow-based implementations to help alleviate this upgrade process, though the conversion can be laborious and time consuming. In this chapter, we have developed a method to facilitate this conversion process by automatically detecting the dataflow models of the core functions, and we have developed techniques to strategically apply the formal model characteristics revealed through such conversion.

We have also developed a generic instrumentation approach that, when combined with traditional profiling tools, can be used to facilitate conversion of legacy designs to DBD semantics. We have demonstrated our instrumentation approach using the lightweight dataflow environment (LIDE) framework and the DSPCAD integrative command line environment (DICE). In addition to supporting our proposed model detection features, this instrumentation-driven approach can be useful in debugging dataflow graphs, measuring performance, and experimenting with system design trade-offs.

Third, we have presented an iterative actor partitioning process that can be used to partition complex actors into simpler sub-functions that are more prone to analysis techniques. In the next chapter, we will extend this instrumentation technique to develop a validation framework that can be used to validate dataflow properties in signal processing systems.

Chapter 5: Instrumentation-driven Validation of Dataflow Applications

In the previous chapter, we presented a generic instrumentation approach that was used to detect instances of well understood models from legacy code. In this chapter, we will extend that instrumentation technique to develop a validation framework that can be used to validate dataflow properties of signal processing applications. Material in this chapter was published in partial, preliminary form in [81].

5.1 Introduction

Dataflow modeling is an important tool often used by the designers of communication and signal processing systems to facilitate system level analysis and optimizations by exposing high level application structure. Such dataflow-based analysis and optimization has the potential to increase design quality in various dimensions — e.g., by taking advantage of parallelism [78], minimizing resource utilization [82], or enhancing the use of vectorization [83, 84]. Developers then apply these optimizations to the implementation of the hardware and software components in the final system.

Formal methods and testing are two important approaches to system verification. Such methods attempt to validate that the system complies with its original specifications. Formal methods use mathematical means to prove that the system meets a specification.

Although such formal methods provide correctness guarantees, their rigor often makes them impractical to prove the correctness of large, complex systems [85].

Testing methods can be used to improve the quality of a product by detecting and removing as many defects as possible, and increasing developer confidence in the proper functioning of the system. Testing methods cannot guarantee the absence of defects, but are easier to use when compared to formal techniques. As a result, testings methods are widely used to test various system properties.

During the implementation process, carefully performed testing and functional validation can mitigate functional errors. Unit tests, in particular, help to validate the correct behavior of each core function by comparing the generated outputs to the expected outputs [86]. However, conventional approaches to testing do not systematically uncover implementation mistakes that cause the violation of dataflow properties specified in the original design. Such violations are difficult to detect if they are not explicitly targeted in the testing process — for example, because they may not result in incorrect output values of a given functional component (actor). However, in signal processing systems, violations of the assumed dataflow properties often lead to sporadically-incorrect behavior of the application as a whole, since key optimizations and design decisions, such as the employed scheduling strategy, may depend strongly on specific dataflow properties.

In this chapter, we present a system level validation technique that complements functional validation that is traditionally done through unit tests. It ensures that system level concerns (e.g., scheduling and buffer management) that make use of declared dataflow properties will not fail due to bugs in the property declarations or due to bugs in the implementations that lead to deviation from the specifications. Our *dataflow valida-*

tion framework (*DVF*) can be used to detect inconsistencies in individual dataflow ports, entire actors, or dataflow subgraphs. The technique builds on a systematic approach to dataflow graph instrumentation introduced in [63].

Our overall DVF validation approach works by (1) having the designer specify expected dataflow behavior for components that the designer wants to test; (2) executing the application, which results in parts of the dataflow graph being instrumented to validate that the observed dataflow behavior matches the expected behavior; and (3) correcting any reported errors that result from the application execution in Step (2). In addition to supporting this dataflow-property-oriented validation approach, DVF facilitates the diagnosis and repair of detected errors by saving the execution state of the application at the time of the violation, thereby providing the developer with insights into the cause of the detected malfunction.

We demonstrate DVF in the context of audio and image processing applications, and show that typical dataflow properties can be validated with minimal amounts of extra processing (i.e., with low run-time overhead). The correctness of the supported validation approach relies on having a thorough set of unit-tests that exercises as much of the dataflow graph as possible. Thus, DVF provides systematic integration of dataflow property considerations into the general framework of unit testing.

5.2 Related Work

By helping to alleviate the problem of system-level validation, the contributions in this chapter address one of the major bottlenecks in design processes for signal processing systems. Our work combines aspects of dataflow modeling, application profiling, and system verification. In this section, we briefly discuss relevant background in these areas and compare our work to the current state of the art.

Many different specification models have been introduced over the years that vary in the strictness of their semantics and their utility for different application domains. Such a system specification approach facilitates the verification and testing processes [37]. For example, the authors in [38] describe the formal specification and verification of a multicore digital signal processor that employs 64 processors. They abstract implementation details by specifying the expected behavior from the programmer's perspective, and then use an assume-guarantee method with a model checker called MOCHA to perform compositional verification.

Dataflow-based specification is used to describe MPEG reconfigurable video coding (RVC) to abstract away implementation specific complexities [39]. In [40], the authors propose using CAL [23] to specify the system, and then perform functional validation using the OpenDF environment [41]. The approach that we present in this chapter complements the functional validation methods provided by these related works in its emphasis on validation of dataflow properties.

Various tools have been introduced to facilitate profiling and complexity analysis for signal processing systems. For example, Ravasi and Mattavelli describe the Software Instrumentation Tool (SIT), which acts as a C virtual machine and allows extraction of information related to fine-grained computation and memory utilization [87]. Like SIT, DVF also instruments the application. A distinguishing aspect of DVF is that it operates at the dataflow level, and focuses on validation of dataflow properties, thus complementing SIT's fine-grained functional analysis.

Causation traces are introduced in [88]. Such traces are obtained from specific executions of the system under development, and represent dataflow actions that have token dependencies. These traces can then be used in offline analysis to improve scheduling and parallelization decisions.

Several high-level modeling languages and tools have been used to carry out system-level analysis, verification and validation, and architectural exploration (e.g., see [35, 36, 89–91]). These tools enable designers to validate the expected behavior of an implementation and check that the resulting software meets the specified performance requirements. Similar to these tools, DVF enables validation of system behavior. However, DVF differs from these tools in its focus on system-level dataflow properties, which are of increasing relevance in the design and implementation of signal processing systems.

Instrumentation-based validation (*IBV*) uses monitors to instrument Simulink applications and find violations of requirements [92]. The validation occurs by checking if the observed run-time behavior captured by the monitors matches that of the requirements, which are encoded as assertions. In [93], the authors demonstrate the utility of IBV by applying it to the validation of automotive controllers. While both IBV and DVF use run-time instrumentation, IBV tests the functional correctness of an application, while DVF uses instrumentation to validate dataflow properties. Although dataflow properties may influence the functional correctness of an implementation, our specialized focus on dataflow properties allows more precise identification of design defects that have such influence.

5.3 Validation Framework

Figure 5.1 illustrates our approach to validating application behavior using DVF. DVF consists of 3 general phases: the offline behavior specification phase, application setup phase, and application execution phase. During the first phase, which occurs offline (before running the application), the designer creates a specification of the expected behavior. The application setup phase consists of processing that specification and instrumenting the dataflow application. During the application execution phase, the instrumentation inserted during the second phase collects run-time information from the application context, and then validates that observed behavior with the expected behavior. The run-time information extracted during the third phase is collected by monitoring and storing information associated with how actors access their input and output edges during dataflow graph execution.

In the remainder of this section, we introduce formal notation that is useful to describe dataflow validation, and we provide further details on the behavior specification phase. Then in Section 5.4 and Section 5.5, we discuss the application setup and application execution phases, respectively.

5.3.1 Dataflow Validation Notation

In this section, we extend the formal notation from Section 2.3 that we use to precisely define the problem that this chapter addresses. We then describe the approach to solve the problem in the ensuing sections, which form the core contribution of this chapter.

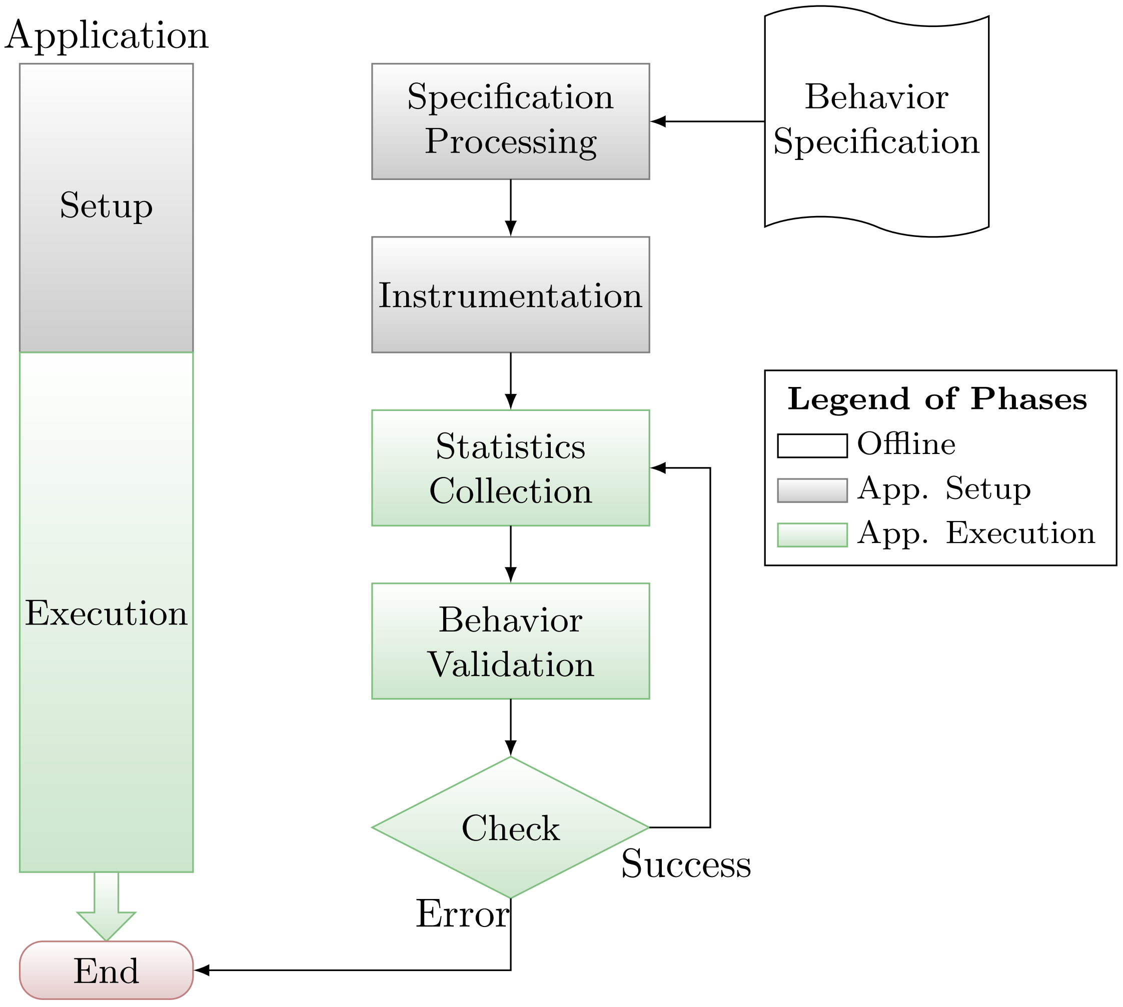


Figure 5.1: An illustration of DVF. The framework includes a setup phase, during which the application is instrumented, and an execution phase, during which run-time behavior is checked against expected behavior.

Recall that an actor's dataflow behavior is a function of the dataflow model to which the actor conforms [5]. Actors exhibiting static behavior have fixed consumption and production rates, while in general production and consumption rates can vary across distinct firings of the same actor. For example, as was shown in Section 2.3, an actor a conforming to the SDF model will by definition have fixed consumption and production rates — $\forall e \in port_{in}(a) : cns(e) \in \mathbb{N}$, and $\forall e \in port_{out}(a) : prd(e) \in \mathbb{N}$, where $\mathbb{N} = \{1, 2, \ldots\}$.

The knowledge that a system's dataflow behavior is static, for example, can lead to better analysis and stronger optimizations (e.g., see [5]). However, these optimization are generally valid only if the assumed dataflow properties hold true. We define $\mathcal{I}$ as the set of declared invariants (or dataflow properties) that the designer believes should hold true throughout system execution.

Applying a minor abuse of notation, we define the invariant set $\mathcal{I}_a \subset \mathcal{I}$ of each actor a as the set of all invariants in $\mathcal{I}$ that define properties for actor a. Similarly, we define the invariant set $\mathcal{I}_e \subset \mathcal{I}$ of each edge e as the set of all invariants in $\mathcal{I}$ that define properties for edge e. Invariants are derived from the dataflow properties encoded in the behavior specification file (described in subsection 5.3.2) during the specification processing stage.

The relevant state of the executing dataflow application can be specified as:

$$Sys = (G, \Pi, \Phi), \tag{5.1}$$

where Π is the set of actors being executed on the available processing elements (PEs), and Φ is the state of the system FIFOs. Here, the state of the FIFO associated with each edge e includes the number of tokens queued on the edge along with the sequence of token values stored.

The validation function maps the set of declared invariants $\mathcal{I}$ and system state Sys to a Boolean value: $Val : \mathcal{I} \times \Sigma \mapsto \mathbb{B}$, where $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$, and Σ represents the set of all possible system states.

For a given system state $x \in \Sigma$, the validation of actor a is successful if

$$\forall i \in \mathcal{I}_a,\ Val(i, x) = \texttt{True}. \tag{5.2}$$

Intuitively, each invariant $i \in \mathcal{I}_a$ is evaluated on the given system state x to determine the validation of actor a. The validation of an edge e can be performed with Equation 5.2 by replacing $\mathcal{I}_a$ with $\mathcal{I}_e$.

Intuitively, a *detectable fault* is a defect in the dataflow graph $G = (V, E)$ (i.e., a defect involving one or more elements of $(\{V \cup E\})$) that leads to a violation of one or more invariants in $\mathcal{I}$ under some system state $x\prime$, where $x\prime$ can be reached by executing the graph from some valid initial state x_o.

In our proposed DVF approach, the statistics collection and behavior validation stages defined in Section 5.5 carry out computations associated with Val, and the process of identifying detectable faults.

5.3.2 Behavior Specification

In the behavior specification phase, the designer specifies the expected dataflow behavior of the actors in the design through a text file, called the *behavior specification file*. This specification file contains a high-level description of the expected behavior for the application components that need to be validated. The description is provided in an XML format, thereby decoupling it from the language-specific implementation of the final design. As a result, the same specification file can be used to validate the high-level, language-agnostic dataflow design as well as the language-specific implementation of the final solution. By using the same specification at the early design stage and later implementation stages, the designer can guarantee consistent, correct behavior and track any errors that may have been introduced during the implementation.

An example specification file for an actor containing three ports is shown in Figure 5.2. Two of the ports are input ports conforming to the SDF model. Each of these ports is specified to have a consumption rate of of 2 tokens per firing. The actor also has one output port, which conforms to the homogeneous synchronous dataflow (HSDF) model, meaning that it has a constant product rate of 1 token per actor firing. By design, the XML-based behavior specification file does not contain any information about the functional behavior of the associated component. This is because DVF is oriented toward orthogonalizing dataflow properties during the validation process so that they can be focused on during testing, and addressed in a method that is independent of the implementation platform or language.

In dataflow graphs containing multiple instances of the same actor, the *actor id* disambiguates which instance needs to be instrumented. Similarly, *port numbers* are used to identify specific ports within a DVF behavior specification file.

5.4 Application Setup Phase

The behavior specification file is applied as input to the application setup phase, and consists of two stages — (1) specification processing and (2) instrumentation — as illustrated in Figure 5.1.

```
<actor>
  <name="adder"/>
    <id=1/>
    <port>
        <number=1/>
        <direction=input/>
        <type=SDF/>
        <behavior=2/>
    </port>
    <port>
        <number=2/>
        <direction=input/>
        <type=SDF/>
        <behavior=2/>
    </port>
    <port>
        <number=3/>
        <direction=output/>
        <type=HSDF/>
    </port>
</actor>
```

Figure 5.2: The contents of a DVF behavior specification file for an adder actor.

5.4.1 Specification Processing

The specification processing stage parses the behavior specification file to construct and initialize interfaces that will be employed in the instrumentation stage. These interfaces will specify how instrumentation code interacts with the associated dataflow graph actors and edges.

For the example of Figure 5.2, the specification processing parser extracts the expected dataflow behavior for each port of the `adder` actor and translates that behavior into a set of invariants that will be checked at run-time. The result of the specification processing phase would be the following set of invariants for the adder actor:

$$\mathcal{I}_{adder} = \{cns(e_1) = 2, cns(e_2) = 2, prd(e_3) = 1\}.$$

The XML parser captures any dependencies that may exist between the ports, and allows the instrumentation and validation stages to utilize such dependency information. For example, a Boolean dataflow (BDF) [94] actor containing a control input port would require the other input ports to specify behaviors for the `True` and `False` cases. This dependency would be captured in the specification processing stage and encoded in the set of invariants held by the expected and observed behavior structures. This knowledge will then be applied during application execution to provide appropriate instrumentation and validation.

5.4.2 Instrumentation

In DVF, the interfaces that are initialized in the specification processing stage dictate the instrumentation required to validate the dataflow behavior of each component. Instrumentation code is generated automatically from the behavior specification file. This includes code to validate the connection of the correct ports in the implementation with the corresponding actors, and results in the insertion of lightweight segments of monitoring code that execute before and after each instrumented actor. We refer to these code segments as *DVF monitoring code segments* (*DMCSs*).

To provide this kind of monitoring code, we have generalized the instrumentation technique introduced in [63] such that the *pre-invoke instrumentation* and *post-invoke* *instrumentation* operations can be applied in a manner that is independent of the type of scheduling strategy used.

The DMCSs for each instrumented port execute an associated *statistics collection function* before and after the corresponding actor executes. The type of statistics collection function applied is determined by the port behavior, as specified in the behavior specification file. Both the specification processing and instrumentation stages are parts of the application setup phase, as illustrated in Figure 5.1, and are carried out before the application starts executing.

In the example of Figure 5.2, DMCSs would be added for each port, resulting in a dataflow graph model of the form depicted in Figure 5.3. As the application executes, the DMCSs for each instrumented port invoke the *statistics collection* process before and after `Actor 1` fires.

5.5 Application Execution Phase

The execution phase of DVF contains two stages, as illustrated in Figure 5.1. In this section, we present details on the operation of these two stages.

5.5.1 Statistics Collection

The separation of specification processing, instrumentation, and statistics collection activities into separate stages results in a flexible framework capable of validating a variety of dataflow behaviors, while simultaneously allowing reuse of common validation components. The *statistics collection* stage collects selected statistics of each in-

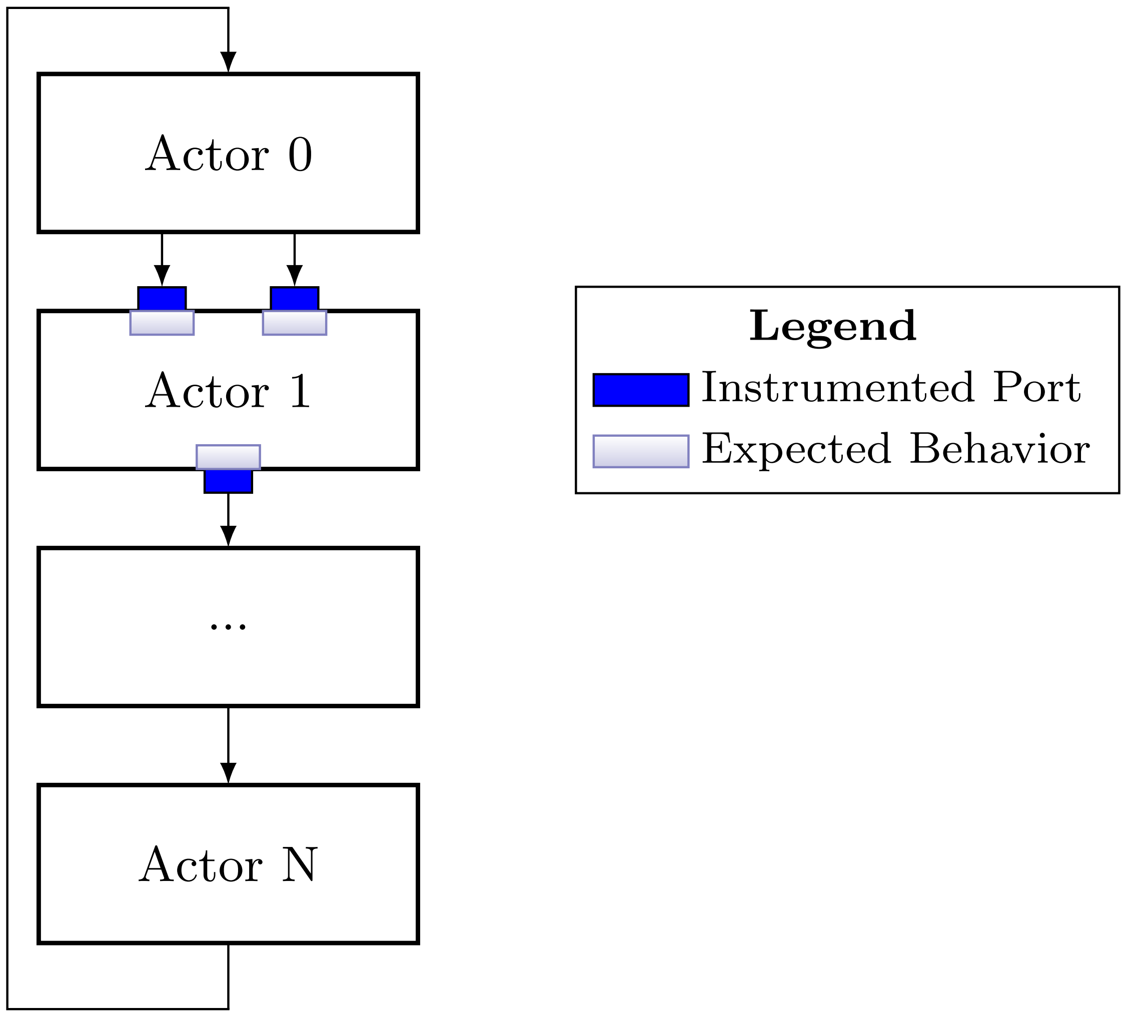


Figure 5.3: The expected behavior for each port, shown in dark, is extracted from the behavior specification file during the specification processing stage. DMCSs, represented by the light shade, are added to each instrumented port such that the associated statistics collection function is called before and after the actor fires.

strumented component when the associated actor fires. The type of data collected varies for each port and depends on the assigned statistics collection function, which is selected from a library of available functions during the instrumentation stage. The flexibility offered by DVF enables the designer to add new statistics collection functions to the library as desired.

The validation of the behavior for static dataflow models, such as HSDF and SDF, requires collecting the consumption and production rates for each instrumented port when the associated actor fires. The resulting statistics collection function collects the FIFO populations before and after the instrumented actor fires, thus enabling computation of the consumption and production rates.

Collection of data required to validate dynamic dataflow behavior generally requires more complex statistics collection functions that collect more kinds of data. For example, the statistics collection function for a BDF actor, whose behavior varies depending on the input to the control port, collects the value of the token at the control port as well as the consumption and production rates for the data ports.

Figure 5.4 shows that the collected information from an instrumented component is stored in an entity called the *observed behavior structure*. The observed behavior structure in DVF can be viewed as a placeholder for an arbitrary data structure that is used to store and organize collected statistics during execution. The complexity of the observed behavior structure depends on the behavior being validated. For example, for an SDF model, the observed behavior structure can be implemented as a vector, where each element corresponds to the consumption or production rate of a specific actor port, and for cyclo-static dataflow (CSDF) [28], one can use an array of vectors with each port represented by a separate vector of non-negative integers.

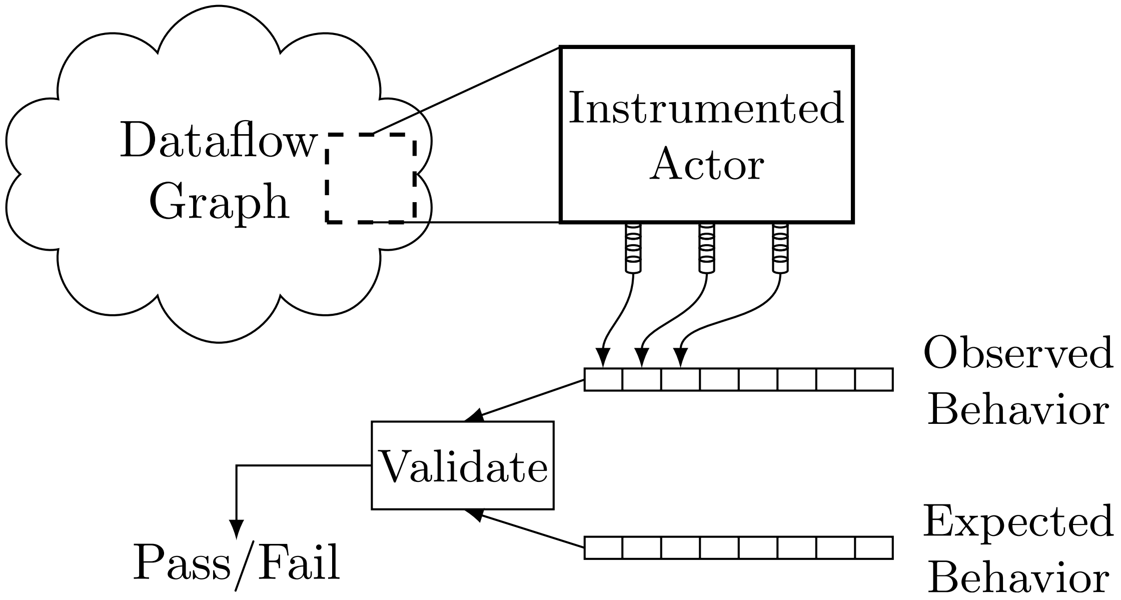


Figure 5.4: The statistics collection stage obtains and stores data about observed dataflow behavior during each actor firing and passes the collected information to the behavior validation stage. The behavior validation stage in turn compares the observed behavior with the expected behavior, and reports any inconsistencies as errors. Details on any detected inconsistencies are also reported to help the designer identify and fix the errors.

5.5.2 Behavior Validation

The behavior validation stage involves comparing the observed behavior structure populated during the statistics collection stage with the expected behavior structure that is defined in the specification processing stage. To ensure that the validation process does not report false errors from the beginning of application execution, the observed behavior structure is initialized to the expected behavior. Such an initialization translates to the premise that all the invariants for the application are assumed to be `True` until proven otherwise.

Because the DVF validation process is carried out concurrently with application execution (rather than as a kind of trace-based post-processing), an invocation of the validation process does not need to compare the observed and expected structures in their entirety. Only the observed behavior structure elements set during the most recent actor firing need to be compared to the corresponding elements in the expected behavior structure, thus resulting in a significant reduction in the required processing and storage requirements. For example, with the observed behavior vector in Figure 5.4, only the most recent elements populated by the statistics collection function would be compared to the corresponding elements in the expected behavior vector.

Matching of the compared elements in the observed and expected behavior structures results in the pass of the validation check (as in Equation 5.2), in which case the application continues to execute and the validation process continues with the statistics collection stage, as shown in Figure 5.1. However, if the validation fails, execution of the application terminates with an error report, and the execution state of the application is saved to a file to facilitate the diagnosis and repair of the detected "dataflow behavior malfunction". If DVF is operating within an actual application deployment (e.g., rather than during testing), a controlled shutdown or warning indication can be initiated upon detection of a dataflow behavior malfunction. Such proactive fault handling is often preferable to silently allowing the system to enter an invalid execution state that is related to mismatches between design time assumptions and actual implementation characteristics.

5.6 Evaluation

5.6.1 Experimental Setup

To further concretize DVF, we have used the framework to validate dataflow properties for an automatic speech recognition (ASR), acoustic tracking (AT), and JPEG encoder applications. In this section, we describe our experimental setup. The description and results for each application can be found in subsection 5.6.2, subsection 5.6.3, and subsection 5.6.4.

Each application was implemented using the lightweight dataflow environment (LIDE) [32]. In particular, we used LIDE-C, which provides libraries and application programming interfaces to construct dataflow actors and graphs using the C language. The dataflow properties for each component being validated were encapsulated in a separate behavior specification file (as described in Section 5.4). A corpus consisting of 20 audio files, 12 bitmap (BMP) files, and 32 acoustic test files was used as input to the ASR, JPEG, and AT applications, respectively, while validating the dataflow properties for each actor. Each experiment was repeated 50 times per input file to better characterize the effects of DVF in the presence of background applications.

We employed 2 different platforms to conduct experiments. The embedded system experiments were conducted on a Raspberry Pi embedded platform that has an ARM1176JZF-S 700 MHz processor with 256 MB of RAM. Experiments were also conducted on a laptop platform with an Intel(R) Core(TM) i7-2675QM 2.20GHz processor and 6 GB of RAM.

5.6.2 Validation of Automatic Speech Recognition Application

5.6.2.1 ASR: Description

We implemented the embedded automatic speech recognition (ASR) algorithm based on [95], and applied this implementation to experiment with our proposed DVF approach.

We separated the ASR system functionality into 7 actors, as shown in Figure 5.5. These actors are described as follows.

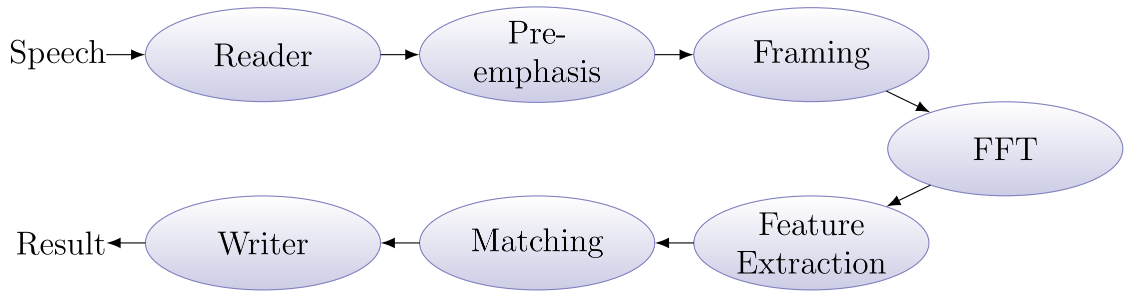


Figure 5.5: The ASR algorithm is separated into seven actors that repeatedly extract a feature set from the input speech signal and compute the closest match against a database of pre-collected speech samples.

Reader

The `reader` detects a voiced signal and partitions the $8\,$kHz-sampled signal into frames that span 256 samples ($31.25\,$ms) each.

Pre-emphasis

The `pre-emphasis` actor processes each frame by applying a first order high-pass filter to compensate for variations in the low and high frequency components of speech.

Framing

The `framing` actor applies a Hamming window to the overlapping frames.

FFT

The `FFT` actor computes the spectrum of the windowed signal.

Feature Extraction

The `feature extraction` actor derives the Mel Frequency Cepstral Coefficients (MFCCs) of the signal by

1. applying the Mel filter bank to the frequency spectrum to compute the Mel spectrum, and
2. applying an inverse discrete cosine transform (IDCT) to the logarithm of the Mel spectrum. The MFCCs are managed as vectors of 15 coefficients each. A single vector of this form is computed for each speech frame.

Matching

The `matching` actor uses a dynamic time warping (DTW) algorithm to find the best match between the input signal and the collection of speech signals.

Writer

The `writer` actor saves the results of the matching actor to a text file.

Dataflow properties for each ASR actor are summarized in Table 5.1. These properties include the dataflow model; the number of input and output ports; and the consumption and production rates. Here, by the "dataflow model" of the actor, we mean the model that the designer declares the actor to conform to or the most specialized model that has been determined for the actor through prior application of relevant analysis tools, such as model detection [63]. The reader produces one extra token corresponding to an SNR value every 20 frames, making it a CSDF actor (the notation in Table 5.1 shows the actor producing 257 tokens during the first firing and then producing 256 tokens during each of the next 19 firings).

Table 5.1: Dataflow properties of ASR actors.

Actor	*Dataflow Model*	*#Inputs/ #Outputs*	*Cons./Prod. Rates*
Reader	CSDF	0/1	0/257,256{19}
Pre-emphasis	CSDF	1/1	257,256{19}/256
Framing	SDF	1/1	200/256
FFT	SDF	1/1	256/128
Feature Extraction	SDF	1/1	128/15
Matching	SDF	1/1	300/2
Writer	SDF	1/0	2/0

5.6.2.2 ASR: Results

Applying DVF to the ASR application in this case study enabled detecting and correcting several critical errors caused by violations of various dataflow assumptions.

Incorrect Buffer Size

A change to the behavior of the reader actor was not propagated to the FIFO connecting it to the rest of the dataflow graph. DVF was able to detect the reader attempting to output more data than what was specified in its behavior specification.

SDF Behavior Violation

To optimize for varying signal-to-noise (SNR) in the input speech signal, the reader was augmented to calculate SNR at the start of every speech sample. During initialization for a given speech sample, the optimized reader actor produces 256 tokens (signal samples), followed by a single token that encapsulates the SNR value. Ensuing invocations of reader result in the output of 256 tokens, as described in Table 5.1, for the remaining processing of the current speech signal. DVF detected the discrepancy between the actual dataflow behavior of the optimized reader, and the specified (obsolete) behavior, which was based on the original reader. This allowed quick detection of the error and corresponding repair of the overall dataflow schedule.

Deadlock Detection

The sizes of the FIFOs connecting ASR actors were adjusted several times during the development process. During one of the design iterations, an error was introduced that caused the ASR application to reach a deadlock because of a misconfigured FIFO between the pre-emphasis and framing actors. DVF was able to identify both actors responsible for the deadlock and provided information about the behavior specification violations that helped in repairing the misconfiguration.

The setup time needed to process the behavior specification and instrument the dataflow graph is shown in the second column of Table 5.2 and Table 5.3. From this, we see that the average setup time is 3.36 ms and 220 µs on the embedded and laptop platforms, respectively. The average time for the entire ASR application without DVF is 70 ms on the embedded platform and 10 ms on the laptop platform. Thus, DVF setup adds about 5% overhead to the run-time of the ASR application on the embedded platform and 2% overhead on the laptop platform.

Recall that in addition to the application setup stage, which occurs once at the start of the application, DVF includes statistics collection and behavior validation stages, which occur every time the instrumented component executes. The run-time overhead caused by DVF for each instrumented component of ASR on the embedded platform is shown graphically in Figure 5.6. For each actor, the figure shows two box plots, where the left plot shows the baseline execution time without DVF and the right plot shows the execution time with the actor being validated using DVF. Table 5.2 shows the mean values of the same results, from which we see that the run-time validation overhead was on average 2.1% of the execution time for the actors tested. The same results for the laptop platform, depicted in Table 5.3, show the run-time validation overhead to be on average 9.1% of the execution time.

Table 5.2: DVF performance results for the actors in the ASR application on the embedded platform.

Actor	*Setup Time*	*Baseline Time*	*With DVF Time*	*DVF Overhead %*
Reader	3.25 ms	56 µs	58 µs	3.6
Pre-emphasis	3.38 ms	102 µs	105 µs	2.9
Framing	3.43 ms	77 µs	78 µs	1.3
FFT	3.41 ms	288 µs	291 µs	1.0
Feature Extraction	3.40 ms	101 µs	105 µs	4.0
Matching	3.46 ms	2.97 ms	2.98 ms	0.3
Writer	3.25 ms	294 µs	300 µs	2.0

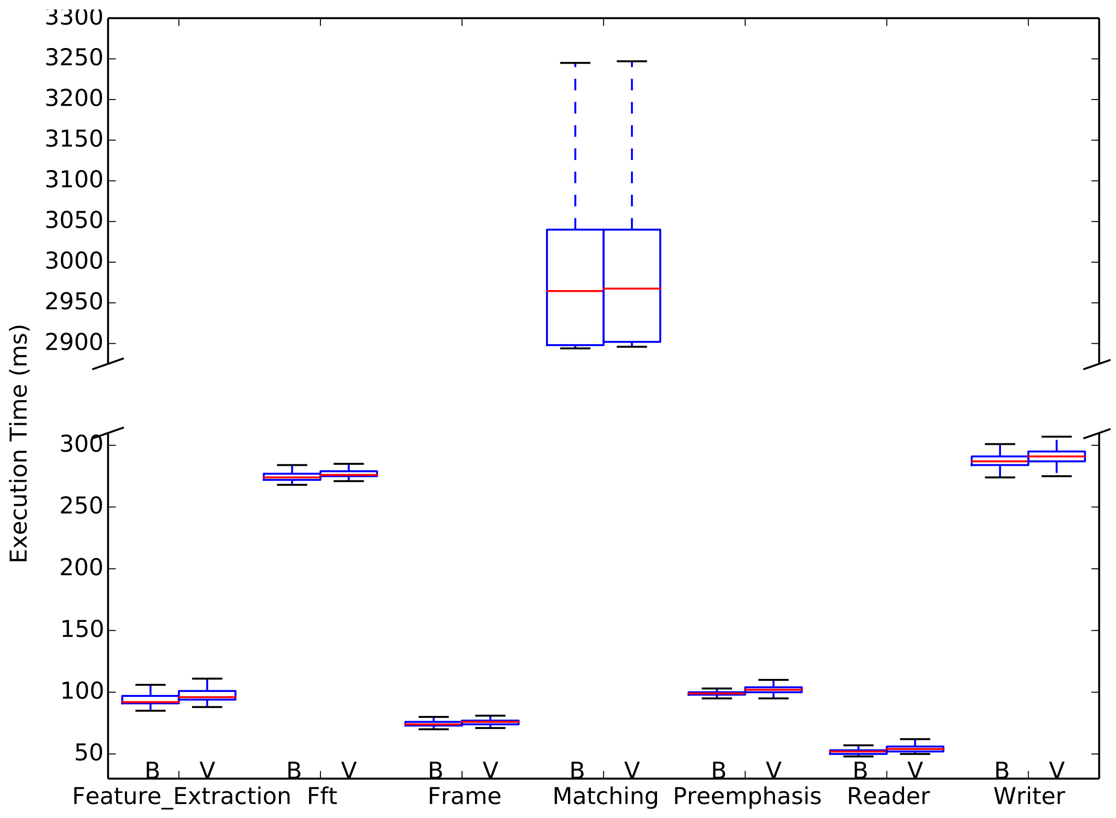


Figure 5.6: Execution time comparison with validation (denoted by V), and without validation ("baseline", denoted by B) on the embedded platform.

5.6.3 Validation of Acoustic Tracking Application

5.6.3.1 Acoustic Tracking: Description

We implemented the acoustic tracking (AT) application based on [96], and applied DVF to validate the 6 actors shown in Figure 5.7. These actors are described as follows.

Source(Data)

The `source` actor reads acoustic data sampled at 8000Hz.

Table 5.3: DVF performance results for the actors in the ASR application on the laptop platform.

Actor	*Setup Time*	*Baseline Time*	*With DVF Time*	*DVF Overhead %*
Reader	157 µs	6 µs	7 µs	16.7
Pre-emphasis	223 µs	9 µs	10 µs	11.1
Framing	209 µs	6 µs	7 µs	16.7
FFT	248 µs	30 µs	30 µs	0.0
Feature Extraction	255 µs	14 µs	14 µs	0.0
Matching	209 µs	758 µs	768 µs	1.3
Writer	144 µs	6 µs	7 µs	16.7

Detection

The `detection` actor detects potential target(s). The detection algorithm works on a frame by:

1. applying a smoothing filter to the data;
2. identifying local minimum and maximum regions; and
3. selecting regions that are outside the desired tolerance (e.g., that deviate from the mean by more than σ^2).

The frame size is configured with a parameter and can take on the values of 10 ms, 100 ms, and 1000 ms.

Feature Extraction

The `feature extraction` actor computes the discrete Fourier transform (DFT) of the envelope from the input data. It then computes the energy in the frequency domain. The feature set consists of DFT coefficients in selected spectrum bands.

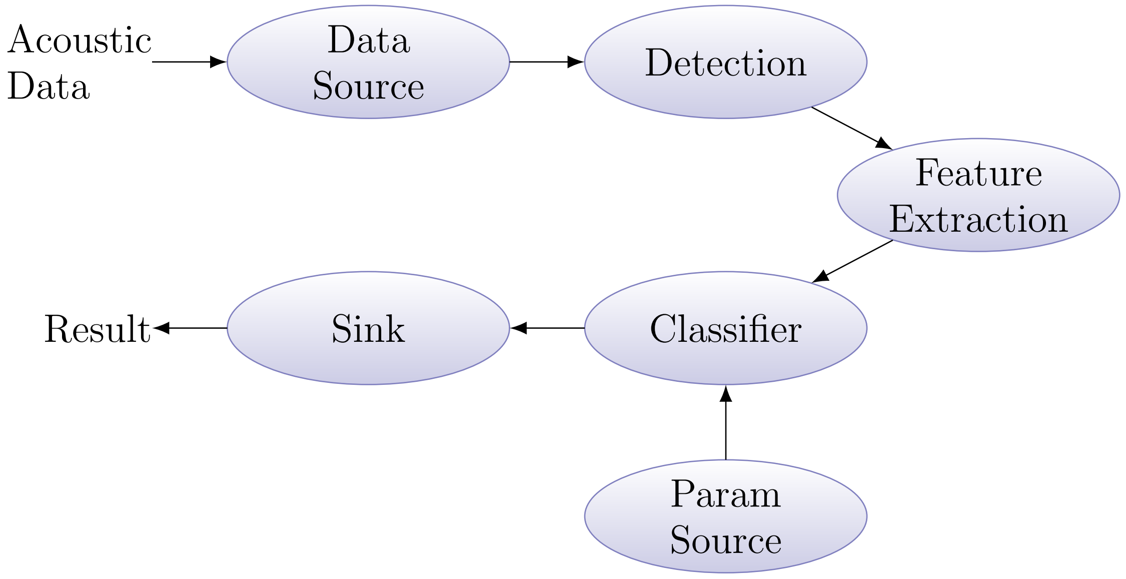


Figure 5.7: The AT algorithm is separated into six actors that repeatedly extract a feature set from the input acoustic signal and compute the closest match against a database of pre-collected acoustic templates.

Classifier

The linear discriminant analysis (LDA) `classifier` actor classifies the signal into one of three classes: human, vehicle, and noise. The actor outputs the classification result and the posterior probability.

Source(Parameters)

The second `source` actor reads the classifier parameters, which are pre-computed from training data.

Sink

The `sink` actor saves the class of each detected target to a file.

The dataflow properties for each AT actor, including the dataflow model, number of input and output ports, and consumption and production rates, are summarized in Table 5.4. The behavior of the detection, classifier, and feature extraction actors can be configured with parameters as shown in the *Cons./Prod. Rates* column of Table 5.4.

Table 5.4: Dataflow properties of actors in the acoustic tracking application.

Actor	*Dataflow Model*	*#Inputs/ #Outputs*	*Cons./Prod. Rates*
Source (Data)	HSDF	0/1	0/1
Source (Params)	PSDF	0/1	0/1
Sink	HSDF	1/0	1/0
Detection	PSDF	1/1	224000 / frame_size
Feature Extraction	PSDF	1/1	frame_size / feature_length
Classifier	PSDF	2/1	30 x num_cases, 6 x feature_length / 1

5.6.3.2 Acoustic Tracking: Results

DVF identified several important violations of the specified dataflow behavior. The following is a list of discrepancies that were found and corrected with the aid of DVF:

- The detection actor was incorrectly specified as SDF. For fixed parameter values, the detection actor is CFDF. As mentioned in chapter 2, CFDF stands for *core functional dataflow*, which is a very general (expressive) dynamic dataflow model [30]. The detection actor consumes and processes input data in one mode, but outputs the data in a different mode. Thus, this CFDF actor needs to be scheduled twice as many times as an SDF version of the actor, which would input, process, and output data in the same mode.

- There was a discrepancy in the specification of the consumption rate for port 2 in the classifier actor.
- As the system design evolved, we had changed the behavior of the Source (Data) actor to output a large frame size (224000 vs. 1), but failed to update the behavior specification file and schedule based on this change.
- An update to the implementation of the classifier actor resulted in a change of CFDF mode assignments and associated consumption and production rates. This change was mistakenly not relayed to the system integrator, but the resulting inconsistencies were detected using DVF.

The setup time needed to process the behavior specification and instrument the dataflow graph is shown in the second column of Table 5.5 and Table 5.6. The average setup time is 3.46 ms and 583 µs on the embedded and laptop platforms, respectively. This is slightly higher than the times seen for the ASR application and can be attributed to the processing of a more complex set of behaviors. The average time to run the entire AT application without DVF is 2.83 sec on the embedded platform and 178 ms on the laptop platform. Thus, the DVF setup adds less than 2% overhead to the run-time of the AT application.

As discussed above, DVF includes statistics collection and behavior validation stages, which occur every time the instrumented component executes. The three right columns in Table 5.5 and Table 5.6 can be used to examine the DVF overhead for the AT actors on each platform. On average, the *baseline* configuration (i.e., execution without DVF) runs 11.4% faster than the configuration with DVF on the laptop platform and

11.9% faster on the embedded platform. The amount of overhead added by DVF is related to the complexity of the behaviors that are being validated. Thus, validation of more complex behaviors will result in higher total execution time. However, the fraction of time that DVF consumes for each individual actor will be highly dependent on the complexity of the actor itself. This explains the large variance in the overheads reported in Table 5.5 and Table 5.6. Because more complex actors typically consume a larger fraction of the application's total execution time, the overall overhead caused by DVF when compared to the baseline is less than 10% on a laptop and less that 15% on an embedded platform.

Table 5.5: DVF performance results for the actors in the AT application on the embedded platform.

Actor	*Setup Time*	*Baseline Time*	*With DVF Time*	*DVF Overhead %*
Source (Data)	3.36 ms	102 ms	103 ms	1.0
Source (Params)	3.34 ms	26 µs	30 µs	15.0
Sink	3.35 ms	53 µs	60 µs	13.2
Detection	3.54 ms	262 µs	274 µs	4.6
Feature Extraction	3.51 ms	319 ms	356 ms	11.6
Classifier	3.67 ms	35 µs	44 µs	25.7

Table 5.6: DVF performance results for the actors in the AT application on the laptop platform.

Actor	*Setup Time*	*Baseline Time*	*With DVF Time*	*DVF Overhead %*
Source (Data)	572 µs	4.45 ms	4.46 ms	0.2
Source (Params)	556 µs	1 µs	1 µs	0.0
Sink	571 µs	7 µs	8 µs	14.3
Detection	599 µs	20 µs	20 µs	0.0
Feature Extraction	595 µs	6.84 µs	7.10 µs	3.6
Classifier	604 µs	2 µs	3 µs	50

5.6.4 Validation of JPEG Encoder

5.6.4.1 JPEG Encoder: Description

We have also applied DVF to validate a JPEG encoder described by the dataflow graph in Figure 5.8. The dataflow properties for each JPEG actor, including the declared dataflow model, number of input and output ports, and consumption and production rates, are shown in Table 5.7. The dataflow properties for each component from Table 5.7 were encapsulated in a separate behavior specification file and were validated using DVF.

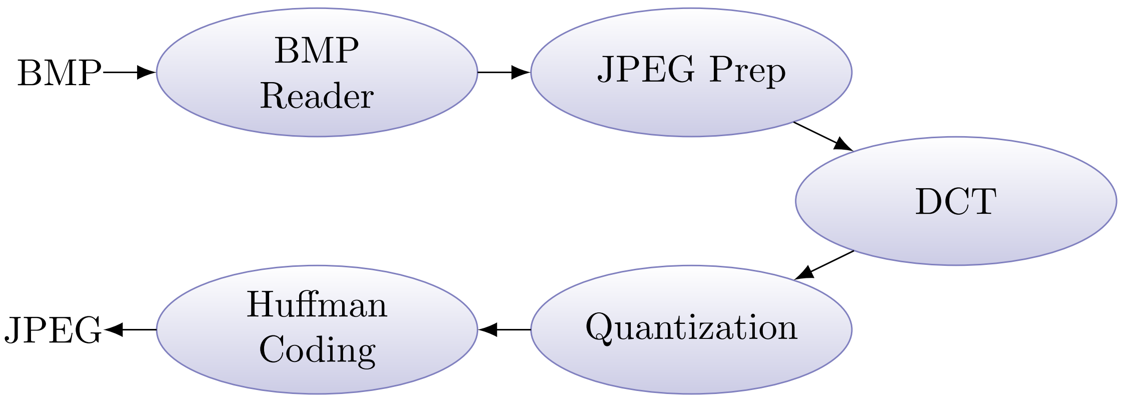


Figure 5.8: The JPEG encoder is separated into 5 actors that encode a BMP file into a JPEG file.

Table 5.7: Dataflow properties of actors in the JPEG encoder.

Actor	*Dataflow Model*	*#Inputs/ #Outputs*	*Cons./Prod. Rates*
BMP Reader	HSDF	1/1	1/1
DCT	HSDF	1/1	1/1
Huffman Encoder	HSDF	3/0*	1/0
JPEG Prep	CFDF	2/2	1/6
Quantizer	HSDF	1/2	1/1

*writes the data to a file

5.6.4.2 JPEG Encoder: Results

The results of our validation experiments are summarized in Table 5.8. DVF found that our initially presumed dataflow model for the JPEG prep actor was incorrect. Thus, optimizations that were made on the assumption that all the actors in the graph exhibit static dataflow behavior were also incorrect and had to be modified.

The setup time needed to process the behavior specification and instrument the dataflow graph is shown in the second column of Table 5.9 and Table 5.10. This setup time is seen to be 2.36 ms and 216 μs on the embedded and laptop platforms, respectively. The average time to run the entire JPEG encoder application without DVF is 510 ms on the embedded platform and 15 ms on the laptop platform. Thus, the DVF setup adds less than 2% overhead to the run-time of the JPEG encoder.

As discussed above, DVF includes statistics collection and behavior validation stages, which occur every time the instrumented component executes. The three right columns in Table 5.9 and Table 5.10 can be used to examine the DVF overhead for the JPEG encoder actors on each platform. On average, the *baseline* configuration (i.e., execution without DVF) runs 8% faster than the configuration with DVF on the laptop platform and 15.1% faster on the embedded platform. Thus, the overall overhead caused by DVF when compared to the baseline is less than 10% on a laptop and less than 20% on an embedded platform.

Table 5.8: Inconsistencies detected with DVF for the actors in the JPEG encoder.

Actor	*Presumed DF Model*	*Actual DF Model*
BMP Reader	HSDF	HSDF
DCT	HSDF	HSDF
Huffman Encoder	HSDF	HSDF
JPEG Prep	HSDF	CFDF
Quantizer	HSDF	HSDF

Table 5.9: DVF performance results for the actors in the JPEG encoder on the embedded platform.

Actor	*Setup Time*	*Baseline Times*	*With DVF Time*	*DVF Overhead %*
BMP Reader	3.41 ms	76.2 µs	76.5 µs	0.3
DCT	3.43 ms	21 µs	24 µs	14.3
Huffman Encoder	3.54 ms	21 µs	25 µs	19.0
JPEG Prep	3.65 ms	55 µs	61 µs	10.9
Quantizer	3.54 ms	16 µs	21 µs	31.3

5.7 Summary

In this chapter, we have extended the instrumentation technique from chapter 4 to develop a framework, called *dataflow validation framework* (*DVF*), for validating dataflow properties during design and implementation of signal processing systems. We have demonstrated the utility of DVF using case studies involving automatic speech

Table 5.10: DVF performance results for the actors in the JPEG encoder on the laptop platform.

Actor	*Setup Time*	*Baseline Time*	*With DVF Time*	*DVF Overhead %*
BMP Reader	232 µs	2.26 ms	2.29 ms	1.2
DCT	204 µs	1.02 µs	1.03 µs	1.0
Huffman Encoder	313 µs	1.24 µs	1.47 µs	18.5
JPEG Prep	196 µs	6.86 µs	7.39 µs	7.7
Quantizer	233 µs	1.05 µs	1.16 µs	10.5

recognition, JPEG encoding, and acoustic tracking applications. Using DVF, we were able to identify and quickly diagnose and repair several violations of dataflow properties. On the case studies that we experimented with, our results show that DVF adds less than 15% execution time overhead to the existing application, making it a suitable approach both for design-time testing and for run-time fault detection. Useful directions for future work include extending the proposed methods in the context of run-time fault detection to help drive efficient and reliable system reconfiguration for fault recovery.

Chapter 6: Conclusion and Future Work

The insatiable desire for newer and better technology has resulted in frequent upgrade cycles during which key system components are updated with the latest trends. In order to meet shorter time-to-market requirements, developers are increasingly relying on an iterative design cycle. The adoption of iterative designs and the complexity of modern embedded systems have resulted in the use of various computer-aided design (CAD) tools to facilitate analysis, debugging, and profiling. In this dissertation, we presented new profile- and instrumentation-based techniques to facilitate design and implementation of embedded systems for signal processing. First, we demonstrated the use of a new profile-based technique to improve the performance of translation-lookaside-buffers used in virtual memory systems. Next, we developed several instrumentation-based techniques to facilitate design and maintenance of signal processing systems that are developed using dataflow-based design (DBD) methodologies. In this chapter, we summarize our contributions presented in this dissertation and provide useful directions for future research.

6.1 TLB Interference Reduction in Multi-tasked Systems

Modern signal processing systems run concurrent applications that share the underlying hardware leading to inter-task interference. Such interference results in not only deteriorated performance, but more importantly for some applications, highly sub-optimal worst-case execution time (WCET) estimates due to the unpredictability of interference. In chapter 3, we presented a Context-aware TLB Preloading (CTP) methodology that can alleviate D-TLB interference in multi-tasked workloads by preloading at context-switch time extended live set (ELS) TLB entries that will be used in the near future.

CTP works through a synergistic cooperation between (1) the compiler, for an application-specific analysis of the task's context, and (2) the OS, for a *run-time introspection of the context* and an efficient identification of the TLB ELS. The CTP methodology includes the following steps.

1. Hotspots are identified, either by manually profiling an application, or using a profile guided optimization compiler (PGO) technique.

2. The compiler generates an ELS preload function (EPF) for each hotspot.

3. Executing applications register active EPFs with the OS.

4. At context-switch time, the OS calls registered EPFs to preload the associated TLBs.

The experimental results, using popular signal processing kernels as synthetic benchmarks, show up to 48% reduction in overall TLB miss rates, which results in more efficient execution, and more accurate estimation of the WCET.

We have demonstrated the benefits of a compiler-OS synergistic cooperation for the problem of reducing TLB interference on a uni-processor. A natural extension of the proposed technique would be to apply it to multiprocessor platforms. TLBs are typically processor-specific; thus, when a task gets scheduled on a new processor core, none of its page-table entries (PTEs) would be present in the TLB. Preloading the TLB with the ELS in such scenarios should significantly improve performance.

Another possible direction for future work includes adapting the compiler-OS synergistic cooperation to the problem of parallel scheduling on heterogeneous embedded systems. A compiler can identify parallelization opportunities within an application and encode them into the program. The operating system can then make run-time decisions, either at launch time or at context-switch time, regarding the parallelization level that can be supported, and taking into consideration the current workload and available resources.

6.2 Instrumentation-driven Dataflow Analysis

Dataflow modeling offers a myriad of tools to improve optimization and analysis of signal processing applications, and is often used by designers to help design, implement, and maintain systems on chip for signal processing. However, maintaining and upgrading legacy systems that were not originally designed using dataflow methods can be challenging.

In chapter 4, we presented a method to facilitate the process of converting legacy systems to dataflow semantics. To achieve this, 1) we developed a generic method for instrumenting dataflow graphs; 2) we used this instrumentation method to automatically detect instances of well-understood dataflow models from core functions being converted; and 3) we presented an iterative actor partitioning process that enables partitioning of complex actors into simpler sub-functions that are more prone to automated analysis techniques.

In chapter 5, we addressed the problem of ensuring consistency between (1) dataflow properties that are declared or otherwise assumed as part of dataflow-based application models, and (2) the dataflow behavior that is exhibited by implementations that are derived from the models. This was achieved by extending the instrumentation technique from chapter 4 to create a novel dataflow validation framework (DVF), which enables identifying disparities between an application's formal dataflow representation and its implementation. We demonstrated the utility of DVF through design and implementation case studies involving several signal processing applications.

Useful directions for future work include enhancing our developed dataflow instrumentation framework to be able to detect and act upon system faults. Extending DVF to detect such faults and reconfigure relevant aspects of a dataflow application is an important direction of future research. Possible reconfiguration options in this context include:

- Buffer reconfiguration: detecting unexpected token production and consumption behavior and dynamically reconfiguring allocated buffers.

- Schedule reconfiguration: detecting incorrectly identified schedule execution behavior (e.g., when dynamic behavior is encountered in a context where static behavior is expected), and switching to a new schedule that departs from the invalid assumptions on which the current schedule is based.

- Actor reconfiguration: applying hierarchical design [97] to actors to enable adaptive selection of actors based on the current operating environment (e.g., selecting a classifier based on observed performance or dynamically-varying, real-time constraints).

Exploring such fault tolerance and reconfiguration techniques at the dataflow level offers an interesting study of system-level trade-offs between required redundancy and system performance.

Bibliography

[1] Connie U. Smith and Murray Woodside, "Performance validation at early stages of software development," in *System Performance Evaluation: Methodologies and Applications*. 1999, CRC Press.

[2] Torsten Kempf, Kingshuk Karuri, and Lei Gao, *Software Instrumentation*, John Wiley & Sons, Inc., 2007.

[3] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[4] Karl Pettis and Robert C. Hansen, "Profile guided code positioning," *SIGPLAN Not.*, vol. 25, no. 6, pp. 16–27, June 1990.

[5] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, second edition, 2013, ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[6] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *Computers, IEEE Transactions on*, vol. 37, no. 9, pp. 1088–1098, Sep 1988.

[7] Massimo Ravasi and Marco Mattavelli, "High-level algorithmic complexity evaluation for system design," *Journal of Systems Architecture*, vol. 48, no. 1315, pp. 403 – 427, 2003.

[8] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A SW performance estimation framework for early system-level-design using fine-grained instrumentation," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, March 2006, vol. 1, pp. 6 pp.–.

[9] Reed Hastings and Bob Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, 1991, pp. 125–138.

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, June 2005.

[11] J.K. Hollingsworth, B.P. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, May 1994, pp. 841–850.

[12] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe, "Dynamic native optimization of interpreters," in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, New York, NY, USA, 2003, IVME '03, pp. 50–57, ACM.

[13] Nicholas Nethercote, *Dynamic Binary Analysis and Instrumentation*, Ph.D. thesis, Computer Laboratory, University of Cambridge, United Kingdom, Nov. 2004.

[14] Dawei Wang, Sikun Li, and Peng Zhao, "System level design in embedded stream media process system-on-chip using application profiling," in *Digital Media and its Application in Museum Heritages, Second Workshop on*, Dec 2007, pp. 353–358.

[15] H. Hubert, B. Stabernack, and K.-I. Wels, "Performance and memory profiling for embedded system design," in *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, July 2007, pp. 94–101.

[16] Jason G. Tong and Mohammed A.S. Khalid, "Profiling tools for fpga-based embedded systems: Survey and quantitative comparison," *Journal of Computers*, vol. 3, no. 6, pp. 1 – 14, 2008.

[17] Po-Hui Chen, Chung-Ta King, Yuan-Ying Chang, and Shau-Yin Tseng, "Multiprocessor system-on-chip profiling architecture: Design and implementation," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, Dec 2009, pp. 519–526.

[18] T. Miyazaka and E. A. Lee, "Code generation by using integer-controlled dataflow graph," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1997.

[19] Hyunok Oh and Soonhoi Ha, "Efficient code synthesis from extended dataflow graphs for multimedia applications," in *In Proc. 39th DAC, 2002*. 2002, pp. 275–280, IEEE Computer Society.

[20] Frederik Nebeker, *Fifty Years of Signal Processing: The IEEE Processing Society and its Technologies 1948-1998*, IEEE History Center, 1998.

[21] Agner Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," Tech. Rep., Technical University of Denmark, 2014.

[22] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software synthesis for DSP using Ptolemy,” *Journal of VLSI Signal Processing*, vol. 9, no. 1, January 1995.

[23] J. Eker and J. W. Janneck, “CAL language report, language version 1.0 — document edition 1,” Tech. Rep. UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003.

[24] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, “DIF: An interchange format for dataflow-based design tools,” in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004, pp. 423–432.

[25] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*, Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, September 1993.

[26] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, “A lightweight dataflow approach for design and implementation of SDR systems,” in *Proceedings of the Wireless Innovation Conference and Product Exposition*, Washington DC, USA, November 2010, pp. 640–645.

[27] E. A. Lee and D. G. Messerschmitt, “Synchronous dataflow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[28] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static dataflow,” *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[29] J. T. Buck, “Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems,” in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, October 1994, pp. 508–513.

[30] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, “Functional DIF for rapid prototyping,” in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[31] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki, “The DSPCAD integrative command line environment: Introduction to DICE version 1.1,” Tech. Rep. UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[32] C. Shen, L. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. S. Bhattacharyya, “The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1,” Tech. Rep. UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[33] M. Morris Mano and Michael D. Ciletti, *Digital Design (4th Edition)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[34] Michel Diaz, *Petri Nets: Fundamental Models, Verification and Applications*, ISTE, 2010.

[35] Bran Selić and Sébastien Gérard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*, Morgan Kaufmann, Boston, 2014.

[36] Yue Ma, Huafeng Yu, Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin, Loic Besnard, and Maurice Heitz, "Toward polychronous analysis and validation for timed software architectures in AADL," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1173–1178.

[37] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, Feb. 2009.

[38] T.A. Henzinger, Xiaojun Liu, S. Qadeer, and S.K. Rajamani, "Formal specification and verification of a dataflow processor array," in *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, 1999, pp. 494–499.

[39] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG reconfigurable video coding framework," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 251–263, May 2011.

[40] K. Jerbi, M. Wipliez, M. Raulet, O. Deforges, M. Babel, and M. Abid, "Fast hardware implementation of an hadamard transform using rvc-cal dataflow programming," in *Proceedings of the 2010 5th International Conference on Embedded and Multimedia Computing (EMC 2010)*, Piscataway, NJ, USA, 2010.

[41] S. S. Bhattacharyya, G. Brebner, J. Eker, J. W. Janneck, M. Mattavelli, C. von Platen, and M. Raulet, "OpenDF — a dataflow toolset for reconfigurable hardware and multicore systems," in *Proceedings of the Swedish Workshop on Multi-Core Computing*, Ronneby, Sweden, November 2008, pp. 43–49.

[42] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann, 2007.

[43] J.L. Hennessy, D.A. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2012.

[44] William Stallings, *Operating Systems: Internals and Design Principles*, Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[45] Arkaprava Basu, Mark D. Hill, and Michael M. Swift, "Reducing memory reference energy with opportunistic virtual caching," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 297–308, June 2012.

[46] M. Cekleov and M. Dubois, "Virtual-address caches part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5, pp. 64–71, 1997.

[47] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Transactions on Computer Systems*, vol. 6, no. 4, pp. 393–431, 1988.

[48] S. Yamada and S. Kusakabe, "Effect of context aware scheduler on TLB," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2008, pp. 1–8.

[49] Ilya Chukhman and Peter Petrov, "Context-aware TLB preloading for interference reduction in embedded multi-tasked systems," in *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, New York, NY, USA, 2010, GLSVLSI '10, pp. 401–404, ACM.

[50] O. Tickoo, H. Kannan, V. Chadha, R. Illikkal, R. Iyer, and D. Newell, "qTLB: Looking inside the look-aside buffer," *Lecture Notes in Computer Science*, vol. 4873, no. 1, pp. 107–118, January 2008.

[51] Isabelle Puaut and Damien Hardy, "Predictable paging in real-time systems: A compiler approach," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2007, pp. 169–178.

[52] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, July 2003.

[53] J. Staschulat and R. Ernst, "Worst case timing analysis of input dependent data cache behavior," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2006, pp. 227–236.

[54] Gokul B. Kandiraju and Anand Sivasubramaniam, "Characterizing the D-TLB behavior of SPEC cpu2000 benchmarks," *SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 129–139, 2002.

[55] J. H. Lee, J. S. Lee, S. Jeong, and S. Kim, "A banked-promotion TLB for high performance and low power," in *ICCD*, September 2001, pp. 118–123.

[56] M. Shalan and V. J. Mooney, "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," in *CODES*, 2002.

[57] M. Kandemir, I. Kadayif, and G. Chen, "Compiler-directed code restructuring for reducing data TLB energy," in *CODES+ISSS*, September 2004, pp. 98–103.

[58] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström, "Recency-based TLB preloading," in *International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 2000, pp. 117–127, ACM.

[59] Gokul B. Kandiraju and Anand Sivasubramaniam, "Going the distance for TLB prefetching: an application-driven study," in *International Symposium on Computer Architecture (ISCA)*, 2002, pp. 195–206.

[60] Arm Ltd., *Dhrystone and MIPs performance of ARM processors, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3885.html*.

[61] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, February 2002.

[62] I. Chukhman, W. Plishker, and S.S. Bhattacharyya, "Instrumentation-driven model detection for dataflow graphs," in *System on Chip (SoC), 2012 International Symposium on*, Oct., pp. 1–8.

[63] I. Chukhman, S. Lin, W. Plishker, C. Shen, and S. S. Bhattacharyya, "Instrumentation-driven model detection and actor partitioning for dataflow graphs," *International Journal of Embedded and Real-Time Communication Systems*, vol. 4, pp. 1–21, June 2013.

[64] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974.

[65] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal, "Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications," in *Proceedings of the International Symposium on System-on-Chip*, 2011, pp. 14–21.

[66] A. Gregerson, M. J. Schulte, and K. Compton, "High-energy physics," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Springer, 2010.

[67] W. Plishker, C. Shen, S. S. Bhattacharyya, G. Zaki, S. Kedilaya, N. Sane, K. Sudusinghe, T. Gregerson, J. Liu, and M. Schulte, "Model-based DSP implementation on FPGAs," in *Proceedings of the International Symposium on Rapid System Prototyping*, Fairfax, Virginia, June 2010.

[68] C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of general data flow actors into known models of computation," in *Proceedings of the International Conference on Formal Methods and Models for Codesign*, 2008, pp. 119–128.

[69] M. Wipliez and M. Raulet, "Classification and transformation of dynamic dataflow programs," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, Oct. 2010, pp. 303 –310.

[70] Matthieu Wipliez and Mickaël Raulet, "Classification of dataflow actors with satisfiability and abstract interpretation," *IJERTCS*, vol. 3, no. 1, pp. 49–69, 2012.

[71] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, April 1994.

[72] Jonathan Piat, Mickael Raulet, Maxime Pelcat, Pengcheng Mu, and Olivier Deforges, "An extensible framework for fast prototyping of multiprocessor dataflow applications," in *Proceedings - 2008 3rd International Design and Test Workshop, IDT 2008*, Monastir, Tunisia, 2008, pp. 215 – 220.

[73] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, January 2003.

[74] R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Taipei, Taiwan, April 2009, pp. 565–568.

[75] Shuvra S. Bhattacharyya et al., "Heterogeneous concurrent modeling and design in java, volume 1: Introduction to Ptolemy II," Tech. Rep. UCB/ERL M03/27, Electronics Research Laboratory, University of California at Berkeley, July 2003.

[76] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773–799, May 1995.

[77] M. G. Main and R. J. Lorentz, "An O(n log n) algorithm for finding all repetitions in a string," *Journal of Algorithms*, vol. 5, no. 3, pp. 422–432, September 1984.

[78] Vivek Sarkar and John Hennessy, "Partitioning parallel programs for macro-dataflow," in *Proceedings of the 1986 ACM conference on LISP and functional programming*, New York, NY, USA, 1986, LFP '86, pp. 202–211, ACM.

[79] CMS Collaboration, "CMS TriDAS project : Technical design report; 1, the trigger systems," Tech. Rep. CERN-LHCC-2000-038, CERN. Geneva. LHC Experiments Committee, 2000.

[80] Richard Geldreich, "Jpeg-compressor," `http://code.google.com/p/jpeg-compressor/`, 2012, [Online; accessed September-2012].

[81] Ilya Chukhman and Shuvra S. Bhattacharyya, "Instrumentation-driven framework for validation of dataflow applications," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, Oct 2014, pp. 1–6.

[82] Gwo Giun Lee, Chun-Fu Chen, and He-Yuan Lin, "Algorithmic complexity analysis on data transfer rate and data storage for multidimensional signal processing," in *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, Oct 2013, pp. 171–176.

[83] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[84] M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained block processing for DSP software optimization," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 163–177, February 2008.

[85] Ralf Kneuper, "Limits of formal methods," *Formal Aspects of Computing*, vol. 9, no. 4, pp. 379–394, 1997.

[86] J. B. Rainsberger, *JUnit Recipes: Practical Methods for Programmer Testing*, Manning Publications, illustrated edition edition, July 2004.

[87] M. Ravasi and M. Mattavelli, "High-abstraction level complexity analysis and memory architecture simulations of multimedia algorithms," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, no. 5, pp. 673–684, May 2005.

[88] J.W. Janneck, I.D. Miller, and D.B. Parlour, "Profiling dataflow programs," in *Multimedia and Expo, 2008 IEEE International Conference on*, June 2008, pp. 1065–1068.

[89] Luis Alejandro Cortés, Petru Eles, and Zebo Peng, "Modeling and formal verification of embedded systems based on a petri net representation," *J. Syst. Archit.*, vol. 49, no. 12-15, pp. 571–598, Dec. 2003.

[90] Paul Le Guernic, Jean pierre Talpin, and Jean christophe Le Lann, "Polychrony for system design," *Journal for Circuits, Systems and Computers*, vol. 12, pp. 261–304, 2002.

[91] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gerard Berry, *Compiling Esterel*, Springer Publishing Company, Incorporated, 1st edition, 2007.

[92] Rance Cleaveland, Scott A. Smolka, and Steven T. Sims, "An instrumentation-based approach to controller model validation," in *Lecture Notes in Computer Science*, San Diego, CA, USA, 2008, vol. 4922 LNCS, pp. 84 – 97.

[93] A. Ray, I. Morschhaeuser, C. Ackermann, R. Cleaveland, C. Shelton, and C. Martin, "Validating automotive control software using instrumentation-based verification," in *Proceedings of 24th IEEE/ACM Conference on Automated Software Engineering*, 2009, pp. 15–25.

[94] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[95] S. Phadke, R. Limaye, S. Verma, and K. Subramanian, "On design and implementation of an embedded automatic speech recognition system," in *Proceedings of the International Conference on VLSI Design*, 2004, pp. 27–132.

[96] T. Damarla, A. Mehmood, and J. Sabatier, "Detection of people and animals using non-imaging sensors," in *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on*, July 2011, pp. 1–8.

[97] Lai-Huei Wang, *HierarchicaL Mapping Techniques For Signal Processing Systems On Parallel Platforms*, Ph.D. thesis, University of Maryland, College Park, 2014.