UNIVERSITY OF CALIFORNIA
RIVERSIDE

MPSoC Simulation and Implementation of KPN Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Chun Shing Cheung

August 2009

The Dissertation of Chun Shing Cheung is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Thank you Prof. Harry Hsieh and Dr. Felice Balarin for their valuable opinions over the years.

To my parents and my wife.

ABSTRACT OF THE DISSERTATION

MPSoC Simulation and
Implementation of KPN Applications

by

Chun Shing Cheung

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2009
Professor Harry Hsieh, Chairperson

Design of *Multiprocessor System-on-a-Chips* (MPSoC) currently suffers from poor tool support. MPSoC is considered to be the next general design platform for embedded system designs. As complex designs such as multimedia and gaming processing become more common in handheld devices and traditional ASIC solutions are too slow and too expensive, MPSoC allows a fast software solution by running multiple low-cost, low-speed, low-power embedded processors in parallel and combining their processing power to solve more complex computation problems. However, current design methodologies for MPSoC generally restrict the specification of the software for the convenience that it can be analyzed statically. Such restrictions prevent MPSoC designs to reach their full potentials.

In this thesis, I propose an MPSoC design methodology that does not impose unnecessary restrictions on the software. Specifically, *Kahn Process Network* (KPN) is used to model the applications such that each process in the KPN process networks can be expressed by the full power of high-level programming languages. Unfortunately, allowing the full power of high-level programming languages prevents the software to be analyzed statically. Therefore, similar to optimizing software for single-processor systems, a profile-based methodology is proposed to explore the vast design space of MPSoC for applications written in KPN.

There are two main ingredients in the methodology. **1.** The MPSoC simulation must be made both fast and accurate. The speed of the simulation must allow designers to modify and experiment different design options in the limited design time allocated for system-level design exploration. At the same time the simulation must be accurate enough for the exploration results to be meaningful. A new MPSoC simulation framework that simulates in the speed close to behavioral simulation and generates performance results with less than 5% error is shown. **2.** An analysis from the simulation must provide accurate MPSoC-specific profiling information about the implementation for guiding the designers to make design decisions. Execution characteristics of MPSoC make such profiling information very different from single-processor systems. A new profiling technique specifically to determine performance-critical information for MPSoC is described.

Three optimization techniques at various implementation levels that use the proposed methodology are shown and applied to an MPEG-2 Decoder design. The experiments show that the optimization techniques using the methodology can efficiently optimize the implementations in term of performance, power and area. The results show that the methodology allows designers to explore the MPSoC design space more efficiently with the accurate MPSoC profiling information.

Contents

List of Figures

List of Tables

Chapter 1

Multiprocessor System-on-Chip

System-on-a-Chip (SoC) is an *Application-Specific Integrated Circuit* (ASIC) design that is generally made for only one or a few well-specified functions by combining standard components such as embedded processors, interconnect buses, on-chip memories and other digital signal processing components. The design and the architecture of SoC can be optimized for the particular functions. This is very different from a general desktop computer, which is designed for general-purpose processing and is made to run any programs. Designers of SoC can make application-specific decisions in both hardware and software such that the design can be produced at lower cost and with higher performance.

Over the last decade, *Intelligential Property* (IP) licensing has become a common practice in SoC designs. As capacity in a circuit scales up exponentially based on the prediction of Moore's law, IP modules are commonly used to accelerate design processes. IP modules are reliable low-cost synthesizable or already synthesized modules that are available to be licensed commercially and used in application-specific designs. In additional to common peripherals such as FIFO, GPIO, UART and buses, embedded processors and microcontrollers are available to be placed closely coupled with custom logics and manufactured on a single chip. In most traditional single-processor designs, SoC consists of one embedded processor / microcontroller, custom logics, and optionally a *Digital Signal Processing* (DSP) processor.

With the increasing in size and density of SoC, traditional design methodology for creating custom application-specific logics has been unable to keep up the pace. *Multiprocessor System-on-a-Chip* (MPSoC) [76] is a promising solution for future SoC. MPSoC refers to designs with multiple processing elements, such as embedded processors, microcontrollers, and *Application-Specific Instruction-set Processors* (ASIP), that are connected together over a complex interconnect network of buses. An abstract figure of MPSoC is shown in Figure 1.1. With the current capacity of SoC and the sizes of embedded processors, designers now have the ability to put tens and hundreds of embedded processors into a single chip. This trend is expected to continue and thousands of embedded processors can be easily fitted into a single chip in the near future. With this new design platform, design methodologies, EDA tools and optimization algorithms have to be created and revised.

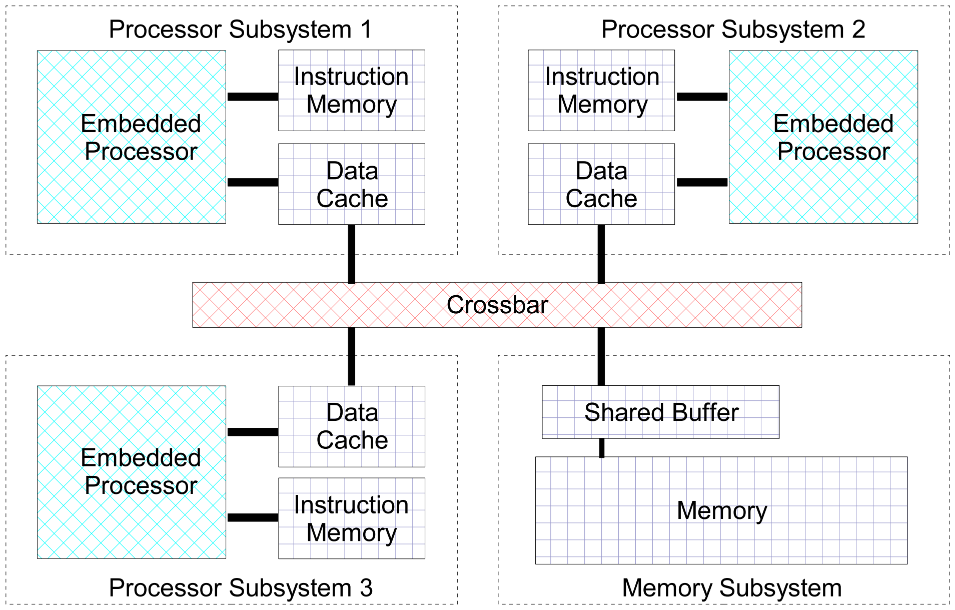


Figure 1.1: A typical MPSoC architecture. It consists of multiple *Processor Subsystems*, where each has an embedded processor / microcontroller / ASIP and private memories for instructions and data. The embedded processors and the memories can be different (heterogeneous) in the processor subsystems. The subsystems are connected with an interconnect network of buses, such as a crossbar.

The benefits of having multiple embedded processors on SoC are more apparent with the current design requirements of the SoC market. Handheld and portable devices are getting more complex that highly sophisticated multimedia and gaming processing has to be able to run on SoC. At the same time, developments of such designs have to be fast and flexible to handle the tight time-to-market requirements, short shelf lives, and ever-changing design functionalities. Here is a list of reasons that make MPSoC preferable over traditional single-processor designs:

- Power consumption, thermal output and reliability have become critical issues in designing high-performance systems. As the capacity of the chip goes up, design of more complex and faster processors is limited by the increase in power consumption. To release the heat generated by the increasing power consumption, heat dissipation has also become a major issue in high performance designs, otherwise the designs would be unreliable. Traditional single-processor designs can no longer provide reasonable power consumption with further increases in processor performance. Multiprocessor designs allow linear increase in power consumption over performance as the total computational power increases linearly with the number of processors. Therefore, MPSoC designs scale when the capacity of the chip increases.

- For high-performance designs, memory accesses become the bottleneck in performance. Increasing the frequency of the processors increases the *memory wall* – the speed difference between the processors and the memories. Improvement in speed of the memories has been slower than that of the processors, hence the performance is limited when memory accesses are involved. MPSoC avoids such memory wall issues by allowing the processors to remain running in slower frequencies while increase the performance by increasing the number of processors, hence maintain the same speed difference between the processors and the memories.

- Software designs allow significantly higher design flexibility over hardware designs. Traditionally, SoC uses custom hardware for its small size, high performance, and low power computing. However, with the ever-increasing design complexities and the shrinking time-to-market requirements, traditional single-processor designs with custom hardware are becoming too slow and difficult for both design and testing. MPSoC provides significant improvements on designers' productivity because software developments are a lot faster and more flexible than traditional custom hardware designs. At the same time, software designs have the ability to be reprogrammed after the chip has been manufactured, providing programmability that does not exist in traditional custom hardware designs.

- Embedded processor IP is easy to license, reliable and well-supported. It allows designers to use multiple embedded processors on their designs without worrying about the internal designs of the processors. As shown in Table 1.1, a number of embedded processors (also known

Company	**Soft Core IP**	**Remark**
ARM	ARMv6 – ARMv11	Most commonly used
Tensilica	Xtensa	Configurable & Extensible
MIPS	MIPS32, MIPS64	
ARC	ARC600, ARC700	Configurable & Extensible
Xilinx	Microblaze, Picoblaze	Configurable & for Xilinx FPGA
Altera	Nios	Extensible & for Altera FPGA
Open Source	OpenSparc, OpenRisc	Free (Open Source)

Table 1.1: Some available embedded processor IP. Most of them target ASIC designs. Some of them are configurable – designers can choose instruction sets available in the processors. Some of them are extensible – designers can add specific instructions into the processors. Some of them are specifically mapped to FPGA.

as *soft cores*) have been used successfully in commercial SoC, most noticeably are those from *ARM* [7], *Tensilica* [154], *MIPS* [102], and *ARC* [6]. These soft cores can be easily instantiated and synthesized into SoC designs. They also provide complete documentation and comprehensive supports on compiler toolsets and simulators. Alteratively, *OpenSparc* [111] and *OpenRisc* [110] provide open-source solutions for synthesizable processors. In the FPGA domain, *Micro/Picoblaze* [166], *Nios* [4], and ARM processors have been natively mapped into the FPGA from *Xilinx*, *Altera*, and *Actel* [2] respectively.

- Placing multiple processors in a single chip has never been easier. With the currently capacity of SoC, which is still expected to be doubled every 18 months, it is easy to fit tens and hundreds of embedded processors into a single chip. On the other hand, traditional single-processor design methodology has not been able to keep up with the capacity of SoC. Therefore, massive parallelism in software is becoming a preferable solution for high-performance computing.

As a result, MPSoC provides a lower cost, lower power and higher performance solution than traditional single-processor systems by allowing multiple software to run concurrently on multiple embedded processors. Embedded processors, such as those licensed by ARM, Tensilica, MIPS and ARC, are optimized for their performance-per-cost ratio, in which the cost includes power and area. Such embedded processors are typically simple pipelined without complex dynamic logics to reduce their costs while providing efficient computing.

1.1 History

In recent years, microprocessor architecture has reach its limit in performance because of its increasing power consumption, worsen thermal output, and decreasing performance-per-cost ratio with more advanced architecture. Multiprocessor designs have become the focus in both computer architecture and embedded system communities. In this section, multiprocessor designs in the last decade are discussed.

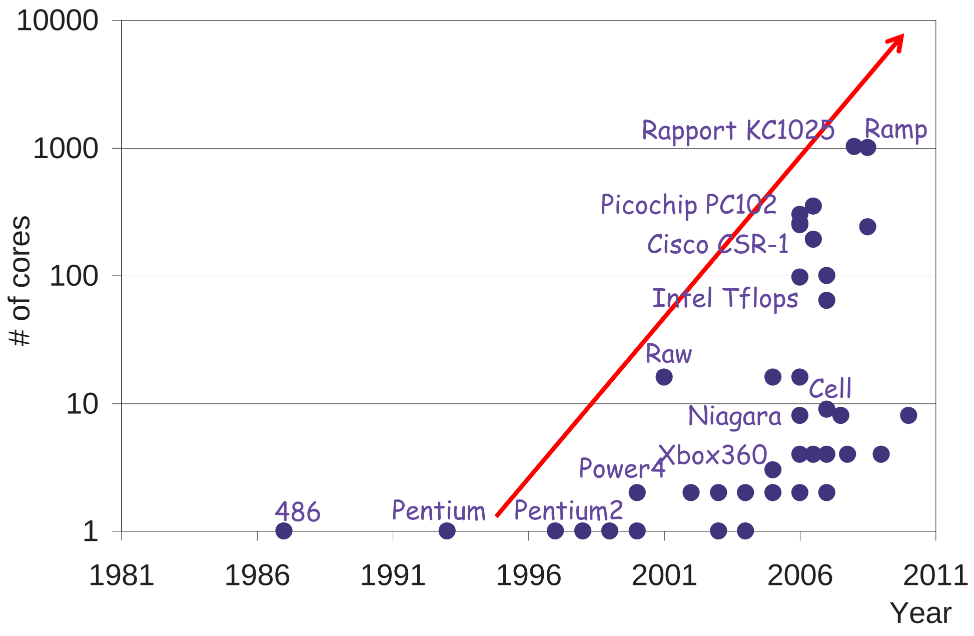


Figure 1.2: Number of processors on designs in the last 20+ years. Some major designs are labeled. The red arrow shows the trend of multiprocessor designs. Data are taken from [60] and [40].

A number of MPSoC designs have been successfully launched over the last decade. Figure 1.2 shows some of the most noticeable products over the years and the number of processors that are used on those products. Multiprocessor designs start to get momentum around late 90's. Since then, the number of processors in designs continues to climb exponentially. As of today, designs are capable to have more than one thousand processors.

SoC designs could be classified as MPSoC for some time because a lot of them consist of a small embedded processor and a *Digital Signal Processing* (DSP) core. The embedded processor is used as the controller because the program can be written in C language and compiled into binary. The DSP core is used for high-performance computing because of its special DSP / *Very Long Instruction Width* (VLIW) / *Single Instruction Multiple Data* (SIMD) operations. The DSP core is not used as the controller because normally no compiler is available and the program has to be written in Assembly language instead of C language. Therefore in this section, I focus on designs with more than two processors. Designs, such as the *Nintendo DS* which has only two processors – one *ARM7TDMI* for touchscreen controls and one *ARM946E-S* for gaming controls, are not discussed.

1.1.1 Xenon

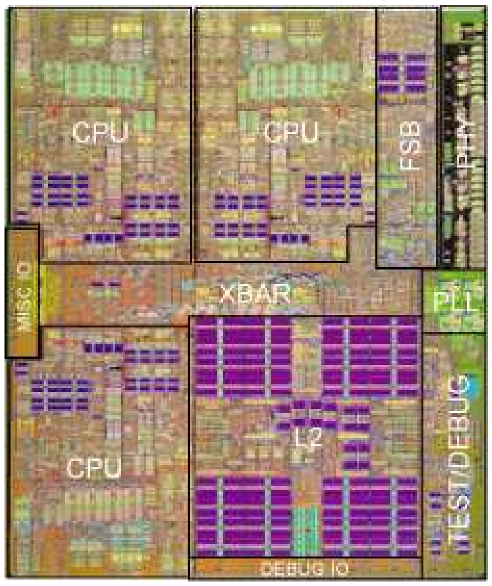


Figure 1.3: Xenon chip layout taken from [23].

Microsoft's Xbox360 gaming console uses *Xenon* [23] (Figure 1.3), which consists of three *PowerPC*-based processing cores, for main processing. Each core is based on *IBM*'s 64-bit PowerPC *Instruction Set Architecture* (ISA) with *VMX-128* SIMD vector instructions. Each core runs at 3.2GHz and can symmetrically process two threads. Each one has a 32 KB L1 instruction cache, a 32 KB of L1 data cache, as well as a shared 1MB L2 cache (shared among all three cores). *Modified, Exclusive, Shared, Invalid* (MESI) protocol is used for memory coherency in the L1 caches. Graphic processing in Xbox360 is separated and is handled with the *ATI Xenos Graphics Processing Unit* (GPU).

1.1.2 Cell Processor

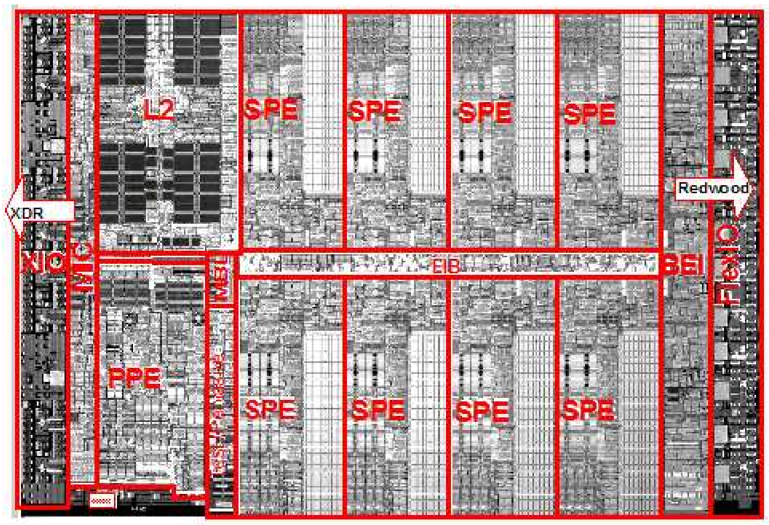


Figure 1.4: Cell processor chip layout taken from [163].

Cell processor [163] (Figure 1.4) is a join development among *Sony*, *IBM*, and *Toshiba*, and is used most noticeably in *Sony's Playstation 3* gaming console. The design consists of one *Power Processor Element* (PPE) and eight *Synergistic Processing Elements* (SPE). The cores are connected with an internal high speed bus called *Element Interconnect Bus* (EIB). The PPE is based on the IBM's 64-bit PowerPC instruction set architecture with VMX-128 SIMD vector instructions and is able to symmetrically process two threads. The SPE is a RISC processor with 128-bit SIMD instructions. The PPE acts as the controller for the eight SPE. The PPE has a 32 KB L1 instruction cache, a 32 KB L1 data cache, and a 512 KB L2 cache. Each SPE has its own 256 KB on-chip memory for both instructions and data (which is known as a *scratchpad memory*).

1.1.3 Cisco CSR-1

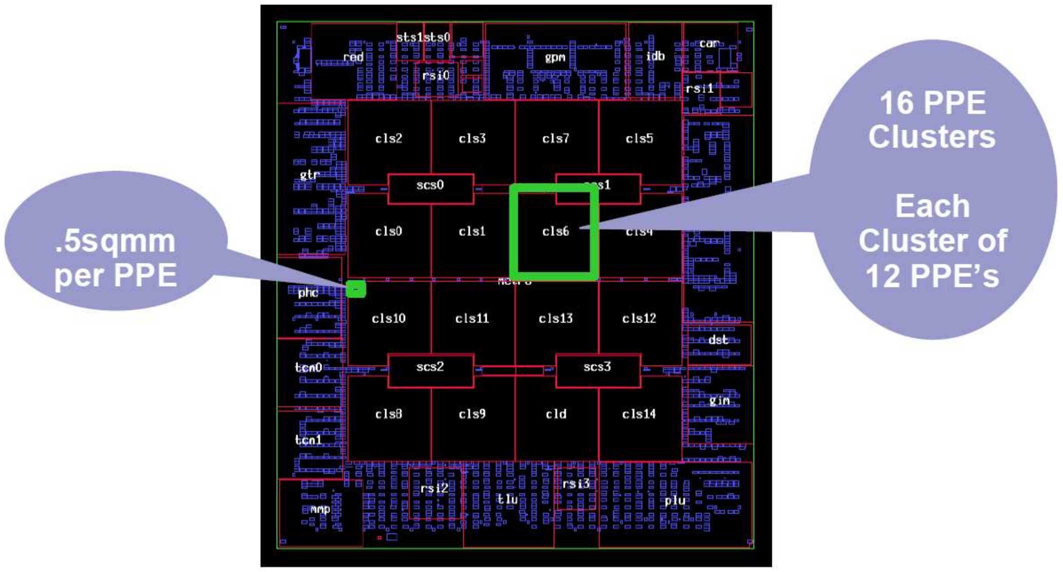


Figure 1.5: Cisco CSR-1 chip layout taken from [50].

Cisco CRS-1 Carrier Routing System (CRS) [50] is the largest production network router in 2008. Each router has multiple slots for expandable routing capacity. A routing chip (Figure 1.5) can be found in a blaze that can be inserted into the slots. Each routing chip has 194 Tensilica's Xtensa processors to parallel process network packets. Specifically among the 194 processors, one is specialized for debugging and one is used as the controller. The remaining 192 processors are separated into 16 clusters of 12 processors (PPE). Each of these clusters is identical in term of the hardware and software and processes one packet at a time. Inside a cluster, the 12 Xtensa processors are customized with *Tensilica's Instruction Extension* (TIE) instructions, which add application-specific instructions to the datapaths of the processors to efficient manipulate different parts of a packet. Under this architecture, each packet is being processed in parallel among the 12 processors within a cluster and 16 packets are processed in parallel in different clusters. Software in the processors can be reprogrammed for additional features and functionalities. With the massive software parallelism, the design is able to process 96 Gigabit of data per second even though it is only running at 250MHz and 35W.

1.1.4 Picochip PC102

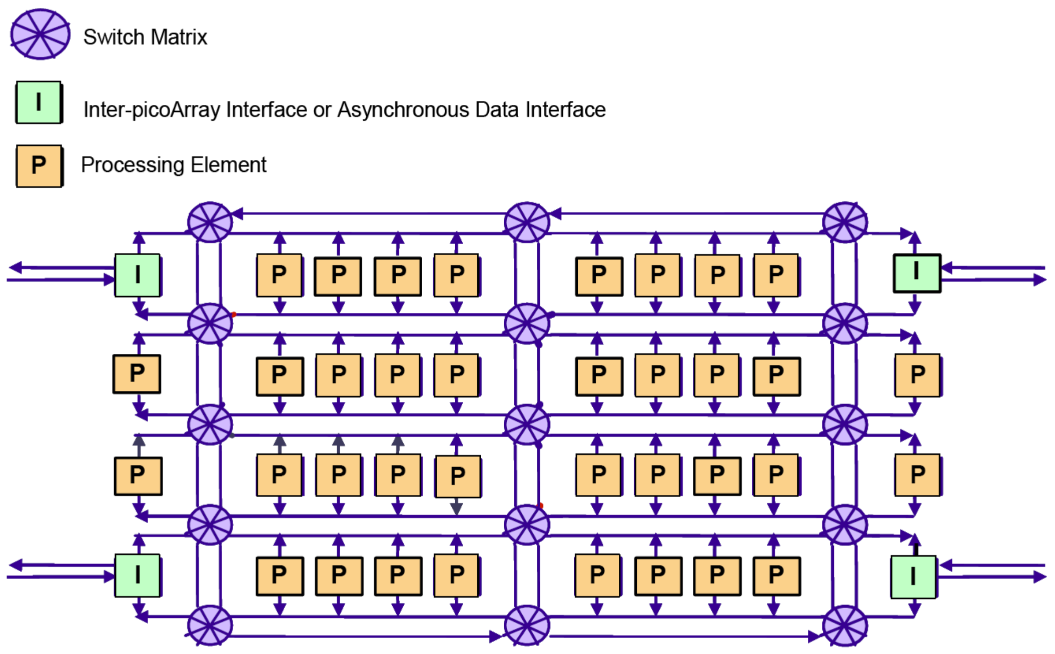


Figure 1.6: Picochip PC102 chip architecture taken from [12].

PicoChip PC102 [12] programmable chip (Figure 1.6) contains 308 16-bit processors. The usage of PC102 is similar to DSP processors but makes use of multiple processors instead of complex DSP instructions. Programming of PC102 uses the *Multiple-Instruction Multiple-Data* (MIMD) programming model, where each processor has its own instructions and processes its own data. Each processor has its own memory (scratchpad memory). Not all processors are the same. There are three types of processors that are different in term of available instructions and memory spaces. 240 of them are processors with a *Multiply-Accumulate* (MAC) instruction and a 768 byte on-chip memory, which make them useful for more computation-intensive instructions. 64 of them are processors with no MAC instruction and a larger 8704 byte on-chip memory, which make them useful for more memory-intensive instructions. And there are four processors that have a much larger 64KB memory, which are mainly used as controllers. The processors are connected with a network of buses and configurable switch matrices. The design has been shown to provide more than 10 times better performance than a standard DSP processor even though it is only running at 160MHz and 5W.

1.1.5 Ambric AM2045

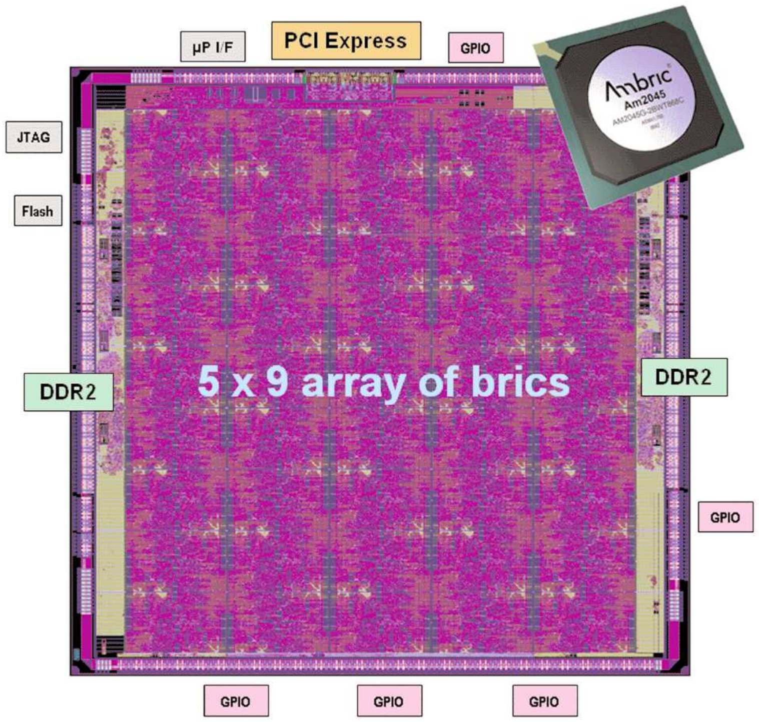


Figure 1.7: Ambric AM2045 chip layout taken from [65].

Ambric AM2045 [65] (Figure 1.7) has 336 32-bit RISC processors that target video processing. The processors are organized into clusters (*Brics*). Each bric consists of eight processors, a 8KB shared on-chip memory, and can run in an independent clock frequency. The eight processors in a bric are RISC processors or RISC processors with DSP instruction extensions. Each of them has a private memory with the size ranging from 256 bytes to 1KB. Processors in one bric can access the shared on-chip memories only in the adjacent brics. The application is written as objects in Java language and the development tools convert the application into parallel machine binaries. The object-oriented approach allows the objects to be distributed among the processors.

1.1.6 Kilocore

Industries and researches are currently capable of designing systems with more than 1000 processors (kilocore). *Rapport KC1025* uses one IBM's PowerPC processor and 1024 small 8-bit processors running at 125MHz. The small processors are organized in a 32x32 mesh configuration.

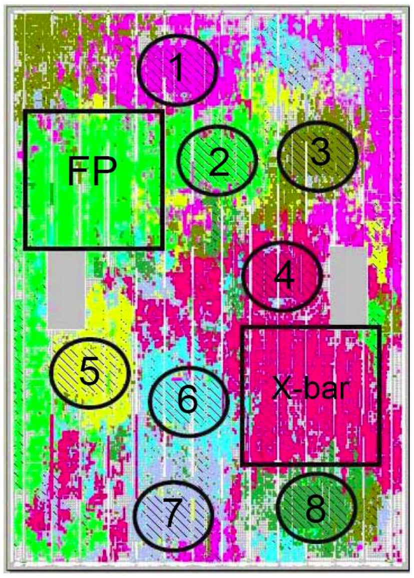


Figure 1.8: FPGA layout on one VirtexII Pro taken from [25] that has eight Microblazes and a floating point unit. The 1008 processor version of RAMP Blue uses a slightly different layout that has 12 Microblazes on one VirtexII Pro with no floating point unit.

The *RAMP* project [25] (Figure 1.8) is a join project of several major universities and companies in the United States, including University of California Berkeley, Stanford University, University of Washington, Massachusetts Institute of Technology, Carnegie Mellon University and Intel. The goal of the project is to build a configurable system with more than 1000 embedded processors for researches and application developments. In 2008, a *RAMP Blue* system with 1008 processors was presented. It is built with 21 BEE2 processing boards [29], each with five *Xilinx's VirtexII Pro* FPGA (where one of them is solely used for controlling) that are connected with high speed fibers. Each of the cores is a Xilinx's Microblaze that is connected with IBM's buses.

1.1.7 Commercial IPs

To facilitate more power and area efficient designs, commercial IP companies have started to license IP with multiple processors. For instance, ARM is licensing its *Cortex-A9 MPCore* [113], which has four ARMv7 processors and snoop cache coherency. Tensilica is licensing its *388VDO Video Engine* [114], which is a video processing IP for codec encoding and decoding. It is composed of two highly configured Xtensa processors.

1.2 Design Space

As shown in the MPSoC designs in the previous section, the MPSoC architectures and application specification can be very different. Both hardware and software design choices are very important in MPSoC. With different combinations of hardware and software design choices, the implementations will have very different performance and cost (in term of area and power) characteristics. It is designers' responsibilities to explore this design space to optimize both hardware and software in order to maximize performance and minimize cost in the implementation.

1.2.1 Hardware

The hardware part of MPSoC determines the performance and parallelism allowed in the design. As shown in the designs in the previous section, there are a lot of options in the hardware. Those designs are very different in term of the number of processors, the type and the Instruction Set Architecture (ISA) of each processor, the cache and memory subsystems, and the way the processors talk to each others.

Processor

The number of processors available and the types of the processors are critical for the cost and the performance of an MPSoC implementation. In term of performance, the number of processors determines the parallelism that is possible in the software, while the types of the processors determine the execution time of the sequential programs in the software. On one hand, a good implementation should have powerful enough processors because the parallelism is limited by the algorithm and the specification of the software. On the other hand, a good implementation should have more smaller processors because smaller and slower processors generally provide a better performance-per-cost ratio. Therefore, an MPSoC implementation must provide a good balance between the number of processors and the types of the processors.

The tradeoff between power consumption and performance can also be achieved by running the processors in different frequencies. As the power consumption of a processor increases super-linearly with the frequency of the processor [75], running a processor in a slower frequency improves its performance-per-power consumption ratio. Therefore, *Voltage Islands* (VI) [87, 91, 108] and *Dynamic Voltage Scaling* (DVS) [59, 119, 165] provide design opportunities for an implementation to have more efficient performance and power characteristics.

Instruction Set Architecture

The ISA determines the efficiency of a sequential program when executing on the processor. For application-specific implementations, DSP instructions speed up arithmetic operations that are common in signal processing and multimedia applications. Such DSP instructions include complex operations (such as *Multiply-Accumulate* (MAC)), *Very Long Instruction Word* (VLIW) and *Single Instruction Multiple Data* (SIMD) operations. These instructions speed up the applications by combining multiple operations into one instruction that can be executed more efficiently. On the other hand, having instructions that are not used in the applications is not cost efficient. Datapaths that implement the unused instructions take up space on the chip, which increase the area and make design synthesis more difficult. The extra datapaths also draw extra static (and possibly dynamic) power, which increase the power consumption.

For a cost-efficient MPSoC implementation, the processors should not be overly complex and have just enough instructions for efficient computing. The processors should be simple and heterogeneous such that simple jobs are executed on smaller and simpler processors and more complicated jobs run on the processors that have specialized DSP instructions. Tensilica [154] and ARC [6] provide customizable and extensible embedded processors that allow processors to be configured depending on the programs running on them.

Memory subsystem & Interconnect

With multiple processors run in parallel, memory subsystems and interconnect networks are very important in order to provide data to the processors in a timely fashion. With more processors, traditional memory hierarchy is not able to keep up the total memory bandwidth required for all the processors. Therefore, memory subsystems in MPSoC use a combination of local on-chip memories, caches, and off-chip memories to maximize the memory bandwidths for the processors.

1.2.2 Software

The software part of MPSoC specifies how the design utilizes the performance and parallelism available in the hardware. The software design must match the hardware design such that no hardware is unused or wasted. The software have to be written in a distributed model, which allows the instructions to run on the multiple processors.

Parallelization

Exploring the parallelism in the software is crucial for the application to be executed in parallel in multiple processors. The application has to be separated into multiple instruction-data segments, where each processor can run a different segment. The degree of parallelism in the application is very important. A finer (hence, more massive) parallelism provides the ability to execute more segments concurrently and each segment is likely to have less computation requirements and need a smaller processor. But at the same time, adding parallelism to the application could increase the overall complexity and require the use of a less efficient algorithm. And too much parallelism not supported by the hardware would simply increase the communication overheads between the segments without improving the performance.

Each instruction-data segment may have different computation requirements. Some of the segments are more important and require faster computation, and some of them can benefit from DSP instructions if they are available. Therefore, it is crucial for the designers to identify parts of the programs that are more important and speed them up by using a more powerful processor or special DSP instructions.

Software Optimization

Software optimizations are very important to maximize the computational power of the processors. With software programs running on different processors in parallel, software optimizations are being introduced with a whole new meaning. Optimizations on a single software program are no longer the primary interest. Software optimizations for MPSoC have to focus on reducing the overall execution time and memory requirement of the whole implementation, which involves multiple programs.

In additional, a lot of embedded processors are extensible, which means customized DSP instructions can be added to the processor datapaths. Based on the software programs running on a processor, specific DSP instructions can be designed and added to the processor datapath such that the programs can benefit from the new DSP instructions. Such instruction customizations are normally done with software optimizations because the instruction customizations are instruction-dependent and software optimizations have to be aware of the instructions that are used.

Design Specification

In order for the software segments to execute as one application, the application should be specified in a distributed *Model of Computation* (MoC). The MoC determines how the distribution of the software can be written and how the instruction-data segments communicate. There are generally two main approaches – *Symmetric Multi-Processing* (SMP) and *Message-Passing*. In SMP, the segments have the same view on the memory and they communicate by reading and writing to the same memory addresses. In message-passing, the segments communicate by sending data from one segment to another.

The MoC that the software uses must be supported by the hardware. Specifically, SMP requires processors to have the same view on the memory, which implies a shared memory architecture with coherent caches – which is not scalable over tens of processors. Therefore, SMP is only useful for small number of processors. To make use of the small private on-chip memories normally available for each processor, the MoC should be *Multiple-Instruction Multiple-Data* (MIMD), where each processor has its own instructions that processes its own data and use message-passing communication mechanism.

1.3 Thesis Overview

This thesis focuses on design space exploration of application-specific MPSoC designs. With a lot of design options in the design space – in both hardware and software, there is currently no structural way to explore the design space to achieve a cost-efficient implementation. MPSoC is very difficult from traditional single-processor designs that a lot of design steps have to be reconsidered in order for the MPSoC designs to reach their full potentials.

The main benefit of implementing a design in MPSoC is that designers can write software, instead of some other specification models which restrict the uses of regular software statements such as while loops and pointers. The efficiency, programmability and flexibility of MPSoC come from the fact that designers can design the applications in the full power of high-level programming languages. Writing software that can be statically analyzed and optimized, although being heavily used in researches, contradicts the reasons of using MPSoC. It is especially true for designs with soft deadlines, which include multimedia and gaming processing that are thriving in MPSoC designs and are the focuses of this thesis. Since an MPSoC design inherits the potential complexity of software designs, using profile-based information is a reasonable approach in optimizing the design.

1.3.1 Outline

In this thesis, I propose a new MPSoC design optimization methodology without imposing unnecessary restrictions on the software specification. Specifically, *Kahn Process Network* (KPN) [79] is used as the distributed Model of Computation for MPSoC. KPN provides a good balance between the expressiveness of the programming languages and its analyzable properties. Its deterministic nature avoids race conditions which are the major pitfalls in MPSoC designs. An application written in KPN also allows it to be easily implemented in MPSoC.

The proposed methodology is software-driven. The software is initially specified in KPN, and an MPSoC hardware implementation is created to match the parallelism and the communication requirements of the software. From there, the proposed MPSoC design optimization methodology is applied to optimize the MPSoC implementation in both hardware and software.

The MPSoC design optimization methodology is a profile-driven optimization flow that makes use of an analysis during simulation. Similar to using profiling information for software optimization in single-processor systems, the MPSoC design optimization methodology involves simulating the implementation, analyzing the design during simulation, determining the important parts of the implementation, and optimizing the implementation based on the analysis results.

A fast and accurate MPSoC simulation is an important part of the methodology. Since the methodology involves repeatedly simulating and analyzing the implementation and iteratively exploring and improving the MPSoC implementation, the MPSoC simulation has to be fast enough to shorten the design exploration time. At the same time, the simulation results have to be accurate such that the analysis and optimizations based on the results are reliable. In the first part of this thesis, I present a simulation framework that uses compile-code simulation techniques and the SystemC simulation kernel to accurately estimate and efficiently simulate an MPSoC implementation. The experimental results show that the simulation framework simulates at the speed close to behavioral simulation, which is more than 1000 times faster than using *Instruction Set Simulators* (ISS), and at the same time has less than 5% error in performance estimation.

To help designers exploring the MPSoC design space, simulation results (profiling information) that show the important parts of the implementation for optimization are provided. MPSoC is very different from single-processor systems because there are multiple embedded processors running in parallel and they interact with each others. Therefore, traditional software profiling information for single processor is not accurate and not useful in optimizing MPSoC designs. In the second part of this thesis, I show an algorithm to find the MPSoC-specific profiling information – the *MP-critical path*. Then I present three optimization techniques that use the profiling information at different design levels: instruction level, segment level and program level. Shown in the experiments on an MPEG-2 Decoder design, the optimization techniques significantly reduce the execution time and power consumption of the MPSoC implementations while limiting the area usage.

In summary, the proposed methodology is very different from a lot of other MPSoC design optimization methodologies because:

- It uses KPN as the design specification, which allows the full power of high-level programming languages such that software can be written easily and more efficiently.
- A profile-based design optimization methodology is used, which is the directly result from the fact that KPN is used to model the application.
- The simulation of MPSoC is both fast and accurate. Since the methodology allows more fine-grain optimizations, the simulation not only has to be fast but also has to be accurate enough to allow small improvements to be reliably shown in the simulation. The simulation framework in the methodology proposes techniques based on compile-code simulation to simulate MPSoC efficiently and accurately. The techniques allow compiler optimizations and memory accesses to be considered accurately and simulation to be done more efficiently.

- A new profiling technique for MPSoC is shown. MPSoC is very different from single-processor systems in term of execution characteristics. Therefore design optimizations have to be based on a different analysis specifically for MPSoC. An MPSoC-specific profiling information is proposed and an efficient algorithm to find such information is described. Such profiling information can be used to optimize MPSoC implementations at different design levels, which is shown by the optimization techniques presented to improve the implementations at instruction level, segment level and program level.

1.3.2 Organization

This thesis is organized as follows:

- Chapter 2 describes some related works on system-level design methodologies, compile-code simulation and MPSoC design optimizations.
- Chapter 3 introduces Kahn Process Network (KPN) specification and the benefits of using KPN as the distributed Model of Computation. The target MPSoC implementation of an KPN application is also specified.
- Chapter 4 proposes a profile-driven design space exploration methodology for KPN-based MPSoC.
- Chapter 5 describes the compile-code simulation framework that allows fast and accurate MPSoC simulation. Specific techniques to consider compiler optimizations, memory accesses and a modeling technique to speed up simulation are discussed.
- Chapter 6 introduces the profile analysis specifically for MPSoC designs. Several optimization techniques at various design levels based on the analysis results are presented.
- The thesis is concluded in Chapter 7.

Chapter 2

Related Work

Over the last decade, MPSoC designs and optimizations have been extensively studied in both researches and industries. In this chapter, I survey some of the major works on system-level design methodologies, simulation techniques and MPSoC design optimizations.

2.1 System-level Design Methodology

Some system-level design methodologies have been proposed over the years and are suitable for designing MPSoC systems. Most noticeably are Polis / Metropolis [9, 39, 88, 153], Simulink [66, 124], Sesame / Spade / Archer [92, 123, 171], Compaan-Laura [67, 83, 147, 148, 157, 158] and TTL [131, 156, 159]. They all focus on the design flow from the design specifications to actual implementations (system-level synthesis). Although some of these methodologies provide platforms for design space exploration and optimizations, they do not have an optimization methodology – which generates and provides useful information to the designers for optimizing their implementations. Hence design optimizations remain ad-hoc and rely on the instincts of the designers. In this thesis, the methodology focuses on design optimizations – which generates and provides useful information about an implementation for optimization purposes. And the design optimization methodology proposed in this thesis can actually be integrated to these methodologies.

2.1.1 Polis / Metropolis

Polis / Metropolis [9, 39, 88, 153] are platform-based design methodologies that can be used to design MPSoC. They aim to provide an uniform design and modeling platform for heterogeneous components across multiple levels of abstraction. In these methodologies, the designs are specified in less-restricted forms, such as *Codesign Finite State Machines* (CFSM) for Polis and *Metropolis Meta Model* (MMM) for Metropolis. They purposely separate the function modeling and the implementation aspects of a design such that different levels of simulation can be done with the same models. The main feature in these methodologies is that they allow properties and constraints to be formally specified in the models and verified with simulation / formal verification tools. Therefore, formal methods for verification and synthesis are integrated into the design methodologies.

2.1.2 Simulink

Simulink [66, 124] is a high-level synthesis methodology that uses a process network model of Matlab programs as the specification. The platform provide multiple levels of abstraction to allow designers to explore the design space at different abstraction levels. There are mainly four levels: a functional simulation level, an abstracted simulation level, a cycle-accurate level, and an implementation level. An automatic mean is provided to allow a common specification to be used in all four abstraction levels and generate scheduled C programs for implementations without additional functional manual modeling.

2.1.3 Sesame / Spade / Archer

Sesame / Spade / Archer [92,123,171] are system-level modeling and simulation methodologies for performance evaluation and exploration. They focus on modeling application behaviors and architectural constraints. The main feature of these methodologies is their hybrid simulation – which allows a mix simulation of components at different levels of abstraction for architectural exploration and simulation model refinements. Therefore, behaviors and architectures can be modeled separately. For design exploration, they transform particular simulation traces into a dataflow graph and use the graph for efficient analysis. Such trace analysis, however, can only be done at a small application and a short simulation.

2.1.4 Compaan-Laura

Compaan-Laura [67, 83, 147, 148, 157, 158] is a high-level synthesis methodology that converts a sequential program written as a Matlab program into concurrent components. Compaan itself compiles a sequential program in Matlab, especially one with nested loops and tight bounds, into a distributed specification in Kahn Process Network. The Matlab specification is used because it provides better dataflow programming than other programming languages and has no difficult-to-analyze constructs such as pointers. With the KPN specification, Laura generates VHDL hardware descriptions for the KPN processes and synthesizes them into FPGA using commercial synthesis tools.

2.1.5 TTL

Task Transaction Level (TTL) Interfacing [131, 156, 159] is an interface-centric approach for designing distributed embedded systems. It provides an unified interface (TTL interface) for both hardware and software interfacing at three different levels of abstraction – functional, architectural, and implementation levels. Hence the behavioral synthesis can be divided into smaller synthesis problems in the interface boundaries. It allows independent behavioral synthesis of the TTL components (hardware or software) because the interfacing is the same before and after the synthesis.

2.2 Compile-code Simulation

Compile-code simulation is a common technique to speed up simulation. Compile-code simulation generally means to execute the behaviors of target components natively in the host development machine, instead of simulating the detail hardware components that provide the same behaviors in the target implementation. Although this term can be used for both hardware and software components, this thesis focuses on software where compile-code simulation means to run the programs natively in the host development machine instead of simulating the way the programs execute on the target embedded processor architectures.

To allow performance estimation in compiled-code simulation, a lot of compile-code simulation techniques aim to add timing delays to otherwise behavioral simulation in order to consider the performance of the implementations. Performance estimation in compile-code simulation is possible because the timing delays are annotated into the behavioral description and are compiled along with the behaviors for simulation, although no detail hardware architecture is simulated. These timing delays annotated are executed along with the programs in the host development machine for performance estimation – such as adding all timing delay values to determine the overall execution time when all programs run on a single-processor system. Some of such estimation techniques are discussed here. These techniques either restrict the program specifications and the compiler optimizations that can apply to the programs, or focus on the retargetability and usability in the corresponding design methodologies. On the other hand, the performance estimation technique proposed in this thesis focuses on the accuracy of the software programs running on an MPSoC system. It does not restrict the program specifications, and at the same time considers the parameters that are important for performance, such as compiler optimizations and memory accesses.

2.2.1 S-Graph

Performance for software programs running on a target system has been estimated for POLIS [39, 153] in the code generated from a language called *Codesign Finite State Machines* (CFSM). The CFSM is compiled to a *Software Graph* (s-graph) to estimate timing information for the C program that would be generated for the target system. In Polis, estimation is based solely on higher-level information that is available on the CFSM. Such estimation does not provide the same level of accuracy as the estimation based on instruction-level information – which is used in the simulation framework proposed in this thesis. In addition, the specification in Polis is limited (not generic C programs) and no compiler optimization effect is considered in the estimation.

2.2.2 Lee *et. al.*

Lee *et. al.* [90] estimates the performance of a software program written in C language on a single-processor target system with a fixed architecture using compile-code simulation. Similar to the simulation framework proposed in this thesis, GCC is used as the analysis and annotation tools for the performance estimation. In Lee's technique, the timing delays of the program is annotated into the intermediate representation using GCC and compiled into a native executable for the host development machine. The program is executed on the host development machine and the timing delays are tallied to estimate the total execution time of the program when running on the target system. However, to generate a native executable for the host development machine, the machine description of the host development machine has to be used, which makes it unable to accurately consider compiler optimizations. In addition, this technique only works for a single software program running on a single-processor system with a fixed architecture. The simulation framework proposed in this thesis solves these issues by using a 2-pass approach – simulation models are generated by GCC targeting the target processors for the timing estimation, then they are compiled again for simulation. The performance simulation is modeled in SystemC. The estimated result is represented by SystemC time instead of a simple counter, which allows simulation of concurrent components. It provides a higher description level for multiprocessor architecture modeling and allows synchronization for multiple processors using SystemC simulation kernel.

Moreover, Lee's estimation only considers instruction types and requires program-specific coefficient to make the estimation accurate. Such accurate coefficient is very difficult to find and no algorithm has been proposed to find this coefficient. In the proposed simulation framework, estimation is based on low-level instruction information and makes use of the detail information available in additional to the instruction types. Such technique allows more accurate timing delay estimation without the need of a program-specific coefficient.

2.2.3 Embedded System Environment toolset

Embedded System Environment (ESE) toolset [73] provides automatic *Transaction Level Model* (TLM) generation from a C program for software and from high-level design parameters for hardware components. For retargetability, *Low Level Virtual Machine* (LLVM) compiler is used as the frontend for the C program and a general processor unit model is used for performance estimations on different target processors. These estimations will be used in the SystemC TLM model generated from the LLVM for simulation. Since the target compilers are not used, no target compiler optimization is considered in the performance estimation under this technique. Also memory accesses are not considered in the simulation.

2.2.4 Brandolese *et. al.*

Brandolese *et. al.* [21] proposed a source-level analysis technique for performance estimation. The technique uses a statistical method where a set of mathematical equations are used as the basis for the estimation. The equations make use the operations and types of the operands in the high-level programming statements in the program. However, since the model is built at the source level, no low-level detail from the target compilers is available in the mathematical model hence is not considered in the estimation.

2.2.5 Posadas *et. al.*

Posadas *et. al.* [125] proposed a technique to directly estimate software performance in SystemC without the need to generate instrumented simulation models. Software delays are integrated into the software models by overloading all the datatypes used in the programs by custom datatype classes in SystemC, hence all the operations in the programs would use the overloaded operators of the custom datatype classes. The overloaded operator functions are annotated with timing delay information to estimate the total execution time of the programs during the SystemC simulation. However, this technique only consider operations at the source level. Such operations can be changed dramatically when compiler optimizations are enabled. Hence such technique does not consider any compiler optimizations that can be applied to the programs by the target compilers.

2.2.6 Lazarescu *et. al.*

To consider compiler optimizations, Lazarescu *et. al.* [11, 89] proposed a technique to generate simulation models from the target binaries (executables). Performance annotations are estimated by analyzing the assembly instructions in the target binaries. Since the target binaries are produced from the target compilers after optimizations are applied, compiler optimizations are considered in the estimation. The estimation is annotated to the equivalent C programs that are generated from the target binaries. However, generating such C programs is not always possible. The jump table of an indirect jump instruction must be recovered in order to generate a semantic equivalence C program.

2.2.7 Schnerr *et. al.*

Schnerr *et. al.* [142] also proposed a performance estimation technique based on the target binaries. Instead of generating the semantically equivalent C programs for annotations, the performance estimation is back-annotated into the original C programs. Therefore a one-to-one matching must be found in order to back-annotate the estimation into the original C programs. Only limited compiler optimizations can be used for compilation in this technique, otherwise it would not be able to find a good one-to-one matching between the target binaries and the original C programs for estimation insertions.

2.2.8 Zivojnovic and Meyr

Zivojnovic and Meyr [172] used compiled instruction set simulation, which is a simulation technique between compile-code simulation and instruction set simulation. It is similar to compile-code simulation where the target programs are compiled into native programs. However, instead of simulating the program behaviors in the host development machine, only the information of the target binary instructions are compiled into the native programs. When executed, the native programs simulate the target processor architectures for behavior and performance information. Therefore, similar to instruction set simulation, target binary instructions are simulated one-by-one in the target processor architectures. Compiled instruction set simulation is faster comparing to instruction set simulation but is still orders of magnitude slower than compile-code simulation.

2.2.9 Other Simulation Technique

Besides compile-code simulation, other techniques have been used to simulate MPSoC systems and estimate the performance of an MPSoC implementation (or embedded software in general).

Static Analysis

Static analysis [19, 97, 112, 132] allows performance to be computed without the use of simulation. Their estimation techniques range from statistical models [132], non-linear equations [19], to classification and neural network [112]. However, multiprocessor system is in general too complex to be analyzed using static analysis. Static performance estimation techniques are based on formal analysis of the program specifications. No simulation is performed and usually the *worst-case estimation time* (WCET) [97, 127] is obtained. Static analysis is useful when software programs have hard deadlines and the performance has to be guaranteed in the worst case. However, most static analysis techniques restrict the specifications of the software programs being analyzed, are unable to recognize the true execution path, and overestimate the execution time. For general applications, there is no effective way to use abstract models or close-form formulas to accurately estimate the performance of an implementation. Due to the inherent complexity, full system formal analysis can only be done at the highest level of abstraction, which often does not contain any performance information.

Instruction Set Simulation

In MPSoC system-level designs, *Instruction-Set Simulators* (ISS) are commonly used for software simulation, such as in MPSim / MPARM [5, 15, 94], XTMP [47] and Synopsys CoCentric Studio [86]. They provide cycle-accuracy by using cycle-accurate ISS and other cycle-accurate models written for a system-level simulator, such as SystemC [109]. However, even in an ideal case, ISS runs in the range of 100KHz [154]. Assuming the target processors are running at 300MHz, for one second simulation of a 10-processor MPSoC implementation, it would take more than 8 hours of simulation. Although it is significantly faster than simulating the same implementation in the lower RTL level, it does not provide simulation result and feedback fast enough to efficiently explore the design space. With an 8-hour simulation, designers can only simulate one, or at most two, design decisions in a day. Even if the designers have two months for system-level design exploration, they can only explore around 100 implementations. This is considered a very small number compared to the implementations that are possible in MPSoC. The slow simulation speed prevents any reasonable design space exploration using performance information to explore different number of processors, software program partitioning, and multiprocessor architectures. ISS is useful for design verification. However, ISS is very slow and is the bottleneck of MPSoC simulation. Only a limited number of design explorations can be done at the ISS level.

Improved instruction set simulation techniques using interpretation [161] or compiled instruction [130] also provide instruction-by-instruction simulation of software. They intended to increase the simulation performance of ISS. However, they can only provide 10X speedup comparing to ISS, as these techniques require simulation of many architectural details and are still very slow and inefficient for system-level design space exploration.

Behavioral Simulation

On the other hand, system-level designs with behavioral simulation do not help driving design space exploration. A number of behavioral-level simulation platforms are developed to make simulation more efficient and allow functional verification at higher level. Metropolis [9], Simulink [72], Sesame [123], TTL [159] MESH [118] and SystemC *transition-level modeling* (TLM) [55, 64] all provide behavioral-level simulation for MPSoC. Simulation runs faster by running software natively on the host development machine. However, behavioral simulation only simulates the functionality of the designs and does not provide any performance information. Behavioral simulation, though useful for functional analysis at system level, does not provide any performance information that is crucial for design and analysis of MPSoC implementations. Although delay information can be manually annotated [13, 42, 61, 85, 128, 161], manual annotation is not automatic, not accurate, and not reliable.

2.3 Simulation Acceleration Techniques

Several simulation techniques have been proposed to improve simulation speed of hardware and software under different scenarios. Some of them deal with hardware-software cosimulation where hardware components and software components are simulated in different environments (simulators). Some of them try to improve simulation speed of the software components running on instruction set simulators. And some of them allow simulation to run on a distributed computing setting, such as in a supercomputer. They are different from the simulation speed improvement technique proposed in this thesis because this technique focuses on simulation in a unified hardware-software simulation environment, such as SystemC, and reduces the simulation overhead due to context switching in a single-processor simulation environment.

2.3.1 Hardware-Software Cosimulation

Yi *et. al.* [169] speeded up hardware-software cosimulation by setting arbitrary synchronization points between the hardware simulator and software simulator. This technique tries to avoid the idle time when one of the two simulator reaches a synchronization point, which is a time in the simulation clock where an interaction between hardware components and software components may occur, and waits for the other simulator to get to the same synchronization point. Such synchronization points may happen very often when the hardware components and the software components can interact anytime. Yi *et. al.* tried to avoid such idle time by setting arbitrary synchronization points that are further apart and only allowing the simulators to communicate all pending interactions at those points. Therefore, simulation speed enhances with loss in accuracy.

2.3.2 Trace-based Simulation

Kim *et. al* [84] and Gao *et. al* [53] proposed techniques to reuse previous simulation results (traces) to speed up software simulation. Kim *et. al* first generated traces from an instruction set simulation and then use the traces generated for subsequent simulation. Since the subsequent simulation does not require simulation of target processor architectures in the instruction set simulators, simulation becomes faster. Gao *et. al* applied a similar technique in the same simulation by searching the instruction set simulation history for trace results of the same instructions. These techniques assume that the performance of the instructions remains the same regardless of changes in the implementation, which is often not true for MPSoC system since programs in the processors interacts. And although such techniques show speedup compared to instruction set simulators, they are not as useful for compile-code simulation which is much faster than instruction set simulators.

2.3.3 Distributed Simulation

Distributed discrete event simulation is commonly used in large scale molecular and bioinformatic simulation in distributed supercomputers. Distributed discrete event simulation seeks to provide more efficient simulation by relaxing the dependencies between the components. Those approaches either restrict the semantic of the system [28] or use speculation [74, 80, 120, 121, 167] to simulate in a distributed system. Such simulation is more suitable for homogeneous systems where each entity is identical and can be easily scaled to simulate in a distributed environment.

2.4 MPSoC design Optimization Techniques

MPSoC design optimizations have been a common research topic for the last decade. Specific researches have been focused on memory subsystem optimizations – which include cache selection [36], data allocation [37], interconnect synthesis [27,41,116,117,129,135,151] and network-on-chip [37,108], synthesis – which includes hardware-software partitioning [30,34,43,81], component selection [17,33,48,49,78,126,144,150,152], software partitioning [18,59,70,71,96,138,155], program mappings [14, 31, 33, 37, 78, 126, 155] and scheduling [8, 30, 34, 43, 69, 96, 133], as well as energy optimizations [31, 59, 75, 95, 98, 107, 108, 119, 134, 141, 165].

In these researches, design optimizations are either based on statistic models [33, 48, 49, 116, 117, 135], which are also profile-based but do not consider inter-dependencies between the programs in the execution and variations of behaviors during the execution, or rely on the statically analyzable properties of some restricted models of computation (such as task graphs [14, 18, 27, 30, 31, 34, 36, 37, 41, 43, 59, 70, 71, 78, 81, 95, 96, 126, 129, 134, 138, 141, 144, 152, 155], periodic tasks [98, 107, 108, 165, 168] and synchronous dataflow graphs [8, 17, 33, 69, 133, 150, 151]), which provide analyzable state spaces and simple high-level behaviors for the applications. Implementation decisions of such models can normally be transformed into mathematic problems. These avoid the difficult simulation and analysis problems that exist in more general specifications, which include KPN. However, designs of such restricted models are more difficult and restrict designers from writing efficient programs.

2.4.1 Task Graphs

Task graphs [14,18,27,30,31,34,36,37,41,43,59,70,71,78,81,95,96,126,129,134,138, 141, 144, 152, 155] are by far the most common specification used in MPSoC design optimization researches because they are easy to analyze statically. A task graph defines the procedures and their dependencies in an application as a *Directed Acyclic Graph* (DAG). Each vertex in the graph is a procedure function (task) with pre-determined execution time that can only be executed after all its parents finish. Due to its acyclic nature, the graph is not designed to express continuous execution. Each vertex only executes once. The state space of any given time can be represented by the remaining execution time of each task, which provides a small state space that can be analyzed exhaustively.

2.4.2 Periodic Tasks

Periodic task representation [98,107,108,165,168] is a well-studied model of computation in researches. In this model, a fixed set of known tasks are invoked periodically. No communication and dependency between tasks are expected nor modeled. Each task is ready to be executed every fixed period of time, takes a pre-determined length of execution time, and has to finish before its deadline. Periodic tasks have been well-studied in both embedded software and distributed system communities. However, it does not provide a mechanism to define dependencies between tasks, which makes it less useful for MPSoC designs.

2.4.3 Synchronous Data-flow

A *Synchronous Dataflow* (SDF) graph [8, 17, 33, 69, 133, 150, 151] is a subset of KPN where each process (actor) has a fixed firing rule. When an actor is fired, it always consumes a fixed number of data in the input FIFO channels and produces a fixed number of data in the output FIFO channels in a known execution time. Due to the fixed firing rules, a SDF application can be statically analyzed in the process network level using matrices without any information inside the processes. However, the restrictions of having fixed firing rules prevent control-oriented and data-dependent applications to be efficiently designed and restrict designers from write regular software. SDF has been extensively studied by the research community because of its restricted semantics and analyzable behaviors. The state space of a SDF graph at any given time can be represented by the number of data in the FIFO channels and the remaining execution time for processes (actors) if they are executing.

2.4.4 More General Specification

A number of more general specifications that impose less specification restrictions have also been used in MPSoC researches. It includes but not limited to *Symmetric Multi-Processing* (SMP) [143], OpenMP [77, 139], *Message-Passing Interface* (MPI) [136, 137] and *Kahn Process Network* (KPN) [45, 101, 160]. These models of computation have been used to demonstrate design methodologies and illustrate MPSoC capabilities. However, due to the unrestricted semantics in the programs, applications written in these specifications are not statistically analyzable and currently no optimization techniques have been proposed to optimize these applications except those using statistical methods, which only provide a rough summary of the implementation without sophisticated analysis of the implementation and do not provide insight on the important parts for design optimizations.

2.4.5 MPSoC-Specific Profiling

Traditional software profiling information on individual processors has been extensively used in optimizing software. During simulation, execution frequencies of statements are recorded [10, 32], and the most frequently executed statements (*hotspots*) in the programs are determined for optimizations. However, such profiling information is only applicable for single-processor designs where all statements run sequentially. Traditional software profiling does not provide accurate hotspot information for MPSoC.

Traces are normally used in post-simulation analysis for MPSoC designs [99]. Visualization and debugging tools such as Vampir [24] and Paje [44] provide interfaces to visualize the executions of the programs in MPSoC simulation. These tools focus on efficient generation, synchronization and interpretation of multiple traces from multiple processors. They do not create hotspot information for the programs and no automatic analysis has been proposed for software optimizations.

Longest path finding in parallel computation is common for hardware designs. Critical paths at logic [3] and gate level [38] have been extensively used to estimate the shortest clock cycle possible in a synchronous digital circuit. Throughput analysis has been applied to Synchronous Data-Flow graphs [8] without the need of simulation. These works rely on well-define semantics of the parallel executing elements that do not exist in many richer models of computation such as Kahn Process Network. This thesis considers MPSoC designs which consist of multiple programs that are large and require long simulation to activate meaningful execution paths.

Chapter 3

Kahn Process Network

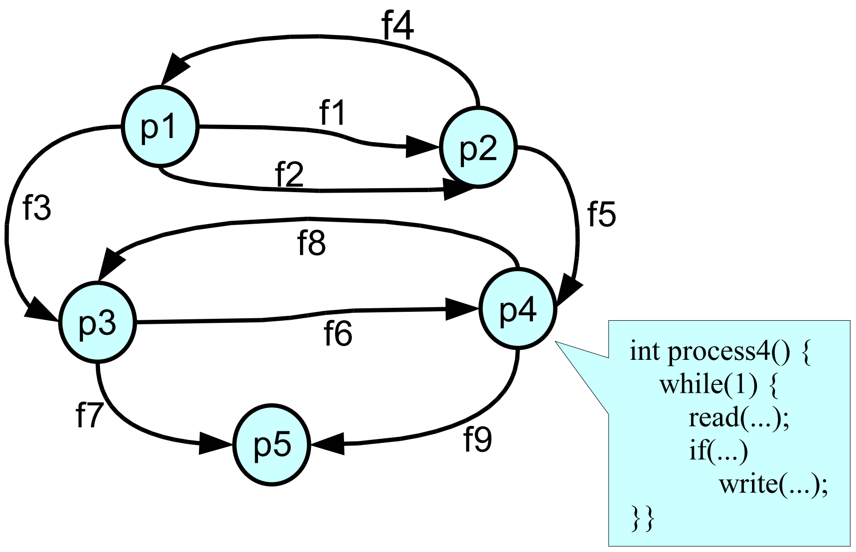
Kahn Process Network (KPN) [79] is a popular computation model to specify functional behaviors related to dataflow and streaming. This class of applications includes streaming-based multimedia and signal-processing applications, such as MPEG encoders and decoders. In a process network, computation and communication are explicitly specified and task-level parallelism in the application is explicitly defined. Implementation details are abstracted away from the functional specification, which provides flexibilities for implementation choices such as different hardware and software options in MPSoC.

KPN specification provides designers a powerful modeling language to model designs with explicit parallelism and complex controls. The freedom to write arbitrary programs in high-level programming languages for the processes with complex control-flows allows designers to write efficient designs. As multimedia and gaming processing are becoming more complex and control-oriented, it is necessary to provide designers with a simple programming model that does not impose unnecessary restrictions. KPN is well suited for that need.

KPN has been used to efficiently develop distributed applications. de Kock [45] used KPN to develop real-world multimedia applications and implemented them in MPSoC. YAPI [46] is a C++ API that implements KPN in a single processor. Compaan project [83] attempts to transform sequential programs, especially those with nested loops and tight bounds, into KPN. Laura [148] and Espam [106] automate the generation of HDL implementations from the higher level system specifications in KPN and use commercial tools to synthesize them into implementations. Saseme / Spade [92] also proposes the use of KPN for application specifications in MPSoC systems. These researches focus on KPN specifications, design methodologies from behavioral level to implementation level, and code generations for custom hardware implementations. They do not have an automated approach nor provide a methodology to simulate and optimize MPSoC implementations.

In this thesis, KPN is used as the specification model. The KPN specification of an application is either designed manually by the designers or generated automatically from a sequential program using tools such as Compaan [83]. The design optimization methodology proposed in this thesis takes the KPN specification as an input and optimizes the MPSoC implementation for performance and cost.

3.1 Specification



```
int process4() {
   while(1) {
      read(...);
      if(...)
         write(...);
}}
```

Figure 3.1: High-level diagram of a Kahn Process Network application. Each vertex is a sequential program and each directed edge is a FIFO channel.

A design written in KPN consists of processes and FIFO channels, as shown in Figure 3.1. They basically form a directed graph where the vertices are the processes and edges are FIFO channels. Processes are regular sequential programs that can have arbitrary data and control flows. They communicate with each other by reading and writing from the FIFO channels. For each FIFO channel, there is only one reading process and one writing process. A FIFO channel can only be read when there is data inside.

A KPN application can be either a close or an open system. A close system means the KPN does not have any external FIFO channels where one side of a FIFO channel is not in the system. On the other hand, an open system has at least one such FIFO channels to be used as a data source or to generate data output. Here I focus on open systems where the input FIFO channels take the input streams, such as MPEG-2 video streams, and the systems generate data output to the output FIFO channels, such as the decoded raw video frames.

3.1.1 Deterministicism

The key benefit of using KPN is that an application specified in KPN is functionally deterministic [79, 82]. The functional behaviors of the application do not depend on the execution order of the processes (timing when the processes execute and the FIFO channels are accessed). The behaviors only depend on the specification of the application and the input streams (for open systems).

This is very beneficial for design space exploration in MPSoC because there is no possibility of race conditions. Race conditions are the major issues in MPSoC because multiple interacting programs run concurrently and asynchronously on a set of processors. If the behaviors of the application change with slightly difference in timing, there is a race condition which makes design of MPSoC very difficult as minor changes in the implementation may render the behaviors different or incorrect. In KPN, functionally correct execution order is guaranteed as long as the processes and FIFO channels are implemented correctly. This property provides huge implementation freedom that can be explored for better MPSoC implementations without worrying about the functional correctness.

3.1.2 Modularization

Another benefit of using KPN is that KPN promotes modularization and code reuse of the software. Since computation and communication of an application is completely separated into processes and FIFO channels, the behavior of one process does not affect the behaviors of other processes. Therefore, an application can be separated into well-defined processes and then implemented and tested independently. Such modularization also promotes code reuse. A common well-defined process can be reused in multiple applications that have some common functionalities, such as the IDCT function used in both JPEG and MPEG designs.

Such modularization can be hierarchical. An open KPN system, which takes inputs and generates outputs outside the system, can be used as a building block for a more complex application. An example is a *Picture-in-Picture* (PIP) application that uses two MPEG decoders to decode two video streams and combine the raw video frames together to form one PIP video frame stream. Such hierarchical design can later be flatten to create one single Kahn Process Network.

3.2 Scheduling

The specification of KPN does not specify any particular way to schedule the processes to be executed. The specification only restricts that a FIFO channel can only be read when there is data in the channel (and data can always be written to the FIFO channels). KPN guarantees that any orders of executing processes would result in the same outputs. This allows highly flexible scheduling mechanism without affecting the functional behaviors of the application. However, the scheduling mechanism does affect the implementation, performance and memory usage.

When implementing a KPN application in SoC with limited memory, the biggest challenge is to bound the runtime memory usage. The scheduling policy cannot allow a process to repeatedly write to a FIFO channel without allowing the other process to read from the FIFO channel. In such case, the design will eventually run out of memory. To resolve this problem, Parks [115] specified a scheduling policy where the memory usage is limited. Specifically, a bound is set for each FIFO channel and a process can write to a FIFO channel only if the bound is not reached. With the FIFO sizes bounded, deadlocks may occur. Parks [115] proposed an algorithm to expand the FIFO sizes uniformly when there are deadlocks. Geilen [54] extended the algorithm to expand the FIFO sizes only for those FIFO channels that are involved in the deadlocks and resolve them whenever they occur. These algorithms can be used at design time with profiling information or at runtime during execution. Such deadlock-free FIFO sizes are properties of the specification and the input streams (for open systems), and do not depend on the implementation. Hence for simplicity, I assume the FIFO sizes are known, and the implementation is deadlock-free when such FIFO sizes are used.

The scheduling of KPN enforces that the memory usage is limited by restricting the processes from reading and writing to the FIFO channels. At any given time, a process is either said to be *ready* or *blocked*. A process is ready when continue executing it does not violate the KPN specification restrictions and FIFO size bounds. Otherwise, the process is blocked. In other words, a process is blocked under one of the following two conditions:

3.2.1 Blocking Read

A process is forced to suspend (blocked) when it is going to read from an empty FIFO channel. The KPN specification defines that a FIFO channel can only be read if there is data. Therefore, a process trying to read from an empty FIFO channel has to wait until another process writes to the FIFO channel. This is done by *blocking* the reading process and *unblocking* it later when there is data available.

3.2.2 Blocking Write

A process is also forced to suspend (blocked) when it is going to write to a full FIFO channel. A full FIFO channel is defined as a FIFO channel where the number of available data has reached the set bound. Therefore, a process trying to write to a full FIFO channel has to wait until another process reads from the FIFO channel. This is done by *blocking* the writing process and *unblocking* it later when there is space available. No overflowing is allowed in the FIFO channels.

3.3 MPSoC Implementation

KPN is a distributed model of computation that can be easily implemented in MPSoC. Each process in a KPN application is a sequential program (in C/C++) that can be compiled and run on the embedded processors without significant modifications. With a proper interconnect network and implementations for FIFO channels, a KPN application can be easily mapped and run on MPSoC (Figure 3.2).

Processes can only communicate using FIFO channels and all process synchronization is embedded inside the data transferred by the FIFO channels. This allows KPN applications to be implemented in MPSoC without expensive synchronization constructs necessary in other computation

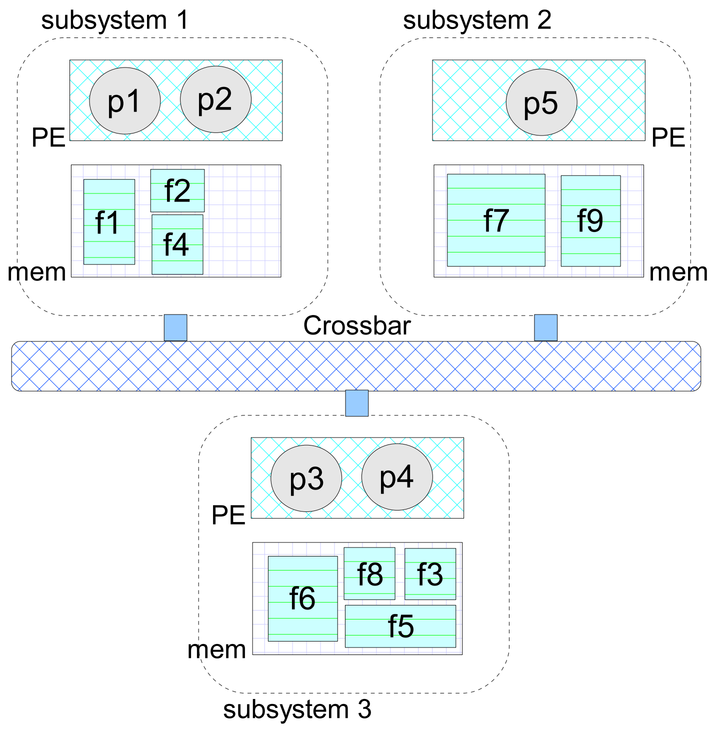


Figure 3.2: An MPSoC implementation of a KPN application. Processes are mapped to the processors and FIFO channels are mapped to memories. FIFO channels are implemented as software FIFO.

models such as in Symmetric Multi-Processing (SMP). Particularly, no cache coherency and atomic accesses are necessary because processes do not have to share memory spaces.

3.3.1 Process

KPN processes are sequential programs extended with the ability to communicate with other processes using FIFO channels. For a KPN application written in a C++ API, such as YAPI [46], the processes are written in C/C++ that can be compiled with C/C++ compilers of the target embedded processors. To allow proper compilation, implementations of FIFO reads and writes have to be annotated onto the processes. An invocation of a FIFO read or write checks the FIFO channel to determine if there is any data or spaces. It blocks the process if no data (for read) or no space (for write) is available. Otherwise, it reads or writes from the FIFO channel.

Multiple processes can run on the same processor. In such case, in additional to the KPN scheduling policy described in Section 3.2, a multi-program scheduling policy has to be used. The multi-program scheduling policy determines which of the ready processes is executed in the processor at any given time when more than one processes are ready.

In the implementation, a *Real-Time Operating System* (RTOS) with priority, round robin or time-slicing scheduling is used. The RTOS keeps the state of each process during execution. A process can be either ready or blocked. If the process is ready, it can be scheduled to execute by the RTOS. If the process is blocked, the RTOS checks whether it is ready periodically and changes its state back to ready. When a process executes and is blocked, it notifies the RTOS and preempts.

Each processor belongs to a processor subsystem. Each processor subsystem has a processor, a local bus, an instruction memory, a data memory, and an interface to access other processor subsystems through the interconnect network. Such architecture is commonly used in commercial products shown in Section 1.1 and allows maximum bandwidths within a subsystem and in between subsystems. Local memory accesses are fast and do not get slow down by other subsystems because of congestions in the interconnect network. All instructions and data that belong to the processor are stored in the local memories for efficient accesses.

In the implementation, processors can have different sets of instructions and run in different voltages and frequencies. With different available instructions and voltage assignments, performance and power consumption of the implementation change dramatically. The desired implementation should satisfy the performance constraints while minimizing the total power and area usages. Local memories for a processor run at the same voltage and frequency as the processor.

3.3.2 FIFO channel

FIFO reads and writes are the only communication primitives in KPN. There are several different implementations of FIFO channels. A FIFO channel can be implemented as a hardware FIFO, which is generally faster because of the dedicated FIFO control and memory but requires more area and power; a FIFO channel can also be implemented as a software FIFO, which is cheaper but slower because of the software FIFO control and shared memory accesses. In my implementation, software FIFO is used because it is cheaper and more scalable.

When implementing a FIFO channel, one important consideration is the size of the channel. The most important requirement is to execute the KPN application correctly without deadlocks.

To avoid deadlocks, I use a FIFO sizing technique based on Geilen [54], which increases the FIFO sizes that are responsible for the cyclic dependencies of the deadlocks. The minimum FIFO sizes to run an application correctly are the property of the KPN specification and are independent from the implementation. The same minimum buffer sizes can be used for different implementations.

Task Transaction Level (TTL) [159] message-passing interface is used to implement FIFO channels in the memories. In TTL, only one program writes to a memory address and only one program reads from the memory address. Therefore in the implementation, the memory space is allocated in the local memory of the reader. Memory latencies of the communication between processors are hidden if the reader is not waiting for the writer because write operations are posted in the processors. Writing to a remote memory is done via a cross-bar on-chip interconnect network. A mix-clock FIFO [35] is used to transfer data across buses of different clock frequencies.

3.4 MPEG-2 Decoder in KPN

An industrial-strength MPEG-2 Decoder design [46] is used as the design example in this thesis. The MPEG-2 Decoder is manually designed such that high-level parallelism of the application is explicitly defined. The decoder is developed in YAPI [46], a KPN C++ API, with complex data-dependent controls and operations. The modeling of the application is independent of the implementation and does not consider any implementation details in MPSoC. The specification of the MPEG-2 Decoder design consists of 9 processes and 63 FIFO channels. The process network is shown in Figure 3.3. In the specification, MPEG-2 stream decoding stages are clearly defined in the processes. The controller process (*Thdr*) controls the dataflow of video streams through decoding stages. The streams is first parsed with variable length decoding (*Tvld*) into frames and macroblocks. Each of them is decoded through inverse scan, inverse quantization (*Tisiq*) and inverse discrete cosine transform (*Tidct*). Prediction processes (*TdecMV* and *Tpredict*) predict the frames and motion compensation. Output processes (*Tadd*, *TwriteMB*, *Toutput*) combine the streams and produce output raw video frames. FIFO channels between the processes are used for both controls and data transfers. The MPEG-2 Decoder is a fully functional design where a raw video frame stream is produced as an output in simulation or execution.

The decoder is implemented in the MPSoC architecture as shown in Figure 3.4. Tensilica's Xtensa LX2 processors [154] that can be configured using *Xtensa Processor Generator* (XPG)

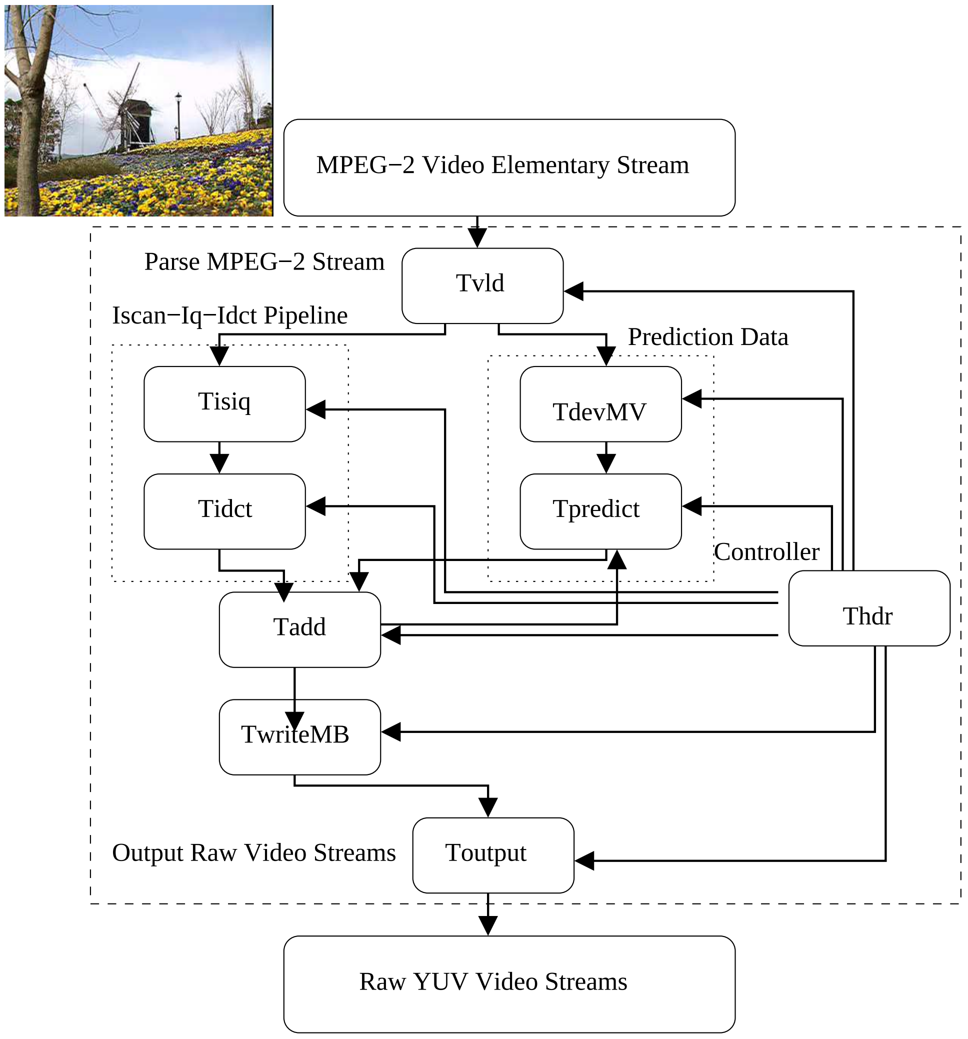


Figure 3.3: MPEG-2 Decoder application in KPN. The decoder is separated into 9 processes and 63 FIFO channels.

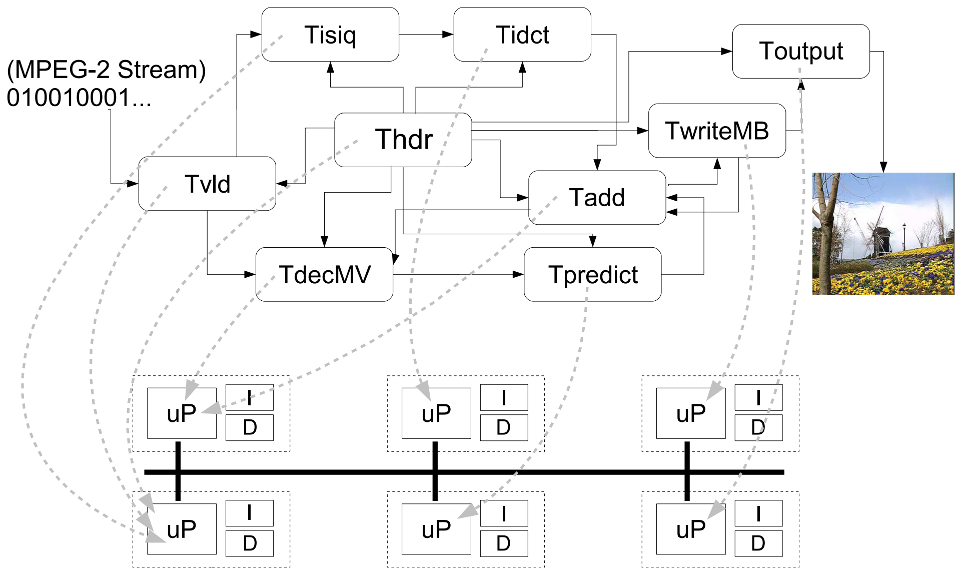


Figure 3.4: MPSoC implementation of the MPEG-2 Decoder application. The 9 processes are mapped into processors and 63 FIFO channels are mapped into memories.

and extended using *Tensilica Instruction Extension* (TIE) are used. Multiple processes can map into the same processors. Each processor has its own combination of instruction memories, data memories and caches. FIFO channels are mapped to on-chip memories that are shared. These processors are connected with an interconnect network of buses, such as a crossbar on-chip network, such that multiple processors can communicate by accessing the shared memories.

For experiments, video streams in MPEG Elementary Streams [104] with two standard resolutions (352x240 in 29.97fps and 704x480 in 14.97fps) are used. The video streams are chosen to characterize the average-case behavior [93]. The minimum total FIFO sizes to decode the videos are 139,596 bytes (for 352x240) and 555,396 bytes (for 704x480). Such FIFO sizes are determined from manual analysis of the application and the FIFO sizing technique based on Geilen [54].

Chapter 4

Design Optimization Methodology

A profile-driven optimization methodology is used to analyze Kahn Process Network applications and gather runtime information of the implementations. Unlike other process network specifications such as Task Graphs and *Synchronous Dataflow* graphs (SDF), a process in KPN can be any arbitrary sequential program, which allows designers to use high-level programming languages such as C language without restrictions. The expressive power of KPN enables applications to be written easily and efficiently. However, KPN places no restriction on the data and control operations in the programs. Analyzing the state space of an individual program is as hard as analyzing any sequential programs and analyzing multiple interacting programs on a multiprocessor system is even harder. Determining static information, such as the processing times of the processes, the number of data that the processes generate to the FIFO channels, and the number of data that the processes consume from the FIFO channels, are all NP-complete problem. Hence, any static or compile-time analysis cannot be done except in very small designs. Furthermore, *Worst Case Execution Time* (WCET) analysis [97, 127] often overestimates the computational requirements and is unable to identify the true path of execution.

This is the same problem software optimization communities face everything. And they have been using profile-based optimizations for a long time to analyze and optimize the software designs. Profiling information is very useful because it provides accurate runtime information of the applications and can be used in any KPN applications. Such profile-driven methodology has been commonly accepted and used for software optimizations in single-processor systems and it is reasonable to extend the use of the profile-based optimization methodology to multiprocessor designs because they are even more difficult to analyze statically.

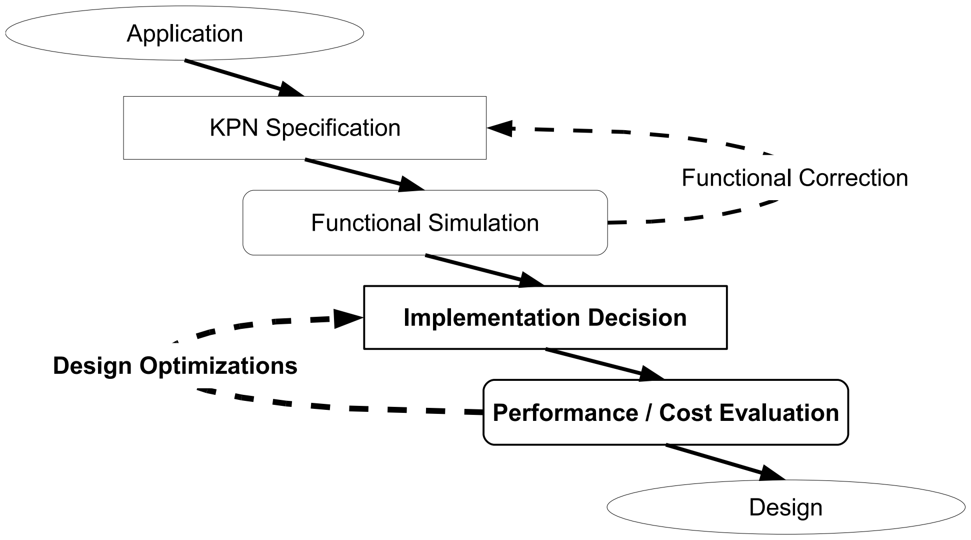


Figure 4.1: Design flow to implement a KPN application in MPSoC. Because the behaviors of KPN do not depend on implementations, functional simulation and verification can be done *in priori*. Implementation optimization decisions can be made by only considering the performance and cost (power and area) of the implementations.

The design flow is shown in Figure 4.1. After an application specification is written and functionally verified, the designers pick their implementation decisions and build a simulation model to evaluate their implementation. When the design constraints are not met and the implementation needs to be optimized, the designers would improve their decisions based on the previous simulation results and re-evaluate their implementation. These steps repeat until the performance and the cost (power and area) are satisfactory.

One drawback of such design flow is that the behaviors of the designs can be input-dependent (for open systems). Input data streams are needed to exercise data-dependent controls and operations in the designs, which can change both the functionalities and timing characteristics. Although worst-case scenario inputs are needed for hard-deadline applications, for multimedia and gaming computing (which are the main applications for MPSoC), not only the worst case happens very rarely and is not meaningful, but also the worst case inputs depend on the implementations. The average-case behavior [93] is actually more important in these applications and has been shown to work very well in the software optimization communities. As a result, designers would choose several input streams that are interesting, and set the performance constraints that are possibly the actual constraints plus margins for errors and input differences. The methodology determines whether an implementation is good based on these input streams and the constraints.

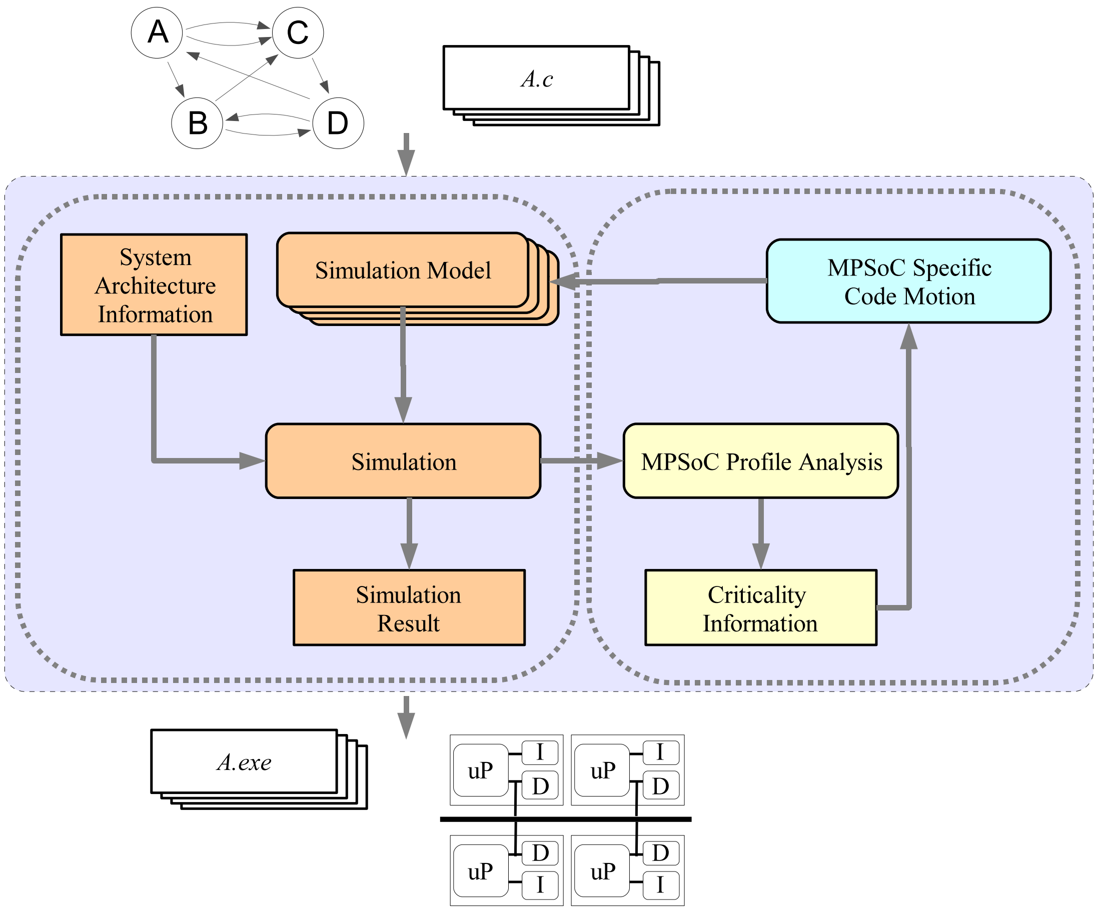


Figure 4.2: MPSoC design optimization methodology. The methodology is mainly composed of two parts: MPSoC simulation and MPSoC-specific profiling analysis.

The proposed MPSoC design optimization methodology is shown in Figure 4.2. To allow efficient software programming, the application is modeled in KPN, which is composed of a process network that describes the connectivity and sequential programs that describe process behaviors. An initial implementation is designed and simulated. During the simulation, an MPSoC-specific profiler analyzes the software programs that are simulating and generates useful profiling information for subsequent design optimizations. If the simulation results are not satisfactory or the results show some violations in the design constraints, designers can apply design optimizations, either manually or automatically, based on the MPSoC-specific profiling information. These optimization steps repeat until the implementation is satisfactory across all interested inputs.

There are two major components in this methodology: the simulator and the MPSoC-specific profiling analysis.

1. MPSoC simulation: The KPN processes run in the processors are regular sequential programs. Unlike Periodic Tasks, SDF and Task Graphs, where the processes (tasks and actors) have fixed execution time and dependencies that can be analyzed at higher level, KPN processes have to be simulated with the input streams in order to determine their behavior and timing characteristics. Because simulation is part of the design optimizations, the simulation must be fast enough to allow more optimization iterations. At the same time, the simulation must be accurate such that design optimizations that only have minor improvements can be applied and shown in the simulation results.

2. MPSoC-specific profiling analysis: Multiprocessor systems are very different from single-processor systems. In single-processor systems, instructions are executed as a single stream of instructions. Reducing the execution time on any of the instructions reduces the overall execution time. However, MPSoC is different that there are multiple processors executing programs in parallel. Reducing the execution time on some instructions in a program does not necessarily reduce the overall execution time. In order to find the important parts of the programs for design optimizations, an MPSoC-specific profiling analysis has to be used.

4.1 Efficient MPSoC Simulation

Chapter 5 presents a simulation framework that allows MPSoC implementations to be simulated and evaluated very efficiently at system level. The advantage of system-level designs is the ability to explore different high-level process and architectural decisions. Therefore, the ability to give fast feedbacks to the designers is very important at this level.

The framework uses compile-code simulation approach [9,11,21,22,39,90,92,123,125], which generates timed simulation models for software programs. In my simulation framework, the simulation models generated are capable to be used in MPSoC simulation with *SystemC* [109]. Specific techniques are proposed to ensure that the approach provides superior accuracy due to the fact that the estimation comes from the instrumented compilation procedures that generate the target software binaries. The simulation framework also considers compiler optimizations and simulates memory accesses. Complex run-time decisions are simulated to provide traceable analysis on the estimation. A simulation acceleration technique is also implemented to speed up simulation.

With the compile-code simulation framework that can simulate more than 1000X faster than *Instruction Set Simulators* (ISS) and is able to accurately estimate the performance with less than 5% error, we provide the designers with the freedom to explore the design space and constant feedbacks for their design decisions. Designers can test their decisions in finer steps and analyze their design decisions with the accurate estimation. With simulation speed exceeding 1000X of ISS, designers can test hundreds of configurations in one day, either manually or automatically. This opens up design space exploration opportunities that were not possible before.

4.2 Criticality-based KPN Optimization

Chapter 6 defines the *MP-Critical Path*, which is the execution path that determines the performance of an MPSoC implementation. The MP-critical path considers MPSoC-specific characteristics, where multiple processors run concurrently and programs interact. Therefore, to capture useful execution information to assist design optimizations in MPSoC implementations, the MPSoC simulation must be analyzed differently and intelligentially to determine the design decisions that are important.

With such profiling information, optimizations specifically for MPSoC can be applied efficiently. The profiling information based on the MP-critical path specifies instructions that are actually important in the MPSoC and provides essential information to the design optimizations. The optimizations can come at several different levels. As shown later in Chapter 6, optimizations can be done at instruction level, segment level and program level, where each of these optimization techniques interpret the MPSoC-specific profiling information differently. These design optimizations allow implementation tradeoffs between performance, area and power consumption that designers can explore to achieve satisfactory cost-efficient implementations.

Chapter 5

Efficient MPSoC Simulation

As the design complexity grows with MPSoC, it is important to reduce the time spent on every stage of the design cycles. Choosing the optimum design decisions is critical to achieve a satisfying cost-efficient implementation. With more design points to be explored, the designers have a lot of freedom to investigate in different design directions for a better quality implementation. Normally, the designers will first choose a specific implementation in mind. Then the implementation will be evaluated in term of the behavior, timing, area, power, and whether it violates any constraints. Based on the characteristics that are observed in the evaluation, the designers modify the implementation, either manually or automatically, to remove the undesirable characteristics or to reduce the cost. To fine tune some of the design decisions, the designers need a fast and accurate simulation framework to experiment different combinations of decisions. Note that this chapter focuses on the software programs because software is the most complicated part of MPSoC for estimation and simulation, and it is often the bottleneck on MPSoC simulation.

To allow designers to experiment implementations early in the design cycles and explore different implementations in a short amount of time, designs of complex systems are leaning forward *system level*. System-level design exploration platform is sometimes referred to as a *virtual prototyping platform*. It allows long simulation to be performed efficiently and provides very quick feedbacks to the designers. In addition, it provides very high visibility on the whole system. Designers can easily observe the execution of every detail of every component in the implementation and figure out the next design changes based on the observations.

The quality of a system-level design exploration result is directly related to the efficiency and accuracy of the simulation. Obtaining performance information about an implementation must be fast in order to meet the tight time-to-market constraints. The number of implementations that the designers can explore is limited by the number of simulation that can be performed within the amount of time allocated for system-level exploration, and a very efficient simulation is necessary. At the same time, accurate simulation is also necessary to confidently explore different implementations and design parameters in order to achieve a cost-efficient MPSoC implementation. With such system-level simulation framework, impacts from design parameter changes can be observed fast and accurately. The simulation framework provides designers an efficient design space exploration environment to optimize MPSoC implementations for performance, power and area.

5.1 Compile-code Simulation

To allow MPSoC to be simulated efficiently, one common technique is to use compile-code simulation. Compile-code simulation focuses on accelerating software simulation. To allow performance estimation, programs are analyzed and annotated with estimated timing delays. To achieve simulation efficiency, compile-code simulation avoids interpretation of target programs by converting them into native executables [9, 11, 21, 22, 39, 90, 92, 123, 125] or compiled instructions [130] using a compiler. The annotated timing delays are then executed natively on the host development machine for performance estimation.

As shown in Figure 5.1, the software simulation models provide performance estimation with timing delays and efficient simulation by running them on the host development machine natively. The left side of the figure shows a typical simulation path using *Instruction Set Simulators* (ISS) while the right side shows a straight-line where the programs are compiled natively for behavioral (functional) simulation. The middle one uses the compile-code simulation. The ISS approach provides cycle-accurate simulation of the software programs by compiling the programs into target binaries and simulating the timed processor hardware architecture that runs the target binaries for performance estimation. Compile-code simulation does not require running the programs on ISS. Instead, the programs are estimated and instrumented with execution times (timing delays), which creates simulation models for timed simulation. Since the simulation models are instrumented with estimated execution times, executing the models simulates the software programs with

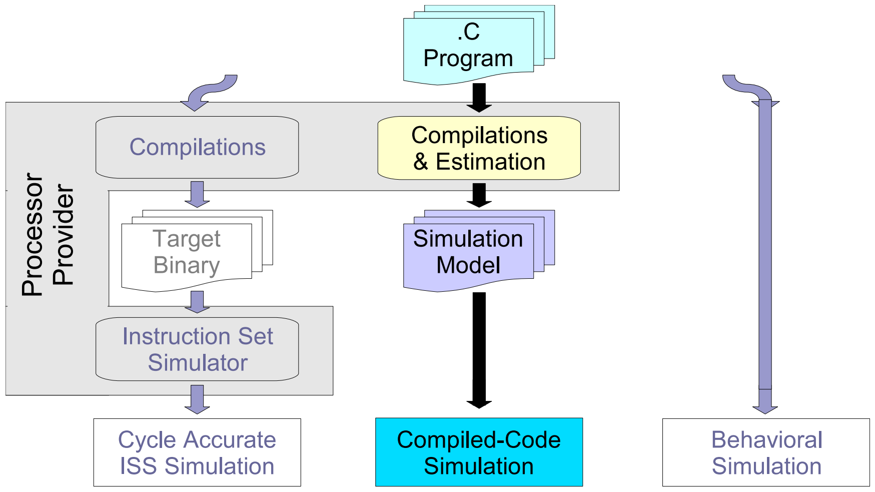


Figure 5.1: Compile-code simulation flow (middle) verse instruction set simulation (left) and behavioral simulation (right).

timing without the need of ISS. At the same time, the models execute directly on the host development machine similar to behavioral simulation. Therefore the simulation is efficient. However, current compile-code simulation techniques [21, 39, 73, 90, 125, 153] do not estimate performance accurately. Particularly, they do not consider compiler optimizations and memory accesses, which are very important on performance.

5.1.1 Software Simulation Model

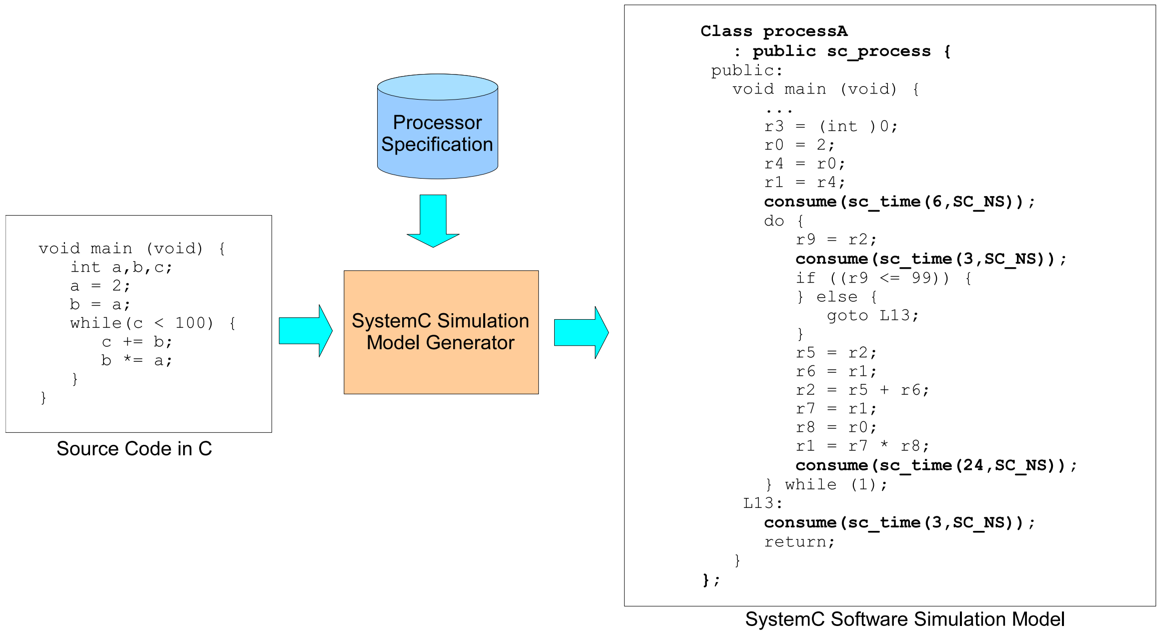
To allow MPSoC simulation, the software simulation model is a software *Transaction-Level Model with Time* (TLM/T) model [161] in SystemC [109]. The software TLM/T model is basically a behavioral-level description of the program annotated with estimated execution times. The model implements *SC_MODULE* in SystemC and has one *SC_THREAD* to execute the main program. Estimated timing delays to execute the program in the target processor are annotated onto the software TLM/T model using *consume()*. *consume()* adds timing delays to the model and synchronizes it with other concurrent simulating models in SystemC simulation. An example of a software TLM/T model is shown in Figure 5.2. In the example, *main()* executes once starting at the beginning of the simulation. The program calculates a set of numbers and explicitly writes to a memory address with a *Transaction-Level Model* (TLM) interface. The behavior of the simulation

```
#include “systemc.h”
SC_MODULE tlm_program                 // declare the software TLM/T module
{
    sc_port<mem_if> mem_port;         // memory port
    SC_CTOR (tlm_program) {           // SystemC constructor for tlm_program
        SC_THREAD (main);             // register main() with SystemC kernel
    }
    void main () {                         // program main function
        int i, j;
        wait (sc_time(18, SC_NS));         // estimation
        for (i = 1; i < 10; i++) {
            for (var[i] = 1, j = 0; j < i; j++) {
                var[i] *= i;
                consume (sc_time(15, SC_NS));  // estimation
            }
            consume (sc_time(24, SC_NS)); // estimation
            mem_port->write(0x400,4,&var[i]); // write 4 byte data to address 0x400
        }
        consume (m_never_notify);         // program stops here
    }
    int var[10];
    sc_event m_never_notify;
};
```

Figure 5.2: Software TLM/T model for MPSoC simulation.

model is the same as the software program and executes directly on the host development machine. The model is a SystemC component which can work with any SystemC models representing other MPSoC components in simulation.

Although compile-code simulation has been used for more than a decade for efficient simulation, it does not estimate software performance accurately, particular when compiler optimizations and memory accesses are involved. Other compile-code simulation platforms focus on simulation flexibility and use compilers only as a mean to achieve simulation efficiency [9, 11, 21, 22, 39, 90, 92, 123, 125]. In the following sections, techniques allowing accurate estimation, considering compiler optimizations and enabling memory access simulation are introduced to make the efficient compile-code simulation accurate such that it can be used for design space exploration in the methodology.


Processor Specification
SystemC Simulation Model Generator
Source Code in C
SystemC Software Simulation Model

```
void main (void) {
   int a,b,c;
   a = 2;
   b = a;
   while(c < 100) {
      c += b;
      b *= a;
   }
}
```

```
Class processA
   : public sc_process {
 public:
   void main (void) {
      ...
      r3 = (int )0;
      r0 = 2;
      r4 = r0;
      r1 = r4;
      consume(sc_time(6,SC_NS));
      do {
         r9 = r2;
         consume(sc_time(3,SC_NS));
         if ((r9 <= 99)) {
         } else {
            goto L13;
         }
         r5 = r2;
         r6 = r1;
         r2 = r5 + r6;
         r7 = r1;
         r8 = r0;
         r1 = r7 * r8;
         consume(sc_time(24,SC_NS));
      } while (1);
    L13:
      consume(sc_time(3,SC_NS));
      return;
   }
};
```

Figure 5.3: Automatic software TLM/T model generation using SystemC simulation model generator.

In the first stage of the methodology, an individual software program written in C is processed through a SystemC simulation model generator (a modified compiler, as shown later in this chapter) and automatically instrumented with timing delays (estimated execution time) when running on the target processor, as shown in Figure 5.3. The simulation model generated is a SystemC model of the software program. The simulation model generated behaves exactly the same as the original software program except with timing delays instrumented (although the program looks completely different, which will be explained later). The simulation model can work with an operating system model to accurately simulate the software program running on the target processor.

5.1.2 SystemC-based MPSoC Simulation

In the second stage of the methodology, the target multiprocessor architecture and interconnect network are built and specified using SystemC. SystemC is used because it is the *de facto* standard for system-level designs and design space explorations. The techniques are also applicable to other system-level modeling and simulation environments such as Metropolis [9]. In a multiprocessor system, the architecture and the interconnect network are as important as the software programs running on the processors. SystemC 2.0 provides a core language that introduces a set of constructs for generalized modeling of communication and synchronization at system level. These constructs include *channels*, *ports*, *interfaces*, *exports*, and *events*. They provide supports for system-level modeling of primitives such as queues, semaphores, memories, and buses. In the simulation framework, SystemC is used to specify the multiprocessor system architecture. The software simulation models generated are simulated with other SystemC *intellectual property* (IP) models (such as *operating systems* (O/S), interconnect network, caches and memories) for a complete multiprocessor system-level performance simulation. The simulation models are simulated as concurrent software programs running on multiple processors in the specified MPSoC architecture. Performance of the implementation can be estimated accurately in the SystemC simulation.

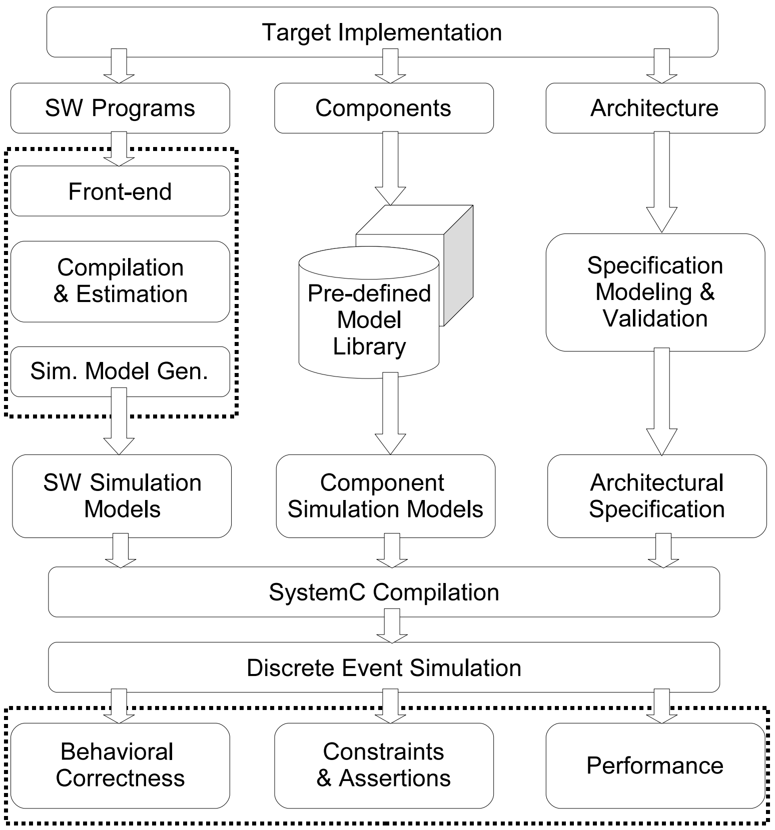


Figure 5.4: MPSoC Simulation Framework Overview.

Figure 5.4 shows the simulation framework used to explore the design space in MPSoC. The designers provide a description of the target implementation, which includes software programs, components, and the architecture of the MPSoC implementation. Based on the description, software simulation models are generated for the software programs and component simulation models are selected from a pre-defined simulation model library. All simulation models are written in SystemC. The architecture description is validated for semantic errors and connectivity problems, and is used to generate an architectural specification in SystemC. Using the simulation models and the architectural specification, a regular SystemC compilation is used to compile them into an MPSoC simulation. The OSCI SystemC simulator [109], implemented as a discrete event simulator, is used to simulate the implementation. From the simulation, the designers can determine whether the implementation behaves correctly and satisfies all constraints. The designers modify the implementation if it behaves incorrectly or fails some constraints. It also determines the performance of the implementation so that the designers can improve the performance or reduce the cost in power and area.

Modeling or obtaining operating system, interconnect network and memory models is a separate research topic and is not discussed in this thesis. The designers are assumed to have a library of SystemC models for these components with sufficient accuracy and speed required for simulation.

In the experiments, pre-characterized timed TLM simulation models are used. Operating systems, interconnect networks, caches and memories have well-defined characteristics and the services they provide are very deterministic, so the estimated timing delay for each service can be determined *in prior* and annotated into the simulation models. These models create the pre-defined simulation model library for components used in the simulation framework.

The *operating system* (O/S) models that provide services such as scheduling, pre-emption, suspension and interruption are proposed in several literatures [68, 103, 105, 140]. The O/S models are generic such that they do not imply any specific implementations of the O/S. The models take into consideration some of the most influencing parameters in the O/S, such as the scheduling policy, context switching time and scheduling latency. And the O/S models are able to provide common operating system services such as events, mutexes and mailboxes. The O/S models are built based on the system-level modeling constructs in SystemC 2.0. The software simulation models generated complement the O/S models by providing accurate estimation of software programs.

Caches and memories are also modeled in timed TLM to provide fast efficient simulation. The cache models are highly configurable. They can be configured with any combination of valid cache size, associativity, block size and replacement policy. Cache coherency can also be enabled to match the target memory subsystem implementation. CACTI library [145] is integrated into the cache and memory models to accurately estimate the access time and the energy consumption of caches and memories with different configurations.

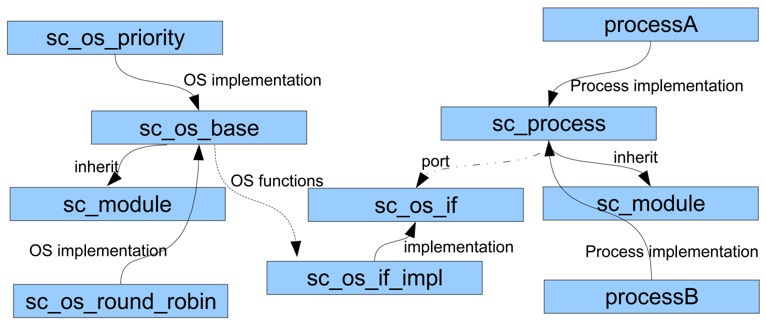


Figure 5.5: C++ class diagrams for simulating software programs and operating systems in SystemC.

Figure 5.5 shows a set of SystemC modules (C++ classes) used for program-O/S modeling. Software simulation models that are generated from the simulation model generator inherit from the virtual *sc_process* base class. The base class provides a SystemC port to connect the software simulation models to the operating system interface *sc_os_if* and access the functionalities provided by the operating system. The base class provides function calls such as *consume*() and *suspend*(), which are used by the operating system to serialize programs in a processor. *consume*() is called by the software simulation models to consume time from the processor and may switch to another program if a preemptive scheduling policy is used. *suspend*() is called in voluntary preemptive scheduling to tell the operating system to switch to another program.

Simulation of operating system relies on the functions *consume*() and *suspend*() in the *sc_os_if* interface, which are called by the simulation models to voluntarily consume time in the processor and preempt to other programs. The behaviors of an operating system are determined by the implementation of the *sc_os_base* base class. The *scheduling*() method in the operating system model determines the scheduling policy used in the processor. It also determines whether preemptive or voluntary preemptive scheduling policy is used. Context switching time and scheduling latency are also annotated into the operating system model to accurately simulate the operating system timings and behaviors.

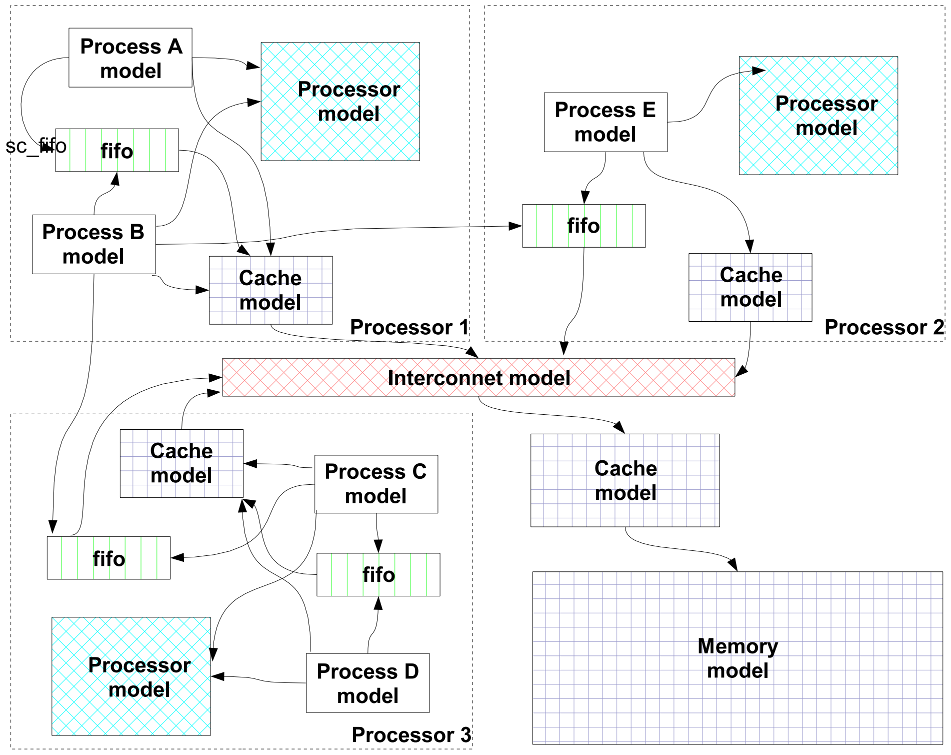


Figure 5.6: Architecture modeling of an MPSoC simulation in SystemC.

Figure 5.6 shows how the simulation models are connected together in an MPSoC simulation. Each of the software simulation models (processA to processE) is generated using the simulation model generator and implements the virtual *sc_process* base class. Software simulation models that link to a single operating system model, which implements the virtual *sc_os_base* base class, are a set of software programs running on the same processor. Memories and other peripherals are also presented to model the multiprocessor system architecture. Multiple processors are simulated using SystemC simulation kernel.

5.2 Consider Instruction-Level Information

In the simulation framework, instruction-level information is used to provide accurate estimation for the software simulation models. With the detail information available at the instruction level, estimation can be applied both efficiently and accurately. The software performance estimation accurately reflect the performance of the software programs similar to using ISS. At the same time, the models simulate at the speed similar to the behavioral models.

The instruction-level representation of a program in a compiler is very detail such that it accurately reflects the software binary that would be generated for the target processor. Hence accurate timing estimation that reflect the executing time on the target processor can be instrumented.

	instruction	branch	interlock
g3fax	65.20%	10.40%	24.40%
matmul	74.90%	4.70%	20.40%

Table 5.1: Execution time breakdowns of two benchmark programs running on a Tensilica's Xtensa processor with base configuration. Only major timing delay components are shown. Other components account for total less than 1% of execution times in both benchmark programs.

To accurately estimate the performance of software programs, an estimation must consider the instruction-level details of the programs in the implementation. Table 5.1 shows the percentages of execution times of two benchmark programs that are attributed to instructions, branches and interlocks. Branch penalties and pipeline interlocks clearly are significant portions of the execution times. An accurate performance estimation must take into consideration the branch characteristics (including branch predictions and delay slots) as well as pipeline interlock hazards. The analysis shows that the software simulation models that consider instruction selections, branch penalties and pipeline interlocks would provide superior performance estimations of the software programs. This is very different from other compile-code simulation approaches [9, 11, 21, 22, 39, 90, 92, 123, 125], which only consider the high-level behaviors and operations of the programs but do not consider the instruction-level details, especially branch penalties and pipeline interlocks. Such instruction-level information gives this simulation framework superior performance estimations over the estimation from other compile-code simulation approaches.

Overall, the estimation is based on:

$$Estimation = Instruction + Branch\ Penalty + Interlock \tag{5.1}$$

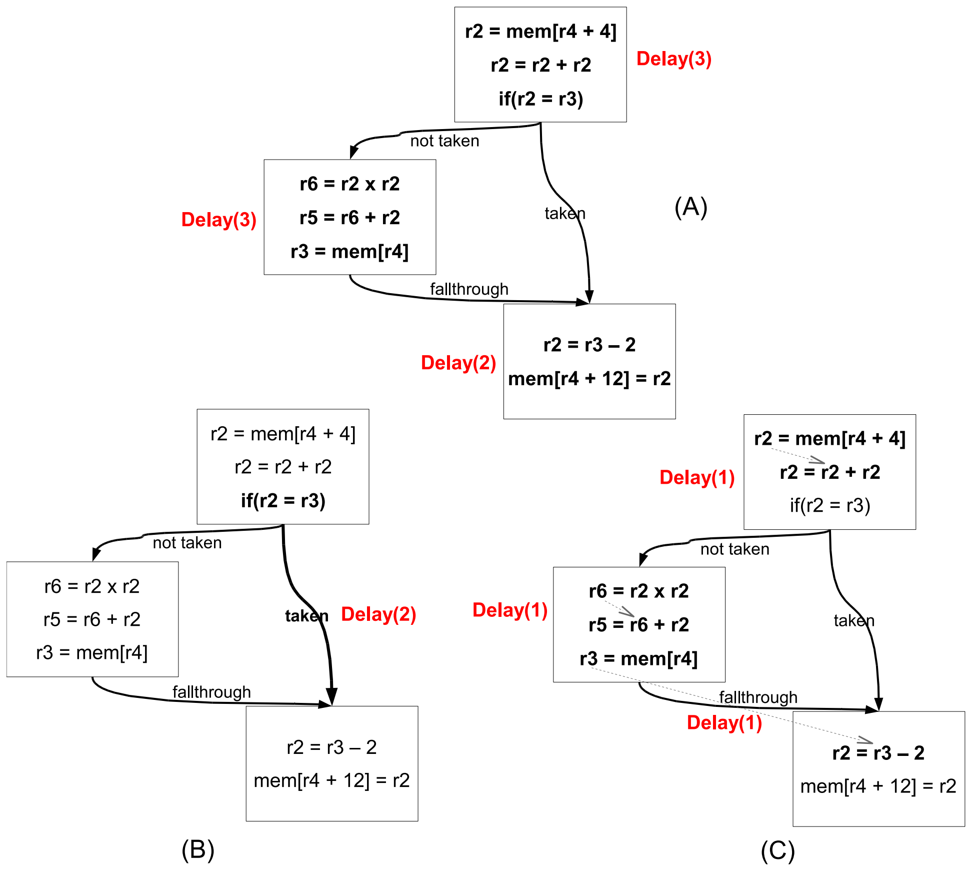


Figure 5.7: Execution time estimations of three major timing delay components: Instruction selection (A), Branch penalty (B) and Pipeline interlock (C).

5.2.1 Instruction Selection

An estimation for an instruction is the *cycle per instruction* (CPI) of the instruction format in the ideal execution condition. Compilations have already been applied and data- and control-flows have been finalized in the instruction-level information. Under a fixed in-order pipeline, most instructions in embedded processors only take one cycle (in the ideal case – branch penalties and pipeline interlocks are estimated separately). Special instructions that take more than one cycles are normally well-defined in the target processor documentations. Execution time for instruction selections can therefore be accurately estimated based on the program at instruction level.

Instruction estimation is applied in the unit of basic blocks. A basic block is a sequence of instructions that has one entrance, one exit and does not contain any jump instructions or jump targets. In MPSoC, a basic block also cannot contain any communication or interactions between programs. Since executing the first instruction in the basic block implies sequential execution of all the instructions in the same basic block, a basic block can be estimated with one timing delay. Such estimations are shown in Figure 5.7A.

For a different *Instruction Set Architecture* (ISA), estimation for instruction selections can be applied in the same way. By specifying the ideal CPI for each instruction format, instructions with different ISA can be estimated accurately.

5.2.2 Branch Penalty

An estimation for a branch penalty is the additional cycles caused by an indirection of the control flow. With static- or compile-time branch prediction, branch penalties can be statistically determined for a given control path. When a conditional branch result is being evaluated, the processor pipeline either stalls or executes speculatively. The fetched instructions may be valid or have to be flushed when the branch is mis-predicted. In either case, branch penalties for a branch taken and not-taken can be determined.

Branch penalties are annotated onto the control-flow. In the example shown in Figure 5.7B, the target processor always predicts a branch to be not taken and executes speculatively. If a branch is actually taken, a branch penalty is added as represented by the extra delay on the "taken" edge.

For a different processor pipeline, a different branch penalty may apply. In Tensilica's Xtensa processors, a branch penalty in a 5-stage pipeline costs two cycles, while in a 7-stage pipeline a branch penalty costs three cycles. For more complex branch prediction techniques, such as bimodal and dynamic predictions, a more complex analysis or a statistic model can be added to the simulation models. However, embedded processors normally use simple branch predictions that can be analyzed by the technique described above.

5.2.3 Pipeline Interlock

An estimation for a pipeline interlock is the additional cycles caused by the data-dependencies in the dataflow. A pipeline interlock happens when the result generated by one instruction is not immediately available to the next instruction as an input operand, which causes a pipeline stall. In most pipelines in embedded processors, the result of a register operation is available to the next instruction by forwarding the result to the next instruction before it is committed (actually written to the register). For more complex operations, such as multiplications, floating-point operations and memory operations, the results take longer to generate. Therefore, some amounts of stalls are required before the results of such operations can be used as operands in the next instructions.

Pipeline interlocks can be observed by analyzing the data-dependencies between consecutive instructions. As shown in Figure 5.7C, such dependencies can be within a basic block or in between basic blocks. Estimation of pipeline interlocks within a basic block can be annotated onto the basic block, while estimation of pipeline interlocks between basic blocks are annotated onto the edge in the control-flow. A window of instructions is used to analyze data-dependencies and estimate interlock cycles across all paths. The length of the window is the maximum distance between two instructions that can have a dependency, which in general is small. The length is one in a regular 5-stage pipeline and three in a 7-stage pipeline.

5.3 Consider Compiler Optimization

		cycle	(cmp-O0)
g3fax	-O0	3586965	
	-O1	1464664	(-59.2%)
	-O2	1348849	(-62.5%)
	-O3	1079985	(-69.9%)
matmul	-O0	1032809	
	-O1	211003	(-79.6%)
	-O2	160299	(-84.5%)
	-O3	113910	(-89.0%)

Table 5.2: Impacts of compiler optimizations in execution times on two benchmarks for Tensilica's Xtensa LX2 processor with base configuration using GCC 4.1 cross-compiler and different optimization options.

To accurately estimate the performance, the simulation framework must consider the impacts of compiler optimizations. Without advance computer architecture components in the embedded processors, compiler optimizations are very important because they are the only opportunities to schedule instructions and reorder basic blocks in a program. Table 5.2 shows the execution times on two benchmarks with different optimization options using *GNU Compiler Collection* (GCC) cross-compiler and Tensilica's Xtensa *Instruction Set Simulator* (ISS). "-O0" represents no optimizations. "-O1" includes dominator-based optimizations. "-O2" adds common sub-expression eliminations and software pipeline scheduling. "-O3" adds loop unrolling and inter-procedural analysis. The designers may turn on or off some of the optimizations because some of those involve space-speed tradeoffs, and some may even generate an incorrect program if the C program does not adhere to the ISO standard (i.e. aliasing). As shown in the table, "-O3" can reduce execution times by up to 89% compared to the un-optimized programs. Even the same optimizations can have different effects on different programs. Therefore, to accurately consider optimizations, optimizations that applied when generating target binaries must be applied without changes when generating the software simulation models.

5.3.1 GCC Compilation

GCC offers high quality compilations for many processors, including common embedded processors such as ARM, MIPS and Xtensa. GCC generates optimized instructions of a program based on the behavioral-level description, normally specified in C language. The optimized instructions of the program are used in the estimation to consider all compilations and optimizations applied.

To understand how to consider compiler optimizations, first an understanding of compilation and optimization procedures is required. In a GCC compilation, the C program is parsed to generate an *Abstract Syntax Tree* (AST). The tree data-structure is converted into an intermediate representation *GIMPLE*. GIMPLE provides a simple data-structure for optimizations such as *Static Single Assignment* (SSA) form optimizations. Based on the instructions provided by the target processor, GIMPLE is translated into the second intermediate representation *Register Transfer Language* (RTL). RTL provides a data-structure that closely represents target instruction sequences such that optimizations based on instruction sequences can be applied.

5.3.2 Apply Target Optimization

Compilations of the program in the compiler and optimizations that are applied to it are controlled by a set of parameters, such as operation primitives, storage layout, stack layout, addressing mode, and instruction format available in the target processor. Even for machine-independent optimizations, optimizations are applied differently based on these information in order to provide better optimization benefits on the target processor.

By specifying *exactly the same* parameters, same compilations and optimizations are applied to the program. The resulting RTL would reassemble the target binary program that would have been generated by the original compiler. Since the software simulation model is generated from GCC with the *same* compilation parameters as those used for generating the target binary for the target processor, all the compiler optimizations are reflected in the software simulation model and its estimation. The specific technique to do it in GCC is discussed later in Section 5.5.1.

5.4 Memory Subsystem Simulation

The memory subsystem, which can account for 50% of performance and energy expenses, is one of the most important architectural decisions to be explored. Cache and memory configurations are very important in MPSoC. Different programs have different memory requirements and cache access characteristics. It is important to configure the caches and memories such that the programs run efficiently under the constraints in performance, area and power. Analyzing cache and memory configurations in MPSoC is difficult because programs on different processors complexly interact. Memory address traces differ dramatically with minor changes in the implementation. Therefore, the address traces generated by ISS – a technique commonly used in single-processor system for cache configuration exploration [56] – cannot be reused for cache configuration exploration in MPSoC.

Cache simulation is currently the only reliable way to obtain accurate information about a particular memory subsystem implementation. Simulation that does not based on cache simulation with accurate memory address traces tends to be inaccurate. For target systems with cache, cache simulation can be done in SystemC using SystemC cache models.

The simulation framework allows cache simulation by accurately exposing memory accesses and the target memory addresses of these accesses. Memory address traces, which are essential for cache simulation, are generated during simulation as if the programs are running in *Instruction Set Simulators* (ISS). Hence caches and memories can be accurately simulated. Specifically, techniques to expose memory accesses in the software simulation models and target memory addresses of these accesses are discussed in the next subsections.

5.4.1 Memory Access

Memory accesses have to be exposed in the software simulation model because memory accesses of a program that are not explicit in the model would not be simulated. Memory accesses can be implicit or explicit in a program. Explicit accesses are those specified in the simulation model using *read()* and *write()*, which are used for communication and interactions between programs. However, implicit accesses, such as reading or writing a de-referenced address, an array, a volatile variable, or a program stack due to lack of registers, do not alter the program behaviors and are not specified in the software simulation model. Therefore, when memory subsystem simulation is not considered, only explicit accesses are instrumented with *read()* and *write()* and simulated in the model.

When a program accesses a memory address with either a *read()* or a *write()*, the simulation model issues a read or a write call as a SystemC call to the corresponding cache model. The cache simulator inside the cache model determines the result and the timing delay of the memory access. Hence, cache related timing information can be accurately simulated.

To expose implicit memory accesses and instrument them into the software simulation models for memory subsystem simulation, the instruction-level information inside the compiler (specifically, the RTL structure) is used. Implicit memory accesses are generated during compilations and optimized during compiler optimizations. Implicit memory accesses generated by the compiler are converted into load and store instructions, which represent accesses to the memory subsystem. The number and locations of the implicit memory accesses depend on the optimization levels in the compiler. In the instruction-level information, implicit memory accesses are clearly defined and can be used to instrument the software simulation models with proper *read()* and *write()*. Without using the compiler to analyze the programs, implicit memory accesses cannot be exposed accurately. Instruction memory accesses, on the other hand, can be annotated statically in the simulation model with the memory addresses of the instructions [142].

With *read()* and *write()* instrumented onto the software simulation models, memory accesses can be simulated for performance and energy estimations of different cache and memory configurations. To the best of my knowledge, no other system-level design platform is able to simulate caches alongside behavioral-level models (annotated or otherwise) with the same level of accuracy and efficiency. This is a great deficiency when the memory subsystem is significant for energy and performance characteristics.

5.4.2 Simulation-time Address Mapping

In addition to exposing all memory accesses in the software simulation models, the target memory addresses for the memory accesses need to be determined in order to simulate the memory subsystem accurately. Data in the implementation are mapped to specific target memory addresses. The target memory addresses determine the runtime characteristics of the caches and memories. The caches behave differently with different memory addresses.

Simulation of software simulation models directly uses the host memory on the host development machine allocated to the instances of the models in the simulation. Since a program is compiled and directly executes on the host development machine, the memory required to store the variable values resides on the host memory, where the host memory addresses are different from the target memory addresses of the same variables.

In general, it is impossible to determine target memory addresses for all loads and stores at compile time. Statically determining target memory addresses for all accesses at compile time is an NP-hard problem, especially when pointer manipulations are involved. Therefore, the simulation framework uses a lookup approach to dynamically determine the target memory addresses during simulation.

Target Memory Address

The memory map of the implementation is provided to the software simulation models to map the data into their target memory addresses. During target compilation, a linker maps data to specific memory addresses based on a linker script for the target binaries. The linker script specifies the memory address for each section: *bss*, *literal*, *stack*, *heap*, etc. Data can be placed into specific sections using compiler directives in the programs. To provide accurate target memory addresses for simulation, such memory map must be provided.

Without a provided memory map, which is common for early system-level design space exploration, one memory map can be generated. The generated memory map maintains the spatial and temporal localities of the memory accesses. Memory subsystem design space exploration can be done with minor loss in accuracy. If the exploration result is satisfactory, the generated memory map can be used to construct a linker script that keeps the same memory map as used in the exploration.

Address Lookup

A simulation time lookup procedure is used to determine the target memory address for each memory access. For each memory access exposed, the software simulation model reads or writes to an address in the host memory where the data is allocated. The host memory address indicates and identifies the data that is being accessed. Therefore, using the host memory address, a *reverse address mapping* (lookup) procedure is used to determine the target memory address of the corresponding access during simulation. A similar ”Address Recovery” technique has been proposed in [53] to forward memory addresses from the target memory space to the host memory space. However, that technique does not support any pointer manipulations, which are very common in multimedia and gaming computing. Reverse address mapping allows pointer manipulations on the host memory addresses and is able to map addresses after pointer manipulations to their correct corresponding target memory addresses because it only depends on the memory addresses after pointer manipulation and does not depend on how the addresses are manipulated.

For reverse address mapping, a reverse memory map *mmap* is used to record the relationships between host memory addresses and target memory addresses. *mmap* is a set of tuples with three fields: *host memory address* $\in H$, *data length* $\in N$ and *target memory address* $\in T$. H is the set of host memory addresses that are accessible directly from the software simulation models. N is a natural number that represents the size of the data in byte. T is the set of target memory addresses of the data in the implementation. All addresses are assumed to be byte addresses.

Property 1 *Uniqueness of host memory addresses:*

$$\nexists (h_1, l_1, t_1), (h_2, l_2, t_2) \in mmap$$
$$\textit{such that } (h_1 = h_2) \wedge (t_1 \neq t_2)$$

Property 2 *Non-overlapping addresses:*

$$\nexists (h_1, l_1, t_1), (h_2, l_2, t_2) \in mmap$$
$$\textit{such that } (h_1 < h_2) \land (h_1 + l_1 > h_2)$$

Since SystemC simulation is simulated in one host memory space, each data in the software simulation models allocates a specific address in the host memory. No host memory addresses of two data are the same or overlapped (Property 1 and 2). Each data in software, when in scope, has an unique host memory address. Each instance of a simulation model allocates a different memory address. Therefore, when a data goes into scope, $register()$ is invoked to add a new entry into *mmap* (Algorithm 1). When the data goes out of scope, $unregister()$ is invoked to remove the entry from *mmap*. $register()$ and $unregister()$ are inserted into the software simulation models when the models are generated. The entries in *mmap* always obey Property 1 and 2. *mmap* also applies to stacks and heaps as they are considered big chunks of memory allocated to the programs.

Algorithm 1: Register and Unregister at Reverse Address Mapping

```
1 $mmap = \varnothing$
2 $register(h \in H, l \in N, t \in T)$
3     check against lemma 1 and 2
4     $mmap = mmap \cup \{(h, l, t)\}$
5 $unregister(h \in H)$
6     $mmap = mmap / \{(h, *, *)\}$
```

On the other hand, target memory addresses are not unique and can overlap. Unlike host memory which is in one memory space, in the implementation programs in different target processors can have asymmetric memory views and access different memories with the same memory address. In this case, two host memory addresses map to the same target memory address, and the memory accesses access different memories in SystemC simulation as the accesses are originated from different processors that are connected to different memories.

Property 3 *In-order memory addresses:*

$$(\forall (h, l, t) \in mmap) \land (\forall i < l)$$
$$\rightarrow lookup(h + i) = t + i$$

Memory for a data (including an array or a data-structure) is always contiguous and data-types for data of the same size are used. Therefore, in the program point of view, if the host memory address h with size l maps to the target memory address t, an offset added to the host memory address ($h+i$), as long as $i < l$, maps to the same offset of the target memory address ($t+i$). This property (Property 3) allows pointers (data addresses) to be used to lookup their corresponding target memory addresses after pointer manipulations.

Algorithm 2: Lookup Reverse Address Mapping

```
1 sort *mmap* with increasing order of h
2 $lookup(s \in H)$
3   use binary search for last $(h,l,t) \in mmap$ such that $h < s$
4   **if** $h+l < s$ **then**
5       return *ERROR*
6   **else**
7       return $(t+(s-h))$
8   **end**
```

The reverse memory map allows the target memory addresses to be generated dynamically during simulation regardless of pointer manipulations in the programs. The function $lookup : H \rightarrow T$ (Algorithm 2) is used to determine the target memory addresses based on the host memory addresses. The complexity of such lookup is $O(lnN)$, where N is the number of entries in *mmap*. Hence, such lookup is scalable for big designs. During simulation, pointer manipulations are applied directly on the host memory addresses, and the resulting addresses are used to lookup the target memory addresses. A valid pointer manipulation always ends up in a memory address of a data that is properly declared in the host memory and registered in the reverse memory map. Unless the programs try to access memory that is not properly declared and registered, pointer manipulations are properly handled.

Address Map Example

An example of a reverse address mapping is shown in Figure 5.8. First, when a data is allocated, *register()* adds an entry into *mmap*. In the example, the array *ary* of total size 12 bytes resides in the host memory starting at 0xa044, depending on the virtual memory allocated by the

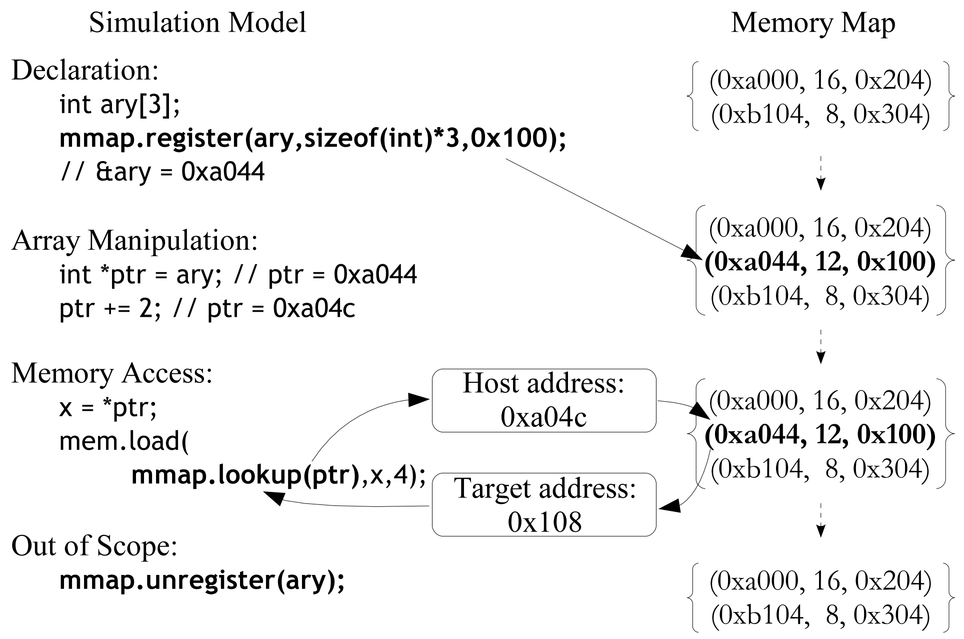


Figure 5.8: An example of Reverse Address Mapping that maps a host memory address after pointer manipulation to the corresponding target memory address.

host operating system. The target memory address for the array *ary* is 0x100, provided by a memory map. The host memory address of the array may be different every time the simulation runs, however it always map to the same target memory address. In *register()*, the entry $(0xa044, 12, 0x100)$ is added to *mmap*. Second, the address *ptr* is calculated by pointer manipulations. As a result *ptr* points to $0xa04c$ in the host memory, the third element in *ary*. Third, *ptr* is then used in a load instruction, where it is used in *lookup()* for the target memory address. An offset of 8 is then applied to the address 0x100, which results in the target memory address 0x108. The address corresponds to the third element of *ary* in the target memory. In general, for a legal load or store, the host memory address must point to a memory space that is properly declared, hence the address can always be used to map to a target memory address. Last, when *ary* goes out of scope, the entry in *mmap* is deleted with *unregister()*. Entries in *mmap* are dynamically added and deleted when variables and arrays go in and out of scope during simulation. This dynamic operation is necessary since the same host memory space can be reused by the simulator for multiple data where their lifetimes do not overlap.

5.5 Simulation Accuracy

By considering instruction-level information, applying target optimizations and allowing memory access simulation, compile-code simulation can be made accurate. All these techniques are implemented in the GCC backend with a machine description. In this section, the specific implementation of the simulation model generator, which is the GCC compiler with a modified compiler backend, is shown along with the accuracy achieved by the simulation framework.

5.5.1 Machine Description

As mentioned in Section 5.3, GCC uses a set of parameters that describe the target processor to guide the GCC compilations and optimizations. GCC offers high quality compilations of C programs for many processors, including common embedded processors from ARM, MIPS and Tensilica. GCC uses a *machine description* to describe the target processor and the accompanying assembly code generation. The machine description is a Lisp-like description that describes many aspects of the target processor. Based on the machine description, GCC creates different passes that affect various stages of the compilations and optimizations (Figure 5.9). In this way, GCC acts like a cross-compiler that uses different target machine descriptions for different target processors. GCC arranges its compilation and optimization passes according to the target processor-dependent parameters in the machine description. Based on the primitives, storages, stack layouts, addressing modes and instruction formats, GCC builds a cross-compiler that creates semantically and syntactically correct target binaries.

To create software simulation models in GCC for a target processor, a modified machine description similar to the original one (which creates target binaries) is used. One machine description is normally applicable for a family of processors, such as Tensilica's Xtensa processors with different configurations, where different compilation options (flags) generate target programs for different processors in the family. The modified machine description is modified as follows:

- The modified machine description has the *same* compilation parameters to generate the same target-dependent instruction-level information (*RTL structure* in Figure 5.9).
- Software estimation based on the target processor documentations – in this case, instruction, branch and interlock – is added;

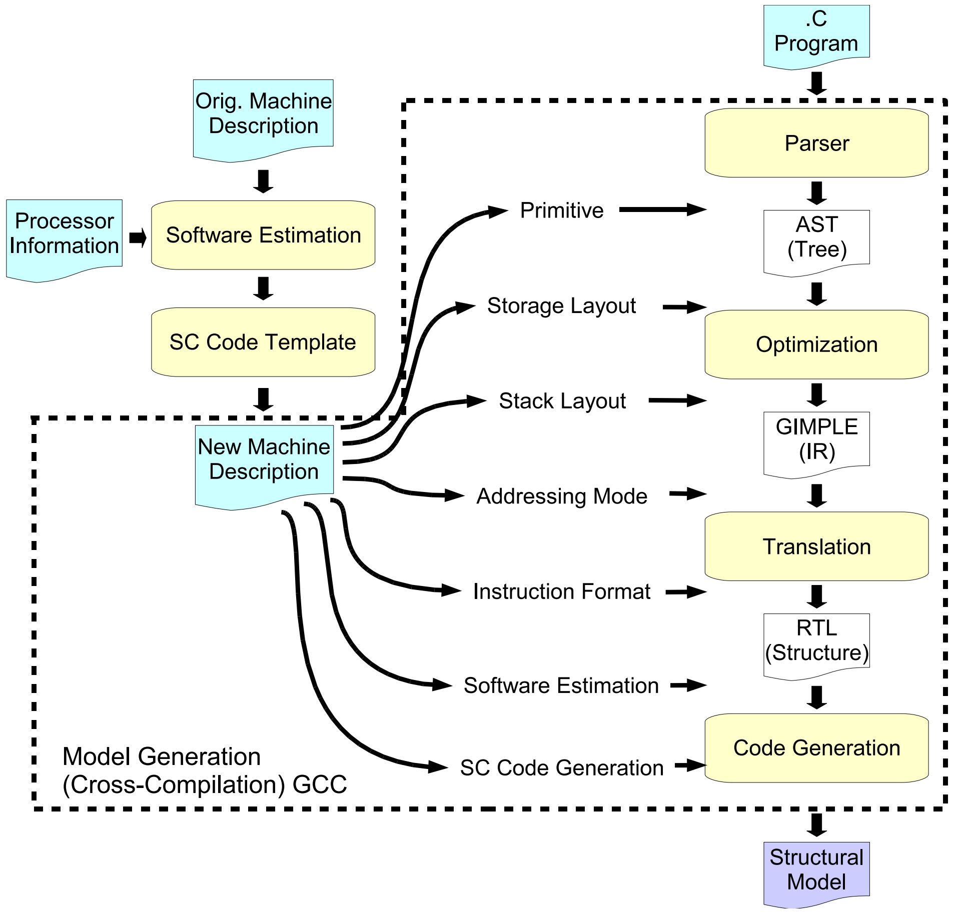
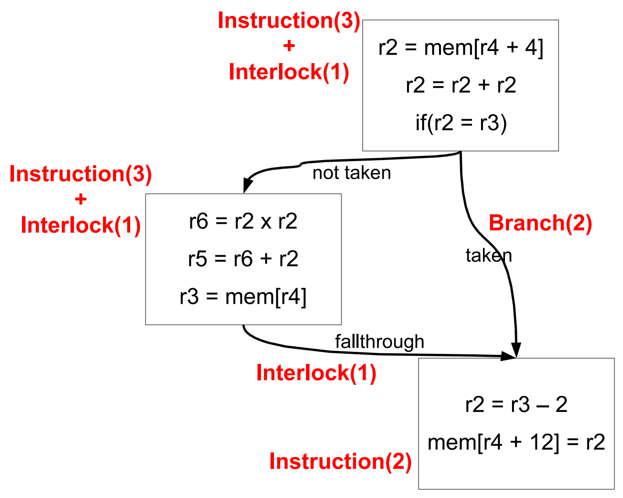


Figure 5.9: New machine description to generate software simulation models using GCC.

- Memory accesses are instrumented using *read()* and *write()* and data allocations are instrumented using *register()* and *unregister()*;
- C instructions as code generation templates are used;
- Compilation hooks to generate a semantically correct SystemC component are added.

5.5.2 Simulation Model Generation

Using the modified machine description, GCC compilation generates software simulation models that are useful for MPSoC simulation. A software simulation model for each program, which does not reassemble the original C program, is created for accurate estimation and MPSoC simulation. Generation of a completely different simulation model is necessary because of all the control- and data-flow changes done by the compiler. Compilations and optimizations in GCC, or any compilers, change the control- and data-flow of a program for the purpose of reducing its execution time or memory usage. However, after the changes there is no longer an exact one-to-one correspondence from the optimized RTL structure to the original C program. For example, a statement in the original program may be duplicated and the two new statements have different execution time because of other compiler optimizations. It is practically impossible to achieve the same accuracy with a manually-annotated or a back-annotated original C program.



```
public softwareA
                : public sc_process {
   public:
          sc_mem_port mem_port;

...
   r2 = *(int *)(r4 + 4);
   mem.load(mmap.lookup(r4+4), r2, 4);
   r2 = r2 + r2;
   CONSUME(4); // 3+1
   if (r2 == r3)
          {CONSUME(2); goto L10;}
   // CONSUME(0)
   r6 = r2 * r2;
   r5 = r6 + r2;
   r3 = *(int *)(r4);
   mem.load(mmap.lookup(r4), r3, 4);
   CONSUME(5); // 3+1+1
L10:
   r2 = r3 - 2;
   *(int *)(r4 + 12) = r2;
   mem.store(mmap.lookup(r4+12), r2, 4);
   CONSUME(2);
...
}
```

Figure 5.10: Software simulation model generated with timing delay estimations and memory access instrumentation based on instruction-level information.

In the software simulation model, only the semantic of the program, the timing delay estimation (based on the instruction selection, branch penalty and data interlock) and memory accesses are present. The modified machine description, as shown in Figure 5.9, allows GCC to generate C instructions with correct semantic along with estimation and annotations. The generated C instructions are wrapped in a SystemC class that implements *sc_process* (as seem in Section 5.1.2). An example of code generation from instruction-level information (as shown in Section 5.2) is shown in Figure 5.10. Processor details, such as register files and pipelines, are not described in the software simulation model hence are not simulated. Only their impacts on performance remain in the simulation models. This allows the simulation models to run at several orders of magnitude faster than that when the register files and pipelines are explicitly simulated (as is done on *Instruction Set Simulators* (ISS)).

Although the generated software simulation model may contain more C instructions than the original C program, they are simple instructions that describe the same semantic as the original program. Using an ideal compiler, both the software simulation model and the original program would compile into the same executable and have the same simulation performance. In reality, simulation of the software simulation model runs at most four times as slow as the original program. However, such overhead is unnoticeable in MPSoC simulation because most of the simulation time is used for synchronizing concurrent components in SystemC that is shown in Section 5.6.

If a different target processor is chosen or a different set of compiler optimizations is called for, another simulation model is generated to reflect the new choices. In all experiments, the time to generate a simulation model is always under 0.1 second – a very small amount of time especially comparing to the length of time involved in simulating a complete MPSoC system.

5.5.3 Estimation Accuracy

The modified machine description for the Tensilica's Xtensa processor family have been created and extensively validated. The modified machine description for the Xtensa processor family allows experiments with different processor configurations, such as different *instruction set architecture* (ISA) configurations and pipeline architectures (5- and 7-stage pipelines). The modifications are mechanical and can also be applied to other machine descriptions such as the ones for ARM and MIPS processor families. Such modifications only have to be done once for each processor family and the number of useful processor families is very limited.

For comparison, Tensilica's cycle-accurate *Instruction Set Simulator* (ISS) is used for single-processor systems and *Xtensa SystemC TLM* (XTSC) simulation is used for multiprocessor systems to simulate multiple instances of Xtensa LX2 processors in the SystemC simulation environment [154] at cycle-accurate level. The results from ISS and XTSC are used to determine the accuracy of the estimation from the compile-code simulation framework. For the experiments, Xtensa tools are installed and run on Linux with Red Hat Enterprise 4 and SystemC simulation is compiled using OSCI SystemC 2.1 library [109]. The simulation model generator for software programs is built with GCC 4.2.2. All experiments run on a Pentium 4 3.3GHz machine with 1GB of memory.

All software programs in the MPEG-2 Decoder design (Section 3.4) are compiled into software simulation models using the GCC compiler with the modified machine description. Each software program is compiled individually by the simulation model generator to create a software simulation model with time delays accurately instrumented. The software simulation model generation is automatic and no manual input is required. Compilation time increases as more optimizations are enabled. But even with the highest level of optimizations in GCC (-O3), generation of each simulation model takes less than 0.1 second.

In the first set of experiment, a set of benchmarks (from EEMBC and PowerStone) are used to demonstrate the accuracy of the software simulation models by themselves. It is a more important measurement because the accuracy of such estimation is not influenced by other components as in the MPSoC simulation (such as the accuracy in the O/S, interconnect network, cache and memory simulation models). The MPEG-2 Decoder design is then used to show the accuracy in MPSoC simulation.

A set of processors with different architectural characteristics are used to illustrate the accuracy, efficiency, and robustness of the estimation approach. *base* (0.68mm^2, 320MHz), *typical* (0.86mm^2, 317MHz), *typ+fp* (1.61mm^2, 314MHz) and *7-stage* (1.05mm^2, 311MHz) are four different configurations of Xtensa LX2 processors. The first three configurations have a normal 5-stage processor pipeline but contain different architectural components. The *base* processor is small and has no hardware multiplier nor floating point coprocessor. *typical* includes a 32-bit single-cycle multiplier. *typ+fp* further adds a floating-point coprocessor for fast floating-point operations. The *7-stage* processor has a similar configuration as *typical* but with a 7-stage pipeline.

	adpcm	g3fax	g721	jpeg	matmul	fpfir
base	1813251	1348849	484987	5767054	539854	378197
typical	505936	1348849	393115	3469753	160299	189361
typ+fp	505936	1348852	393111	3469753	160280	68675
7-stage	582000	1679730	478413	4045035	189176	213324

Table 5.3: Benchmark execution time (in cycles) in a diverge set of benchmarks and several Xtensa LX2 processors with different configurations. The benchmarks are compiled with "–O2" optimization flag in GCC.

Several diverge benchmarks are used to highlight the accuracy of the estimation approach on different types of programs. *g3fax*, *g721* and *jpeg* have different branch and interlock characteristics. *adpcm* uses multiplications and *fir* uses floating-point operations extensively. Table 5.3 shows the cycle-accurate execution time obtained from running simulation on ISS. The optimization level is set to "-O2", which is common for embedded software because it provides the best performance without increasing the code size. Each software simulation model takes less than 0.1 second to generate. The accuracy of the simulator does not depend on the optimization flags. Although not shown in the results, the estimation is accurate in any combinations of optimization flags – even when optimization passes are individually enabled or disabled in GCC. For the rest of experiments, "-O2" is used unless specified otherwise. The execution times in the table clearly illustrate the need to match processors with applications. All benchmarks benefit from using the *typical* configuration except for *g3fax*, which does not contain any multiplications. The only benchmark that benefits from the floating point unit is *fpfir*, since it is the only one with floating point operations. Having a 7-stage pipeline allows slower 2-cycle memories although the processor is bigger and slower, which affects some benchmarks more than the others.

The estimations instrumented in the software simulation models are accurate across different benchmarks and processor architectures. The experimental results are shown in Figure 5.11. For comparison, *unopt. instr.* is an estimation based on an un-optimized program and each instruction takes one cycle without considering branches and interlocks. It represents other compile-code simulation techniques that use estimations based on behavioral-level (or source-level) descriptions. Since no optimization impact can be observed and branch/interlock information is not relevant, the errors on the estimations are up to 380%. To show the importance of all three timing delay components in the estimation, estimations without one of the three timing delay components are shown for comparison. Instruction selection is the major part of the estimations. Without instruction selection (*w/o instr.*), the errors on the estimations are up to 97%. Without branch penalty (*w/o branch*), the errors on the estimations are up to 33%. Without pipeline interlock (*w/o interlock*), the errors on the estimations are up to 14%. When considering all three components together (instructions, branches and interlocks), the estimations are always within 1% from the cycle-accurate results obtained from ISS.

Memory accesses from each program are transactionally correct. Because memory accesses are exactly represented in the software simulation models and the reverse address mapping

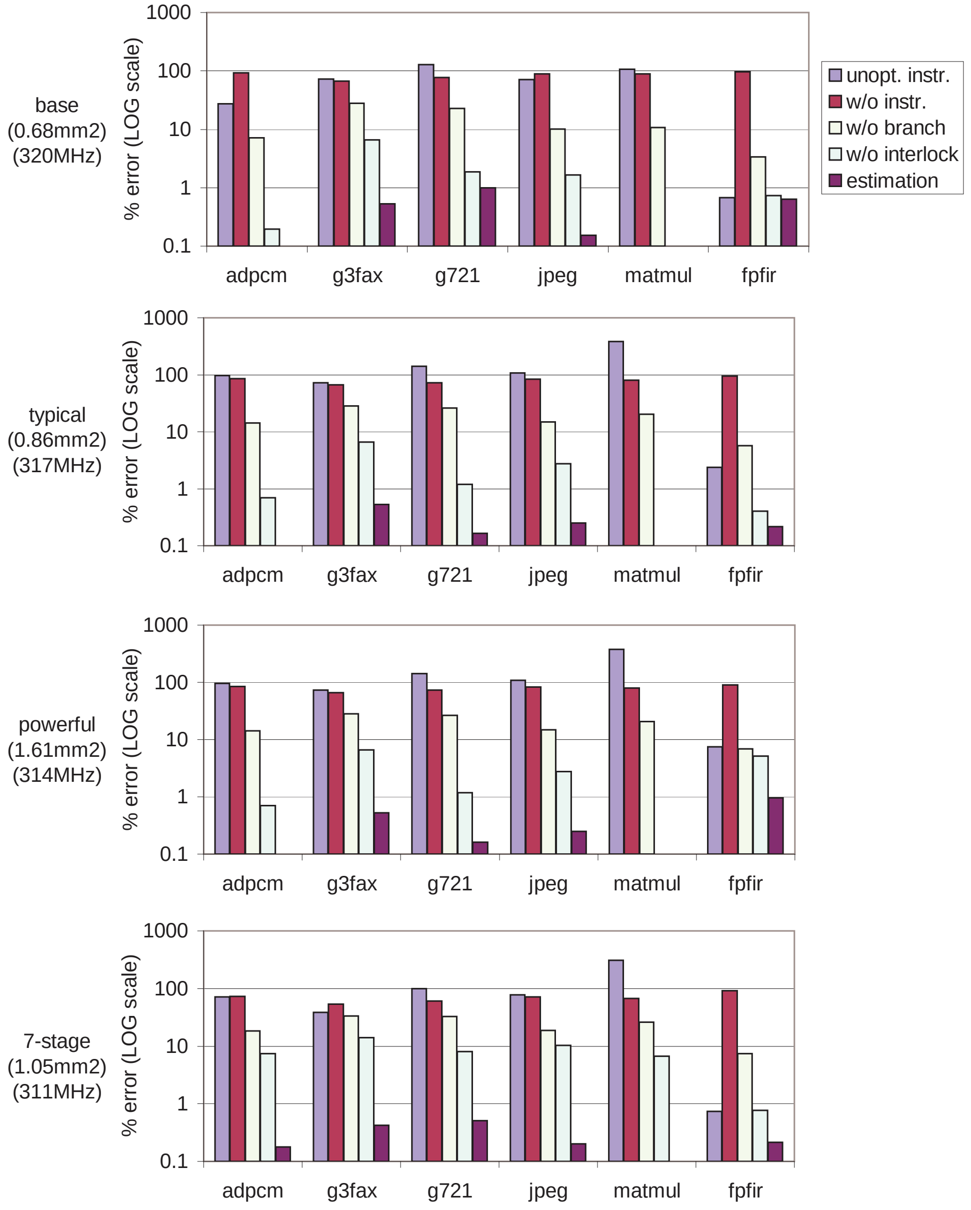


Figure 5.11: Estimation accuracy of the generated software simulation models on a diverge set of benchmarks and several Xtensa LX2 processors with different configurations.

maps the accesses correctly into their target memory addresses with a provided memory map, memory accesses from each program have no error.

For MPSoC simulation, the estimation from the multiprocessor simulation using the software simulation models in SystemC is compared to the cycle-accurate results from Tensilica's XTSC simulation. The experiments on the MPEG-2 Decoder design measure the execution times to produce every other raw video frames. Two 3-processor implementations are used to compare the estimations and the cycle-accurate results. *Implementation A* maps the programs together based on their connectivities. The programs are more likely to map to the same processors if the programs heavily interact. *Implementation B* maps the programs to the processors based on the workloads. The processors are assigned with programs with similar total workloads.

Frames	Cycle-accurate	Estimation	Error
2	152.79 ms	160.14 ms	4.81%
4	200.40 ms	208.72 ms	4.15%
6	306.77 ms	313.34 ms	2.14%
8	389.06 ms	400.51 ms	2.94%
10	434.06 ms	444.59 ms	2.43%
12	551.34 ms	570.38 ms	3.45%
14	630.18 ms	651.56 ms	3.39%
16	676.50 ms	697.85 ms	3.16%
18	785.80 ms	810.05 ms	3.09%
20	867.24 ms	894.82 ms	3.18%
22	913.72 ms	939.83 ms	2.86%

Table 5.4: Estimation accuracy in MPSoC simulation of the MPEG-2 Decoder design in implementation A, which is a three-processor implementation with mapping to reduce the communication between processors.

As shown in Table 5.4 and 5.5, the estimated execution times from the compile-code simulation framework accurately estimate the performance on both implementations at various points of the simulation. Even when the times between frames vary due to the difference frame types in an MPEG-2 video stream (I-, P- and B-frames), the estimation is accurate at any point of the simulation in both implementations. For every comparison point, the relative error between the estimation and the cycle-accurate result from XTSC is between -0.8% and 4.8%. The difference mainly comes from the inaccuracy in the modeling of O/S, buses, caches and memories, which are modeled manually and the estimation on the models is less accurate than the software simulation models.

Frames	Cycle-accurate	Estimation	Error
2	183.08 ms	185.52 ms	1.33%
4	245.25 ms	247.72 ms	1.01%
6	390.77 ms	390.11 ms	-0.17%
8	500.60 ms	500.42 ms	-0.04%
10	558.95 ms	556.33 ms	-0.47%
12	710.47 ms	707.48 ms	-0.42%
14	815.24 ms	810.24 ms	-0.61%
16	875.78 ms	869.47 ms	-0.72%
18	1025.51 ms	1017.19 ms	-0.81%
20	1133.78 ms	1125.79 ms	-0.70%
22	1191.72 ms	1183.63 ms	-0.68%

Table 5.5: Estimation accuracy in MPSoC simulation of the MPEG-2 Decoder design in implementation B, which is a three-processor implementation with mapping to balance the processor workloads.

Similar to single-processor systems, for a program to run on a different target processor or use a different set of compiler optimizations, a new simulation model with different estimation needs to be generated. Software simulation model generation does not introduce any overheads because each software simulation model takes less than 0.1 second to generate even at “-O3” optimization level. Such time is negligible comparing to the length of time involved in simulating an MPSoC implementation.

5.6 Improve Simulation Speed

	un-timed	timed	overhead
9uP	10.50s	217.67s	20.7X
3uP w/RR	10.50s	172.82s	16.5X
3uP w/Pr	10.50s	176.50s	16.8X
1uP w/RR	10.50s	149.11s	14.2X

Table 5.6: Simulation overheads in timed simulation when simulating different implementation of the MPEG-2 Decoder design. Different number of processors and scheduling policies are used. RR is round-robin non-preemptive scheduling. Pr is priority-based non-preemptive scheduling. Simulating timing delays in the simulation models lengthens the simulation time by 14X–21X comparing to the same simulation without simulating the timing delays.

Although the compile-code simulation framework provides very efficient MPSoC simulation compared to cycle-accurate and *Register-Transfer Level* (RTL) simulation, a timed simulation in SystemC – a regular discrete event simulator – is more than 14 times slower than an un-timed simulation in the same simulator. Table 5.6 shows the simulation overheads for timed simulation to simulate the MPEG-2 Decoder design in different MPSoC implementations. Implementations with different number of processors (1, 3 and 9 processors) and different scheduling policies (Round-robin and priority-based) are used. Similar result has been observed on [42]. Slower simulation tremendously reduces the number of design points that the designers can explore in the design space in a limited amount of time. Unfortunately, timed simulation is necessary to evaluate implementations and verify them against performance constraints. By simply simulating timing delays in the software simulation models and the components, the MPEG-2 Decoder design simulates more than 14 times slower. Based on the profiling result of the simulation, more than 95% of the simulation time is associated with activities related to switching between the component simulation (such as the simulation kernel and simulation clock manipulations). Simulation speed can be improved if such simulation overheads are reduced.

5.6.1 Timed Simulation Overhead

The compile-code simulation framework uses the SystemC simulator – a discrete event simulator – to simulate MPSoC implementations. Discrete event simulation is an event-based simulation. Events are triggered in an instant in the simulation clock and the corresponding components are simulated until timing delays are requested. *OSCI SystemC* [109], which is the base of almost all SystemC simulator derivatives, uses discrete event simulation. In the MPSoC simulation, a timing delay is requested every time before a communication channel, which is used for interactions between components, is accessed. A timing delay is requested using the SystemC *wait()*. For every *wait()*, the SystemC simulation kernel switches out the current simulating component, calculates the next invocation of the component based on the arguments passed by the *wait()*, and switches to the next earliest ready component.

Discrete event simulation provides a strictly chronological ordering in simulating the components and communication channels. A discrete event simulator keeps a global simulation clock which indicates the current time inside a simulation. The simulation clock of a channel access is always the same as the global simulation clock at the time the channel access function is called. Since the global simulation clock only goes forward (increases monotonically), components and communication channels are simulated chronologically. For a channel access, there is an implicit assumption that all other accesses that are called before the current simulation clock have been called. When a channel is accessed, the channel should be able to response, both behaviorally and timingly, at the time of the access (a channel can hold a channel access if it is a blocking call, which is also considered as a response). However, a high simulation overhead is resulted when the simulation requires very frequent switching to maintain such strictly chronological order.

A producer-consumer example is used to illustrate the switching overhead associated with timed simulation. The example consists of a dedicated FIFO channel and two components: a producer and a consumer. The producer generates an item every 10*ns* and writes it to the FIFO channel; the consumer reads an item from the FIFO channel every 15*ns*. The blocking interface is used such that the producer blocks when it writes to a full FIFO channel and the consumer blocks when it reads from an empty FIFO channel. The components are unblocked when the FIFO channel becomes not

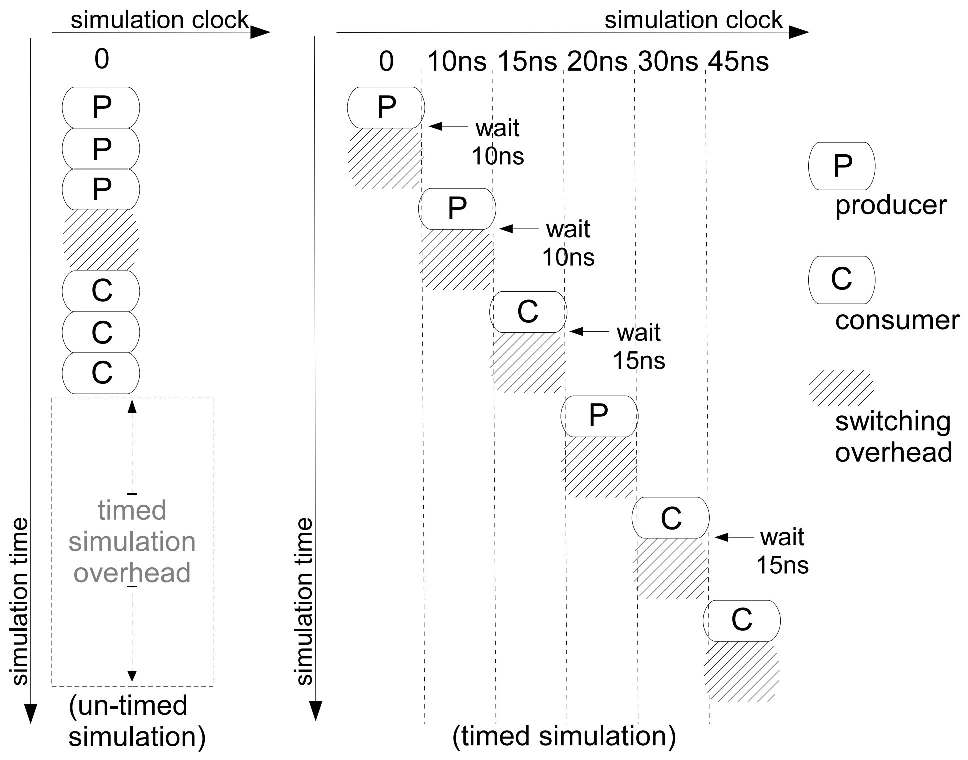


Figure 5.12: Timed simulation overhead in a producer-consumer example. On the left, the producer-consumer example is simulated without timing delays. The producer can keep writing to the FIFO channel without switching out. The producer only stops when the FIFO channel is full. A switching overhead is imposed before the consumer starts simulating. On the right, the same example is simulated with timing delays. Before the producer can write its first data to the FIFO channel, it has to wait for the global simulation clock to reach 10ns – which is the length of time the producer takes to generate a data. So the producer switches out, which imposes a switching overhead, and waits for the global simulation clock before switching back in. Same switching happens for every FIFO read and write, which creates significant overheads in timed simulation.

full or not empty, respectively. In an un-timed simulation, the components do not switch for every FIFO read or write, and the simulation takes only 0.24 second to finish. In a timed simulation, FIFO reads and writes are simulated in strictly chronological order. Therefore, the producer and the consumer are alternatively scheduled in the discrete event simulator as shown in Figure 5.12. Since there is a switching overhead associated with each switch, such ordering imposes significant overhead to simulate. As a result, the timed simulation takes 4.10 seconds to complete, which is 17 times slower than the un-timed simulation.

5.6.2 Partial Order Channel Modeling

Majority of the simulation overheads associated with switching between components can be avoided by simulating the system in a non-chronological order. By decoupling the simulation clocks of the components from the global simulation clock in the simulation kernel, some delay requests do not require advances in the global simulation clock. Majority of the simulation overheads can be avoided by reducing the number of switches between the components. With individual simulation clocks in the components, some components may simulate ahead of the global simulation clock. As a result, components in the system are simulated in a non-chronological order instead of strictly chronological order. The term *partial order method* is used to describe this simulation technique.

To avoid the simulation overheads associated with switching, the simulating components are kept active as long as possible. Each switch has an associating simulation overhead. By reducing the number of switches during simulation, simulation speed is improved. Therefore, the partial order method follows the following guidelines to maximize the simulation speed.

- When a component is simulating, simulate it as long as possible until the behavior or the timing of the component is no longer known based on information available in the simulator;
- When a component has to wait, wait as long as possible to gather more information from other components for later behavior and timing computations.

Simulating channel accesses in chronological order is a sufficient condition for correct behavior and timing simulation, but such order is not always necessary. One access has to be simulated before the other only if the earlier access affects the behavior or the timing of the later access. Chronological order ensures that such dependency is satisfied because no access can be affected by an access that happens later in the simulation clock. If events at the same time affect each others, it is a race condition and the behavior and timing are not guaranteed to be simulated correctly in this situation.

Simulation Principle

To prevent excessively switches between components, the partial order method allows channel access simulation not to follow the chronological order. When a component tries to access a channel after a timing delay, it accesses the channel immediately with a *timestamp* of the access. If the result of the access, in both behavior and timing, can be determined in the simulation, the access returns immediately with the result along with the timestamp in which the access should return. Therefore the delay request to advance the global simulation clock is not necessary and no simulation overhead due to switching is involved.

To allow components to simulate separately from the global simulation clock, an individual (local) simulation clock is added in each component to keep its timestamp. The local simulation clock advances when the component requests a timing delay without switching out. In a communication channel access, the component calls the channel interface and uses the local simulation clock as the timestamp of the access. If the access successes, the returned timestamp will be used as the local simulation clock and the simulation of the component continues. The channel access may block the component if the result of the access cannot be determined at that point of simulation. Such blocking access can either wait for an event that has yet generated or wait for advances of the local simulation clocks in other components. The ability to calculate the access result depends on the semantic of the communication channel. Some examples of communication channel modeling are shown in the next subsection.

Communication Channel Modeling

To allow partial order method in the simulation, the communication channel interface must take the local simulation clock (timestamp) of the calling component as an input parameter. For a blocking interface, which the access can be returned at a different time in the simulation clock, the output of the interface also includes a timestamp to indicate the simulation clock when the access returns. Determining the access result relies on the modeling of the communication channel. The channel is responsible to determine whether the result of an access at the simulation clock given by the timestamp can be computed. If the result can be computed, the behavior and the the timing are immediately returned and no timing advance in the global simulation clock is necessary. Otherwise, the calling component preempts and waits until both the behavior and the timing of the access can be computed. Although the partial order method requires additional modeling on the channel, it is a beneficial feature instead of a burden because system-level modeling is meant to capture very high-level information of the system.

Algorithm 3: Blocking FIFO read for partial order simulation

```
    Input: $process\_ts$
1 begin
2     if $FIFO$ $is$ $empty$ then
3         wait for a FIFO write
4     read $data$ and $data\_ts$ from the FIFO
5     return $data$ $and$ $max\{process\_ts, data\_ts\}$
6 end
```

Algorithm 4: Non-blocking FIFO read for partial order simulation

```
    Input: $process\_ts$
1 begin
2     if $FIFO$ $is$ $empty$ $and$ $ts(writer) < process\_ts$ then
3         wait until $process\_ts$
4     if $FIFO$ $is$ $empty$ $or$ $data\_ts > process\_ts$ then
5         return $read$ $fail$
6     read $data$ and $data\_ts$ from the FIFO
7     return $data$
8 end
```

For a FIFO channel, the partial order simulation method allows an access to be complete if enough information is available in the FIFO channel. Because only one component can read from the FIFO channel and only one can write to the FIFO channel, the result of a *read* can be determined immediately if a data is available in the FIFO channel (and verse vise for a *write*). The algorithms for both blocking and non-blocking reads are shown in Algorithm 3 and 4. To accommodate timestamp calculations, each FIFO data is annotated with a timestamp stating the simulation clock when the data is written, and each FIFO space is also annotated with a timestamp stating the clock when the space is available. Initially, all spaces in the FIFO channel are annotated with initial time *zero*.

Algorithm 5: Blocking mutex lock for partial order simulation

```
  Input: process_ts
1 while true do
2     if mutex not locked and (ts(lockers) >= process_ts or
      unlock_ts >= process_ts) then
3         take mutex
4         return max(unlock_ts, process_ts)
5     else if mutex not locked then
6         wait until process_ts
7     else
8         wait for mutex unlock
9 end
```

Besides FIFO channels, the partial order simulation method also works for other communication channels such as a mutex or a shared variable. For a mutex, a component can *lock* a mutex if all other components that are capable to lock the mutex have later timestamps than the accessing component (Algorithm 5 and 6). *Unlocking* a mutex always succeeds without switching since the component holding the mutex can unlock the mutex anytime independent from other components (Algorithm 7). In the partial order mutex model, the mutex is either locked or unlocked. If it is in the unlocked state, there is an annotated timestamp to state the simulation clock when the mutex is unlocked (initially set to time *zero*).

For a shared variable, the channel keeps a record on the values of the variable over the simulation clock. A *write* to the shared variable always succeeds and adds a (value, clock) tuple to

Algorithm 6: Non-blocking mutex lock for partial order simulation

```
Input: $process\_ts$
1 while true do
2     if mutex not locked and $ts(lockers) >= process\_ts$ and $unlock\_ts <= process\_ts$
      then
3         take mutex
4         return $process\_ts$
5     else if (mutex not locked and $unlock\_ts > process\_ts$) or (mutex locked and
      $ts(lockers) > process\_ts$) then
6         return lock fail
7     else
8         wait until $process\_ts$
9 end
```

Algorithm 7: Mutex unlock for partial order simulation

```
Input: $process\_ts$
1 begin
2     unlock the mutex
3     set $unlock\_ts = process\_ts$
4     notify mutex unlock event
5     return $process\_ts$
6 end
```

Algorithm 8: Shared variable write for partial order simulation

```
Input: $data$, $process\_ts$
1 begin
2     add an entry $(data, process\_ts)$ to the ordered array
3     return $process\_ts$
4 end
```

Algorithm 9: Shared variable read for partial order simulation

```
    Input: $process\_ts$
1 begin
2     if $ts(writers) < process\_ts$ then
3         wait until $process\_ts$
4     read the value with $max(ts < process\_ts)$ in array
5     return value (and process_ts)
6 end
```

the record (Algorithm 8). A *read* succeeds only when all the writers have later timestamps than the reader. In the partial order shared variable model, the record of variable values over the simulation clock is kept in an ordered list (Algorithm 9). The list is ordered w.r.t. the clock. Writing to the variable adds an entry to the list. When the reader reads the variable and all writers have later timestamps, the reader looks for the value that was last written based on the reader's timestamp.

5.6.3 Producer-Consumer Example

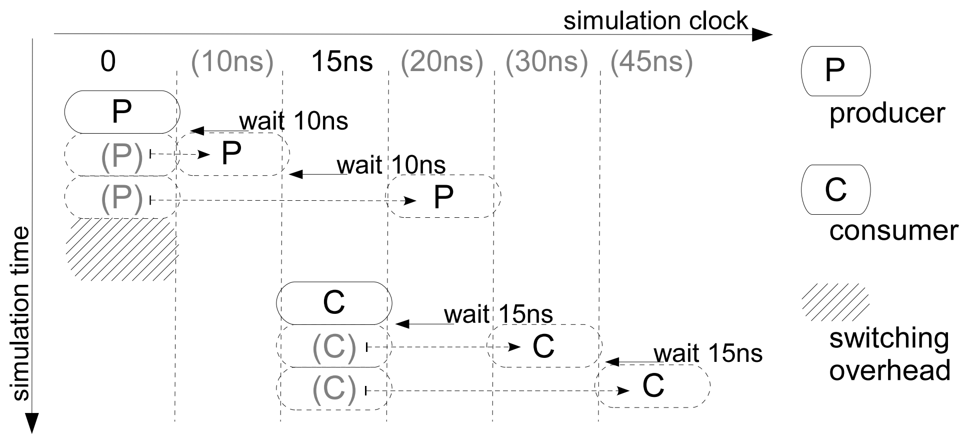


Figure 5.13: Producer-consume example in partial order simulation method, which has less switches comparing to the chronological timed simulation. At the same time the simulation gives the same simulation results.

The producer-consumer example is used again here to illustrate the saving of switching overheads in timed simulation by using partial order method. The schedule of the simulation is shown in Figure 5.13. When a timing delay is requested in the producer or the consumer before accessing the FIFO channel, the request is not transform into a *wait()* that causes a switch. Instead, the request increments the local simulation clock of the producer or the consumer and the clock is subsequently used as the timestamp for the FIFO access. As the result, as shown in the figure, the producer and the consumer do not need to switch between every read and write, hence reducing the simulation overheads. The simulation is in partial order, which is shown in the example as the third write from the producer is simulated earlier than the first read from the consumer, although the write happens later in the simulation clock. Partial order method improves the simulation speed of the producer-consumer example by avoiding excessive switches between the producer and the consumer. As a result, the example simulates in 0.29 second, comparing to 0.24 second for an un-timed simulation and 4.10 seconds for a chronological timed simulation.

5.6.4 Backward Compatibility

Communication channels modeled with partial order method can also be simulated alongside with unmodified communication channels that are not modeled with partial ordered method. To make the unmodified channels simulate correctly, the assumptions of discrete event simulation without partial order method have to be reenforced: accesses to a channel have to be made in strictly chronological order and the simulation clocks of the accesses can be obtained from the global simulation clock. These assumptions can be simply enforced by calling *wait()* from the components before the accesses to force the components to wait. This updates the local simulation clocks of the components to match the global simulation clock, and hence the simulation clocks of the accesses can be obtained from the global simulation clock when the channel accesses are called. Since global simulation clock only increases monotonically, the accesses to the channel are made in chronological order. However, in a mixed simulation with modified channels and unmodified channels, speedup for partial order method only applies to those accessing the modified channels, and the speedup of the simulation is not maximized.

5.7 Simulation Performance

5.7.1 Extending Simulation Model

The partial order simulation method is implemented as user-code in SystemC. The SystemC classes are extended (derived) to integrate partial order method into the simulation. A timestamp is added to the derived class of *sc_module* to keep the local simulation clock of a component. Several methods are also added to the class to allow easy accesses to class members and advances of the local simulation clocks. Channel interfaces are also extended to add the channel access methods with timestamps.

Since the implementation of the time data-structure *sc_time* in SystemC 2.1 is too inefficient for the manipulations needed in partial order method, a new timestamp data-structure is designed for partial order channel modeling. The new timestamp data-structure can be easily converted to and from *sc_time*.

5.7.2 Simulation Speed

The MPSoC simulation speed is evaluated by comparing the time taken to decode half a second of an MPEG-2 stream in the MPEG-2 Decoder design at different simulators. Such simulation length is required to evaluate an implementation because each MPEG-2 *group of pictures* (GOP), the minimum repeating sequence in an MPEG-2 stream, contains half a second of video. The simulation times of it running on different simulators are shown in Figure 5.14.

Using ISS, the simulation is very slow and takes more than one day to decode half a second of the MPEG-2 stream, which prevents the designers from using it for design space exploration that requires a number of simulation. Compile-code simulation, even when the simulation models are running chronologically in the discrete event simulator, is more than 100X faster than ISS because the simulation models are running natively on the host development machine. However, the simulation speed is not maximized due to the timed simulation overheads. Majority of the simulation time is used for controlling the simulation clock and switching between simulation models to keep them running in chronological order. When non-chronological order simulation using partial order method is used, significant number of the switches are avoided and the simulation is additional 10X faster, which provides the overall speedup of 1000X over ISS. The simulation that

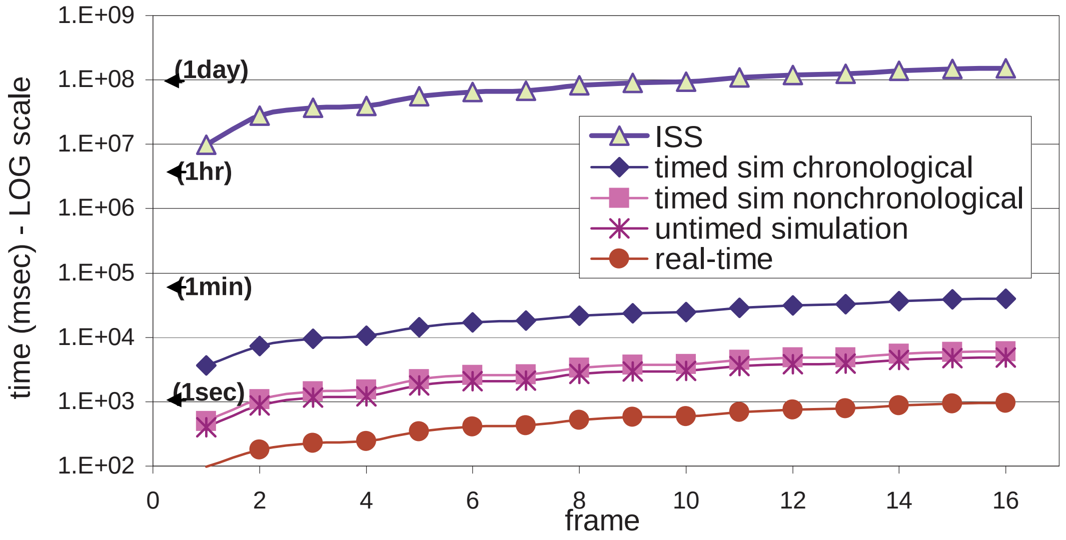


Figure 5.14: Time to simulate an MPEG-2 Decoderdesign using **1.** cycle-accurate simulation with XTSC (*ISS*), **2.** the compile-code simulation framework in chronological simulation – no partial order method (*timed sim chronological*), **3.** the compile-code simulation framework with partial order method (*timed sim nonchronological*), **4.** untimed simulation (*untimed simulation*) and **5.** speed of the target implementation (*real-time*) for comparison.

takes ISS more than a day now only takes seconds to finish, and at the same time provides accurate performance estimations.

For comparison, the speed of the compile-code simulation framework is also compared to untimed simulation and the actual implementation. The timed simulation is only a fraction slower than untimed simulation. (Note that untimed simulation is only useful for functional simulation and does not simulate performance. Additional simulation time is required in order to obtain execution time information.) And the timed simulation is only 10X slower than the actual implementation, which is a three processor system running at 300MHz. Such difference between the simulation time and the real-time can obviously be reduced by using a more powerful development machine. It is here to show that this compile-code simulation framework can be used for very realistic simulation with real design inputs and exploring different implementations efficiently.

With this compile-code simulation framework, simulation of realistic industrial MPSoC designs now requires only seconds of simulation time, which is sufficient for effective design space exploration. It introduces a new direction for efficient system-level simulation and allows designers to explore a much larger design space for better implementations.

5.8 Energy Model

Although this chapter mostly focuses on simulation accuracy and performance, energy models can also be integrated into the simulation framework. The energy model used here is commonly available in instruction-level simulation (such as in ISS). Generally, processors can switch among different modes. The three main modes are running, idle, and power-off modes. When a processor is running, it consumes both dynamic and static power. The energy consumed in the running mode are instrumented into the software simulation models based on the instructions in the basic blocks, which is similar to the way timing delays are instrumented in the models. The processor consumes only static power when it is idle and negligible amount of power when it is power-off. Energy for memory accesses and on-chip interconnect networks are also included in the overall energy consumption.

To compute both static and dynamic power consumption of a processor, the energy model in [100] is used. The same energy model has been used in [26, 75, 170]. Specifically, the dynamic power is computed by

$$P_{dynamic} = C_{eff} V_{dd}^2 f \tag{5.2}$$

where C_{eff} is the effective switching capacitance of the processor for a given instruction, V_{dd} is the supply voltage, and f is the optimizing frequency of the processor. Dynamic power reduces rapidly with lower supply voltage and lower frequency. It becomes negligible when the processor is clock-gated in idle mode. Static power mainly comes from subthreshold leakage and reverse bias junction current. The static power is computed by

$$P_{static} = L_g (V_{dd} I_{sub} + |V_{bs}| I_j) \tag{5.3}$$

$$V_{th} = V_{th1} - K_1 V_{dd} - K_2 V_{bs} \tag{5.4}$$

$$I_{sub} = K_3 e^{K_4 V_{dd}} e^{K_5 V_{bs}} \tag{5.5}$$

where V_{bs}, V_{th1}, I_j, K_1, K_2, K_3, K_4, K_5 are constants for a given technology and L_g is the size of the processor [100]. Static power reduces sublinearly with the supply voltage and can be further reduced in power-off using DPM [16], which is used in the implementations. The cycle time of the processor is computed by

$$T_{cycle} = \frac{L_d K_6}{(V_{dd} - V_{th})^{\alpha}} \tag{5.6}$$

where K_6 is a constant and L_d is a processor-specific parameter [100]. Constants for the 70mm technology [75] are used in the experiments.

Chapter 6

Criticality-based KPN Implementation

An MPSoC design contains multiple interacting programs. Determining the specific objects for design optimizations is difficult because the programs run in parallel instead of in series. Design optimizations based on traditional single-processor software profiling [10, 32] are not reliable for MPSoC. Traditional software profiling for single processor assumes the programs in a design run sequentially, hence reducing the execution time on any parts of the programs reduces the overall execution time of the design. Traditional software profiling weights each statement execution equally and tries to find the statements that execute most frequently. However, this assumption does not apply to MPSoC designs because programs run in parallel. Some statement executions are more important than the others for the overall execution time. Therefore, traditional software profiling results from individual processors do not reveal the statements that are critical. A new MPSoC-specific profiling technique to accurately determine the important statements is needed.

The goal of such MPSoC-specific profiling technique is to find the statements that are critical for the overall execution time in an MPSoC design. For a design written as multiple programs, the design is composed of interacting programs running on separate processors. Programs block and unblock each others during execution, which allow necessary communication and synchronization between the programs. As a result, the programs have dependencies that keep them from running continuously. Programs routinely wait for each other for input and output data to keep the integrity and consistency of the design. Such dependencies cannot be ignored because they determine the performance of the implementations. In the next section, a symbolic model is used to analyze an execution of an MPSoC implementation and the *MP-Critical Path* (LDP) is defined as the execution path among the programs that is critical for performance. The MP-critical path is shown to contain very different information from the traditional software profiling. An iterative algorithm is proposed to find this path dynamically during simulation with reasonable simulation time overhead.

Three techniques are then proposed that make use of the MP-critical path information to efficiently optimize MPSoC implementations. These techniques are classified into three different design levels, which demonstrate the information in the MP-critical path is useful throughout design decision procedures. The efficiency of design optimizations is demonstrated in the MPEG-2 Decoder design. At the instruction level, custom instructions are used to speed up the most frequently executed statements in the path; at the segment level, instruction reordering techniques take the information of the basic block segments in the path and reduce the length of the path; at the program level, power consumption is reduced while satisfying the performance constraints using voltage islands by determining the importance of each program in the path.

6.1 MP-Critical Path for Performance Analysis

The goal of the performance analysis us to assist the designers in design optimizations by automatically narrowing the correct hotspots into a small number of statements. Hotspot information is very important for design optimizations. The famous 80–20 rule states that approximately 80% of execution time is based on 20% of the statements. Hence design optimizations often focus on the small number of statements that are important for the performance. For that reason, finding these statements becomes a crucial first step in design optimizations.

For example, if an MPSoC simulation shows that a performance constraint is violated, automatically generated correct hotspot information would help the designers to resolve the constraint violation with design optimizations. Such hotspot information for MPSoC allows designers to focus on small number of statements to optimize their designs. In single-processor systems, such hotspot information can be found by traditional software profiling in a simulation, which determines which statements are executed most frequently and take most time. However, MPSoC is very different from single-processor systems such that information from traditional software profiling is inaccurate for design optimizations.

6.1.1 Assumption

In this analysis, each program in the design is first assumed to run on one processor in an MPSoC implementation. With multiple processors in the system, each processor has a well-defined role and is dedicated to one job. Each program runs on one processor only. This assumption is lift once a specific RTOS scheduling policy is determined in the experiments.

The *program steps* in the MPSoC design have strictly precedence relationships. The programs run cooperatively instead of independently. If the execution of a program is considered as a sequence of steps, certain steps have dependencies to other steps in other programs. These steps have to wait for their parent steps in other programs to finish before they can execute. Such assumption is reasonable for MPSoC designs and applies to many models of computation, including Kahn Process Network. The symbolic model for the relationships and how the relationships apply to Kahn Process Network are shown in the later subsections.

6.1.2 MP-Critical Path

To reveal the correct hotspot information in the programs for design optimizations, determining the execution path that is critical is needed. A critical path is with respect to a processing step, where the time the processing step is executed determines the performance of the implementation. For a latency-constrained application, the processing step is the step that generates the output at the end. For a throughput-constrained application, there is a sequence of steps that generate outputs and the processing step that is interesting is the one generating output late and violating the performance constraints.

An MP-critical path with respect to a processing step is defined as the sequence of executed statements among all programs that contribute to the earliest starting time of the step. As each program runs sequentially and some steps wait until their parent steps in other programs to finish under the strictly precedence relationships, there is an execution path among the programs that leads to the earliest starting time of the processing step. Reducing the execution time on any part of the path allows an earlier starting time.

An example of an MP-critical path with respect to a processing step is shown in Figure 6.1. *p1* to *p4* are four programs running on an MPSoC system. In the figure, *block* means a processing step in a program cannot execute right away because of the strictly precedence relationships.

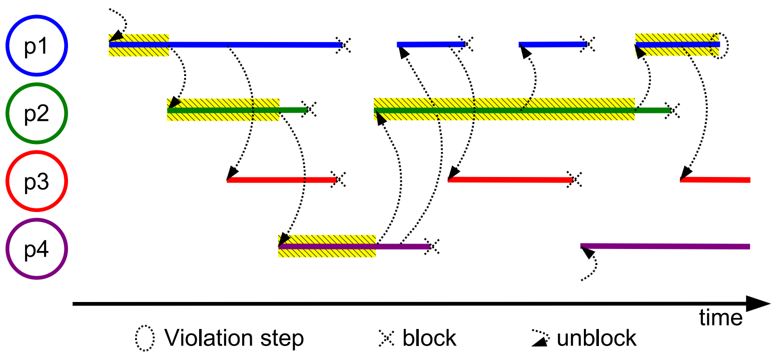


Figure 6.1: An illustration of an MP-critical path on an execution trace. Horizontal lines represent active processing in the processors, and crosses and arrows represent strictly precedence relationships. The MP-critical path is shown in the highlighted path.

When the parent steps are later finished, the step is *unblocked*. The figure shows the blocking and unblocking between the programs over the execution time. In the example, the goal is to make the last processing step of *p1* execute earlier. The MP-critical path of the processing step, as highlighted in the figure, is the execution path among the programs that leads to the earliest starting time of the processing step. According to the path, program *p2* is responsible for majority of the delay in the MP-critical path. Therefore, design optimizations should focus on the statements on *p2*. On the other hand, optimizing *p3* cannot improve the MP-critical path. Definition of the MP-critical path and an efficient algorithm to find such path is presented in the following subsections.

6.1.3 Blocking / Unblocking Relationship

The computation model of KPN complies with the software model with strictly precedence relationships. Each process in KPN is a sequential program that only communicates to other processes using FIFO channel. A FIFO f is an ordered queue where data is produced and consumed in the same order. A FIFO channel contains a sequence of data. The i-th data in the FIFO f is denoted as f^i. $Prod(f,i)$ and $Cons(f,i)$ denote production (writing) and consumption (reading) of the data f^i.

The semantics of a FIFO channel restrict the starting time of production and consumption of the data. Specifically, the production of the data f^{i+1} must happen after the production of the data f^i (i.e. $\tau_{Prod(f,i)} \leq \tau_{Prod(f,i+1)}$), and the consumption of the data has to be in the same order (i.e. $\tau_{Cons(f,i)} \leq \tau_{Cons(f,i+1)}$). In addition, the data must be produced before it can be consumed (i.e. $\tau_{Prod(f,i)} \leq \tau_{Cons(f,i)}$). If a program tries to consume the data from the FIFO channel before the data is produced, the program has to wait until the data is available. This is commonly referred to as "blocking reads". For a FIFO f with size N, the production of the data f^{i+N} cannot occur before the consumption of the data f^i. As a result $\tau_{Cons(f,i)} \leq \tau_{Prod(f,i+N)}$. This is commonly referred to as "blocking writes". These restrictions can be represented in the MPSoC execution model as strictly precedence relationships.

	blocking	unblocking
shared variable	spinwait	write
message-passing	blocking read/write	write/read
handshaking	synchronize	synchronize

Table 6.1: Blocking Mechanism available on different MPSoC models of computation

Blocking mechanism is also very common in other multiprocessor models of computation. As shown in Table 6.1, blocking and unblocking are commonly used in MPSoC to synchronize multiple asynchronously executing programs. For example, in *Symmetric Multi-Processing* (SMP), blocking can be achieved by a spinwait on a shared variable, and the spinwait is unblocked when a proper value is written to the variable. Although such blocking and unblocking mechanisms are not as explicit as in KPN, they can still be modeled for the MP-critical path analysis.

6.1.4 MPSoC Execution Model

In this subsection, the symbolic model to analyze *an execution* of an MPSoC design is described. Each program in an MPSoC design is modeled as a step transition, and strictly precedence relationships are used to model interactions between the programs.

Each program in the MPSoC design is modeled as a step transition. Each non-repeating step σ represents a processing step in a program. $\sigma_i^P \in S$ is the step i in program P and σ_0^P is the beginning of the program. S is the set of all steps in all programs. $\tau_i^P \in R^+$ is the starting time of the step σ_i^P, and $\delta_i^P \in R^+$ is the execution time (timing delay) of the step σ_i^P.

A step represents execution of a set of statements in the program and its execution history (hence non-repeating). A statement that can be blocked by other programs always starts a step, and a statement that can unblock other programs always ends a step. Therefore, blocking and unblocking always occur between steps.

Property 4 *Step sequence in a program:*

$$\textit{For all steps } \sigma_i^P, i \in [0,\infty) \textit{ in a program P,}$$
$$\textit{the step sequence is } \sigma_0^P \sigma_1^P \sigma_2^P \ldots$$

Property 5 *Starting time restriction on consecutive step:*

$$\textit{For all steps } \sigma_i^P, i \in [1,\infty) \textit{ in program P,}$$
$$\tau_{i-1}^P + \delta_{i-1}^P \leq \tau_i^P$$

Property 4 specifies that each program executes sequentially. Each step σ_i^P is an execution of a set of statements and δ_i^P is the execution time of the statements. A program runs sequentially and subsequent step cannot start before the previous step finishes. Therefore a program is executed following the step sequence and the earliest starting time of each step is restricted by Property 5.

Property 6 *Starting time restriction on precedence relation:*

$$\textit{Step } \sigma_j^Q \textit{ strictly precedes step } \sigma_i^P$$
$$\rightarrow \quad \tau_j^Q + \delta_j^Q \leq \tau_i^P$$

Interacting programs have strictly precedence relationships. If a dependency is implied such that the step i in program P cannot start before the step j in program Q finishes (such as those described in Section 6.1.3), the dependency can be specified as a starting time restriction. Property 6 shows the starting time restriction of a strictly precedence relation. The relation limits the earliest starting time of the steps in addition to the restriction shown in Property 5.

Environment events σ_i^E are defined as a sequence of steps from the environments (outside an open system). σ_0^E is an event representing the beginning of the execution. $\sigma_i^E, i > 0$ are steps that read and write from the buffered inputs and outputs, which interface the open system and the environment. Since the environment is independent of the system, all starting time and execution time of the environment events (τ_i^E and δ_i^E) are pre-defined based on the performance constraints. For example in the MPEG-2 Decoder design, the environment inserts MPEG-2 streams into the input FIFO channels based on the rate of the input methods and expects to read one frame of raw video from the output FIFO channels every $\frac{1}{30}th$ of a second.

6.1.5 Earliest Starting Time

With each program runs on a processor, program steps start *as soon as possible* after all restrictions in Property 5 and 6 are satisfied. For a step σ_i^P that depends on steps σ_j^Q and σ_k^R, its starting time is the latest among the finish time for its previous step σ_{i-1}^P in the same program and all the parent steps.

$$\tau_i^P = max(\tau_{i-1}^P + \delta_{i-1}^P, \tau_j^Q + \delta_j^Q, \tau_k^R + \delta_k^R)$$

Step σ_i^P is called *blocked* when it cannot start immediately after its previous step σ_{i-1}^P. i.e. $\tau_{i-1}^P + \delta_{i-1}^P \neq \tau_i^P$. The step is *unblocked* by one of the parent steps σ_j^Q when $\tau_j^Q + \delta_j^Q = \tau_i^P$.

$Pre : S \rightarrow S$ defines the *immediate prior step* relationships. If a step is not blocked, $Pre(\sigma_i^P) = \sigma_{i-1}^P$ because the step immediately follows its previous step in the same program. If it is blocked, $Pre(\sigma_i^P) = \sigma_j^Q$ where σ_j^Q is the step that unblocks σ_i^P. No one step is assumed to be unblocked by two steps at the same time. Therefore, the immediate prior step of the step σ_i^P is defined as follows:

$$\forall \sigma_i^P, i \in [1, \infty), Pre(\sigma_i^P) = \{\sigma_{i'}^{P'} | \sigma_{i'}^{P'} \text{ unblocks } \sigma_i^P\}$$

Immediate prior step of a step σ_i^P can come from the environments if σ_i^P is blocked by reading from an input buffer when the buffer is empty or by writing to an output buffer when the buffer is full. In such case, $Pre(\sigma_i^P) = \{\sigma_j^E\}$ where σ_j^E is the environment event. The immediate prior step of any environment event σ^E is $\emptyset$.

6.1.6 MP-Critical Path

The MP-critical path of a processing step can be defined using the symbolic model. The MP-critical path is the sequence of statements that contribute to the earliest starting time of the processing step. The path can also be represented by a set of steps in S. Since the earliest starting time of step σ_i^P depends on the step $Pre(\sigma_i^P)$, The MP-critical path of the step σ_i^P ($LDP(\sigma_i^P)$) can be defined as follows.

$$LDP(\sigma_i^P) = \{\sigma_{i'}^{P'} | (\sigma_{i'}^{P'} = \sigma_i^P) \\ \vee (\sigma_{i'}^{P'} = Pre(\sigma \in LDP(\sigma_i^P)))\}$$

The MP-critical path of the processing step is used to determine hotspot information for design optimizations. The processing step σ_i^P is normally a read from an input FIFO channel or a write to an output FIFO channel, which the designers would want it to happen earlier. MP-critical path $LDP(\sigma_i^P)$ represents the execution path among the programs that leads to the FIFO read or write. Such path provides designers important information on where to optimize their designs.

6.2 MP-Critical Path Finding Algorithm

To find the MP-critical path, first a naïve algorithm is presented. However, the naïve algorithm requires complete execution traces generated by all the processors for the duration of simulation. Therefore, an iterative algorithm is presented. Similar to traditional software profiling in single-processor systems, no trace has to be generated in simulation for later analysis. Instead, a set of counters are used to keep track on the performance-critical information.

6.2.1 Naïve Algorithm

Since the MP-critical path has a recursive definition, a naïve algorithm uses execution traces from simulation and back-tracks the execution path from the processing step. Such algorithm can be used alongside with trace-based analysis tools [24, 44]. Starting from the step σ_i^P, the simulation traces can be used to back-track the immediate prior step $Pre(\sigma_i^P)$. The immediate prior step is responsible for the lateness of the starting time of the step σ_i^P. The MP-critical path of the processing step can be built up by recursively back-tracking until the immediate prior step is an environment event σ^E, which could be the beginning of the execution σ_0^E. The naïve algorithm is shown in Algorithm 10.

Although the naïve trace-based algorithm allows the MP-critical path to be derived from execution traces, the algorithm relies on complete simulation traces and post-simulation analysis. Generating such traces is very expensive in term of both simulation speed and disk spaces. The algorithm is not scalable for long simulation and may generate very long traces that are difficult

Algorithm 10: Naïve trace-based algorithm for finding an MP-critical path.

```
Input: Traces from all $P$, Target step $\sigma_{iv}^{Pv}$
Output: $LDP(\sigma_{iv}^{Pv})$
1 $LDP(\sigma_{iv}^{Pv}) = \{\sigma_{iv}^{Pv}\}$
2 $\sigma_i^P = \sigma_{iv}^{Pv}$
3 while $Pre(\sigma_i^P) \neq \sigma^E$ do
4     $LDP(\sigma_{iv}^{Pv}) = LDP(\sigma_{iv}^{Pv}) \cup \{Pre(\sigma_i^P)\}$
5     $\sigma_i^P = Pre(\sigma_i^P)$
6 end
7 return $LDP(\sigma_{iv}^{Pv})$
```

to analyze. Therefore, an iterative algorithm is present to derive the MP-critical path dynamically during simulation that does not require generation of any traces.

6.2.2 Iterative Algorithm

The iterative algorithm comes from the definition that the MP-critical path of any step σ_i^P is based on the MP-critical path of its immediate prior step $Pre(\sigma_i^P)$. When expanding the definition of the MP-critical path $LDP(\sigma_i^P)$, the path includes the step σ_i^P itself and the MP-critical path of its immediate prior step $LDP(Pre(\sigma_i^P))$. Therefore, the following definition can be used to iteratively build up the MP-critical path in each step during simulation.

$$LDP(\sigma_i^P) = \{\sigma_i^P\} \cup LDP(Pre(\sigma_i^P))$$

The algorithm used to keep track of the MP-critical path in each step dynamically during simulation is shown in Algorithm 11. When each step starts, it copies the MP-critical path from its immediate prior step and adds itself into the path. The immediate prior step can be an environment event σ^E if the step is blocked by a read or a write from an empty input FIFO channel or a full output FIFO channel.

Only the MP-critical path of the currently executing step in each program is needed to be kept to determine the MP-critical path of the processing step at the end. The paths for other steps are not kept since only the path of the interested processing step is important. Therefore, one

Algorithm 11: Iterative algorithm for finding an MP-critical path.

```
  Input: Target step $\sigma_{iv}^{Pv}$
  Output: $LDP(\sigma_{iv}^{Pv})$
1 forall program P do
2     $LDP(\sigma_0^P) = \{\sigma_0^E\}$
3 end
4 repeat
5     foreach step $\sigma_i^P$ starts do
6         $LDP(\sigma_i^P) = \{\sigma_i^P\} \cup LDP(Pre(\sigma_i^P))$
7     end
8 until $\sigma_i^P$ violates a performance constraint
9 return $LDP(\sigma_i^P)$
```

MP-critical path of the currently executing step in each program is kept and the paths of those steps that are finished are discarded.

The iterative algorithm allows the MP-critical path to be derived very efficiently in a simulation. In the experiments, the MP-critical path profiling using the iterative algorithm slows down the simulation at most twice. Such overhead is reasonable and is in the same scale as traditional software profiling. On the other hand, the trace generation needed for the naïve algorithm slows down the simulation more than ten times. Note that the algorithm only proposes how to trace the MP-critical path during simulation and does not restrict the information that is kept for profiling. Such profiling information depends on the design level of the optimizations being applied. For optimizations that require more information being kept in the profiling results, addition simulation overhead applies for additional calculation, memory accesses and I/O operations. However, there is no reason to profile more information than those being used in the design optimizations.

6.3 Instruction-level Optimization

Information from the MP-critical path profiling can be used for design optimizations at different design levels. In this thesis, three optimization techniques at instruction level, segment level and program level are used to demonstrate the usefulness of the MP-critical path profiling information. First, instruction-level optimization using custom instructions at Tensilica's processors is presented.

6.3.1 Custom Instruction

At the instruction level, statements in the programs are profiled and those that are executed most frequently in the MP-critical path can be determined and are optimized. Custom instructions [162], hardware accelerators [63, 149] and library routines [122] are common techniques to speed up software executions at this level. They provide speedups by replacing the hotspots with faster executions using specific software or hardware. Since hotspot information for MPSoC is necessary to apply these techniques, correctly identifying hotspot information in the programs is crucial to optimize software in limited design time.

In this thesis, custom instructions added to the embedded processors are used. Tensilica's Xtensa LX2 processor with typical configurations is used in the experiments. This processor is extensible such that custom instructions can be designed and integrated into the processor datapath using *Tensilica Instruction Extension language* (TIE) [162]. These custom instructions improve the performance by having more compact instructions that perform operations that normally requires more instructions. Processors in the MPSoC design become heterogeneous by adding different custom instructions into the processors. However, such design optimizations do not come for free. Custom instructions add logics to the processor datapaths, which add area to the processors and may impact the maximum processor frequencies if the processors become too large.

6.3.2 Experiment and Result

To allow instruction-level optimizations, instruction execution frequencies are profiled. Specifically, execution frequency of each basic block is profiled. A basic block is a sequence of instructions that has one entrance, one exit and does not contain any jump instructions or jump targets.

Information from the traditional software profiling and the MP-critical path are compared in a simulation that decodes one group of pictures of an MPEG-2 stream. Each group of pictures contains half a second of video. Table 6.2 shows the execution frequencies of the statements in the traditional software profiling and the MP-critical path. The frequencies of the 14 most frequently

program	line #	profiling (%)	LDP (%)	diff (%)
Twritemb	299-300	9.96%	0.02%	-99.80%
Tpredict	400-401	7.56%	14.66%	+93.92%
Toutput	401-402	6.51%	0.71%	-89.09%
Tidct	203-237	5.79%	10.51%	+81.52%
Tpredict	382-390	5.48%	14.31%	+161.13%
Tadd	266-268	5.30%	0.59%	-88.87%
Tadd	278-285	4.59%	0.20%	-95.64%
Tpredict	367-369	2.72%	5.07%	+86.40%
Tpredict	338-339	2.17%	3.93%	+81.11%
Tpredict	351-357	2.01%	5.28%	+162.69%
Tidct	147-181	1.90%	3.79%	+99.47%
Tpredict	296-299	1.87%	4.64%	+148.13%

Table 6.2: Comparing instruction execution frequencies in traditional software profiling (*profiling*) vs. MP-critical path (*LDP*).

executed statements in the traditional software profiling are used for comparison. These 14 statements are responsible for more than 50% of the total execution time in both results. As shown in the table, although some statements execute very frequently and take a long time to execute (i.e. line 299-300 in *Twritemb*), the statements do not contribute to the MP-critical path. Optimizing these statements does not provide any performance improvements in the MPSoC design. On the other hand, some statements (i.e. line 382-390 in *Tpredict*) show to be more significant in the MP-critical path. These statements are important in the MPSoC design and optimizing these statements provides substantial performance improvements.

The overhead to keep track of the instruction execution frequencies in the MP-critical path in each step dynamically during simulation increases the simulation time by about 70%. Such overhead is comparable to traditional software profiling in single-processor simulation and several times faster than generating simulation traces for off-line analysis.

Based on the hotspot information from both profiling results, software optimizations are applied to the programs in the MPEG-2 Decoder design. Tensilica's *XPRES* compiler [58] is used to generate custom instructions for the most frequently executed statements. The XPRES compiler is directed to optimize the statements according to the hotspot information from Table 6.2. The default options are used to combine multiple instructions in the original programs into a lesser number of complex instructions. Table 6.3 shows the speedups of the custom instructions on the statements used in the experiments. The generated custom instructions reduce the execution time of the statements themselves by 36% to 55%. The table also shows the numbers of gates (area overhead) required to implement the custom instructions.

#	program	line #	gate	runtime
1	Twritemb	299-300	12,683	-42%
2	Tpredict	400-401	7,421	-38%
3	Toutput	401-402	12,693	-37%
4	Tidct	203-237	27,295	-36%
5	Tpredict	382-390	8,727	-50%
6	Tadd	266-268	12,779	-42%
7	Tadd	278-285	5,356	-38%
8	Tpredict	351-357	8,247	-47%
9	Tpredict	367-369	5,218	-55%
10	Tpredict	296-299	4,747	-40%
11	Tpredict	338-339	4,927	-55%
12	Tidct	147-181	17,861	-47%

Table 6.3: Speedup for custom instructions in the basic blocks used in the experiments.

Performance improvements using the hotspot information from the traditional software profiling and the MP-critical path are compared. For the traditional software profiling, the custom instructions are applied in the order of execution frequencies shown in the column *profiling* in Table 6.2. For the MP-critical path, custom instructions are iteratively applied to the most frequently executed statements shown in the MP-critical path. The total area for custom instructions is limited to 90K gates.

The software optimization results are shown in Figure 6.2. Using the MP-critical path, the important statements that can speed up the MPSoC design can be correctly determined. Therefore, a performance improvement can be observed in every custom instruction applied. On the other hand, traditional software profiling does not reveal the statements that are important in the MPSoC design. With imprecise hotspot information, designers will waste their time optimizing an unimportant part of the programs and can only discover later that the optimization shows very little performance improvements in the simulation. As a result, the software optimization using the MP-critical path information offers 50% better performance improvement than using the traditional software profiling when the area overhead is limited to 90K gates. In the scenario where 10% performance improvement is required to meet the performance constraints, custom instructions using

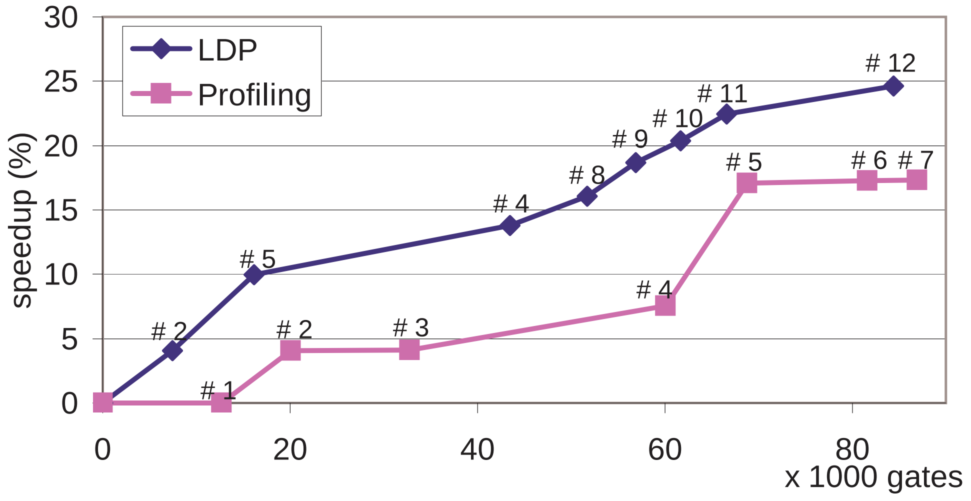


Figure 6.2: Software optimization results with custom instructions using information from the MP-critical path (*LDP*) and the traditional software profiling (*profiling*).

information from the MP-critical path take 16K gates, while custom instructions using information from traditional software profiling take 69K gates. The MP-critical path analysis provides designers correct hotspot information and allows designers to optimize the designs efficiently at instruction level.

6.4 Segment-level Optimization

In program segment level, compiler code motion techniques that specifically improve performance on MPSoC can be applied. In today's compilation flow of KPN applications, each program is compiled and optimized independently. During compilation of an individual program, the compiler, built for single-processor program compilation, only applies code motions that benefit the program on single-processor systems. Current compiler optimizations only consider the conditions when the program is executed on a single processor without interference from outside the program. These optimizations do not consider dependencies between the programs in the same design. Current compilers do not apply optimizations that can only be useful in MPSoC, nor do they have enough information to apply the optimizations. Therefore, code motion techniques that can only be useful in MPSoC are not implemented in the compilers.

With the MP-critical path information, code motion techniques that benefit the overall performance of the MPSoC design can be applied to the programs. In general, most optimizations that are used in single-processor compilers are also useful in MPSoC. However, some code motion techniques that do not provide improvements on single-processor systems are actually useful in MPSoC. Such techniques include specialized instruction reordering, loop restructuring and function inlining. Although these techniques are commonly used in compiler optimizations, their applications on MPSoC high-level optimizations are new and different from those have been done before.

6.4.1 Instruction Reordering

The MPSoC code motion techniques proposed are mainly based on instruction scheduling at higher level. Instruction scheduling is often used to improve instruction-level parallelism, such as to reduce pipeline hazards in a superscalar processor [1] or to expose all functional units in a VLIW processor [51, 62]. The proposed techniques extend instruction scheduling to MPSoC such that it improves task-level parallelism for an application written in KPN. The instructions are reordered to reduce the lengths of the *critical code segments* in the programs, therefore increase the parallelism at the segment level. This is done before the programs are individually compiled with the target compiler and –O2 optimizations.

A critical code segment is a segment in the MP-critical path where the sequence of instructions are in the same program. Each critical code segment begins with a read/write and also ends with a read/write. As shown in the MP-critical path analysis, a segment always starts with a program being unblocked, where the program is previously blocked because of a read or a write. At the end of the segment, the program unblocks another program by a read or a write. Although a critical code segment always starts and ends with reads or writes, without MP-critical path analysis it is impossible to identify the critical code segments because both reads and writes can start and end the segments and not all reads and writes start or end the segments.

Classification

To reorder the instructions using instruction scheduling, the instructions in a critical code segment are first classified into one of the four classes: *critical*, *prologue*, *epilogue* and *independent*. Critical instructions are instructions that have to be executed inside the segment. The prologue instructions are instructions that can be executed before or inside the critical code segment. The epilogue instructions can be executed during or after the critical code segment. The independent instructions can be moved independently from the critical code segment. Classifications of the instructions provide directions on how the instructions should be moved in order to reduce the length of the critical code segment. Note that instructions being classified also include other reads and writes that are inside the critical code segments.

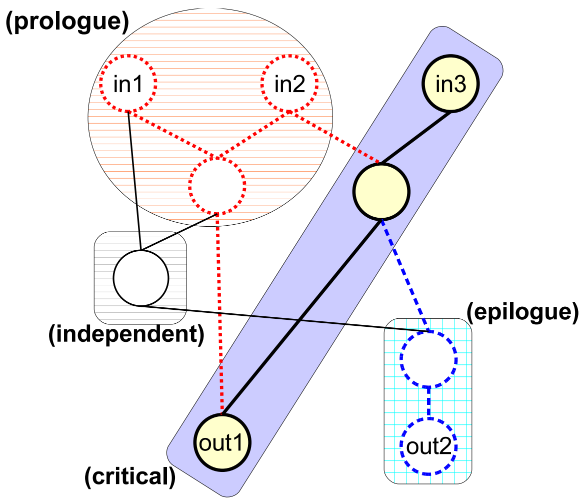


Figure 6.3: Classification of instructions in a critical code segment represented in a dataflow graph. The code segment starts with *in*3 and ends with *out*1. Instructions are classified into one of the four types: critical, prologue, epilogue and independent.

For dataflow statements, instructions can be classified with respect to a critical code segment using a dataflow graph. An example of the classification is shown in Figure 6.3. The example has three inputs (labeled *in*1, *in*2 and *in*3) and two outputs (labeled *out*1 and *out*2). The critical code segment begins with *in*3 and ends with *out*1. The *critical* instructions are those require the input data from *in*3 and compute the output data for *out*1. Such critical instructions are labeled with $(critical)$ in the figure. Instructions that compute the output data for *out*1 but do not require the input data from *in*3 are *prologues* (labeled with $(prologue)$). These instructions can be executed before reading *in*3 and therefore can be moved outside the critical code segment. Instructions that do not compute the output data for *out*1 but require the input data from *in*3 are *epilogues* (labeled with $(epilogue)$). These instructions can be executed after writing *out*1 and therefore can also be moved outside the critical code segment. The remaining instructions that neither require the input data from *in*3 nor compute the output data for *out*1 are *independents* (labeled with $(independent)$) and can be moved either before or after the critical code segment.

Specifically, the classification is defined as follows. An instruction *s* is considered to take a set of inputs and compute a set of outputs, which denoted as $in(s)$ and $out(s)$. The transitive inputs and outputs is denoted as

$$in*(s) = in(s) \cup \{s' \mid s' \in in(t), \exists t \in in*(s)\}, \text{ and} \tag{6.1}$$

$$out*(s) = out(s) \cup \{s' \mid s' \in out(t), \exists t \in out*(s)\}. \tag{6.2}$$

The critical instructions from i to j are those in the intersection: $in*(j) \cap out*(i)$. The prologues are those in $in*(j) - out*(i)$. The epilogues are those in $out*(i) - in*(j)$. Independents are those in $\neg(in*(j) \cup out*(i))$. The algorithm for classification only requires simple breath-first searches and set operations.

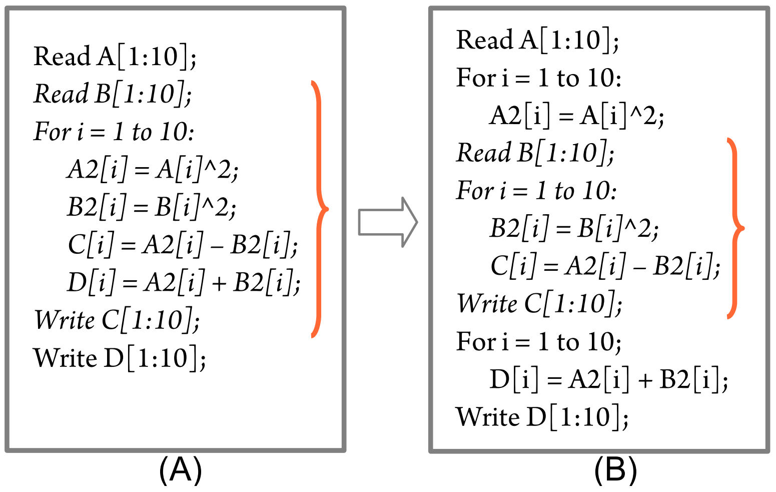
Reordering

Instructions are reordered such that the lengths of the critical segments are reduced. Instructions are reordered in the following ways: instructions that are classified as prologues are moved *backward* in the control-data flow graph before the read/write that begins the critical code segment; instructions that are classified as epilogues or independents are moved *forward* after the read/write that ends the segment. As a result, the instructions that are not critical are moved outside the critical code segments, hence there are fewer instructions inside the segments.

The instruction reordering also allows instructions to be moved past simple control-flow structures. When attempting to move an instruction forward beyond a branch, the instruction is duplicated and applied to all the branch targets in the control paths. For a join, the instruction can be moved beyond the join if the instruction can be moved forward from all the incoming branches. The same principle applies for moving instructions backward over the control-flow structures.

Note that similar to other instruction scheduling algorithms, this instruction reordering maintains the semantics of the programs. The instruction reordering is automatically done on the programs in the *static single assignment* (SSA) form. Every variable is assigned exactly once and there is only one version for each variable since it cannot be reassigned. Representing the programs in SSA form allows simple and efficient implementation of the instruction reordering.

Loop Restructuring



```
Read A[1:10];
Read B[1:10];
For i = 1 to 10:
    A2[i] = A[i]^2;
    B2[i] = B[i]^2;
    C[i] = A2[i] – B2[i];
    D[i] = A2[i] + B2[i];
Write C[1:10];
Write D[1:10];
```

```
Read A[1:10];
For i = 1 to 10:
    A2[i] = A[i]^2;
Read B[1:10];
For i = 1 to 10:
    B2[i] = B[i]^2;
    C[i] = A2[i] – B2[i];
Write C[1:10];
For i = 1 to 10;
    D[i] = A2[i] + B2[i];
Write D[1:10];
```

Figure 6.4: An example of loop restructuring that creates more freedom for instruction reordering.

An instruction cannot be moved beyond a loop because of the control dependencies. A loop contains a backward branch which makes the instruction difficult to be moved through using simple instruction reordering. As shown in the example in Figure 6.4, loops may need to be restructured to take advantage of the optimizations that use the MP-critical path information and code motion techniques. In the example, the critical code segment starts in reading the input *B* and ends in writing the output *C*. Without restructuring the loop, instructions inside the loop cannot be moved outside the critical code segment (Figure 6.4(A)). By restructuring the loop into separate parts, instructions that only use *A* and those only compute *D* can be moved out from the segment (Figure 6.4(B)), hence reducing its length.

In the design optimizations, loops are simply unrolled to expose the opportunities for instruction reordering. Loop unrolling is a common loop transformation technique that rewrites the loop as a sequence of independent statements at the expense of increasing the code sizes. By unrolling the loops, the maximum freedom is provided to the instruction reordering to move instructions forward and backward in the control-data flow graph, which allows the proposed design optimizations to be applied more efficiently. Therefore, if the loop unrolling is enabled in the high-level optimization tool, loops are unrolled automatically if they are on the critical code segments, and the unrolled loops are reverted if no benefit can be achieved.

Function Inlining

Similar to loops, instruction reordering also does not work well with functions. Although using functions is a desirable programming practice in procedural programming, it obscures the program analysis and instruction reordering. Because of the potential increase in complexity, most optimization techniques do not consider optimizations across functions. For a function call inside the critical code segment that is related to the input and output of the segment, the function call cannot be moved outside the segment since the effects of the function call are not clear. Everything that executes inside the function, regardless of whether it actually uses the input or computes the output, remains inside the critical code segment. Interprocedural analysis is necessary to further optimize the MPSoC design.

Function inlining is used to place all instructions inside the function on its caller such that instructions can be freely reordered. A function normally takes more than one inputs and computes more than one outputs. Not everything in the function is necessary to be in the critical code segments. Hence the instructions originally in the function can be classified (into critical, prologues, epilogues and independents) and reordered accordingly after inlining. Similar to loop unrolling, function inlining increases the code sizes of the programs. Therefore, if function inlining is enabled in the high-level optimization tool, functions are inlined automatically if they are on the critical code segments, and the inlined functions are reverted if no benefit can be achieved.

6.4.2 Experiment and Result

To illustrate the benefits of instruction reordering with a control-data flow graph. the IDCT component inside the MPEG-2 Decoder design is used to show how each optimization step changes the performance and code size. The IDCT component is a major component that directly affects the performance of the decoder. The IDCT component is based on the fast discrete cosine transform algorithm in [52]. The component operates first on rows then on columns in an input matrix. In the process network, three processes work on the rows (*IDCT-ROW*) and another three work on the columns (*IDCT-COL*). All high-level optimizations are automatically applied. Each program is compiled independently using *GCC* with $-O2$ optimizations, in additional to the code motions applied.

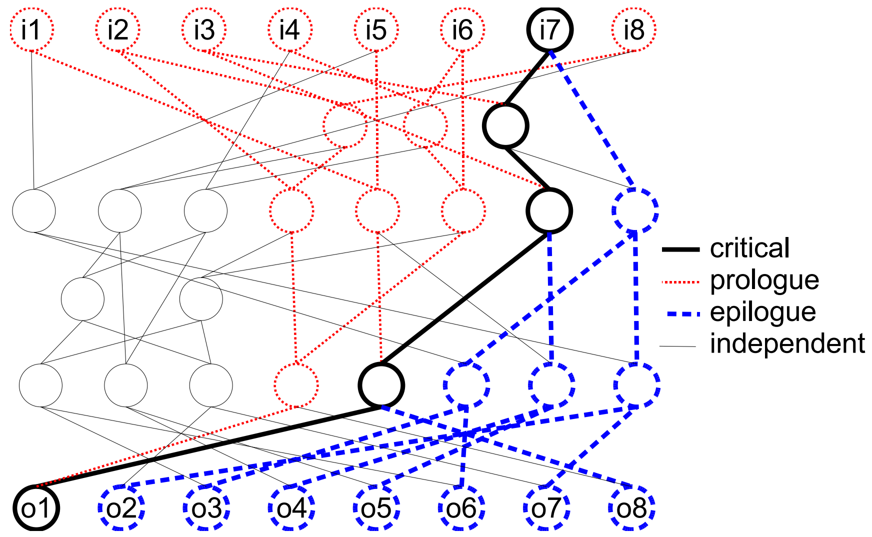
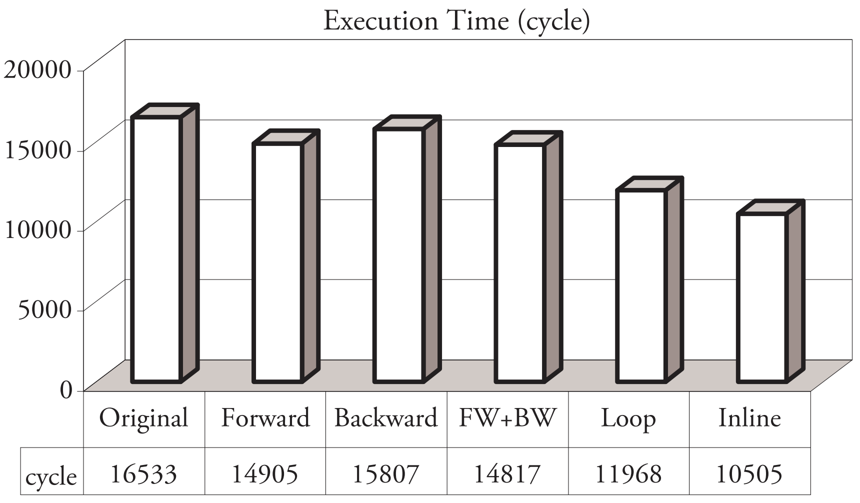


Figure 6.5: Instruction Classification in *IDCT-COL*.

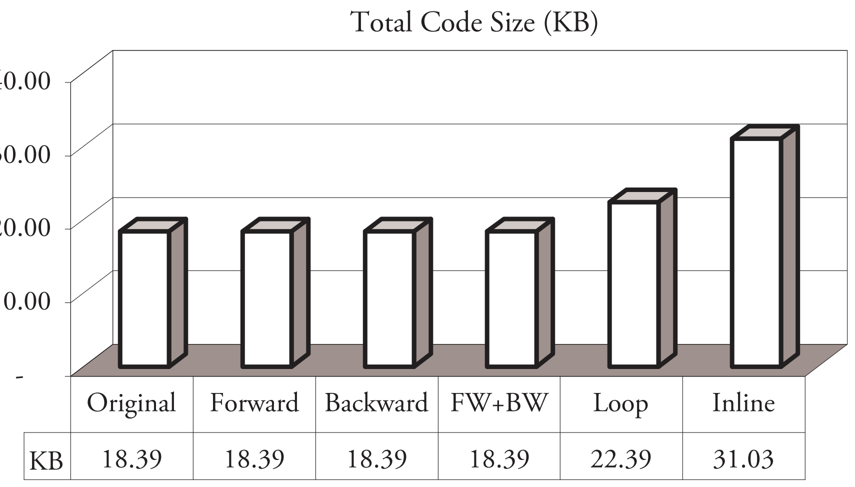
The optimization techniques are illustrated using the *IDCT-COL* process (Figure 6.5). The process transforms a column of eight data in the input matrix using a set of operations. Each of the eight outputs ($o1$–$o8$ on the bottom) is computed using all eight inputs ($i1$–$i8$ on the top). In one instance, the critical code segment starts in $i7$ and ends in $o1$. By restructuring the loops and inlining the functions, enough freedom is provided to reorder the instructions. As shown in Figure 6.5, only five out of 37 operations in the dataflow graph have to be placed inside the critical code segment. Armed with this information, high-level optimizations can be applied to reorder the instructions such that the lengths of the critical code segments are reduced.

Note that in a single-processor compiler, no such optimization would be applied because: **1.** no MP-critical path information is available by analyzing only one program, and **2.** the MPSoC specific optimizations do not improve the performance in single-processor systems.

With such optimizations, the critical code segments are shortened and the performance of the application is improved. Figure 6.6 shows the execution time and the total code sizes of the programs after the optimizations. *Forward* and *Backward* are the results of instruction reordering without loop restructuring or function inlining, where *Forward* only allows instruction to be moved



	Original	Forward	Backward	FW+BW	Loop	Inline
cycle	16533	14905	15807	14817	11968	10505



	Original	Forward	Backward	FW+BW	Loop	Inline
KB	18.39	18.39	18.39	18.39	22.39	31.03

Figure 6.6: Execution time and total code size results for instruction reordering. *Forward* only moves instructions forward in a program. *Backward* only moves instruction backward. *FW+BW* allows instructions to move both forward and backward, without loop restructuring or function inlining. *Loop* allows loop restructuring if the loops are inside the MP-critical path and help instruction reordering. *Inline* does the same for function inlining. A tradeoff between execution time and total code size is shown when more aggressive optimizations are applied.

forward in the control-data flow graph and *Backward* only allows instructions to be moved backward without breaking down loops and functions. *FW+BW* allows instruction reordering to move instructions both forward and backward. As shown in the figure, instruction reordering alone only provides limited performance improvements (<8%). These optimizations do not increase the total code sizes because very limited code is added or duplicated. These optimizations can be consider free and should be applied if the MP-critical path information is available.

To further improve the performance, a finer control on the instruction order is required. Loop restructuring and function inlining provide instruction reordering more freedom to operate. However, such transformations also increase the total code sizes, so they should be considered only for critical code segments (i.e. those that lie on the MP-critical path). Restructuring loops and inlining functions themselves also improve performance, but using them without the proposed instruction reordering only provides minor improvements. Significant improvements are shown only when they are used in conjunction with instruction reordering. By restructuring loops that lie on the MP-critical path with instruction reordering (*Loop*), the performance improves by 26% while the total code size increases by 22%. By inlining functions and unrolling loops that lie on the MP-critical path with instruction reordering (*Inline*), the performance improves by 35% while the total code size increases by 69%. There are obviously tradeoffs between performance and total code size, which is expected in the optimizations. Loop restructuring and function inlining increase the freedom to reorder instructions based on the MP-critical path information at the cost of larger code sizes. Loop unrolling and function inlining by themselves do not improve the performance appreciably. Only when these techniques are used alongside instruction reordering based on MP-critical path would they achieve the dramatic improvements.

6.5 Process-level Optimization

Voltage Island is a design technique that runs parts of the implementation at different voltages to obtain the desired power-frequency characteristics [87]. Due to the super-linearity of the power-frequency characteristic, it is beneficial to run an instruction at a slower speed to reduce the power consumption. When a processor is running at a lower voltage, both static and dynamic power are reduced but the maximum frequency also decreases. The processor consumes less energy to execute one instruction. Therefore, low voltage processors can be used to execute programs that are less important for performance. Unlike *Dynamic Voltage Scaling* (DVS) [26, 75, 119], voltages of the processors are not dynamically changeable during execution time. It is a compile-time decision to determine the voltages in which the processors are running.

At the program level, programs can map to the processors differently and the processors can run in different voltages. Hence in the program level design optimization, the program mappings and voltage assignments are determined. In voltage island, each processor runs on the same voltage. Multiple programs mapped to the same processors are restricted to run in the same voltage. At the same time, communication and interactions between the programs in the same processor can use the local memory and do not require data transfers on the on-chip interconnect network, which is significantly slower and consumes more power. For this optimization, the granularity for MP-critical path analysis is at the program level. The design optimization technique uses the program-level MP-critical path information to determine the desirable program mappings and voltage assignments.

6.5.1 Program Mapping and Voltage Assignment

Power optimizations have been commonly applied at transistor level [146] and circuit level [164]. Most embedded processors support clock-gating as an energy-saving feature [7, 154]. Fine-grained clock-gating, which involves a small amount of gates, is usually automatic and does not require user controls. A processor can also be coarsely clock-gated to disable all switching activities in the processor when it is idle, except minimum logics to detect interrupts to bring the processor back up. Such optimizations are orthogonal to system-level power optimizations described here, which take system-level information into considerations.

With transistors scaling down, leakage power consumption becomes as significant as dynamic power consumption [20]. Simply cutting down switching activities alone can no longer reduce enough power consumption especially when the processor is idle. *Dynamic Power Management* (DPM) allows power-off of the processors such that no leakage or dynamic power is consumed. In the experiments, the *break-even* time [16] policy is used in the implementations.

Dynamic Voltage Scaling (DVS) [26, 75, 119] allows processors to conserve power by scaling the voltages to adapt to different workloads during execution. However, voltage scaling overhead is so large that frequent scaling is neither performance nor energy efficient. Therefore, DVS is considered as an extension of voltage island for different modes of operation that have different workloads, such as decoding video streams with different resolutions and framerates. Since these mode changes happen very infrequently, voltage island assignments can be computed statically and then applied dynamically according to different operating modes.

Current techniques on voltage island can only be applied to designs that are written as task graphs with known execution time [91, 108]. However, embedded software cannot always be converted into task graphs. In this thesis, KPN is considered, where each process is a sequential program written in C language that communicates with other processes using FIFO channels.

The optimization goal here is to find an implementation (i.e. program mappings and voltage assignments) that minimizing the power consumption while satisfying the performance constraints. Power is consumed when executing an instruction, regardless of whether the instruction is part of the MP-critical path. Hence, power consumption is the power consumption of every instruction executed and every data transferred over the on-chip interconnect network. The optimization tries to minimize the power consumption of an implementation estimated by:

$$\begin{aligned} Min\{&\Sigma_i(w_i \times en(V_i))+ \\ &\Sigma_i\Sigma_{j<i}(G_{ij} \times t_{ij} \times E_{global}\}+ \\ &\Sigma_i\Sigma_{j<i}((1-G_{ij}) \times t_{ij} \times E_{local})\} \end{aligned} \tag{6.3}$$

where w_i is the computation cycle in program i. V_i is the voltage of the processor that program i maps to. $en(V_i)$ is the energy per cycle for voltage V_i. G_{ij} is 1 when program i and program j are mapped to different processors, and 0 otherwise. t_{ij} is the amount of data transferred between program i and program j. E_{global} and E_{local} are energy consumption of each data transfer in the on-chip network and a local bus, respectively. The first term of the equation represents the energy spent on computation in the programs. The processors are switched off when there is no computation. The second term is the energy spent on global communication over the on-chip interconnect network, and the last term is the energy spent on local accesses for FIFO reads and writes.

With MP-critical path analysis, each program and FIFO channel may have, respectively, a fraction of computation and a fraction of communication that lie on the MP-critical path. Performance constraint requirements that the desired implementation must satisfy can be set. For the MP-critical path captured, the performance constraint is

$$\begin{aligned} \text{Maximum allowed time} \geq &\Sigma_i(N_i \times w@path_i \times cyc(V_i))+ \\ &\Sigma_i\Sigma_{j<i}(G_{ij} \times t@path_{ij} \times L) \end{aligned} \tag{6.4}$$

where N is the number of programs in the processor which program i maps on. $w@path_i$ is the computation cycle in the path on program i. $cyc(V_i)$ is the cycle time of the processor in voltage V_i.

$t@path_{ij}$ is the FIFO traffic in the path between program i and program j. L is the time for transfers via the on-chip network.

6.5.2 Optimization Algorithm

The optimization algorithm determines the MPSoC implementation based on the information about the overall computation (w_i for each program and t_{ij} between each pair of programs) and the MP-critical path ($w@path_i$ and $t@path_{ij}$ from the MP-critical path analysis). Based on the program-level information, the optimization algorithm is applied.

Algorithm 12: Greedy optimization algorithm to find program mappings and voltage assignment using program-level MP-critical path information.

```
FIND IMPLEMENTATION()
    start with lowest power implementation
    repeat
        simulate and find the MP-critical path
        EXPAND()
        REDUCE()
    until performance constraints are satisfied
```

Although the optimization problem can be solved using an *Integer-Linear Programming* (ILP) solver, for scalability reason a greedy algorithm is proposed. The algorithm (Algorithm 12) consists of mainly two steps: `EXPAND()` and `REDUCE()`. The algorithm starts with an implementation with the lowest possible power consumption, where all programs mapped to one processor running at the lowest voltage. In `EXPAND()`, the algorithm gradually increases the voltages of the processors and separates programs into processors (Algorithm 13) such that the constraints are satisfied. To minimize the power consumption while improving the performance, the change that allows maximum performance improvement over energy increase ($\frac{\triangle performance}{\triangle energy}$) is chosen and applied. Therefore, the implementation tries to give up the least power to meet the performance constraints. After the first step, an implementation that satisfies the performance constraints without using too much additional power is found.

Algorithm 13: *EXPAND* method used in the greedy optimization algorithm in Algorithm 12.

```
EXPAND()
  repeat
    forall processors do
      find current $\frac{\triangle performance}{\triangle energy}$ for processor voltage $\uparrow$
    end
    forall programs do
      find current $\frac{\triangle performance}{\triangle energy}$ for program separation
    end
    apply maximum $\frac{\triangle performance}{\triangle energy}$
  until current MP-critical path is satisfied
```

Algorithm 14: *REDUCE* method used in the greedy optimization algorithm in Algorithm 12.

```
REDUCE()
  for N times do
    forall processors do
      find current $\frac{\triangle performance}{\triangle energy}$ for processor voltage $\downarrow$
    end
    forall programs do
      find current $\frac{\triangle performance}{\triangle energy}$ for program custering
    end
    try minimum $\frac{\triangle performance}{\triangle energy}$, reject if a constraint violated
  end
```

With `EXPAND()`, the implementation may be over-designed such that the voltages of the processors are too high or the implementation is over-partitioned into too many processors. In `REDUCE()` (Algorithm 14), the implementation is reduced such that the power consumption is reduced without violating the constraints. Processor voltage decreases or program clusterings are tested in the order of their estimated impacts on the performance. For efficiency reason, only a pre-defined number of *N* best candidates are tested. The algorithm is very efficient since it works with the program-level information from simulation. Each inner iteration only takes linear time to the size of the design. In the experiments, the optimization algorithm always takes less than one second to complete.

6.5.3 Experiment and Result

Programs that have heavy computation workloads are not necessarily important for performance. Importance for performance depends on the computations that lie on the MP-critical path (*critical cycles*). The computation workload (in cycles) in each program for decoding one second of an MPEG-2 stream and the corresponding critical cycles in the MP-critical path when the programs are running at the same voltage in separate processors are shown in Table 6.4.

program	computation(w)	w@path	fraction@path
Tadd	114.6 Mcyc	2.10 Mcyc	1.83%
Tdecmv	8.2 Mcyc	0.0 Mcyc	0.00%
Thdr	0.10 Mcyc	0.004 Mcyc	4.23%
Tidct	152.6 Mcyc	100.4 Mcyc	65.78%
Tisiq	39.8 Mcyc	0.03 Mcyc	0.07%
Toutput	86.1 Mcyc	1.55 Mcyc	1.81%
Tpredict	207.2 Mcyc	186.9 Mcyc	90.23%
Tvld	83.0 Mcyc	0.01 Mcyc	0.01%
Twritemb	94.9 Mcyc	0.09 Mcyc	0.09%

Table 6.4: Comparing program-level importance in computation workloads (*computation*) *vs* critical cycles (*w@path*). Percentage of the computation workload in each program that lies on the MP-critical path is shown in *fraction@path*.

The table shows that the computation workloads and the critical cycles can be very different in each program. Although the computation workloads for the *Tadd* and *Tidct* programs are similar, they are represented very differently in the MP-critical path. Less than 2% of the computation workload in the *Tadd* program lies on the MP-critical path, while more than 65% in the *Tidct* program lies on the path. Therefore, it is more important for the *Tidct* program to run at a higher voltage than the *Tadd* program. By exploring such differences, the power consumption can be reduced while satisfying the performance constraints using voltage islands. As shown in the results, determining an implementation without considering MP-critical path information and solely based on computation workloads leads to low quality implementations.

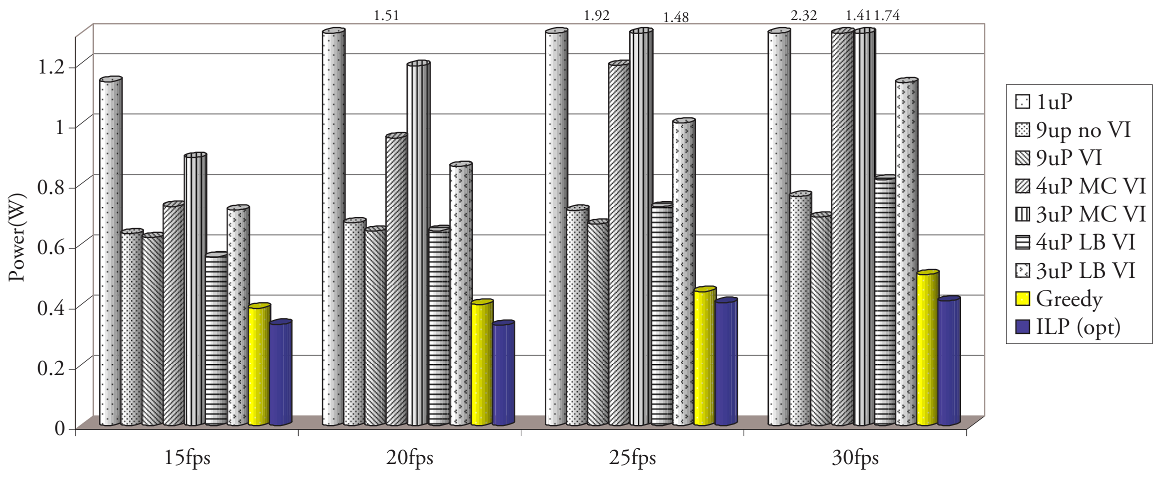


Figure 6.7: Power consumption of implementations with different program mappings and voltage assignments. Experiments are done on decoding an MPEG-2 stream with different framerate requirements.

Figure 6.7 shows the power consumption of the implementations using the MP-critical path information comparing and other approaches without using the information. Without the MP-critical path information, implementations can only base on more observable information, such as the computation workload in each program. Note that existing techniques for task graphs [91, 108] cannot apply here because no dependency information is available. Other reasonable implementations using computation workloads are used here for comparison. *1uP* is a single-processor implementation where the processor voltage is set to be just high enough to meet the performance constraints. The voltage is found using multiple simulations and changing the voltage with binary-search until a minimum voltage that meets the performance constraints is found. *9uP no VI* is an implementation where each program is mapped to individual processors running at the same voltage just high enough to meet the performance constraints. *9uP VI* have separate voltage islands for each processors and the voltages are scaled such that the processor frequencies are proportional to the computation workloads, which are the only performance-indicating number available without MP-critical path information. *4uP MC VI* and *3uP MC VI* are 4-processor and 3-processor implementations where the program mappings are based solely on *min-cut* to minimize the overall on-chip communication. Voltages are scaled based on the computation workloads. *4uP LB VI* and *3uP LB VI* are implementations where the program mappings are based solely on *load-balancing* to balance the computation workloads in the processors. The results using the program-level MP-critical path information and the greedy algorithm are shown with *Greedy*, and the optimal solutions using ILP solver are shown with *ILP (opt)*.

The performance requirements are set to decode an MPEG-2 streams of different framerates (from *15fps* to *30fps*). The higher the framerate, the tighter the performance constraints and the higher the voltages required in the processors. As shown in the figure, the power consumption of the greedy solutions are at least 30% better than other approaches across different framerate requirements of the MPEG-2 Decoder design. This is because the MP-critical path information provides more accurate information about the importance of each program in performance. The algorithm determines an implementation with a good combination of program mappings and voltage assignments such that the performance constraints are satisfied without using too much power.

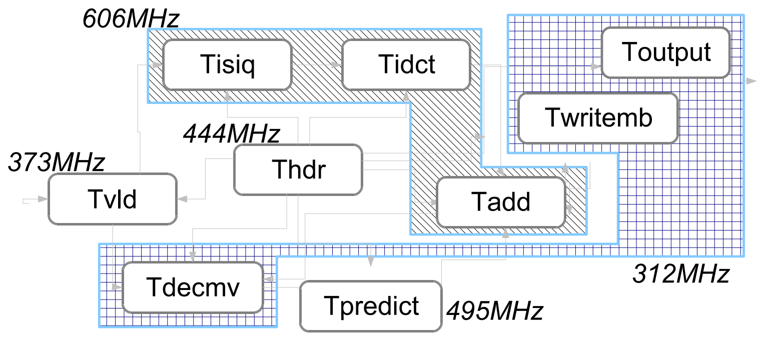


Figure 6.8: Implementation of the MPEG-2 Decoder with framerate requirement of *20fps*.

The implementation for the framerate requirement of *20fps* found using the greedy algorithm is shown in Figure 6.8. The programs are mapped to five processors. Each shaded box in the figure represents multiple programs running on one processor at the specified frequency. Those without boxes are running on separate processors at the specified frequencies. As shown in the implementation, the *Tpredict* and *Tidct* programs, which represent almost all the MP-critical path found, are running at higher frequencies. The *Tisiq*, *Tidct* and *Tadd* programs are mapped to the same processor because there are heavy communication traffics between them. Mapping them to the same processor conserves power by reducing the traffics on the on-chip interconnect network, although such mapping requires the processor to run at a higher frequency. On the other hand, *Tpredict*, *Tadd* and *Twritemb* are running in separate processors even with heavy communication traffics between them. This is because mapping them together in the same processor would require a much faster processor, which offset the energy saving on the on-chip interconnect network.

The greedy results are compared to the optimal ILP solutions of the same models. With the problem setting and constraints, there are more than 1936 constraints and 238 variables even though there are only 9 programs. The ILP problems are solved using the GNU Linear Programming Kit (GLPK) [57]. When comparing to the optimal solutions found in the ILP solver, the implementations found using the greedy algorithm consume slightly more power than the optimal solutions. However, the ILP approach is not scalable and is not applicable for more complicated designs. The number of constraints and variables scale in the order of $O(n^3)$ where n is the number of programs, and the complexity of the ILP problem scales exponentially with the number of constraints and variables. In the experiments, the ILP solver takes hours and days to find the solutions, while the greedy algorithm takes less than a minute including the simulation time to find the MP-critical path. Such results show that the design optimization technique is very good in finding power-efficient implementations using program mappings and voltage assignments at program level.

Chapter 7

Conclusion

In this thesis, I explained the benefits of using Kahn Process Network (KPN) in designing Multiprocessor System-on-a-Chip (MPSoC). KPN provides programming freedom in the software that is crucial for the success of MPSoC designs. To allow software and hardware design space exploration and design optimizations of an MPSoC design based on KPN, a profile-based optimization methodology was used. Fast and accurate MPSoC simulation and MPSoC-specific profiling information were two key elements in the methodology.

For fast and accurate multiprocessor system simulation, a simulation framework based on compile-code simulation was proposed. Since the software is the focus in MPSoC, the simulation model generator automatically generates software simulation models with accurate timing delays. The timing delay estimation is based on instruction-level information of the software programs in the target compiler, while accurately considering compiler optimizations and memory accesses. To further improve the performance of compile-code simulation, a simulation reordering technique was proposed to simulate the design in a non-chronological order. Our study shows that we can obtain performance results with less than 1% error for individual programs and 5% error for the overall MPSoC simulation. The simulation speed is more than 1000X faster than using Instruction Set Simulators (ISS).

For design space exploration, I defined an MP-critical path as the execution path that is important for performance in an MPSoC implementation. The MP-critical path correctly identifies the hotspots for efficient design optimizations in MPSoC. With such information, design optimizations at different design levels can be applied efficiently to optimize the implementations. Several optimization techniques at instruction level, segment level and program level were used to demonstrate the usefulness of the MP-critical path information.

Bibliography

[1] R D Acosta, J Kjelstrup, and H C Torng. An instruction issuing approach to enhancing performance in multiple functional unit processors. *IEEE Trans. Comput.*, 35(9):815–828, 1986.

[2] Actel. http://www.actel.com.

[3] Vishwani D. Agrawal. Synchronous path analysis in mos circuit simulator. In *DAC '82: Proceedings of the 19th conference on Design automation*, pages 629–635, Piscataway, NJ, USA, 1982. IEEE Press.

[4] Altera. http://www.altera.com.

[5] Federico Angiolini, Jianjiang Ceng, Rainer Leupers, Federico Ferrari, Cesare Ferri, and Luca Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1145–1150, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[6] ARC. http://www.arc.com.

[7] ARM. http://www.arm.com.

[8] Felice Balarin, Luciano Lavagno, Praveen Murthy, and Alberto Sangiovanni-vincentelli. Scheduling for embedded real-time systems. *IEEE Des. Test*, 15(1):71–82, 1998.

[9] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[10] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[11] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 82–86, New York, NY, USA, 2000. ACM Press.

[12] BDTi. picoChip PC102 software development tools and programming model, 2008.

[13] G. Beltrame, D. Sciuto, C. Silvano, D. Lyonnard, and C. Pilkington. Exploiting tlm and object introspection for system-level simulation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 100–105, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[14] A. Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 190–197, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[15] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[16] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. pages 231–248, 2002.

[17] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[18] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Comput.*, 37(1):48–57, 1988.

[19] G. Bontempi and W. Kruijtzer. A data analysis method for software performance prediction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 971, Washington, DC, USA, 2002. IEEE Computer Society.

[20] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[21] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 98–103, New York, NY, USA, 2001. ACM Press.

[22] Gunnar Braun, Andreas Hoffmann, Achim Nohl, and Heinrich Meyr. Using static scheduling techniques for the retargeting of high speed,compiled simulators for embedded processors from an abstract machine description. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 57–62, New York, NY, USA, 2001. ACM.

[23] Jeffrey Brown. Application-customized cpu design: The microsoft xbox 360 cpu story, 2005.

[24] Holger Brunst, Dieter Kranzlmüller, and Wolfgang Nagel. Tools for scalable parallel program analysis - vampir ng and dewiz. pages 93–102. 2005.

[25] D. Burke, J. Wawrzynek, K. Asanovi, A. Krasnov, A. Schultz, G. Gibeling, and P.-Y. Droz. Ramp blue: Implementation of a manycore 1008 processor fpga system. In *RSSI '08*, 2008.

[26] Zhen Cao, Brian Foo, Lei He, and Mihaela van der Schaar. Optimality and improvement of dynamic voltage scaling algorithms for multimedia applications. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 179–184, New York, NY, USA, 2008. ACM.

[27] Vikas Chandra, Anthony Xu, Herman Schmit, and Larry Pileggi. An interconnect channel design methodology for high performance integrated circuits. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21138, Washington, DC, USA, 2004. IEEE Computer Society.

[28] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, 1979.

[29] Chen Chang, John Wawrzynek, and Robert W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Des. Test*, 22(2):114–125, 2005.

[30] Jui-Ming Chang and Massoud Pedram. Codex-dp: co-design of communicating systems using dynamic programming. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 114, New York, NY, USA, 1999. ACM.

[31] Po-Chun Chang, I-Wei Wu, Jyh-Jiun Shann, and Chung-Ping Chung. Etahm: an energy-aware task allocation algorithm for heterogeneous multiprocessor. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 776–779, New York, NY, USA, 2008. ACM.

[32] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[33] Liang-Fang Chao and Edwin Hsing-Mean Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Trans. Parallel Distrib. Syst.*, 8(12):1259–1267, 1997.

[34] Karam S. Chatha and Ranga Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):193–208, 2002.

[35] Tiberiu Chelcea and Steven M. Nowick. A low-latency fifo for mixed-clock systems. In *WVLSI '00: Proceedings of the IEEE Computer Society Annual Workshop on VLSI*, page 119, Washington, DC, USA, 2000. IEEE Computer Society.

[36] G. Chen and M. Kandemir. Code restructuring for improving cache performance of mpsocs. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 271–274, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Guangyu Chen, Feihui Li, S. W. Son, and M. Kandemir. Application mapping for chip multiprocessors. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 620–625, June 2008.

[38] Hsi-Chuan Chen, David H. C. Du, David H. C. Du, and Li-Ren Liu. Critical path selection for performance optimization. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 547–550, New York, NY, USA, 1991. ACM.

[39] Massimiliano Chiodo, Paolo Guisto, Attila Jurecska, Luciano Lavagno, Ellen Sentovich, Harry Hsieh, Kei Suzuki, and Alberto Sangiovanni-Vincentelli. Synthesis of software programs for embedded control application. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 587–592, New York, NY, USA, 1995. ACM Press.

[40] Chip Multi Processor Watch. http://www.gigascale.org/mescal/maw.

[41] Jason Cong, Guoling Han, and Wei Jiang. Synthesis of an application-specific soft multiprocessor system. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 99–107, New York, NY, USA, 2007. ACM.

[42] Jérôme Cornet, Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 9–14, Munich, Germany,, March 2008.

[43] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. Cosyn: hardware-software co-synthesis of embedded systems. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 703–708, New York, NY, USA, 1997. ACM.

[44] Jacques Chassin de Kergommeaux and Benhur de Oliveira Stein. Paje: An extensible environment for visualizing multi-threaded programs executions. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 133–140, London, UK, 2000. Springer-Verlag.

[45] E. A. de Kock. Multiprocessor mapping of process networks: a jpeg decoding case study. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 68–73, New York, NY, USA, 2002. ACM.

[46] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *DAC '00*, pages 402–405, New York, NY, USA, 2000. ACM Press.

[47] Ashish Dixit. Networking applications for xtensa configurable processors. In *Linley Tech*, 2006.

[48] Basant Kumar Dwivedi, Anshul Kumar, and M. Balakrishnan. Automatic synthesis of system on chip multiprocessor architectures for process networks. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 60–65, New York, NY, USA, 2004. ACM.

[49] Basant Kumar Dwivedi, Anshul Kumar, and M. Balakrishnan. Synthesis of application specific multiprocessor architectures for process networks. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 780, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Will Eatherton. The push of network processing to the top of the pyramid. In *ANCS '05*, 2005.

[51] P. Faraboschi, J. A. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11):1638–1659, November 2001.

[52] E. Feig and S. Winograd. Fast algorithms for the discrete cosine transform. *IEEE Transactions on Signal Processing*, 40(9):2174–2193, September 1992.

[53] Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 325–330, New York, NY, USA, 2008. ACM.

[54] M. Geilen and T. Basten. Requirements on the execution of kahn process networks, 2003.

[55] Frank Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[56] Arijit Ghosh and Tony Givargis. Analytical design space exploration of caches for embedded systems. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10650, Washington, DC, USA, 2003. IEEE Computer Society.

[57] GNU Linear Programming Kit. http://www.gnu.org/software/glpk.

[58] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147, New York, NY, USA, 2003. ACM.

[59] M. Goraczko, Jie Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and Feng Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 191–196, Anaheim, CA,, June 2008.

[60] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[61] J. P. Grossman, Cliff Young, Joseph A. Bank, Kenneth Mackenzie, Douglas J. Ierardi, John K. Salmon, Ron O. Dror, and David E. Shaw. Simulation and embedded software development

for anton, a parallel machine with heterogeneous multicore asics. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 125–130, New York, NY, USA, 2008. ACM.

[62] R. Gupta. Employing register channels for the exploitation of instruction level parallelism. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 118–127, New York, NY, USA, 1990. ACM.

[63] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. pages 5–17, 2002.

[64] Ali Habibi and Sofiene Tahar. Design for verification of systemc transaction level models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 560–565, Washington, DC, USA, 2005. IEEE Computer Society.

[65] Tom Halfhill. Ambrics's new parallel processor: Globally asunchronous architecture eases parallel programming, 2006.

[66] Sang-Il Han, Xavier Guerin, Soo-Ik Chae, and Ahmed A. Jerraya. Buffer memory optimization for video codec application modeled in simulink. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 689–694, New York, NY, USA, 2006. ACM.

[67] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. Compilation from matlab to process networks realised in FPGA. In *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, volume 1, pages 458–462, Pacific Grove, CA, USA, 2001.

[68] M. AbdElSalam Hassan, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai. Rtk-spec tron: A simulation model of an itron based rtos kernel in systemc. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 554–559, Washington, DC, USA, 2005. IEEE Computer Society.

[69] P. D. Hoang and J. M. Rabaey. Scheduling of DSP programs onto multiprocessors for maximumthroughput. *IEEE Transactions on Signal Processing*, 41(6):2225–2235, June 1993.

[70] Junwei Hou and Wayne Wolf. Process partitioning for distributed embedded systems. In *CODES '96: Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, page 70, Washington, DC, USA, 1996. IEEE Computer Society.

[71] James P. Huang. Modeling of software partition for distributed real-time applications. *IEEE Trans. Softw. Eng.*, 11(10):1113–1126, 1985.

[72] Kai Huang, Sang il Han, Katalin Popovici, Lisane Brisolaraand Xavier Guerin, Lei Li, Xiaolang Yan, Soo lk Chae, Luigi Carro, and Ahmed Amine Jerraya. Simulink-based mpsoc design flow: case study of motion-jpeg and h.264. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 39–42, New York, NY, USA, 2007. ACM Press.

[73] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 3–8, New York, NY, USA, 2008. ACM.

[74] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[75] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 275–280, New York, NY, USA, 2004. ACM.

[76] A Jerraya and W Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2004.

[77] Woo-Chul Jeun and Soonhoi Ha. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 44–49, Washington, DC, USA, 2007. IEEE Computer Society.

[78] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for fpga-based soft multiprocessor systems. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 273–278, New York, NY, USA, 2005. ACM.

[79] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[80] Nasser Kalantery. Time warp - connection oriented. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pages 71–77, New York, NY, USA, 2004. ACM.

[81] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[82] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[83] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *CODES '00*, pages 13–17, New York, NY, USA, 2000. ACM Press.

[84] Dohyung Kim, Soonhoi Ha, and Rajesh Gupta. Parallel co-simulation using virtual synchronization with redundant host execution. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1151–1156, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[85] Tim Kogel and Matthew Braun. Virtual prototyping of embedded platforms for wireless and multimedia. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 488–490, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[86] Tim Kogel and Denis Bussaglia. Systemc based design of an ip forwarding chip with cocentric system studio. In *SNUG '02: Proceedings of Synopsys Users Group*, 2002.

[87] David E. Lackey, Paul S. Zuchowski, Thomas R. Bednar, Douglas W. Stout, Scott W. Gould, and John M. Cohn. Managing power and performance for system-on-chip designs using voltage islands. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 195–202, New York, NY, USA, 2002. ACM.

[88] Marcello Lajolo, Mihai Lazarescu, and Alberto Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign*, pages 85–89, New York, NY, USA, 1999. ACM.

[89] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 167, Washington, DC, USA, 2000. IEEE Computer Society.

[90] Jong-Yeol Lee and In-Cheol Park. Timed compiled-code simulation of embedded software for performance analysis of soc design. In *DAC '02*, pages 293–298, New York, NY, USA, 2002. ACM Press.

[91] Lap-Fai Leung and Chi-Ying Tsui. Energy-aware synthesis of networks-on-chip implemented with voltage islands. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 128–131, New York, NY, USA, 2007. ACM.

[92] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprttere. System level design with spade: an m-jpeg case study. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 31–38, Piscataway, NJ, USA, 2001. IEEE Press.

[93] Yanhong Liu, Samarjit Chakraborty, and Wei Tsang Ooi. Approximate vccs: a new characterization of multimedia workloads for system-level mpsoc design. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 248–253, New York, NY, USA, 2005. ACM.

[94] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a mpsoc environment. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20752, Washington, DC, USA, 2004. IEEE Computer Society.

[95] Jiong Luo and Niraj K. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 719, Washington, DC, USA, 2002. IEEE Computer Society.

[96] Zhe Ma and Francky Catthoor. Scalable performance-energy trade-off exploration of embedded real-time systems on multiprocessor platforms. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1073–1078, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[97] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 147–152, New York, NY, USA, 1997. ACM Press.

[98] Diana Marculescu and Siddharth Garg. System-level process-driven variability analysis for single and multiple voltage-frequency island systems. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 541–546, New York, NY, USA, 2006. ACM.

[99] Grant Martin. Overview of the mpsoc design challenge. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 274–279, New York, NY, USA, 2006. ACM Press.

[100] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 721–725, New York, NY, USA, 2002. ACM.

[101] Sjoerd Meijer, Bart Kienhuis, Johan Walters, and David Snuijf. Automatic partitioning and mapping of stream-based applications onto the intel ixp network processor. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 23–30, New York, NY, USA, 2007. ACM.

[102] MIPS. http://www.mips.com.

[103] R. Le Moigne, O. Pasquier, and J-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *DATE '04: Proceedings of the conference on Design, automation andtest in Europe*, page 30082, Washington, DC, USA, 2004. IEEE Computer Society.

[104] MPEG Elementary Streams. ftp://ftp.tek.com/tv/test/streams/element.

[105] Hiroaki Nakamura, Naoto Sato, and Naoshi Tabuchi. An efficient and portable scheduler for rtos simulation and its certified integration to systemc. In *DATE '06: Proceedings of the conference on Design, automation andtest in Europe*, pages 1157–1158, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[106] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Multi-processor system design with espam. In *CODES+ISSS '06*, pages 211–216, New York, NY, USA, 2006. ACM Press.

[107] Koushik Niyogi and Diana Marculescu. Speed and voltage selection for gals systems based on voltage/frequency islands. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 292–297, New York, NY, USA, 2005. ACM.

[108] Umit Y. Ogras, Radu Marculescu, Puru Choudhary, and Diana Marculescu. Voltage-frequency island partitioning for gals-based networks-on-chip. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 110–115, New York, NY, USA, 2007. ACM.

[109] Open SystemC Initiative. http://www.systemc.com.

[110] OpenRISC. http://www.opencores.org.

[111] OpenSparc. http://www.opensparc.net.

[112] Márcio Seiji Oyamada, Felipe Zschornack, and Flávio Rech Wagner. Accurate software performance estimation using domain classification and neural networks. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 175–180, New York, NY, USA, 2004. ACM.

[113] ARM White Paper. The ARM Cortex-A9 processors, 2007.

[114] Tensilica White Paper. Adding video to SOC: The diamond 388VDO video engine, 2007.

[115] T. Parks. Bounded scheduling of process networks, 1995.

[116] Sudeep Pasricha and Nikil Dutt. Cosmeca: application specific co-synthesis of memory and communication architectures for mpsoc. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 700–705, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[117] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Constraint-driven bus matrix synthesis for mpsoc. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 30–35, Piscataway, NJ, USA, 2006. IEEE Press.

[118] JoAnn M. Paul, Alex Bobrek, Jeffrey E. Nelson, Joshua J. Pieper, and Donald E. Thomas. Schedulers as model-based design elements in programmable heterogeneous multiprocessors. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 408–411, New York, NY, USA, 2003. ACM.

[119] Trevor Pering, Thomas Burd, and Robert Brodersen. Voltage scheduling in the iparm microprocessor system. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 96–101, New York, NY, USA, 2000. ACM.

[120] Kalyan S. Perumalla. usik: A micro-kernel for parallel/distributed simulation systems. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 59–68, Washington, DC, USA, 2005. IEEE Computer Society.

[121] Kalyan S. Perumalla. Scaling time warp-based discrete event execution to 104 processors on a blue gene supercomputer. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 69–76, New York, NY, USA, 2007. ACM.

[122] Armita Peymandoust, Giovanni De Micheli, and Tajana Simunic. Complex library mapping for embedded software using symbolic algebra. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 325–330, New York, NY, USA, 2002. ACM.

[123] Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transations on Computers*, 55(2):99–112, 2006. Member-Andy D. Pimentel and Student Member-Cagkan Erbas.

[124] Katalin Popovici and Ahmed Jerraya. Flexible and abstract communication and interconnect modeling for mpsoc. In *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 143–148, Piscataway, NJ, USA, 2009. IEEE Press.

[125] H. Posadas, F. Herrera, P. Sanchez, E. Villar, and F. Blasco. System-level performance analysis in systemc. In *DATE '04: Proceedings of the conference on Design, automation andtest in Europe*, page 10378, Washington, DC, USA, 2004. IEEE Computer Society.

[126] Shiv Prakash and Alice C. Parker. Synthesis of application-specific multiprocessor architectures. In *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 8–13, New York, NY, USA, 1991. ACM.

[127] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[128] M. Radetzki and R. Salimi Khaligh. Accuracy-adaptive simulation of transaction level models. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 788–791, New York, NY, USA, 2008. ACM.

[129] Balaji Raman, Samarjit Chakraborty, Wei Tsang Ooi, and Santanu Dutta. Reducing data-memory footprint of multimedia applications by delay redistribution. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 738–743, New York, NY, USA, 2007. ACM.

[130] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 758–763, New York, NY, USA, 2003. ACM.

[131] Víctor Reyes, Tomás Bautista, Gustavo Marrero, Antonio Nú nez, and Wido Kruijtzer. A multicast inter-task communication protocol for embedded multiprocessor systems. In

CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pages 267–272, New York, NY, USA, 2005. ACM.

[132] K. Richter, M. Jersak, and R. Ernst. A formal approach to mpsoc performance verification. *Computer*, 36(4):60–67, April 2003.

[133] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pages 285–296, Venice, Italy, October 1993.

[134] Martino Ruggiero, Andrea Acquaviva, Davide Bertozzi, and Luca Benini. Application-specific power-aware workload allocation for voltage scalable mpsoc platforms. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 87–93, Washington, DC, USA, 2005. IEEE Computer Society.

[135] S. Saha, S. S. Bhattacharyya, and W. Wolf. A communication interface for multiprocessor signal processing systems. In *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 127–132, Washington, DC, USA, 2006. IEEE Computer Society.

[136] M. Saldana and P. Chow. TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, August 2006.

[137] Manuel Saldana, Emanuel Ramalho, and Paul Chow. A message-passing hardware/software co-simulation environment to aid in reconfigurable computing design using tmd-mpi. In *RECONFIG '08: Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, pages 265–270, Washington, DC, USA, 2008. IEEE Computer Society.

[138] F. Salice, L. Del Vecchio, L. Pomante, and W. Fornaciari. Partitioning of embedded applications onto heterogeneous multiprocessor architectures. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 661–665, New York, NY, USA, 2003. ACM.

[139] Mitsuhisa Sato. Openmp: parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 109–111, New York, NY, USA, 2002. ACM.

[140] Gunar Schirner and Rainer Domer. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 122–127, Munich, Germany,, March 2008.

[141] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. A co-design methodology for energy-efficient multi-mode embedded systems with consideration of mode execution probabilities. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10960, Washington, DC, USA, 2003. IEEE Computer Society.

[142] Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. High-performance timing simulation of embedded software. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 290–295, New York, NY, USA, 2008. ACM.

[143] Benaoumeur Senouci, Aimen Bouchhima, Frederic Rousseau, Frederic Petrot, and Ahmed Jerraya. Fast prototyping of posix based applications on a multiprocessor soc architecture: "hardware-dependent software oriented approach". In *RSP '06: Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, pages 69–75, Washington, DC, USA, 2006. IEEE Computer Society.

[144] Seng Lin Shee and Sri Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 811–816, New York, NY, USA, 2007. ACM.

[145] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model.

[146] Supamas Sirichotiyakul, Tim Edwards, Chanhee Oh, Jingyan Zuo, Abhijit Dharchoudhury, Rajendran Panda, and booktitle = DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation year = 1999 isbn = 1-58133-109-7 pages = 436–441 location = New Orleans, Louisiana, United States doi = http://doi.acm.org/10.1145/309847.309975 publisher = ACM address = New York, NY, USA David Blaau, title = Stand-by power minimization through simultaneous threshold voltage selection and circuit sizing.

[147] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 7–12, New York, NY, USA, 2002. ACM.

[148] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: The compaan/laura approach. In *DATE '04*, page 10340, Washington, DC, USA, 2004. IEEE Computer Society.

[149] Greg Stitt, Frank Vahid, and Shawn Nematbakhsh. Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. *Trans. on Embedded Computing Sys.*, 3(1):218–232, 2004.

[150] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 777–782, New York, NY, USA, 2007. ACM.

[151] Sander Stuijk, Marc Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC '06: Proceedings of the*

43rd annual Design Automation Conference, pages 899–904, New York, NY, USA, 2006. ACM.

[152] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *VLSID '05: Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 551–556, Washington, DC, USA, 2005. IEEE Computer Society.

[153] Kei Suzuki and Alberto Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 605–610, New York, NY, USA, 1996. ACM Press.

[154] Tensilica. http://www.tensilica.com.

[155] Nattawut Thepayasuwan and Alex Doboli. Layout conscious approach and bus architecture synthesis for hardware/software codesign of systems on chip optimized for speed. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(5):525–538, 2005.

[156] Walter Tibboel, Victor Reyes, Martin Klompstra, and Dennis Alders. System-level design flow based on a functional reference for hw and sw. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 23–28, New York, NY, USA, 2007. ACM.

[157] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A compile time based approach for solving out-of-order communication in kahn process networks. In *ASAP '02: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, page 17, Washington, DC, USA, 2002. IEEE Computer Society.

[158] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, New York, NY, USA, 2004. ACM.

[159] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, WidoKruijtzer, and Gerben Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM Press.

[160] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007.

[161] Emmanuel Viaud, François Pêcheux, and Alain Greiner. An efficient tlm/t modeling and simulation environment based on conservative parallel discrete event principles. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 94–99, Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[162] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 184–188, New York, NY, USA, 2001. ACM.

[163] David Wang. The cell microprocessor. In *ISSCC '05*, 2005.

[164] Qing Wu, M. Pedram, and Xunwei Wu. Clock-gating and its application to low power design of sequentialcircuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(3):415–420, March 2000.

[165] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 664–669, New York, NY, USA, 2007. ACM.

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

[167] Jinsheng Xu and Jinghua Zhang. Efficiently unifying parallel simulation techniques. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 228–232, New York, NY, USA, 2006. ACM.

[168] Ti-Yen Yen and Wayne Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(11):1125–1136, 1998.

[169] Youngmin Yi, Dohyung Kim, and Soonhoi Ha. Fast and accurate cosimulation of MPSoc using trace-driven virtual synchronization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(12):2186–2200, December 2007.

[170] Sushu Zhang and Karam S. Chatha. Approximation algorithm for the temperature-aware scheduling problem. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 281–288, Piscataway, NJ, USA, 2007. IEEE Press.

[171] Vladimir D. Zivkovic, Erwin de Kock, Pieter van der Wolf, and Ed Deprettere. Fast and accurate multiprocessor architecture exploration with symbolic programs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10656, Washington, DC, USA, 2003. IEEE Computer Society.

[172] V. Zivojnovic and H. Meyr. Compiled HW/SW co-simulation. In *Design Automation Conference Proceedings 1996, 33rd*, pages 690–695, Las Vegas, NV, USA, June 1996.