Topics in Power and Performance Optimization of Embedded Systems

by

Michael A. Baker

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2011 by the
Graduate Supervisory Committee:

Karam S. Chatha, Chair
Gregory B. Raupp
Sarma B. K. Vrudhula
Aviral Shrivastava

ARIZONA STATE UNIVERSITY
May 2011

ABSTRACT

The ubiquity of embedded computational systems has exploded in recent years impacting everything from hand-held computers and automotive driver assistance to battlefield command and control and autonomous systems. Typical embedded computing systems are characterized by highly resource constrained operating environments. In particular, limited energy resources constrain performance in embedded systems often reliant on independent fuel or battery supplies. Ultimately, mitigating energy consumption without sacrificing performance in these systems is paramount. In this work power/performance optimization emphasizing prevailing data centric applications including video and signal processing is addressed for energy constrained embedded systems.

Frameworks are presented which exchange quality of service (QoS) for reduced power consumption enabling power aware energy management. Power aware systems provide users with tools for precisely managing available energy resources in light of user priorities, extending availability when QoS can be sacrificed. Specifically, power aware management tools for next generation bistable electrophoretic displays and the state of the art H.264 video codec are introduced. The multiprocessor system on chip (MPSoC) paradigm is examined in the context of next generation many-core hand-held computing devices. MPSoC architectures promise to breach the power/performance wall prohibiting advancement of complex high performance single core architectures. Several many-core distributed memory MPSoC architectures are commercially available, while the tools necessary to effectively tap their enormous potential remain largely open for discovery. Adaptable scalability in many-core systems is addressed through a scalable high performance multicore H.264 video decoder implemented on the representative Cell Broadband Engine (CBE) architecture. The resulting agile performance scalable system enables efficient adaptive power optimization via decoding-rate driven sleep and voltage/frequency state management. The significant problem of mapping

applications onto these architectures is additionally addressed from the perspective of instruction mapping for limited distributed memory architectures with a code overlay generator implemented on the CBE. Finally runtime scheduling and mapping of scalable applications in multitasking environments is addressed through the introduction of a lightweight work partitioning framework targeting streaming applications with low latency and near optimal throughput demonstrated on the CBE.

For Bethany.

ACKNOWLEDGEMENTS

I would like to start by thanking my graduate advisor, Dr. Karam Chatha, foremost for agreeing to take me on as a student, for introducing me to the world of embedded systems research and its international community, and ultimately for educating and leading me through to the culmination of my formal education. Your thoughtful guidance and intuitive ability to sort through the conceptual milieu and focus on the kernels that lead to genuine contribution has been central to any success I have been able to achieve at ASU. I am privileged to have had this opportunity to expand my understanding of our world while entertaining your interest in my perspective through our many energetic conversations in various corners of the world.

I deeply appreciate the professional and personal mentorship provided me by my graduate committee. I am truly grateful to Dr. Greg Raupp for embracing my educational goals predating my application to the university. Without your interest and involvement, from introducing me to the CSE department head to supporting my transition into the Ph.D. program, I would not be where I am. You were always available and willing to support my efforts despite your significant duties and responsibilities, Bethany and I have not forgotten how you took the time to give us a personal tour of the Flexible Display Center (FDC) during our first visit to Tempe. I want to express my great appreciation to Dr. Aviral Shrivastava for your guidance and interest throughout my time at ASU. It was in my first graduate level course that you introduced me to power optimization in the context of embedded systems. Your enthusiasm and thoughtful guidance started me on a fruitful path right from the beginning and has contributed immeasurably to any of my successes throughout my time at ASU. I would like to thank Dr. Sarma Vrudhula for agreeing to join my committee, and for your continued interest over these past few years as well as your dedicated leadership at the Embedded Systems Consortium, not to mention your enduring dedication to the endorsement of my million dollar "invention," the *pocket presentation slide booklet*.

There have been several people at the US Army Research Laboratory (ARL) and the FDC whose support, encouragement, and continued interest in my success have been invaluable to me. In particular, I would like to thank Dr. John Pellegrino for taking the time to mentor me over the last couple of years despite what must be an innumerable set of other priorities, for your professionalism, your interest in my research and success, and for always working to keep me on the right path. I am deeply indebted to Dr. Eric Forsythe for the continuous mentorship despite an awe inspiring travel calendar, and for the endless hours you endured with me picking your brain about the ins and outs of supporting a consortium for the Army with a mission that's likely to forever change handheld computers as we know them. I am grateful to Dr. David Morton who somehow manages a more hectic travel schedule than Eric but still finds time to address my questions and, along with Eric, put out the trail of fires I have been known leave in my wake.

Of particular note at the FDC, I would like to thank Dr. Nick Colaneri for putting up with my constant intrusions into otherwise peaceful and quiet FDC activities. It has been a genuine pleasure having the opportunity to work with Dr. Jann Kaminski, Dr. Sameer Venugopal, Kris Gillis, and Brian Hawley as part of the capstone demonstrator projects over the past three years, the entire experience has been fantastically rewarding and educational. I would also like to acknowledge Jann who has taken the time to work with me from the beginning looking for ways to integrate my work at the FDC, Kris who was always interested in what I had going on and finding opportunities for me to support the FDC, Sameer for his continuous interest and support and for donating his time to Experience Your Future, and Brian for his constant enthusiasm, leadership, and for always being ready to give a hand. I want to thank Henry Girolamo of the Natick Soldier Center for taking me under his wing the minute he met me. I am grateful to Dr. David Allee for a world class education in VLSI design, and for taking the time out of his very busy schedule on several occasions to show me

what he was working on and allowing me to give my feedback, he has never hesitated to offer his support. I also want to thank Jacki Healy for her dedicated support to America's soldiers, Stephanie Furlanetto for her cheerful and enduring support, and Shawn O'Rourke for his support and dedication in addition to giving his time to participate at Experience Your Future, and Nell Lawrence.

In appreciation for those who selflessly gave their support as I worked my way into this program, I want to especially thank Wilbon Davis of Texas State University, who serving as a mentor and a friend had the courage and generosity to endorse me as I strove to enter the program, and whose sustained interest and support provided me with a series of valuable check points over the course of my education. I am particularly indebted to COL Mike Zarbo, a mentor whose faith in me exceeded my own as I worked my way into this advanced education program, for his enthusiastic endorsement during the application process, and throughout my tenure at the Army Project Executive Office for Simulation Training and Instrumentation (PEO STRI). I'm equally indebted to COL (Ret.) James Ralph and Dr. James Blake (COL USA, Ret.) for their endorsement and for their mentorship at PEO STRI. I am uniquely indebted to Martha Vander Berg, who as my academic advisor at ASU has been an outstanding and trustworthy guide and whose intimate involvement with my progress since my first contact with ASU has frequently had a considerably positive impact on my life. I'm also deeply grateful to Dr. Sethuraman "Panch" Panchanathan whose support during my first visit to ASU, the application process, and throughout my time at ASU is deeply appreciated. I want to thank the Uniformed Army Scientist and Engineer (UAS&E) affiliated program staff who ushered me in and forward, LTC (Ret.) Aaron Brown and CW4 (Ret.) Jo Ann Wright at the US Army Research Development and Engineering Command (RDECOM), Paula Bettis, Cece Latimer-Bridges, and Linda Smith at Army Human Resources Command (HRC), all of whom would never hesitate to address an issue, and Paula in particular for whom I was probably a special project, Yvette Evans

at the Defense Finance and Accounting Service (DFAS) who was always more than helpful, even when the accounting system seemed less so.

From PEO-STRI, I owe a great deal to Jim Kay who as the chief of military personnel forwarded the Acquisition Support Center newsletter with Paula's announcement about the UAS&E Ph.D. selection board that started all of this. I'm thankful for the brief time I was able to work with Dr. Bill Swearengin who as I was applying for the advanced degree program was completing his long running attempt at a Ph.D. in computer science, it was he who first suggested that I fly out to Arizona during the application process and who provided me with endless three decade old stories about sleeping under work benches waiting for cart-loads of punch cards to come back down the hall with pages of cryptic error messages from the mysterious mystical unseen *computer*. Thank you to my other long suffering colleagues at PEO-STRI with whom I shared duties at the Joint Readiness Training Center (JRTC), for their support an encouragement as I was inching my way into this program, LTC (Ret.) Pat McManamon, a glorious hybrid of mentor and clown, CW4 (Ret.) Matt Brothers, who I have to describe as my JRTC brother in the truest fraternal sense, Henry Lastra, who instantly took me under his wing on our first meeting at JRTC, and continued his interest and support throughout my time in Orlando even as he rapidly took on additional responsibilities, COL (Ret.) Joe Just, a mentor and a world class car dealer, LTC Charlie Stein, a great sponsor and mentor, SGM (Ret.) Duncan Hurst, LTC (Ret.) Tom Willmuth, and Dr. Bo Terry.

I am deeply thankful to my classmates and peers at ASU who in some cases have carried me through obstacles I could never have navigated alone. I will always have a special appreciation for my fellow Computing Systems Research Lab troopers, most notably those who have at one time or another shared in the pain, an experience with a tendency to bring people closer: Glenn Leary, Sushu Zhang, Nipun Java, Krishna Metha, Weijia Che, Chris Ostler, Amrit Panda, Nik Gadghe, Jyothi Swaroop,

and Ani Kadne. I am compelled to pay special tribute to those among my classmates with whom at one time or another I have shared a special burden, in particular Glenn and the Kosovar, Betim Deva, without whom something would have had to give, as together they represented fully two thirds the total membership of at least five separate, significant, and arguably successful project teams with simultaneous deadlines, even if all the while Betim was quite literally stirring an international incident about which I can add nothing more. I would like to acknowledge Amrit, Nik, and Ani for taking on part of my burden, as well as M. Muqsith with whom, like Glenn, I have succeeded and suffered more or less in parallel since my arrival. I'm equally grateful to Viswesh Parameswaran, Arun Kannan, Sudhee Kadri, Deepa Kannan, Aaron Williams, and Sam Leshner, all of whom have contributed measurably either to the success of this work or otherwise in my ASU career.

I want to thank my US Army Student Detachment (USASD) and UAS&E comrades at ASU, CPT Brent Odom, MAJ Kwane Welcher, LTC Joseph Hitt, MAJ Matt Rogers, and 1LT John Orsini for the mutual support at ASU and for never failing to give me something to chase during our semiannual runs. I am also deeply appreciative to the USASD personnel for their dedicated support.

I would also like to offer my thanks and appreciation to individuals at ASU, ARL, and beyond who over the course of my time here have directly contributed to my growth and learning, or through their selfless support, interest or participation in my work have left a deep and lasting impression: From ARL and RDECOM I would like to thank ARL Director John M. Miller for his mentorship and interest since my arrival at ARL, Dr. Paul Amirtharaj, Dr. Gary Wood, John Eicke, Tom Bower, LTC Fred Ludden, LTC Victor Nakano, LTC Kathy Moses, LTC Marjorie Grantham, LTC Keith Harvey, SFC Jemall Pittman, Tracie Dean, Christina Musse-Torres, Paul Scanlon, Andrew Ladas, Sharon Sanchez, Michael Pollock, SSG Travis Privott, SGM Timothy Weatherspoon, and SGM Steven Hornbach. From ASU I am grateful to Dr. Goran

Konjevod, Dr. Hessam Sarjoughian, Dr. Chaitali Chakrabarti, Dr. Guoliang "Larry" Xue, Dr. Rida Bazzi, Dr. Hasan Davulcu, Dr. Charlie Colbourn, Dr. Yann-Hang Lee, Dr. Donald Miller, Debra Calliss, the CSE capstone design teams, notably team leaders Craig Teegarden and previously mentioned Brian Hawley, the Innovation Space design teams, Scott Lapora, Lincoln Slade, Araxi Hovhannessian, Audrey Avant, Teresa Chai, Lee Reynolds, Wayne Woodland, Monica Dugan, and beyond ASU, Dr. Nikil Dutt of the University of California, Irvine and Dr. Sri Parameswaran of the University of New South Wales in Australia.

I would like to directly acknowledge the RDECOM leadership, without whom this educational opportunity would not exist. In particular, BG R. Mark Brown whose support and tenacity are arguably the reason there was such a program, MG Fred Robinson who took time to offer me guidance, MG Nick Justice and BG Harold Greene, who continue to support those few Army servicemembers in this program and who have both been mentors to me during my time at ARL.

I am uniquely indebted to the Unites States Army which has repeatedly exhibited courageous optimism about my potential and continually forced me to become a better person in ways I am convinced no amount of experience in my alternate civilian life could have approached. I am supremely appreciative of the Army for the incredible, often priceless, opportunities that have been made available to me, and for the support I have consistently been given throughout my career.

I am eternally indebted:

To my family, the fantastic clans woven together to whom I owe so much, for the enviable network of love and support I have been blessed with and for the foundation that it gives all of us.

To my irreplaceable true friends and brothers who, because you have always been there, have an understanding that rises to a level as yet unknown to the social sciences, you hardly require my thanks, but you absolutely deserve it.

To my Dad Aaron and my Mom Sandra, you have literally provided everything there is of me to offer, for your unqualified love and support, for trusting me with my own path from the beginning, and for always being there.

To 1LT Thomas Edison "Tommy" Flanders, a Marine, a Soldier, a brilliant engineer, a quintessentially American personality, for inspiring me to seek understanding, to you I substantially attribute whatever curiosity and ability I might have within me to successfully approach a problem.

To COL William Moss "Tex" McVeigh, an Airman, a combat veteran, and a world class leader, for your advocacy, faith, and inspiration, to you I substantially attribute whatever ambition I have had to seek challenges and to choose the harder right over the easier wrong in all things.

To CPT Park Ray Baker, an Airman, a combat veteran, for your service and your legacy, you have influenced me in ways that possibly only those who knew you can understand, in ways that I can never come to know.

To Joshua Paul Baker, for brotherhood, the shared experiences that substantially defined us both as brothers and men, perspective, and the implicit requirement for Man's eternal soul such that we may meet again.

Most significantly, I must express my infinite debt and gratitude to my wife Bethany, love of my life, for your eternal faith, immeasurable inspiration, enduring foundation, irreplaceable motivation, unearned dedication, inordinate sacrifice, vital curiosity, immortal grace, boundless humanity, incomparable beauty, and love as it propels me. Words are wholly inadequate for expressing my admiration, appreciation, and love, but there it is.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

ENIAC, "the world's first operational, general purpose, electronic digital computer," was commissioned by the United States Army for the purpose of computing artillery trajectories [118]. Solutions tediously calculated with pencil and paper by scores of engineers during the Second World War could suddenly be produced with astonishing speed and accuracy thanks to electronic digital processing.

Those original machines weighing in at several tons and dependent on immense power budgets have steadily given way to ever smaller, lighter, faster, and more energy efficient machines capable of solving a myriad of previously unimaginable problems. In addition, such machines in the embedded domain are capable of performing such tasks in the harshest possible environments under extraordinarily demanding constraints. The processing power of those original massive ballistics calculators has been dwarfed by processing systems literally embedded inside individual artillery rounds today.

The importance and pervasiveness of Embedded computing systems have exploded in recent years. Embedded systems have become nearly ubiquitous in everyday objects, and demands on computational power in such systems continue to rise. Today, we commonly see high performance signal processing and complex control systems present in everything from automobiles to mobile phones. Handheld devices once designed to perform niche tasks such as mobile phone service, photography, voice and video recording, music and video presentation, navigation, portable gaming, and personal data management have moved substantially toward converging into a single multifunctional device.

As the complexity and computational demands on such systems increase, a need for improved system architectures such as multiprocessor systems on chip (MPSoCs)

and improved power and performance techniques continues to grow. The motivations for developing power and performance enhancements in embedded systems are easy to highlight when considering embedded applications in military operations.

In the context of artillery projectiles, ballistics information once processed well in advance and based on necessary assumptions about atmospheric conditions and even the Earth's rotation can today be carried out in flight for dramatic improvements to accuracy and precision. The task of remotely engaging hostile targets on the battlefield has gone from what was once truly Napoleon Bonaparte's art to a genuine science. What once required barrages, with today's technology, can be accomplished with a single round.

The role of digital computation on the battlefield has grown well beyond the artillery tables motivating early digital computers. Nearly every facet of military operations depends on intense embedded computation. While sophisticated aircraft and naval systems have traditionally dominated the military embedded systems domain, the presence and sophistication of embedded systems in ground based equipment has dramatically increased over the past two decades. Everything from Battle Command Systems, communications, radar, sensors, guided ordinance, and deeply integrated unmanned and autonomous systems depend on the abilities of resource constrained embedded computational systems.

Overcoming power limitations in such systems is paramount since any form of energy comes at a premium on the battlefield. Logistics capabilities are a fundamental bottleneck in any operation. Limited transportation resources and often dangerous conditions on important supply routes means that every ounce of logistical payload can come at extraordinary cost. For the US Army in Afghanistan, estimates of the fully burdened cost of getting a single gallon of fuel to the end user in combat are as high as $1000 [102] – literally $1,000/gallon.

Power to run electronic devices comes from petroleum dependent electric generators or batteries, both of which are capable of heavily burdening supply chains at all levels. Such costs are evident in remote areas of Afghanistan where hostile activity and limited transportation routes, which are also extremely difficult to secure, can require helicopters for any successful resupply effort. If an assumption can be made that embedded computing systems are here to stay, this problem provides ample motivation in the search for solutions to the energy cost problem in embedded computation.

1.1 Addressing Energy Resource Constraints

The power supply problem in embedded systems may be addressed through both power and performance optimizations. The *energy efficiency* of a computational system, given in energy per unit of computation, imposes a constraint on the amount of achievable computational throughput in terms of available energy. Power and performance optimization enables increased workloads under fixed energy constraints or fixed workloads under more restrictive energy constraints, indicating improved energy efficiency.

Research aimed at improving energy efficiency in embedded systems covers a broad array of principles and techniques at every level of computer design abstraction. Significant areas of research include device physics, improved lithographic techniques, power and signal distribution design and materials, novel logic circuits such as threshold logic, reduced complexity techniques for basic arithmetic operations, power aware architectural layouts etc. At higher levels of abstraction, we see techniques including power and clock gating, low energy state management, voltage and frequency management, multicore and distributed architectures with data and functional parallelization techniques, programming and compiler techniques such as minimizing memory access and other types of overhead, operating system scheduling techniques. System level techniques include power aware, user and context aware power management schemes,

and other improvements such as the introduction of improved batteries and power supplies and low power peripheral device technologies.

In this work power and performance optimizations center on two primary approaches. The first approach supports *power aware* systems introduced in the following section. The second, larger approach, centers on effective utilization of *multiprocessor systems on chip* (MPSoCs) which have become the dominant architectures in response to the power/performance barrier experienced in microprocessor design in the last decade. The MPSoC paradigm is discussed further in Section 1.1.2.

1.1.1 Power Aware Systems

Consider a situation where a portable multimedia device user wants to watch a video program for the duration of an airline flight, but remaining battery power only affords video for half the duration. Under these circumstances users are willing to sacrifice Quality of Service (QoS) in exchange for increasing the time the service is available. Power Aware applications provide users the flexibility to exercise this type of trade-off.

The concept of Power Aware devices as presented by Lian et al. [100] asserts that a device such as a handheld computer or smart phone should be able to choose an appropriate power saving mode by direct user input, based on the user's history, or automatically by sensing the environment. In essence, an array of system performance modes enable device users to prioritize certain kinds of behavior in exchange for improved power performance. Examination and implementation for two such schemes are presented in the following chapters. Brief introductions are given below in Sections 1.2.1 and 1.2.2.

1.1.2 Multiprocessors

Historically, integrated circuit complexity has marched upward largely in accordance with Moore's law describing the geometric annual doubling of the number of transis-

tors on an integrated circuit due to two dimensional downscaling of feature sizes [117]. Thanks to increasing architectural complexity in conjunction with increasing clock speeds and reduced operating voltage, processor performance had followed a similar trend before hitting a *power/performance barrier* in the last decade. The primary contributing factors in the emergence of this barrier have been power and thermal density, and submicron device subthreshold leakage current. The increasing performance and density of transistors suggests that power density, or unit energy per unit time per unit area, would reach a point at which heat generated due to energy dissipation could not be removed from the chip fast enough to maintain a stable operating temperature. Additionally, contributing to this problem was the increasing contribution of transistor subthreshold leakage current to overall chip power consumption. Not only were power densities becoming unsustainable so was overall power consumption.

To overcome the power/performance wall, it is necessary to consider the relationship between power, P, operating voltage, V, and operating frequency, f. This relationship can be given by the approximation $P = C_{switching} \cdot V^2 \cdot f + V \cdot I_{leakage}$. Where $C_{switching}$ describes capacitances in the circuit which are charged and discharged during operation and is associated with *active* power consumption, while $I_{leakage}$ describes the sub-threshold leakage current which characterizing the transistor technology and is associated with *passive* power consumption. Processor performance can be increased by increasing the system clock frequency, but higher processor frequencies require increasing the operating voltage since circuit capacitances must be charged and discharged more quickly. In effect, increasing clock frequency results in a cubic increase in power consumption and is consequently not viable once the power/performance barrier is reached. Increasing performance through other means, however, can provide the necessary conditions for surmounting the power/performance wall.

Thanks to the massive numbers of transistors available in integrated circuits, it is possible to place multiple or even many processing elements (PEs) on a single

chip. Ideally, while running the multiprocessor system at a fixed frequency and voltage, performance will scale linearly with the number of processors added to the system while increasing power requirements only linearly. Since increasing clock frequency to achieve the same performance gains would result in a geometric increase in power consumption, the multiprocessor system gives improved performance while circumventing the power/performance wall. This principle has driven the recent dramatic rise of multicore and parallel processing solutions. Given appropriate implementation from both the architecture and programming perspective, multiple smaller and simpler computational cores can outperform a single core processor in terms of power, throughput, and latency, while operating at slower, less power hungry voltages and frequencies.

The MPSoC paradigm takes advantage of the huge number of transistors available in state of the art integrated circuits to implement multicore processors and multiple system components on the same chip. Processing elements, application specific cores, data networks, routers, memory elements, and a litany of other components are regularly integrated on the same chip, often reducing circuit area requirements and eliminating off-chip communication latencies and power considerations which would otherwise be required. The MPSoC is thus a logical design for embedded systems where chip size, power, and efficiency are important.

The primary difficulty garnering significant attention with multiprocessors in general arises from the need to efficiently program them. Historically dominant single core programming schools produce code which can be difficult or impossible to map efficiently onto multiprocessors. The problem can be loosely described in terms of Amdahl's law regarding the parallelization limits of programs. Multicore speedup is limited by the non-parallelizable portion of the code [9]. The problem for automating multicore programming arises from the need to identify which portions are sequential, then separate and evenly partition the remaining work across available PEs. Addressing the mapping of a complex codebase, automated program mapping, and power and

performance efficient implementations of programs for arbitrary MPSoCs are the motivating problems addressed here. Contributions addressing these problems are briefly introduced below in Sections 1.2.3, 1.2.4 and 1.2.5.

1.2 Contributions

Overviews of the contributions presented in this work are given in the following sections. Detailed presentations are found in the subsequent chapters.

1.2.1 Power Aware QoS Managemet for Bistable Displays

At the system level, we can address our limited energy budget by identifying the most power hungry system elements and finding ways to improve their efficiency. One important subsystem commonly seen in handheld embedded devices is the liquid crystal display (LCD). LCDs have long been dominant in a myriad of handheld embedded handheld devices and military applications such as the Handheld Terminal Unit [53], but they can consume as much as 60% of the system's entire power budget [36]. New technologies such as bistable electrophoretic displays (EPDs) can significantly reduce power requirements. In certain applications, such as electronic books, EPD power consumption can be many orders of magnitude less than LCDs. In addition, the bistable nature of the display presents opportunities which can be exploited in power aware systems. In Chapter 2 power benefits of EPDs are addressed along with a display driving scheme which significantly reduces power consumption and enables power aware system management by taking advantage of the display's bistable memory properties at video frame rates where the EPD power advantage is otherwise diminished [17].

1.2.2 Power Aware QoS Management in H.264 Decoding

Digital signal processing (DSP) applications are among the most important in embedded systems. Most video, imaging, communications and other sensor applications are extremely dependent on efficient DSP implementations. A particularly important ap-

plication is video decoding needed to display and analyze encoded video streams used for reconnaissance, surveillance, and communications. H.264 is a state of the art video coding and decoding (codec) standard which gives a substantial compression advantage over previous standards. High compression enables significant improvements in video quality when transmitted over bandwidth limited wireless networks, such as those used to download data from unmanned aerial vehicles (UAVs), thanks to reduced bitrates. In Chapter 3 a power aware supporting scheme enabling users to trade quality of service for increased throughput and reduced power consumption in the H.264 CODEC is presented [16].

1.2.3 Low Power High Performance Scalable Multicore Video Decoding

The power/performance barrier introduced in Section 1.1.2 is addressed, at least in part through the transition to MPSoCs, where the computational workload is spread across multiple, often reduced complexity, processing elements running at reduced clock frequencies. However, as previously discussed, programming for this system has introduced new problems as well as new opportunities. Chapter 4 describes a multicore implementation of the H.264 video decoder on the IBM Cell Broadband Engine (CBE) designed to scale with increasing processor array sizes in multicore SoCs. In addition, a novel multicore dynamic frequency and voltage management scheme in introduced and examined on the CBE [14].

1.2.4 Instruction Mapping in Scratchpad Memories

In addition to moving toward multicore chip designs, the introduction of reduced power memory structures such as scratchpad memories can help to reduce overall system power consumption. SPMs have reduced physical complexity when compared with cache memory, but they also hand the burden of memory management, once conducted in hardware, over to the programmer or compiler. The additional burden for program-

mers can be significant. A lightweight heuristic for constructing instruction mappings in SPMs is presented with the H.264 analysis in Chapter 4. A more thorough investigation of instruction mapping techniques in addition to more advanced heuristics are further presented in Chapter 5 [15].

1.2.5 Lightweight Run-Time Stream Scheduler

As the number of cores available in multicore SoCs grows, the problem of extracting parallelism from applications and efficiently mapping them onto many cores in order to maximize processor utilization becomes increasingly difficult. Stream programming is a paradigm designed to aid programmers in explicitly exposing parallelism in their applications to the compiler. Unfortunately the proliferation of multicore architectures and their often independent programming models means that sophisticated compiler tools and developer effort are still required to successfully implement a single application on two or more different hardware configurations.

The problem is exacerbated by the coming need for multitasking multiple applications with user-defined priorities onto a multicore processor. Availability of hardware resources including the number of processing elements will change dynamically at runtime, suggesting that all possible configurations must be addressed if a compile-time approach is used. Efficient algorithms for generating and managing such mappings, particularly run-time implementations are still an important research area. In Chapter 6 a lightweight runtime stream scheduler is presented addressing these issues. The basic requirements for an automated multicore model are laid out, a high performance solution for mapping applications onto available resources is presented, and simulated performance as well as an implementation and evaluation on the Cell Broadband Engine are examined [13].

Chapter 2

POWER AWARE QUALITY OF SERVICE MANAGEMENT IN ADVANCED BISTABLE DISPLAY TECHNOLOGY

Today displays represent a significant fraction of the power required by common handheld portable computing devices. Displays often account for anywhere from 30-60% of the power consumption in these devices, primarily due to backlight power consumption in the ubiquitous Liquid Crystal Displays (LCD) [36]. A typical QVGA display in a handheld device may consume 220mW of power, 91% of which is directly consumed by the backlight [121]. Maturing alternative technologies such as bistable Electrophoretic Displays (EPD) offer promising possibilities for significant power savings by eliminating the backlight and providing a zero power capability when displaying a static image.

A smart driver concept is introduced here for next generation full motion and color EPDs taking advantage of the image stability these displays offer even when the device is turned off, effectively making the display itself a form of static memory. The smart driver offers substantial power savings of between 30% and 50% of display switching power while displaying video when compared to a naive driver without loss of quality by selectively updating only portions of the image which change from one frame to the next. We also introduce a more aggressive lazy driver which seeks to improve power savings even further albeit in exchange for video quality. Both methods are analogous to inter-frame compression used in common video compression algorithms. As a suite of operating modes the smart and lazy drivers enable an array of options to power aware systems working to extend resource availability based on the user's needs.

A great deal of effort has gone into optimizing power consumption in LCDs [57], and as battery powered computing devices become increasingly powerful and pervasive, that effort continues to grow in importance. As lower power display tech-

(a) Detail of colloidal suspension capsules

(b) electrophoretic display on plastic

Figure 2.1: Capsules containing colloidal suspension in an EPD display.

nologies like EPDs become more capable, the effort to optimize those technologies for use with handheld devices will also increase. Currently EPDs are the technology of choice for electronic paper due to the small amount of energy required, very high contrast, and suitability for use in flexible displays. As EPD technology improves, this type of display will increase in importance. With handheld computing devices growing more powerful and more versatile, they will require larger lower power displays than those found in portable devices today. Imminent improvements to full color and full motion capability in EPDs justifies looking into optimization of these devices under those conditions. EPD switching power consumption is not directly evaluated in comparison with LCDs here; however, we find that the power consumed in an EPD with a refresh rate comparable to an LCD displaying full motion video are of the same order. The 9% of the LCD's power budget which goes into updating and maintaining the displayed image is comparable to the total energy consumed by an EPD in the same environment.

2.1 Previous Work

A great deal of effort has gone into optimizing power consumption in LCDs [36] [40] [57] [106] [107]. As battery powered computing devices become increasingly powerful and pervasive, such efforts continue to grow in importance. As lower power display technologies like EPDs become more capable, the effort to optimize those technologies for use with handheld devices will also increase.

Electrophoretic display technology has been studied for decades. Dalisa [49] and Hopper et al. [69] characterized and examined the performance and manufacturing considerations in such displays in the 1970's. More recently, as EPDs began finding commercial success and feasible manufacturing methods, there has been an increased interest in these displays. Comiskey et al. [45] describes an electrophoretic ink and its manufacturing process leading to the founding of Eink Corp., the leading manufacturer of electrophoretic ink. Inoue et al. [79] and Takao et al. [146] present EPD display systems. Previous EPD models are presented by Vermael et al. [153] as well as Hopper et al. These models lay the groundwork for the power model used in this work.

2.2 EPD Technology

EPDs use the electrophoretic force of an electric field on charged pigment particles in an encapsulated colloidal suspension to alter the appearance of a display image. When charged pigment particles are forced to the front of the display though electrophoresis, the capsule appears the color of the pigment particles. Two color particle displays use capsules like the one illustrated in Figure 2.2a. Here particles of opposite charge and color are switched between the front and back of the display to produce pixel color. This process can produce color and contrast comparable to printed paper.

An important distinction between LCD and EPD technology is the concept of bistable pixels. In a bistable display, individual pixels are stable in both "on" and "off" orientations so that an image once established on the display will remain there, even if the display is turned off. This is an extremely useful property, which is particularly interesting when power conservation is important. LCDs must be refreshed continually in order to maintain an image, even if the image is constant, while an EPD may retain an image for months after the device has been turned off. It is this distinction which drives the concepts presented in this work.

2.2.1 Advantages

EPDs have a number of properties which are highly desirable in portable battery powered devices. High contrast in ambient light eliminates the requirement for a backlight found in all transmissive LCD display devices. The display's purely reflective nature also makes it suitable for use with night vision devices, an application for which emissive display technologies are poorly suited. The bistable nature of EPDs means that once an image is drawn on the display, there is no need to refresh or even power the display until a new image must be drawn. For static viewing applications, such as text viewing, once the image is drawn the display might be powered down for minutes, hours, or even days until the viewer is ready to go to the next page. This bistable property can mean substantial power savings in handheld devices. Figure 2.1 also demonstrates that display resolution is not limited by microcapsule size. Image resolution is actually dependent only on the size or shape of the electrodes in the backplane.

2.3 Power Characterization of EPD at the Subpixel Level

2.3.1 Electrophoretic Capsule

In order to characterize the power consumed by these next generation displays, it is necessary to understand the amount of energy required to move a pigment particle in the colloidal solution across an EPD capsule and the amount of leakage current we should expect to occur in the capsule. The physical characteristics of EPD capsules are based on contemporary examples of EPD display technology. Table 2.1 lists the fundamental properties chosen for power characterization. Each display pixel is divided into three subpixels corresponding to the RGB color components in the display. Power is characterized at the subpixel level because each color component or subpixel has its own data value and storage capacitor.

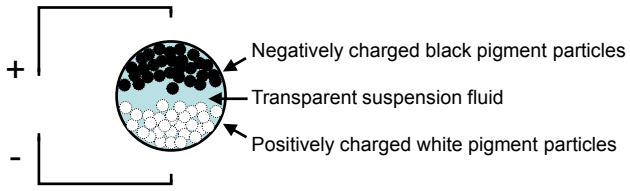
Table 2.1: Simulated EPD attributes.

	Value	Unit
pigment particle radius (r) [22]	0.5	μm
pigment particle charge (q) [22]	4.8E-16	Coulomb
microcapsule diameter [79]	50	μm
supply voltage [153]	15	Volts
suspension resistivity [49]	1.0E12	Ωm
particle concentration [153]	2E16	part./m^3
microcapsules/subpixel [79]	6	capsules

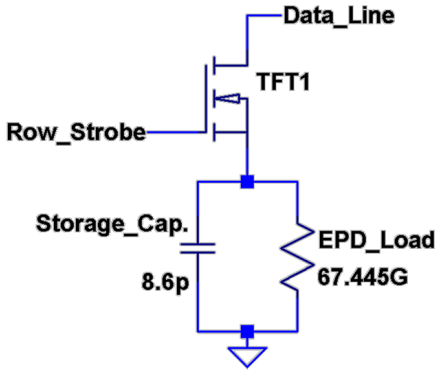
Capsule Switching Power The physical motion of charged pigment particles accounts for most of the power actually consumed in each microcapsule. Here a method for calculating capsule switching power is introduced. For simplification, the amount of power consumed during particle motion is considered to be a constant dependent upon the velocity of the particles as they transit the capsule. Particle motion inside the capsule can be represented as a laminar flow, and the time required to establish laminar flow after the electric field has been applied is very small with respect to the switching timescale, even when we are considering full motion video at 60Hz. In [45] this time is calculated at 55ns, while the frame-write period at 60Hz is approximately 17ms. Here the velocity of each particle is considered as it traverses the capsule, assuming the particle travels the diameter of the capsule anytime the capsule is switched. The velocity is determined by the distance which must be covered by the particles in the amount of time it takes to write one frame at the driving frequency. From the particle velocity v in meters per second, the applied voltage V, and the capsule diameter in meters d, the mobility μ of the pigment particles in solution may be calculated [45] :

$$\mu = v/E, \text{ where } E = V/d \tag{2.1}$$

In defining the $15V$ driving voltage, capsule diameter, particle radius, and particle charge, fixed properties of the next generation EPD must be considered. In order to achieve the required mobility, it is necessary to identify the viscosity of the sus-



(a) EPD capsule with a transparent electrode at the top making the black pigment particles visible to an observer above after a voltage is applied as shown.



(b) EPD capsule model with TFT gate and component values simulated in LTSpice [101].

Figure 2.2: EPD capsule physical and electrical characteristics.

pending fluid. Here we can manipulate the equation for mobility in [45]. Applying the charge per pigment particle in Coulombs and the radius of the pigment particles in meters we are able to find the viscosity needed to achieve switching times necessary for the required frame refresh rate:

$$\begin{aligned} \mu &= q/12\pi r\eta \\ \eta &= q/12\pi r\mu \end{aligned} \tag{2.2}$$

Using the values V = 15 volts, $d = 50\mu m$, $r = 0.5\mu m$, $r = 0.5\mu m$ [22] and a frame refresh rate of 60Hz, it turns out that we must have a viscosity no greater than 0.00255 Ns/m^2 or 2.55 cP, which is much smaller than value used in [79].

To find the power consumed through physical movement of a single pigment particle, we can determine the force acting on the particle in Newtons: N = qE where

Table 2.2: Subpixel power consumption.

	Watts
Steady-state power consumption due to electrophoretic particle motion	3.24E-9
Capsule leakage power	8.84E-13

q is the particle's charge in Coulombs, and E is the electric field inside the capsule as described earlier. The work performed on each particle is this force multiplied by the particle's velocity, in this case 3.00E-3$m/s \cdot$ 1.44E-10 N = 4.33E-13 W per pigment particle. Using the pigment particle concentration of $2 \cdot 10^{18} m^{-3}$, and multiplying by the volume per capsule we have approximately 1300 particles in a capsule. It is not necessary to distinguish between the positive and negatively charged pigment particles for a two particle EPD in this calculation, as the work in either case is indistinguishable. To calculate the power consumed at the subpixel level, this number is multiplied by the number of particles per capsule, and the number of capsules per subpixel to find 3.24E-9 W per subpixel during switching.

Capsule Leakage Power As long as a voltage is applied across the capsule, a small amount of leakage current will flow through the fluid dependent on the resistivity of the colloidal solution in the capsule. Typical resistivities should be $> 10^{12} \Omega \cdot$ cm [49]. Leakage power is calculated for each capsule based on resistivity ρ of the colloidal solution. The height h and radius r of the capsule are used to determine its resistance:

$$R = \rho h / \pi r^2 \tag{2.3}$$

Capsule leakage power is found using the capsule's resistance and the operating voltage: $P = V^2/R$. Multiplying this value by the number of capsules per subpixel we get a leakage power of 8.84E-13W per subpixel. Since this value is four orders of magnitude smaller than the power consumed due to particle motion, it can be safely ignored when calculating subpixel power consumption.

2.3.2 Storage Capacitor

To achieve a frame rate acceptable for viewing video, the storage capacitor shown in Figure 2.2b stores the energy needed to switch a display subpixel between two states, such as black to white or white to black. With a storage capacitor, the display driver can quickly write a row of pixels by charging the capacitor, and then move on while the capsules are driven by the capacitor. The capacitor is charged during the row-write operation, and the particles in the EPD capsules complete their motion under the electric field provided by the charged capacitor during the period of one frame-write operation. At 60 Hz, the frame-write period is approximately 16.7ms. The row-write period is determined by the number of pixel rows in the display. Here 320x240 QVGA is used, where 240 is the number of rows in the image. Dividing the frame-write period by the number of rows gives us a row-write period of 0.0694ms.

The storage capacitor must charge during the row-write period, and then maintain the switching voltage across a subpixel through one frame-write period while performing the work required to switch the subpixel in the same period. The amount of energy lost from the capacitor during a frame-write period is calculated from the amount of work per subpixel derived earlier multiplied by the amount of time the work is performed, in this case the frame-write period: 3.24E-9W $\cdot$ 1.67E-2s = 5.23E-11j. Using the spice model shown in Figure 2.2b, we can calculate the amount of energy lost in the storage capacitor based on the voltage drop during the frame-write period. In the presented model, with an 8.6pF storage capacitor, the subpixel loses 0.47 volts from 14.53V when the capacitor is charged to 14.06V at the end of the frame-write period, expending 5.78E-11j.

The total power consumed in writing to each subpixel is completely dependent on the size of the storage capacitor. At the beginning of each row-write cycle, the storage capacitor is shorted to ground to discharge the remaining energy. This is done

because the image information stored in the associated subpixel is assumed to be stable. The next pixel value is also likely to require the opposite charge used to attain the current state, in which case discharging the capacitor reduces the energy required by the driver to charge the capacitor to the opposite polarity [106].

2.3.3 Thin Film Transistor (TFT)

EPD drivers must use active matrix pixel addressing since the electrophoresis of the pigment particles does not have a threshold voltage. Passive matrix addressing will invariably affect an entire row and column when a single pixel is addressed [49] [69] [146], while active matrix pixel arrays are switched at the pixel level using a TFT to protect pixels from signals intended for other rows. The EPD driver writes a row of display data at a time by applying the appropriate row image voltage values to the columns in the addressing matrix and applying a separate row strobe to turn on the TFT switches for each row in sequence [106].

2.4 Improved Driver Concept

2.4.1 Naive Driver

Standard display drivers update every pixel on the display each time the frame is refreshed. Constantly refreshing pixels is important to the operation of LCD displays because after a charge has been applied to place an LCD pixel in the desired state, the pixel immediately begins moving back to its quiescent state. As a result, LCD displays must be refreshed constantly even if the image displayed remains static. Similarly, the naive EPD driver updates every pixel in the display according to the frame refresh rate.

2.4.2 Smart Driver

Without any requirement to refresh a static image, the display driver need only write the image once. An image might remain on the display with the display driver and even the display itself turned off for days. This property enables some possibilities for smarter

ways to update moving images. When displaying a video with an EPD, a smart driver would know that it need not update the display faster than the frame rate of the video displayed. Such a scheme results in immediate power savings, as a display capable of updating at 60fps showing a video at 32fps instantly conserves energy by only updating the display when a new frame arrives at half the native rate with no loss of performance.

Common user interfaces present largely static screens such as a user desktop. Interacting with the display interface frequently results in small changes to the displayed image as is the case with drop-down menus and mouse pointer motion. Only a very small fraction of pixels actually change from one frame to the next. In this case a smart driver might very efficiently reduce the amount of work performed at each frame refresh by updating less than 1% of the pixels during each refresh. This concept extends to full motion video and is analogous to inter-frame compression used for MPEG video. Even at 60fps, some fraction of the pixels on the display will likely remain unchanged from one frame to the next. The smart driver can detect these static portions of the image by comparing RGB values on the display to those in the new frame, and choosing not to address them when the new frame is written. While choosing to update only pixels that have changed from one frame to the next, the display driver will conserve energy without any loss of image quality in a bistable display.

2.4.3 Lazy Driver

The concept of a smart driver for a bistable display can be taken a step further. If power conservation is paramount, the user might be interested in enabling power aware management features and sacrificing image QoS for power savings. A lazy driver can further reduce power consumption by loosening the definition of a pixel which is unchanged from one frame to the next. The lazy driver might only compare the most significant part of a pixel's color value and then only update pixels which have changed by some threshold amount before bothering to update them during a frame refresh. For

example, the color value of a pixel is represented with a 24 bit number. The three bytes in this number represent the three primary colors in the order red, green, and blue. The lazy driver might compare only the most significant 7, 6, 5, or 4 bits of each byte instead of comparing all 8 as in the smart driver to determine whether a pixel should be updated or not. In effect by doing less work to compare the image on the display with the next frame to be displayed, the lazy driver further conserves energy by updating fewer pixels depending on how many bits were compared. The lazy driver might produce significant power savings over a smart driver, but it also degrades the image quality of the displayed video. Image degradation can increase significantly as the number of bits compared decreases.

2.5 EPD Smart Driver Simulation

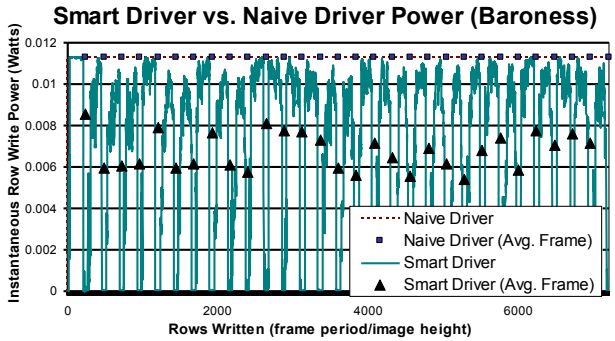
2.5.1 Bistable Display Simulator

In order to characterize the potential power savings achieved by upgrading the bistable display driver, a simulator in Java has been developed which takes as its input a sequence of bitmap images extracted from video benchmarks. Each image is a frame from the original video.

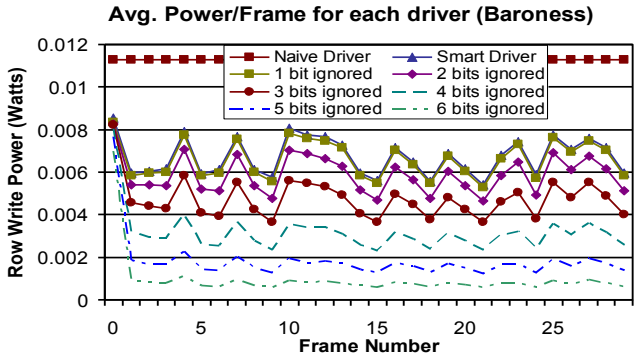
Seven QVGA video segments were extracted from MP4 files provided by Elecard Ltd. [104]. The simulator calculates the power consumed by a naive driver updating every pixel of every frame, the smart driver which updates only pixels that differ from one frame to the next, and six iterations of the lazy driver which compares only the most significant bits of each subpixel value for the next frame with the same bits from the image currently on the display.

2.5.2 Simulator Power Characterization

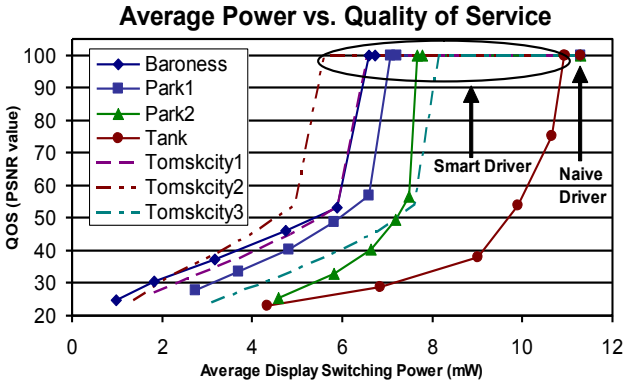
The display is modeled such that the elements in Figure 2.2b represent a subpixel, or a single RGB component color of each pixel. Power is calculated separately for each



(a) Smart driver reduces power consumption by 40% over the naive driver for *Baroness*.



(b) Simulated power consumption for each driver configuration over 29 frames of *baroness*, here the lazy driver produces power savings ranging from 42-91%.



(c) Average video power consumption vs. QoS for each video (line) and each driver configuration (point), all videos consume constant power using the naive driver.

Figure 2.3: Smart driver power performance.

subpixel according to the figures in Section 2.3. The total energy added to the row of capacitors during a row write is used to determine the power consumed during a row write operation. At the beginning of each row write, the storage capacitor is shorted to ground, quickly discharging any stored energy.

The continuous power consumed by the display is determined by the series of row-writes which occur sequentially as each frame is written to the display. The row power is calculated based on the total number of pixels written. The rate of power consumption over the total row during one row write period represents the power during that time period. The total power calculated for the next row represents the power consumed during the next row-write period. When the end of the frame is reached, the next frame begins with the first row.

Power consumption is calculated for every pixel which is written during a frame-write. In the case of the naive driver, every pixel is re-written. The smart driver only updates pixels when any 8 bit RGB component in the next frame is different from the current displayed image. The lazy drivers only compare the first $8 - n$ bits of each RGB component for each pixel, where n is the number of least significant bits ignored during the comparison.

2.5.3 Smart Driver vs. Naive Driver

In Figure 2.3a, the instantaneous simulated display switching power consumption is plotted over 29 frames of *baroness*. The x-axis represents each row written in turn over the course of 29 frames such that every 240 steps along the x-axis represent the instantaneous power over the course of one frame-write. The y-axis gives the power consumed in Watts for each row. The naive driver results in constant power consumption represented by the flat line at 11.3mW. The squares mark the average instantaneous power consumed over the course of each frame by the naive driver. The smart driver produces a display image identical to the naive driver, but with the instantaneous power

Table 2.3: Driver configurations.

Driver	Image degradation	Power savings
Naive	No degradation	0%
Smart	No degradation	3%-50%
Lazy	Selectable degradation	3%-91%

consumption swinging between 0.0mW and 11.3mW. This video is in a "letterbox" format with a black stripe at the top and bottom of each frame. This constant portion of the image corresponds to zero power consumption by the smart driver at the beginning and end of each frame. The triangles mark the average at the end of each frame for the smart driver. The simulated smart driver resulted in power savings averaging 33% across 7 video clips with savings ranging from 3% to 50% without any degradation to the displayed image.

2.5.4 Lazy Driver vs. Smart and Naive Driver Results

Figure 2.3a shows a plot of the average power consumed by each frame over the course of 29 frames of *baroness* for each driver configuration. Each line plotted represents a different driver configuration for the same video. The relative power consumption using the lazy driver is anywhere from 35-90% less than the naive driver with both drivers updating at 60Hz.

In order to characterize the quality of service for each of the drivers, the Peak Signal-to-Noise Ratio (PSNR) is used to compare the resulting image to the original image. A PSNR value 100dB is used to indicate no change from the original image. Lower PSNR values represent lower quality images. Typical values for compressed video are 20-40dB. The lazy drivers achieve significant power savings, but at increasing degradation to image quality.

Figure 2.3c illustrates the relationship between power savings and loss of image quality. In all cases, the PSNR value of the naive and smart drivers are 100dB indicating no measurable change from the original image. When asking the lazy driver to compare

(a) Original quality.

(b) Last 5 bits ignored, PSNR value for this frame is 30.5dB.

(c) Last 6 bits ignored, trail left by basket moving from left to right is clearly visible, PSNR value is 24.6dB.

Figure 2.4: Detail from *baroness* frame 29

fewer bits while deciding whether to update a pixel, the power consumption decreases as the image quality declines. Ignoring 1 bit resulted in a very small power savings and no image degradation in all 7 videos. Ignoring 6, 5, 4, 3, or 2 bits corresponds to the first five points plotted on each line with PSNR values ranging from 20dB to 60dB.

Assuming that the lowest PSNR value we are willing to accept is 30dB, clearly in all cases ignoring 6 bits results in unacceptable image quality, while all other configurations result in acceptable image quality. In this case we would choose to have the driver ignore 5 bits minimizing power consumption with acceptable loss of image quality. It is also interesting to note that when the lazy driver is configured to ignore 3, 2, or 1 bits, the PSNR values are greater than 40dB, which is better than typical video compression. Figure 2.4 illustrates the original image quality, compared with degraded video frames ignoring 5 and 6 bits after 29 frames.

2.6 Conclusion

A method for calculating power consumption in EPD capsules based on the physical motion of pigment particles inside the capsule has been presented. Additionally, smart driver and lazy driver schemes have been presented providing display power savings of up to 50% for these devices without reduction in quality of service with additional savings where reduced quality of service is acceptable. These improved driver schemes are applicable to any bistable or image-stable display. Degradation in quality of service was found to be very small in certain lazy driver configurations. Such power savings using smart bistable display implementations can substantially reduce resource constraints for low power embedded systems.

2.6.1 EPD Switching Time

Slow switching speed remains a dominant issue with EPDs. EPDs typically require relatively high supply voltages to obtain desirable response times. Most EPD pixels today are driven at around 15V which is somewhat high compared to LCD (4.5V) [56]. The switching frequency of the cells is limited by the viscosity of the suspending fluid and the physical motion of charged particles. EPDs historically have switching times on the order of tens to hundreds of milliseconds when driven at reasonable voltages for portable devices with few devices currently operating at the fast end [49] [22]. LCDs on the other hand are easily able to switch on the order of 10 milliseconds [23]. Optimizing the switching speed of EPD capsules is primarily a problem of maximizing the mobility of the pigment particles in suspension. Pigment particle mobility, described in Equations (2.1) and (2.2), is dependent upon several factors. Increasing the supply voltage can improve response times, but here the 15V common in EPDs today is considered an upper limit of what we would like in a battery powered device. Increasing the charge we can attach to pigment particles increases their mobility for a given

electric driving field, but engineers must be careful with large charges attached to the pigment particles as associated ions in the solution can screen the electrical field reducing the efficiency of the capsule [143]. Reducing the pigment particle size increases its mobility, but will reduce the charge which can be attached via the surfactant and may increase the risk of agglomeration reducing the lifespan of the capsule [49]. Finally, reducing the viscosity of the suspension fluid increases particle mobility, but viscosity must be considered along with the specific gravity which must be carefully matched with the specific gravity of the pigment particles to stabilize the colloidal solution and enable image retention [49].

2.6.2 Grayscale in a Bistable Display

Another important problem with EPDs is the difficulty in achieving grayscale. Because of the bistable nature of these displays, putting a pixel into an intermediate state required to display anything other than black or white can be problematic. The state of the pigment particles inside a capsule is not well defined for interim states required to achieve grayscale. Grayscale can be achieved through area ratio scaling using varying numbers of subpixels to achieve relative lightness or darkness in a pixel; however, the levels of gray achievable through this method are limited by the number of subpixels that can be addressed [79]. It is possible to partially switch a pixel resulting in partial transition from black to white or white to black. This is currently done by erasing or blanking the display, then precisely controlling the amount of energy applied to achieve the desired state. Without first resetting the display to a known state, the amount of energy and the supply voltage polarity required to achieve a specific state is directly dependent upon the current state of the pixel where state is characterized by the imprecise location of the pigment particles inside the associated capsules.

Chapter 3

POWER AWARE QUALITY OF SERVICE MANAGEMENT FOR EMBEDDED H.264 DECODING

The Power Aware H.264 system presented here provides user selectable degrees of power saving effort in exchange for *controlled* QoS reduction. The goal is to reduce the amount of computation required to decode the H.264 stream. This goal is achieved by skipping carefully selected blocks during video stream decoding. Based on the number of skipped blocks, the resulting speed-up in the decoder may be predicted and Dynamic Voltage and Frequency Scaling (DVFS) used to reduce power consumption. The presented H.264 encoder-decoder system classifies macroblocks in each frame into slice groups using Flexible Macroblock Ordering (FMO) described in the H.264 standard [157]. The prioritization algorithm determines which blocks in each frame are the most expendable and organizes them into slices accordingly. Slice groups corresponding to different QoS measures are then selectively omitted from the decoding process based on user preference. The effect on QoS and power savings using two types of error concealment are compared. Simple *Copy Forward Error Concealment* (CF-ERC) provides replacement data from the previous frame with a very inexpensive copy based solely on macroblock location. *Motion Vector Error Concealment* (MV-ERC) is more computationally expensive, but provides substantially improved QoS.

The experimental results in Section 3.6 demonstrate that both techniques are capable of enabling significant power savings of up to 53% and 29% compared to the fully decoded video stream. In some cases, particularly using Motion Vector error concealment, the power savings are significant, while degradation to QoS is minimal. Significantly, the encoded stream is also compatible with standard H.264 decoders without support for the slice dropping scheme.

Previous work will be discussed in Section 3.3, in Section 3.4 the encoder-

decoder system is introduced, and modifications to the JM version 12.4 reference encoder and decoder [63] are discussed. The experimental setup is explained in Section 3.5 followed by results and conclusion.

3.1 H.264

The H.264/Advanced Video Coding (AVC) codec is part of the MPEG-4 suite of multimedia standards. The first revision of the standard, completed in May 2003, sought to address increasing demand on video bandwidth associated with growing services like High Definition Television and network based streaming video applications [157]. H.264 implements a number of improvements over previous video codecs which collectively result in reductions of up to 50% in bandwidth requirements for the same level of image quality. The savings result from extensive analysis and optimization in the encoder which serves to minimize redundancy in the video stream, but comes at a significant cost in computational complexity. The H.264 decoder's complexity is about 2.4 times that of a comparable H.263/MPEG2 decoder [72].

Reduced bandwidth makes H.264 bitstreams particularly useful for hand-held applications using video streams transmitted over a wireless network where bitrates are limited and bandwidth comes at an absolute premium. In addition, storage capacity in a mobile device may also be limited. Unfortunately the increased computational complexity that comes with reduced bandwidth has obvious negative consequences for portable devices relying on battery power supplies.

3.2 Quality of Service

In this work, QoS is objectively defined as the level of distortion introduced into a video at the individual frame level and across several frames as a result of discarding residual data from macroblocks. Peak Signal to Noise Ratio (*PSNR*) calculated between the unmodified video and the same video with discarded data is used. A *PSNR* value

around 25dB is considered acceptable here for a low power mode. Values close to 35dB and higher are considered acceptable for more expensive computations. This range corresponds with the expected *PSNR* performance of a lower quality low bit-rate video stream vs. a high bit-rate [66]. More details are provided in Section 3.4.3. The Video Quality Measurement Tool from MSU is used to collect *PSNR* data for individual video frames [119].

3.3 Previous Work

Several publications have focused on the effects of network packet loss on streaming video quality, methods for recovering from packet loss and encoder optimizations designed to improve video streams' robustness against packet loss. Krasic et al. [94] introduces a priority drop scheme which manages prioritized data flow on a network for frame dropping, and dynamically variable coefficient quantization enabling graceful QoS degradation under variable network conditions. The encoded stream is suited for similar packet prioritization, but without necessary modifications to the video coding standard. Additionally, reducing quantization levels in the decoder does not provide significant computational savings relative to block or frame dropping, and slice dropping provides better resolution version of frame dropping. Huang et al. [74] analyzes the video stream in the compressed domain, pruning the compressed data based on a model which estimates the effect on distortion. This scheme is aimed at mobile devices on a network where each device has limited computational ability considered by the video server which appropriately modifies the compressed data stream. The transcoding scheme focuses on computational constraints while minimizing the introduced distortion, but it relies heavily on frame dropping and does not provide measured distortion injection in contrast to the present approach.

Video decoding workloads are notoriously variable. Significant effort has gone toward predicting workload for the purpose of DVFS for power optimization. Son et

al. [139] gives two methods for predicting workload in MPEG streams in order to effectively scale frequency and voltage. Henning et al. [66] describes a method for dynamically choosing IDCT algorithms in order to exchange quality for energy. Hua et al. [73] introduces three techniques for taking advantage of multimedia applications tolerant to deadline misses enabling opportunities for DVFS. Here, data points are dropped from the stream exchanging quality for power savings; however, unlike the present scheme, the impact of individual data points on QoS is not considered.

3.4 Power Aware H.264 Application

The encoder-decoder pair is implemented by modifying the JM H.264/AVC reference encoder and decoder version 12.4. The goal was to free the decoder from the maximum amount of computation while minimizing the degradation introduced into the decoded video stream. This is accomplished by selectively skipping or dropping from the video stream blocks which introduce the least amount of distortion into the video output. The decoder need not perform dequantization or inverse integer transform operations on the dropped blocks, effectively reducing the time required to decode a given video stream. The resulting speedup enables application of Frequency and Voltage Scaling for potentially significant reductions in power consumption.

The encoder generates H.264 video streams inserting an *I* frame every twelfth frame from which no blocks will be dropped followed by eleven *P* frames with prioritized blocks. The Group of Pictures (GOP) sequence used is:

$$\{I, P, P, P, P, P, P, P, P, P, P, P\}$$

The *I* frame uses intra-coding exclusively so that no previously introduced errors can propagate beyond this frame which is decoded independently from any previously decoded frame. The *P* frames in the sequence use inter-coding in which blocks from each frame may be reconstructed using data from the previous *I* or *P* frame in the video

sequence. These inter-coded frames will propagate errors from one frame to the next when a motion vector points to areas in the previous frame impacted by dropped blocks.

At this point *B* frames which use motion vectors in both the forward and reverse direction are avoided in order to simplify analysis of the decoded video stream. However, adding *B* frames to the encoded stream in the future will improve bandwidth efficiency, and reduce the propagation of distortion introduced by dropped macroblocks. Since *B* frames are not normally used as reference frames for Copy Forward or Motion Vector Copy operations, blocks dropped from B frames will not propagate at all. Additionally, inserting a *B* frame between two *P* frames means that the effect of distortion spreading to multiple blocks through multiple motion vectors is limited to one stage over three frames instead of two as is the case with a sequence of three *P* frames. Adding *B* frames to the stream does; however, increase the computational load on the decoder, potentially reducing power efficiency.

3.4.1 Error Concealment

H.264 includes provisions for error concealment to minimize video quality degradation in the event of lost data such as dropped network packets or missed decoding deadlines. Error concealment functions typically replace an entire missing frame or block with buffered data from a previously decoded frame. In the JM reference software, "conceal by copy" replaces a missing block with its predecessor from the previous frame. A "conceal by trial" replaces the missing macroblock after evaluating the surrounding blocks to find one whose motion vector points to data minimizing the distortion at the edge of the missing block. These two actions loosely correspond to the error concealment actions taken by the two decoder schemes. The "conceal by copy" method is used for error correction in the simple CF-ERC decoder, and a modified version of "conceal by trial" for the MV-ERC decoder. Since available blocks are selectively dropped in the decoder, any motion vector data calculated for a given macroblock is still available

to us. This provides the opportunity to replace the macroblock using its motion vector data rather than a simple copy forward. This method is guaranteed to introduce the minimum amount of distortion into the video stream as the unmodified H.264 motion compensation algorithm in the encoder has carefully selected the motion vector data for each block to do exactly that.

3.4.2 Error Propagation

Error introduced in one video frame decreases as it propagates forward due to leakage in the prediction loop [92]. Although distortion introduced through macroblock dropping need only be considered over the course of a window of several frames [92], errors for a given frame can be approximated as the sum of the propagation errors since the last I frame [93].

Errors introduced through macroblock dropping propagate to the next frame each time a macroblock in the following frame references the dropped block via motion vector. All or part of the distortion caused by the dropped macroblock is carried forward when each dependent block in the following frame is reconstructed from its correct residual data added to the distorted data present in the previous (reference) frame.

The amount of error or *distortion* introduced into the video stream by a single macroblock (MB), is defined in terms of Mean Squared Error (*MSE*). In this work, *MSE* defined between the nth block in the current frame m, MB_n^m and the block in the same location from the previous frame, MB_n^{m-1} is given by

$$\begin{aligned} &MSE(MB_n^m) \\ &= \tfrac{1}{16^2} \sum_{i=0}^{15} \sum_{j=0}^{15} ||MB_n^m(i,j) - MB_n^{m-1}(i,j)||^2 \end{aligned} \tag{3.1}$$

where $A(i, j)$ is the Y component of the pixel at position (i, j) in the 16×16 pixel block A. The frame distortion, D_m, is derived from the sum of $MSE(MB_n^m)$ values across all

n macroblocks in the frame

$$D_m = \sum_{i=0}^{n-1} MSE(MB_i^m) \tag{3.2}$$

The total distortion of frame m, $\hat{D}_m$ can be estimated as the sum of the distortion due to dropped macroblocks added to the distortion already present and propagated forward from the previous frame

$$\hat{D}_m = D_m + \hat{D}_{m-1} \tag{3.3}$$

Since the presented system does not drop macroblocks from I frames, and I frames are decoded independently without inter-coding, the distortion in any I frame is considered to be zero. Thus, the total distortion, $\hat{D}$ present in the first P frame, m following an I frame is $\hat{D}_m = D_m$.

We can adjust for the attenuation of distortion from the previous frame, $\hat{D}_{m-1}$, by multiplying by an empirically determined scaling factor $0 \leq \alpha \leq 1$ before adding D_m giving

$$\hat{D}_m = D_m + \alpha \cdot \hat{D}_{m-1} \tag{3.4}$$

where $\alpha \leq 1$.

3.4.3 Encoder Modification

The modified JM12.4 reference encoder prioritizes and orders blocks within each frame according to the amount of distortion, $MSE(MB)$ we would introduce into the stream if the block were dropped by the decoder. Blocks with the smallest *MSE* are considered more expendable, and are placed at the beginning of the list. Blocks with larger *MSE* are considered more important and are placed at the end of the list.

After prioritizing the macroblocks, the encoder begins dividing the blocks into 6 slice groups numbered from 0 to 5. The most expendable blocks are placed in *slice*

group 5 which will be the first slice dropped in the decoder. The most important blocks are placed in *slice group 0* which is never dropped in the decoder. Each slice group has associated with it an acceptable level of total distortion, $\hat{D}_m$ which acts as a threshold when assigning macroblocks to slice groups. The threshold is estimated from user selected Y-component *PSNR* values for each slice group. *PSNR* is given by

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX^2}{MSE}\right) \tag{3.5}$$

where MAX is the maximum pixel value, in this case 255. Given a desired value for *PSNR*, we can calculate the associated *MSE* by

$$MSE = \frac{MAX^2}{10^{PSNR/10}} \tag{3.6}$$

Since this value is an average across all macroblocks in one frame, we multiply *MSE* by the number of macroblocks in the frame to find the allowable distortion or *distortion threshold* for slice group s, D_s. For CIF video, 352×288 gives us 22×18 blocks, or 396 macroblocks per frame, and $D_s = 396 \cdot MSE$.

The encoder builds slice groups for each frame starting at the low distortion end of the prioritized macroblock list and adding macroblocks to the lowest priority slice group as long as the sum of their distortions does not exceed the previously calculated distortion threshold for that slice group. The result is a slice group for each distortion threshold such that dropping any slice and all higher numbered (more expendable) slices results in a level of distortion described by the distortion threshold for the dropped slice. During this process, the encoder also checks each macroblock's distortion in all three components (YUV) against an individual macroblock threshold and immediately adds failing blocks to *slice 0*. This check prevents the encoder from adding blocks when plenty of room for additional distortion is available to an expendable slice group, but adding it would introduce obvious artifacts into the video stream.

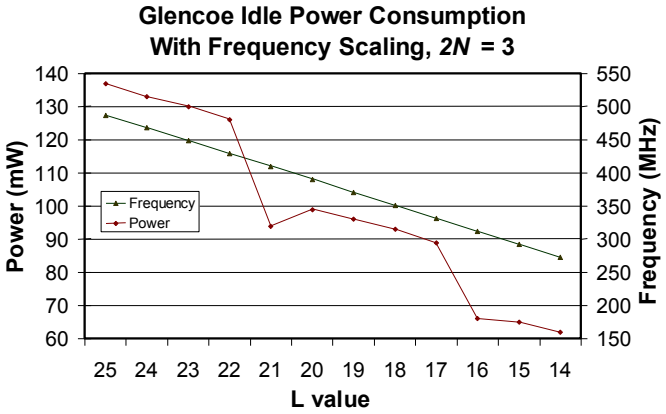


Figure 3.1: Glencoe DVFM characteristics.

In the current implementation, the threshold is defined as 1/5 the smallest of the most distorted blocks from the previous 20 frames in the *Y* component and 1/2 of the average distortion for the *U* and *V* components.

3.4.4 Decoder Modification

The JM12.4 reference decoder has been modified to drop slices from the modified H.264 stream in accordance with a user selected mode. The user chooses a desired level of QoS performance in order to reduce power consumption in the decoder by selecting the number of slice groups to keep in the stream. For example, placing the decoder in *6 slice mode* will not drop any slices. On the other hand, placing the decoder in *1 slice mode* will drop slice 1 and any higher numbered slices, decoding only the most important set of MBs, slice 0. Based on the user selected mode, the decoder performs error concealment to replace data in the skipped slices. Two decoders have been implemented—one using macroblock CF-ERC and the other using MV-ERC as described in section 3.4.1.

The decoding loop treats *I* frames and slice groups numbered smaller than the mode number normally, reading and decoding each macroblock. Slice groups with identification numbers equal or greater than the mode number are skipped in the decoder. The CF-ERC decoder simply does not read or decode blocks in these slices so

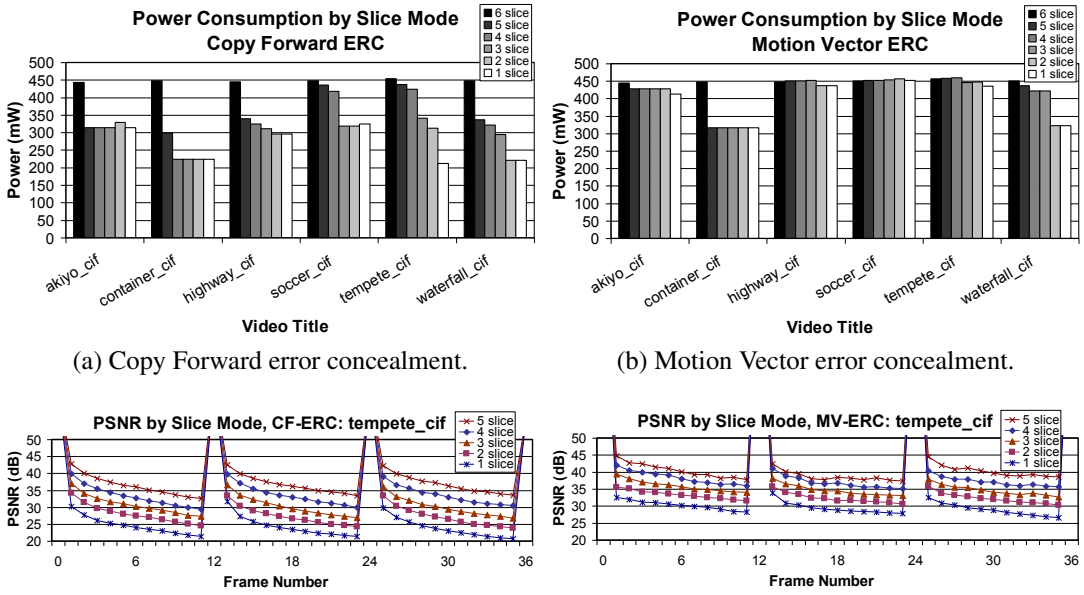
that they are treated as lost by the error concealment function. The error concealment function is also modified to ensure that conceal by copy is the only method used for error concealment, avoiding the conceal by trial function in order to minimize computational complexity for performance reasons. The MV-ERC decoder does read each macroblock in a dropped slice in order to obtain the associated motion vector data, but the coefficient data is not read, and the block is never decoded. Each block from a dropped slice is marked as lost in order to trigger error concealment, and a modified version of the conceal by trial function is used to recover the motion vector data for each marked macroblock copying data from the previous (reference) frame in accordance with the motion vector information.

In the testbed, preliminary results indicated that using motion vectors for error concealment in all three video components (YUV) was so computationally expensive that it wiped out any savings achieved through avoiding integer transform and dequantization operations. The process of rebuilding blocks by performing motion compensation can take 55% of the decoder's effort [34]. As a result, motion vector error concealment is used only for the Y component, and simple copy forward to replace dropped UV data.

The CF-ERC decoder provides better power performance than the MV-ERC alternative. In addition to skipping block decoding, it avoids the expenses of recovering motion vector information from the video stream and later dereferencing it to perform error concealment. However, the improved power performance implies reduced QoS. Although the encoder classifies blocks based on their distortion with respect to the same block (same position) used in CF-ERC, the MV-ERC method will regularly reconstruct a replacement block with less distortion resulting in improved QoS as discussed in Section 3.4.1.

Table 3.1: Benchmark video subset.

Video	Brief Description
akiyo_cif	news reader
container_cif	container moving in harbor
highway_cif	rolling highway
soccer_cif	fast panning and action
tempete_cif	zoom out with falling leaves
waterfall_cif	zoom out on distant waterfall



(a) Copy Forward error concealment.

(b) Motion Vector error concealment.

(c) Copy Forward error concealment.

(d) Motion Vector error concealment.

Figure 3.2: Power figures by slice dropping mode for Copy Forward (a), and Motion Vector ERC (b). The *PSNR* charts show QoS performance for each decoder mode on *tempete_cif* for Copy Forward (c) and Motion Vector ERC (d).

3.5 Experimental Setup

H.264 video streams were generated for 16 benchmark videos using the modified JM12.4 encoder. These streams were then decoded through the two modified JM12.4 decoders on a Linux server, with an Intel(R) Xeon(TM) CPU running at 2.80GHz with 4GB RAM and on the Intel Glencoe Development Platform running Linux 2.6.9 on the XScale-PXA270 rev 4, with 64MB RAM. Power measurements were obtained for a subset of the 16 benchmark videos using the PXA270 V_{core} voltage touch point on the

Glencoe Development Platform with a Keithley 2000 Multimeter sampling at 100Hz with the decoder running each video in each mode. Voltage samples were collected and analyzed in Labview to obtain an average power reading over several iterations of 1024 samples each. These results are presented in the next section. The Linux port for the Glencoe Development board includes a utility for Dynamic Voltage and Frequency Management (DVFM) [78]. Frequencies are set by giving the utility a clock multiplier, L, and a turbo mode multiplier, $2N$. The base clock frequency of 13MHz is multiplied by $L \cdot N$ to obtain the desired frequency [80].

In order to obtain a piecewise-linear power function in terms of L and $2N$, it was necessary to fix the $2N$ value. A value of $2N = 3$ has been chosen, giving the suitable although non-ideal function of power in terms of frequency for system idle seen in Figure 3.1. Frequencies used range from the default value of 468Mhz to 273MHz indicated in the chart with L values from 24 down to 14. This range of frequencies crosses three available voltage levels as can be seen in the figure. The high step on the left corresponds to $V_{DD} = 1.5V$, the middle step corresponds to $V_{DD} = 1.4V$, and the low step on the right corresponds to $V_{DD} = 1.3V$. The power performance chart for the idle Glencoe board gives an idea of the power savings we should expect at various frequencies after implementing DVFM in the decoder. Reductions in V_{DD} provide significant power savings as expected from the quadratic relationship between power and supply voltage, $P = C_{switching} \cdot V^2 \cdot f$.

The goal is to characterize the decoders' power performance given H.264 streams generated by the modified encoder. In the first step, the relative speedup experienced by the decoder when dropping slices is found and compared with the speed of the decoder decoding all slices. Relative speedup determines our ability to slow the decoder with frequency scaling in order to reduce power consumption. To collect decoder timing data, a 100 frame modified H.264 video stream was generated for each test video. Decoder frame rate data was then collected over several iterations for each video in

Table 3.2: Speedup and DVFM frequency for decoder in 1-slice mode for Copy Forward and Motion Vector decoders.

	CF		MV	
Video	Speedup	f(MHz)	Speedup	f(MHz)
akiyo_cif	25%	351	9%	429
container_cif	57%	312	16%	409.5
highway_cif	43%	331.5	6%	448.5
soccer_cif	30%	370.5	5%	448.5
tempete_cif	65%	292.5	14%	429
waterfall_cif	57%	312	17%	409.5

Table 3.3: Power and *PSNR* for decoder in 1-slice mode for Copy Forward and Motion Vector decoders.

	CF		MV	
Video	P	*PSNR*(dB)	P	*PSNR*(dB)
akiyo_cif	29%	43.0	7%	43.8
container_cif	50%	34.9	29%	37.6
highway_cif	33%	34.5	2%	35.7
soccer_cif	27%	28.7	0%	32.7
tempete_cif	53%	24.1	5%	30.1
waterfall_cif	51%	30.5	29%	33.8

each slice dropping mode on the Linux server and the Glencoe board in order to verify the Glencoe's performance.

The frame rate the Glencoe decoder achieves for the full six slice video is taken as the nominal frame rate against which improved frame rates for slice dropping modes are compared for frequency scaling purposes. For instance, if the decoder in 1 slice mode decoding *akiyo_cif* finishes 25% earlier than the same video fully decoded, then we can slow the clock by 25% and still decode at the same frame rate. The adjusted frequency is given by $f' = t' \cdot f_{default}/t$ where t' is the improved decoding time, and $f_{default}$ is the default frequency 468MHz. The calculated f' value must be adjusted up to the next available frequency in the DVFS scheme. For *akiyo_cif*, the calculated frequency, $f' = 348.7$MHz must be adjusted up to the next available frequency, 351MHz at $V_{DD} = 1.4V$.

3.6 Results

Timing results for the CF-ERC and MV-ERC decoders in 1-slice mode are given in Table 3.2 with the computed DVFM frequency used for each video on the Glencoe

(a) *akiyo_cif*

(b) *soccer_cif*

Figure 3.3: Example of last P frame decoded before next GOP demonstrating image quality for the Copy Forward (middle) and Motion Vector Error Concealment (right) vs. the fully decoded video (left).

board. The associated power savings for a given video and frequency are listed in Table 3.3 along with average *P* frame *PSNR*. Significant power savings were obtained from the CF-ERC decoder, in some cases without significant impact on *PSNR*. The MV-ERC decoder achieves much smaller power savings due to the significant complexity of handling motion vectors during decoding, but it does a much better job of preserving image quality.

The power measurements obtained for each video and decoder slice dropping mode are shown for both decoders in Figures 3.2a and 3.2b. The impact of voltage scaling is evident as power drops significantly each time reducing frequency makes a lower V_{DD} available as predicted in Figure 3.1. Performance varies for both CF and MV schemes with variations in the transmitted video stream. There may be signifi-

cant variation in MB inter-dependence and motion-vector densities from one scene or even one frame to the next. The MV-ERC scheme is particularly susceptible to these variations due to the expense of decoding with motion estimation.

Power savings in the Copy Forward encoder were significant, ranging from 27% to 53% for 1 slice mode over the fully decoded video. QoS for these videos proved to be well controlled for each video, although the decoder does not achieve a strict *PSNR* for the same decoding mode across different videos. Figure 3.2c clearly indicates distinct *PSNR* levels for each decoder mode. Figure 3.2d demonstrates the improved QoS associated with Motion Vector Error Concealment, seen in the upward shift of the *PSNR* plot lines. Figure 3.3 presents subjective image data from two videos for comparison between the two schemes. In the chart, the left image is the fully decoded reference image, the center image uses CF-ERC, and the right image has been constructed using MV-ERC. Image artifacts are more prevalent in the CF-ERC decoder, while they are much less noticeable in the MV-ERC frames.

The two videos with the lowest *PSNR* values decoded using the CF-ERC decoder see the largest QoS improvement when the MV-ERC decoder is used. An example of QoS performance of both encoders is presented over all slide dropping modes for *tempete_cif* in Figures 3.2c and 3.2d.

3.7 Conclusion

The presented H.264 block prioritization scheme enables significant power savings in concert with DVFM. The objective QoS measurements indicate acceptable quality performance, but there are several opportunities for improvement. The subjective quality of the decoded videos can be improved through further tuning of the parameters in the encoder to reduce artifacts in the decoded image, particularly focusing on distinguishing foreground objects from background. Tuning a number of other factors may also improve performance. Some examples are reducing the number of slice groups

which decreases decoding complexity and bandwidth, considering motion vector data in addition to distortion data when prioritizing MBs, using motion vectors less aggressively in the decoder for MV-ERC, considering B frames, and varying the GOP size between I frames.

Chapter 4

POWER MANAGED SCALABLE H.264 VIDEO DECODING

High resolution video coding is an example of a computationally expensive application in which real time performance is very difficult to achieve without dedicated hardware. Advances in video parallelization promise to unlock the strengths of MPSoCs and deliver high performance video processing while reducing power consumption to meet tight constraints in embedded applications.

The H.264/Advanced Video Coding (AVC) standard is the current state of the art codec designed to achieve dramatic compression ratios necessary for the storage and transmission of large format, high quality video. Unfortunately the high compression achieved with the H.264 standard comes at a cost of significant increases in encoding/decoding complexity, posing a problem for hardware developers and end users. The computational complexity is a problem for the embedded world in particular where power consumption and processing speed come at a premium. To further complicate the issue, because the codec is designed specifically to compress video image data by discovering and eliminating redundancy within and between frames, the encoded stream is a nightmare of data interdependencies from the multicore programmer's perspective.

In this work a scalable parallelized implementation of the H.264/AVC decoder is presented for the Cell Broadband Engine (CBE) addressing workload and data partitioning in the face of complex data dependencies, and performance and processing element (PE) scalability in terms of throughput in the context of embedded multicore architectures. PE scaling also enables extremely high agility to precisely match the target decoding rate, proving valuable for run time power management. Although efficient multicore implementations alone offer tremendous power advantages over monolithic designs, dynamic power management makes significant additional power savings possible. A dynamic power management (DPM) scheme designed to improve power per-

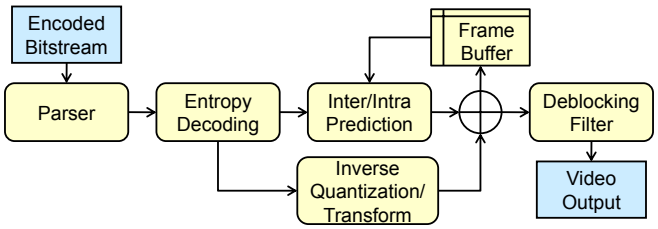


Figure 4.1: H.264 decoder.

formance in the parallelized H.264 decoder is demonstrated which takes advantage of agile PE scaling to maximize the effects of Dynamic Frequency and Voltage Management (DVFM) for just in time frame decoding while minimizing frame deadline misses.

Background is provided in the next section followed by previous work in Section 4.2. The parallelization scheme is presented in Section 4.3. The parallelized decoder implementation and optimizations are presented in Section 4.4. Raw performance results for the parallelized decoder are discussed in Section 4.5. The agile PE scaling and the DPM scheme are presented in Section 4.6. Finally experimental setup, results and discussion of the decoder's power performance are given in Section 4.7 followed by conclusions.

4.1 Background

4.1.1 IBM Cell Broadband Engine

The Cell Broadband Engine (CBE) has the potential to provide exceptional performance for complex data intensive operations such as H.264 video decoding given an effective parallelization scheme. In designing a decoder parallelization scheme for the CBE, limitations such as available bandwidth for communication between processing units, memory limitations for code, data, stack, and heap in the SPU, and the need for synchronization management in the PPU must be considered. See Appendix B for a brief overview of the CBE.

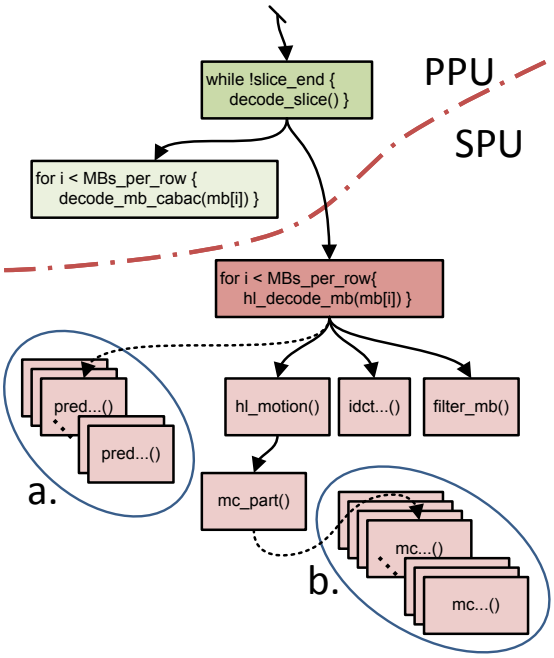


Figure 4.2: *FFmpeg* main MB decoding loop, a. Intra decoding functions, b. Inter (motion compensation) decoding functions.

4.1.2 H.264

The H.264 codec is commonly used for mass video storage applications such as Blu-ray Disc and broadcast/streaming video where minimizing bandwidth and storage space without loss of perceived video quality is extremely important. As HD and larger format video becomes more popular, this kind of improvement in storage efficiency is increasingly urgent. However, due to the increase in computational complexity associated with H.264, the cost of decoding 1920x1080 resolution HD streams cannot be ignored. Decoder implementations capable of handling real time HD decoding without dedicated hardware acceleration are exceedingly rare.

FFmpeg is a popular open source audio and video conversion and streaming project capable of handling a wide range of video and audio codecs based on the *libavcodec* library [19]. Since *FFmpeg* supports a large number of codecs and encapsulation, the code base is large with over 250k lines. The present decoder implementa-

tion's base code is the *FFmpeg* command line tool source code configured to include only the H.264 decoder and YUV encoder. Unused source files (associated with other codecs) are removed.

The data structures and control flow built into the *FFmpeg* H.264 decoder center on decoding one Macroblock (MB) at a time. The decoder maintains a tree of elements and structures rooted in a decoding context data structure which holds the current decoding state, including modes and data associated with the current MB, arrays for accessing information about neighboring MBs, references to the current output picture and any reference pictures etc. A pointer to this structure is passed from one function to the next conveying the state of the decoder and the MB under consideration.

Fig. 4.2 illustrates key control flow elements and functions of the main MB decoding loop within *decode_slice*(). After returning from the entropy decoding phase, *decode_mb_cabac*(), the context structure is filled with all information and block data needed to decode the current MB. This data is passed to *hl_decode_mb*() where the MB is predicted, residual data is recovered, and deblocking is performed on the resulting image data which is stored in a current picture buffer. An important program flow characteristic is that this process is repeated for each MB until the decoder reaches the end of the current slice when control exits the loop and writes the frame to the output if there are no more slices. This process is repeated for each slice in the video.

4.2 Previous Work

4.2.1 Parallelized Video Processing

H.264 parallelization, particularly targeting encoding and high resolution decoding in real-time has been an active area of research. Detailed analysis of opportunities for parallelization in the encoder and decoder across several levels of data partitioning and some functional partitioning schemes are available. Meenderinck et al. [112] provides a good survey of previous encoder and decoder efforts including MPEG-1, MPEG-2, and

H.263 codecs. Encoding presents some unique problems with respect to parallelization compared with decoder implementations such as increased reference data sizes and memory requirements, but the inherent data dependencies are essentially the same for both applications leading to similar advantages and disadvantages for each of the basic parallelization methods.

Efforts to improve encoder performance include coarse grained data partitioning exemplified by work from Intel in the context of hyper-threading technology for frame and slice level parallelization as well as opportunities for SIMDization [35] [98]. Rodríguez et. al [133] evaluates a combination of GOP and frame level parallelization. Jacobs et. al [84] proposes a slice level data partitioning scheme. Roitzsch [134] uses slice level parallelism and workload prediction metrics to achieve load balancing. Coarse grained data partitioning can require vast resources, as even at the sub-frame slice level a large number of slices in flight simultaneously represents a very large memory requirement and long latencies. A finer grained MB level data partitioning scheme for the H.264 encoder is presented by Sun et al. [144] where regions of MBs are assigned to each processor to meet intra dependencies.

Functional partitioning schemes for H.264 encoding have also been evaluated. Chen et al. [33] examines functional partitioning for hardware oriented pipelining, while Jagmohan et al. [85] implements frame level data partitioning and functional partitioning within each frame on the CBE. From a pool of work tasks, each SPU is assigned either motion vector estimation, spatial transform and quantization, or entropy encoding for a complete frame. Hwang et al. [75] uses data flow graph transformations to functionally partition encoder subsystems, but with limited scalability.

Several H.264 decoder studies and implementations are also available. Tol et al. [151] looks at functional and data partitioning and present an argument for the use of data partitioning for scalability. Chong et al. [43] looks at tracking data dependencies in the front end for parallelization addressing issues that make static scheduling difficult

or impossible in video coders. Alvarez et al. [6] presents a characterization of the H.264 decoder's performance emphasizing HD and suggesting better media architecture support and multiprocessor support. Major et al. [105] describes an implementation using dynamically reconfigurable instruction cell based architecture with code tailoring for instruction level optimization. Zhao and Liang [161] presents a wavefront parallelization data partitioning and task scheduling scheme for the H.264 encoder.

In their study on parallel scalability of video decoders, Meenderinck et al. [112] [111] examines the amount of parallelism which can be extracted from the decoder and present a 3D wavefront scheme for extremely high scalability. They look at data structures and dependencies, and point out scalability problems with slice, frame, and intra coded MB levels of data parallelism. Alvarez [5] discusses the scalability of MB-level parallelism for H.264. Seitner et al. [136] gives a simulation based comparison of several MB level parallelization approaches with resource-restricted environments in mind. Their *single-row approach* effectively describes the scheme implemented in this work for the CBE. The reported advantages of this scheme are good scalability and reduced synchronization complexity. Seitner points out that the disadvantage of this scheme is the relatively high inter-processor data dependency which can result in increased bandwidth requirements.

Comparable CBE H.264 decoder implementations have been presented by the author [14], Baik et al. [12], Chi et al. [37], and Cho et al. [39]. In Baik's implementation, parallelism is derived at the inter coded block level where intra block-dependencies need not be addressed. The design utilizes both data and functional partitioning by allocating MBs from inter coded frames among the available SPUs in a load balanced fashion, and dedicating an additional SPU to deblocking. The PPU manages entropy decoding, intra decoding, and other related overhead. Compared to Baik, the implementation presented in the present work achieves greater scalability since the data dependencies imposed by intra coded blocks whether the frame is an

intra (I) or inter (P,B) coded frame are addressed. Unlike Baik, it also distributes the deblocking effort, and removes all decoding activities from the PPU except for the inherently non-parallelizable entropy decoder, minimizing the impact of the PPU as a throughput bottleneck.

Chi and Cho have presented subsequent scalable H.264 decoder implementations on the IBM Cell building on the previous MB row parallelization CBE implementation presented by the present author in [14]. Chi et al. implemented the single-row 2D wavefront scheme for the CBE similar to the one presented in [14] as well as an MB level *task pool*. The results demonstrate that the single-row approach provides near ideal performance and scalability. Chi's implementation assumes an infinitely fast entropy decoder, decoding the video stream offline and enabling demonstration of effective decoder scalability beyond the limitations of the entropy decoding implementation. The implementation consumes more than 50% of available SPU scratchpad memory by storing a full row of MBs for use in inter SPU communication. This data structure can reduce communication bandwidth requirements, but it may also limit opportunities for increased image resolutions.

Cho takes essentially the same parallelization approach with the additional step of parallelizing the entropy decoder at frame level with the help of multithreading in the CBE's PPU. The implementation also executes MB data transformations in the SPU in order to facilitate DMAs which are normally frustrated by the noncontiguous representation of MBs in memory. Manually executed SIMD optimizations as well as SIMD compiler optimizations are also implemented for improved performance. Performance optimizations with respect to available instruction memory vs. required data memory are not specifically addressed beyond the need to avoid repeated code overlay misses by avoiding switches between MB decoding and deblocking stages.

Both Chi's and Cho's implementations serve to validate the single-row parallelization approach implemented in [14] for H.264 performance and scalability. In

the present work, the previous implementation has been extended to explore dynamic power management in the context of scalable H.264 decoding.

4.2.2 Dynamic Video Power Management

DPM has a long history, including the use of sleep states, and voltage and frequency scaling. Weiser et al. used instructions per unit energy as a metric to evaluate scheduling for power management, including frequency and voltage scaling in 1994 [156]. Burd et al. [28] presented a system architecture for dynamic voltage scaling in 2000.

In general, DPM can be characterized as an online problem optimized in the face of competing criteria such as available energy vs. video frame deadlines. The survey and tutorial from Irani et al. [83] provides a good foundation in *competitive* and *adversarial* power management strategies.

A great deal of work has gone toward improving effectiveness of frequency and voltage scaling for video applications. In real time video decoding applications typical of embedded hand held devices, frequency throttling proves problematic due to the need for accurate workload prediction with respect to deadlines. HD video decoding in particular has proven difficult to implement using DPM in recent commercial processors [58]. Overhead, mispredictions and over throttling lead to unacceptable frame jitter, interrupting playback and degrading video quality.

Several previous efforts have been directed at improving power performance in single core video processing applications. Much of the work hinges on accurately anticipating the complexity of the upcoming work element in order to select the lowest clock frequency to meet the deadline and minimize energy consumption. Workload prediction efforts can be broadly classified as either online or offline.

Mesarina et al. [114] presents an offline preprocessing algorithm for MPEG streams exchanging Quality of Service (QoS) for energy and exploring the effect of

buffer constraints. In [16], Baker et al. present a power aware energy for QoS exchange scheme for H.264 by prioritizing work in the encoder for selective workload scaling enabling DVFM in the decoder. Akyol et al. [3] describes an improved proactive complexity model to address the increased difficulty in predicting future workloads in the H.264 decoder. Their system depends on complexity hints embedded in the video stream offline by the encoder, and utilizes larger buffer sizes of up to 16 frames which may not be feasible in many embedded applications, particularly at higher video resolutions. Offline solutions can provide more accurate workload prediction at runtime, but offline analysis may be prohibitively expensive or impossible in applications such as real-time capture, transmission, and display.

A frame level online MPEG workload prediction algorithm is presented in Choi et. al [41] which attempts to mitigate deadline misses by dividing the work into frame dependent and frame independent components. In the event of overly aggressive frequency scaling for the frame dependent workload, a potential deadline miss may be preempted by compensating with a higher frequency in the relatively fixed independent frame workload. Another online work estimation DVFM scheme is presented by Son et al. [139] using slack time and frame drop rates of a GOP as metrics to drive frequency state selection. Pouwelse et al. [129] presents a runtime workload modeling scheme using frame size and type for the H.263 codec relying on video streams augmented with frame size information. Lee et al. [96] proposes and evaluates a frame level technique along with other coarse grained offline and online techniques.

Single core power management schemes for video decoding with online workload estimation tend to use relatively coarse grained work units for analyzing workload prediction as well as power management state selection. Such techniques are susceptible to significant deadline violations which must be corrected by either speeding up subsequent frames or implementing large frame buffers. Frequent frequency and voltage changes are possible with modern processors, and as we will see, fine grained mul-

ticore work distribution enables very precise power state management with extremely accurate power state to workload matching. The presented MPSoC scheme takes advantage of agile PE scaling to circumvent the overhead and shortcomings associated with workload prediction. Thanks to fine grained decoding rate control, power management states are effectively used to substantially reduce power consumption, simultaneously reducing deadline misses and frame buffer requirements.

Recently along with increased hardware support for runtime power management, work targeting DPM in the context of multicore architectures has gained importance. Annavaram et al. [11] uses an emulation test bench to explore DPM for asymmetric multicore processors by executing sequential portions of the code at high clock rates, and parallelizable portions on multiple cores at lower clock rates. Li et al. [99] discusses DPM for parallel computation on MPSoCs. The problem is addressed two dimensionally in terms of available frequencies and available processors. The implemented DPM manager and scheduler effectively addresses MPSoC DPM with a binary search heuristic, but their technique relies on fixed workloads which is unrealistic in applications such as video decoding. Cho et al. [38] presents a model for minimizing power consumption in multicore systems which accounts for the amount of parallelism available in the program, and technology specific properties of the MPSoC characterized by the ratio of static power consumption to dynamic power consumption. For systems in which PEs can be put to sleep individually, Cho reports that minimum total energy is not dependent on the total number of PEs used during execution while the energy efficiency is optimized when the maximum number of cores is used in the parallel portion of the code. This principle suggests that for rate sensitive parallel applications, DPM schemes like the one presented in the paper which use the maximum number of cores at the slowest possible frequency will approach ideal power performance.

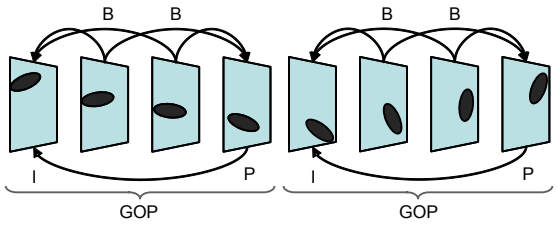


Figure 4.3: Frame types.

4.3 Parallelization Scheme

4.3.1 H.264 Parallelization Opportunities

Frame Level

The video stream is composed of a series of video frames organized into many Groups of Pictures (GOPs) with a sequence of frames starting with an I frame followed by a sequence of P and B frames. The I frame is independently coded, but the P and B frames have data dependencies with other frames in the GOP. Fig. 4.3 illustrates the dependency relationships between the GOP, I, P, and B frames. The GOP is not dependent on another GOP for decoding purposes which exposes a level of parallelism. However, GOP level parallelism imposes an enormous memory requirement for HD resolution and embedded applications in particular and can experience undesirable latencies [112]. Frame level parallelism is also possible based on the dependencies between P and B frames, but this level of parallelism is generally not scalable beyond two or three simultaneous frames and still imposes significant memory requirements.

Slice Level

Within the image frame another straightforward scheme for parallelization presents itself at the slice-level. Slices are groups of independently encoded MBs within a frame which could be decoded independently in parallel. The current *FFmpeg* H.264 decoder already supports thread parallel decoding at this level. Unfortunately, H.264 encoded video frames are not generally encoded using a large number of slices which limits par-

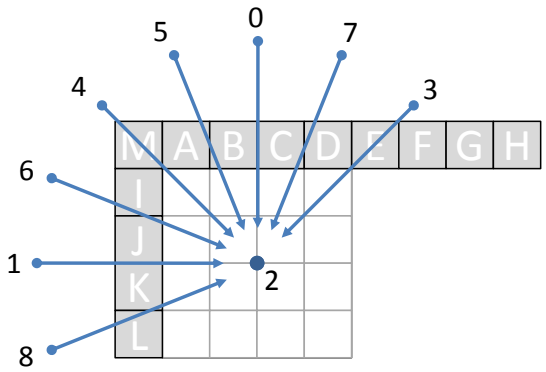


Figure 4.4: Intra coding modes and dependencies [132]. The nine 4x4 intra modes (0-8) are indicated with the direction of an arrow into the 4x4 pixel block in the figure. Mode 2 is the DC mode. The empty pixels in the 4x4 block are predicted using an algorithm to sweep the indexed pixels in the direction of the arrow.

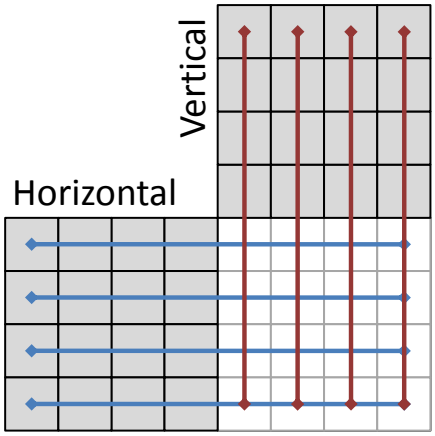


Figure 4.5: Data dependencies of the 4x4 edge filtering algorithms are indicated by the diamond ended horizontal an vertical lines. The filter may use four pixels on either side of the top horizontal or the left vertical border of the 4x4 block and *may modify* three pixels on either side.

allel scalability. Although increasing the number of slices in an encoded video frame can improve error correction performance when streaming over a lossy network, increasing the number of slices also reduces the compression efficiency of the encoder by eliminating many opportunities for exploiting redundancies in the data. For this reason H.264 encoded videos commonly use one slice per frame.

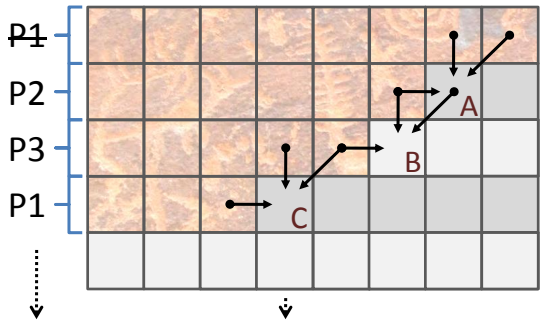


Figure 4.6: Dependencies and synchronization between three PEs. Each row of MBs in a video frame are assigned to one PE in round-robin order. P1 has completed the first row and started working on row 4. P2 is working on MB A and P1 is working on MB C whose dependencies have been satisfied. P3 is stalled on MB B awaiting completion of MB A.

Macroblock Level

Another possibility for parallelization is at the MB level, inter coded blocks are always independent from other inter coded blocks in the same frame exposing a tremendous amount of data parallelism, possibly thousands of MBs per frame in HD video. When considering intra coded MBs on the other hand, it is necessary to deal with inter-block dependencies within the frame. Consequently, some blocks must be decoded before others within the same frame.

Fig. 4.4 illustrates the possible prediction modes used to decode an intra coded MB. In case of mode 0 (Vertical), the bottom row of pixels from the MB in the previous row, directly above the current MB, is used to predict or partially reconstruct the current MB. Fig. 4.5 illustrates the data dependencies associated with the 4x4 deblocking filter. The filter depends on almost the same neighboring blocks as intra prediction, only requiring more pixels from those blocks. It is clear from the figure that there are no intra dependencies on MBs in any other row while decoding the first row of a frame, but intra coded MBs in all other rows depend on MBs in the previously decoded rows.

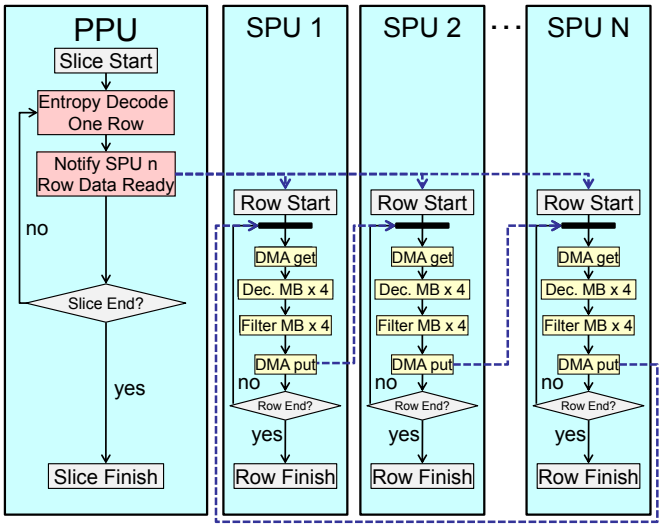


Figure 4.7: Scalable decoder partitioning scheme. Dotted lines represent communication signals. DMA transfers between SPUs and main memory are not indicated. The PPU loop generates one row at a time of entropy decoded MBs. A PPU-SPU signal assigns the row to an SPU and notifies the SPU that data is ready for decoding. The SPU loop completes decoding each MB in the row and returns the results to main memory. Except for synchronization signals, data is passed from SPU to SPU via the current picture store in main memory.

4.3.2 Single MB Row Partitioning Scheme

The presented H.264 decoder parallelization scheme implemented on the CBE focuses on data parallelism at the Macroblock level. Dependencies between Intra-coded MBs are addressed by partitioning a video frame into rows of MBs and assigning one full row of MBs to each decoding core. The scheme assigns a row of MBs to a single SPU, the next row is assigned to the next SPU and so on in round-robin fashion. Fig. 4.6 illustrates inter SPU data dependencies and row to SPU assignment. A stall occurs when unfinished decoding of the reference MB from the previous row results in unresolved data dependencies as seen in MB *B* in the figure. Note that horizontal dependencies occur only within an SPU.

The data and functional partitioning schemes are illustrated in Fig. 4.7, and Fig. 4.2 highlights the dividing line between PPU and SPU in the control flow graph.

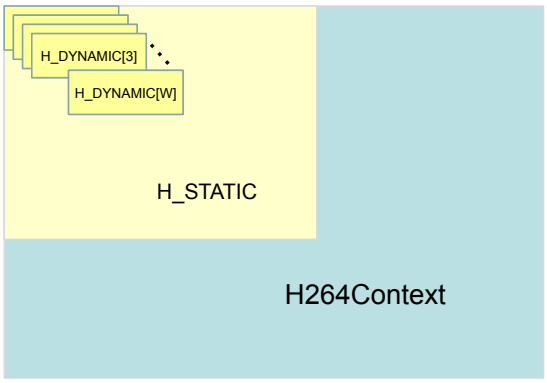


Figure 4.8: Data structure modifications reducing memory requirements in the local store. *W* is the width of the video frame in macroblocks.

Decoding overhead and entropy decoding tasks are executed in the PPU. One row of decompressed MBs is set aside for each SPU in raster scan order. The SPUs retrieve the data as required using DMA transfers and must handle synchronization required to meet intra dependencies. Since all MBs are issued in row order regardless of their prediction mode, we can easily handle dependency synchronization requirements for intra coded MBs whenever they occur in a P or B frame in addition to I frames. A theoretical analysis of similar parallelization schemes for the H.264 encoder is presented in Appendix C.

4.4 Implementation

4.4.1 Data Structures

A critical first step in executing any decoding software on the Cell SPUs is understanding and managing memory requirements. The SPU's 256kB Local Store (LS) is relatively generous by current distributed memory MPSoC standards, but it cannot hold the approximately 30k lines of code plus data we will need for MB decoding. Fig. 4.8 illustrates modifications to the decoding context structure, *H264Context*, for use in the SPU. Only the subset of the decoding context represented by data actually used for MB decoding are loaded into the SPU. Those data fields which are fixed over the course of an MB row are referred to as *static*, represented in the SPU LS once by the *H_STATIC* structure in the figure. The data fields which vary from one MB to the next are referred

to as *dynamic*, and the associated data structure, *H_DYNAMIC*, is replicated in the SPU depending on the number of MBs loaded. Instances of the dynamic memory structure present represent a sliding window of MBs along the SPU's assigned row.

4.4.2 Code Overlay

The 256KB SPU Local Store scratchpad memory is partitioned and shared among the program instructions, program data, the execution stack, and heap for any dynamically allocated structures. In this work, the stack size is constrained by the tree depth of the function call graph for the portion of H.264 decoder code to be executed on the SPU. The basic structure of this function call graph is seen in Fig. 4.2. Stack memory requirements for this portion of the decoder are well bounded as there are no recursive function calls. Although the tree is wide, it is also relatively shallow, with a (somewhat simplified) depth of three as seen in the figure. It is experimentally determined through modifying the available SPU LS that the stack may approach 100KB which must be reserved in the LS memory map to prevent a stack overflow. This stack requirement leaves us with only 156KB of LS space for data and instructions.

The width of the tree as indicated in Fig. 4.2 suggests a large number of possible functions may be called. The implementation includes just under 200 functions totaling approximately 160KB in total LS memory requirements. Additionally, while minimizing memory requirements where possible, the implementation requires at least 50KB in data memory to operate on four MBs at a time. With the 100KB requirement for stack/execution memory, 100KB of available memory in the 256KB LS must accommodate the 160KB of code.

This 60KB memory shortage is overcome using a technique called *code overlay* which allows us to map multiple segments of code to a single address in the LS, and manage which code segments are available in the scratchpad memory. An overlay manager manages a table to track which functions are available and uploads called function

code when it is found to be missing from the LS. The overlay manager implements a direct mapping scheme with a user defined mapping to write missing functions into the LS. The user provides a linker script which describes the overlay mapping. The overlay mapping is defined in terms of *segments* and *regions*. Each segment refers to a specific block of code, and a region refers to a specific location in the memory mapping. If two code segments are mapped to the same region, then they will share the same address in memory, and only one may be present in the LS at a time. An example of the code overlay memory mapping framework is provided in Fig. 4.9.

The linker supports mapping one object file to each segment. Object files may be placed into their own segments or grouped together to form a larger segment. Since the resolution at which the linker handles code segments is the object file, the configurability of the implementation may be controlled with respect to the LS memory map by arranging source code into separate source files for each potential code segment. Practically, this means placing the source code for each function into a separate source file in order to achieve the greatest resolution. In this way, we have the freedom to implement an SPU mapping which addresses the sequence in which functions will be called and the possibility of additional overhead associated with each function call. In this section it is assumed that each function is mapped to a segment, and each segment has only one function mapped. Consequently, the term *function* is used interchangeably with *segment* here.

When an absent function is loaded into the LS, the *interfering* function which is present and mapped to same region is overwritten. The implication is that anytime a function is called and its code is missing from the LS, tasks which stall program execution must be executed resulting in additional function call overhead. This overhead comes from a small cost of the overlay manager's table lookup, and the potentially significant cost of a DMA operation to bring the missing code into the LS. While working to design an overlay mapping which not only enables the SPU executable to fit into the

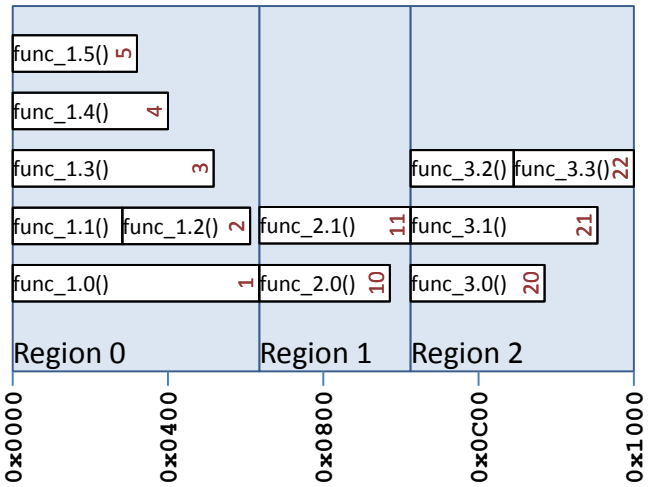


Figure 4.9: Example code overlay mapping showing a 4K section of memory with addresses labeled across the bottom. Memory is divided into regions. Multiple segments may be mapped to the same region or address. Region sizes are determined by the largest segment they contain. Functions or object files are mapped to segments, and multiple objects may be mapped to the same segment.

LS, but also gives us the greatest benefit in terms of performance or throughput, it is this overhead which must be addressed.

Designing an Overlay Mapping

In order to maximize performance with respect to code overlays, it is necessary to minimize the number of overlay miss penalties incurred. Given a trace of the program execution and the overlay mapping, the number of miss penalties can be deterministically identified and an optimal mapping scheme is possible. This is true of many digital signal processing algorithms. Unfortunately, H.264 presents some problems which make finding a general optimal solution impossible. Uncertainty derives from the many possible modes and cases which must be handled such as the nine intra prediction possibilities for each 4x4 block of pixels, anywhere from one to 32 motion vectors for each inter-coded MB, a series of decisions and filters for each edge of each 4x4 block etc. The effect is that the decoder's execution trace will vary dramatically across individual videos, frames, and MBs. In the presence of restricted memory re-

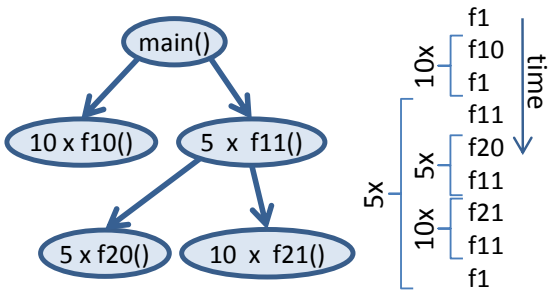
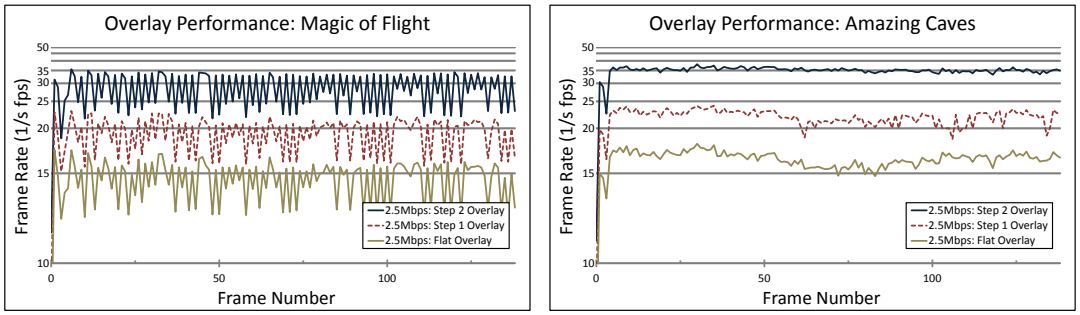


Figure 4.10: Example function call graph.

sources, no one solution will give optimal results for every video. Even if it is possible to use a decoding execution trace to find an optimal overlay mapping, solving for the optimal solution remains prohibitive since it is necessary to consider the interactions of hundreds of functions to choose from an astronomically large set of feasible mappings.

To address this issue, heuristic steps taking into account key areas of interference between functions in the function call graph are investigated, and profiling information is used to help with understanding which functions are called most frequently. The example function call graph in Fig. 4.10 illustrates a key point about interference between functions and profiling data. The nodes in the graph are given in function call order from left to right with each function looping 5 or 10 times. The execution sequence on the right indicates how many times the given function has instructions issuing, and consequently must be present in the LS. Profiling data tells us that $f21()$ is called 50 times, but it is clear that the number of times its caller, $f11()$, appears in the execution trace is 80 due to its caller-callee relationship with $f20()$ and $f21()$. $f11()$ is exposed to at least 60% more interference than any other function even though the profiler tells us it has been called at least 50% less often than any other function.

An example of this type of function is *hl_decode_mb*(), as indicated in Fig. 4.2. This function calls a wide range of other functions, and contains several loops with function calls as in the example above. Such *key* functions also tend to have elevated weights with respect to code overlay costs, as they tend to require more memory, con-



(a) *Magic of Flight* (b) *Amazing Caves*

Figure 4.11: Code overlay mapping performance for two videos. Each chart shows the worst case single region flat overlay, step one results after pulling key functions out of the flat region, and step two results after pulling priority profiling functions out. The horizontal bars indicate frame rates plotted on a time scale.

tributing further to the cost of a miss through increased DMA delays any time they must be loaded into the LS.

Overlay Design Approach

Step 1: The first step in the current approach is to ensure that interference due to key functions is always addressed. The starting point is a single region containing every function. This flat overlay mapping represents the worst case performance, but it also ensures that the code will fit into the SPU LS, as no further vertical compression is possible. The function call graph is examined to identify and prioritize key functions according to their overlay overhead cost. The cost is a function of the number of functions they are expected to call including any functions called from within loops, the priority of the called functions as discussed below, and the key function's size. Functions with the correct structure, but which do not have interference relationships with important leaf functions may be given low priority and ignored. Key functions are removed from the original region and placed into a new region as long as memory is available. The SPU portion of the call graph in Fig. 4.2 has two key functions, *hl_decode_mb*() and *mc_part*().

Step 2: Next it is possible to continue improving overlay performance as long as memory is available. The overlay mapping from the previous step overlays many functions with each other in the same region, as with those associated with inter prediction in Fig. 4.2. Since many of those functions are called with very high frequencies, they tend to interfere with one another. By profiling several videos, we can identify and prioritize the functions which tend to have very high call frequencies, and select those with the highest priority to be removed to a new region. Each move is guaranteed to improve performance as long as any other function in the losing region is called. We can continue to move high priority functions out of the shared memory location into new regions until there is no more room in memory. At this point we have prioritized and removed interference costs for the most expensive functions in the code.

The metrics used to determine function priorities are empirically determined. Function size is a factor, but in reality we find that for the vast majority of functions, it contributes only to the consumption of available memory space, and not to performance of the overlay scheme. The reason for this is the intrinsic overhead of the DMA operation which we find results in nearly identical DMA delays for functions close to 2KB in size and smaller. There are only a few exceptions in the codec where the DMA time is impacted by function size, and these tend to be key functions which were put into their own regions in the first step.

Overlay Performance

Overlay mapping performance is presented in Fig. 4.11. The effect of overlay mapping optimizations is consistent across different videos. It's also clear from the figure that the speedup in terms of decoding time from optimizing out a few key functions in step 1 is similar to the step 2 speedup which applies to more than 100 additional functions.

4.4.3 Synchronization

The dashed lines in Fig. 4.7 represent the required synchronization signals. Although the SPU-SPU signals are implemented using a 128bit DMA, all PPU-SPU signals have been implemented using the blocking SPU in and out mailboxes. A PPU-SPU signal occurs when row data is ready and the row-designated SPU is notified that work can begin. SPUs notify the PPU of their busy state by passively placing a message in their out box which the PPU checks before overwriting an SPU's dynamic MB data structure array.

Image data is not transferred between SPUs in this implementation. Instead, SPU *n* writes its results back to main memory before reporting any progress to SPU *n+1* via DMA into the next SPU's LS. Each SPU checks the progress of its predecessor before working on blocks with unresolved dependencies. Once the previous SPU reports sufficient progress, resolving any necessary dependencies, a DMA is initiated to acquire data for intra decoding and deblocking.

4.4.4 Further Optimizations

Performance improvements achieved using methods outlined in the following subsections are presented in Table 4.1. The improvements given are compared to the optimized code running on six SPUs without the individual optimization implemented.

Direct Memory Access (DMA)

DMA operations can be time consuming and are fundamental to multicore programming on the CBE. Poorly scheduled DMA operations can have dramatic adverse impact on performance. Video decoder throughput has been maximized by minimizing the number of DMA operations, minimizing the size of DMA operations, and scheduling DMA barriers in order to minimize stalls due to outstanding memory accesses.

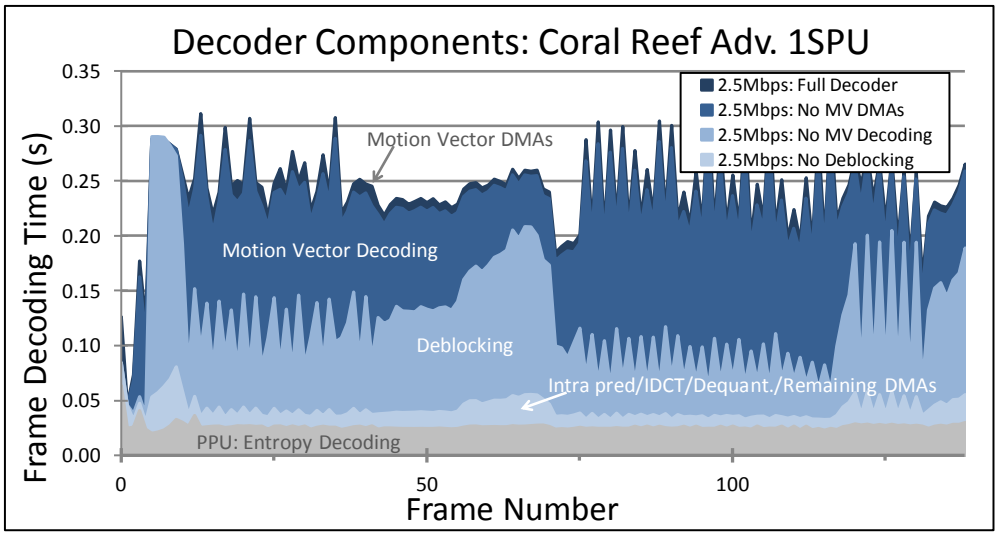


Figure 4.12: Breakdown of decoder performance by component using a single SPU. Top line in the graph indicates the decoding time for each frame. The shaded areas represent the contribution of a portion of the decoder to the total time. Motion vector decoding and deblocking are the most expensive components. The band at the bottom is the PPU (entropy decoder) contribution.

Optimization	Improvement
DMA Scheduling	8.5%
Inlining	7.2%
SIMD Compiler	2.1%
IDCT unrolling	0.3%

Table 4.1: Optimization improvements.

The parallelized decoder implementation requires four important memory operations: 1. Before beginning the row decoding process, SPUs must retrieve the static MB data structure. 2. Before decoding a group of MBs, image data from the previous row for deblocking (which includes data needed for intra prediction) must be gathered. 3. During motion compensation, referenced image data must be retrieved for each motion vector. 4. Decoded image data must be returned to main memory. DMA completion barriers are pushed as far out as possible in order to hide the DMA operation with other work in the SPU. In this way it is possible to significantly reduce or prevent stalls associated with DMA barriers.

Inlining

The process of dividing the *FFmpeg* source code into separate object files for each function has the side effect that the *gcc* compiler is no longer capable of inlining functions as specified in the original source code. Due to the importance of code overlay performance, it is not always desirable to inline the same functions in the parallelized code as the original sequential code. Since inlining changes the size of the calling function by inserting the callee into its code, the DMA overheads and LS memory are both affected. Experiments indicate that in most cases inlining did not improve performance, and in some cases reduced performance. However, a couple of very small functions, *av_clip()* and *copy_block()*, which are called with extremely high frequencies enabled significant performance improvements when rewritten as macros to force inlining.

Single Instruction Multiple Data

One of the major strengths of the CBE is its SIMD capabilities, available in both the PPU and SPU. The portion of code implemented for the PPU is not well suited for SIMD instructions, but there are many opportunities for SIMD execution in the SPU code. Some intra prediction functions, IDCT, and deblocking have properties suitable for SIMDization. The PPC Altivec functions available in the *FFmpeg* source provide this targeted improvement, but the Altivec functions must be transcoded to execute on the SPU, and have not been included in this work. Extensive experimentation with IBM's SIMDizing compiler, *spu-xlc*, on applicable functions achieved limited SIMD improvement. Alvarez et al. [6] also reported very limited performance improvement in integer transform functions using SIMD.

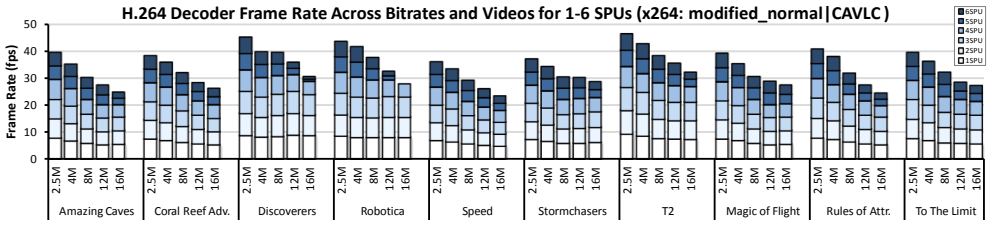
Video	Description
Amazing Caves	cave scenery
Coral Reef Adv.	active underwater scenes
Discoverers	night sky, lab workers
Robotica	fast panning and action
Speed	sporting activities
Storm Chasers	flight, storm clouds
T2	dark scene from cinema preview
Magic of Flight	glider, fast panning
Rules of Attr.	modern multiframe cinematography
To the Limit	mountain climbing

Table 4.2: Benchmark videos.

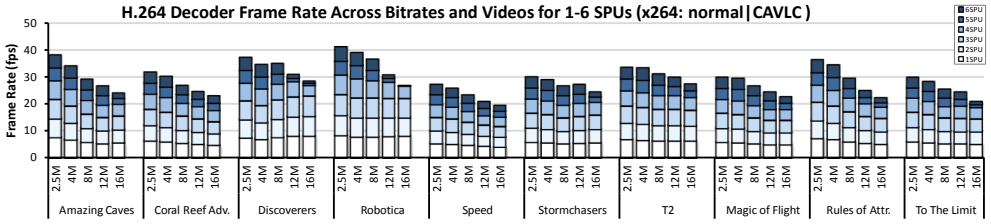
4.5 Raw Decoder Performance Results

The benchmark videos used to test the decoder implementation were taken from Microsoft's WMV HD demonstration page [115], and are listed in Table 4.2. The source videos were transcoded into H.264 1920x1080 (1080p) format at five different bit rates from 2.5Mbps to 16Mbps CAVLC and CABAC using the *x264* H.264 encoder [154] integrated into *FFmpeg*. The videos were encoded using the *x264* presets: *baseline*, *normal*, and *hq*. Also tested was a modified version of the normal preset with B frame encoding removed, referred to in Fig. 4.13 as *modified-normal*.

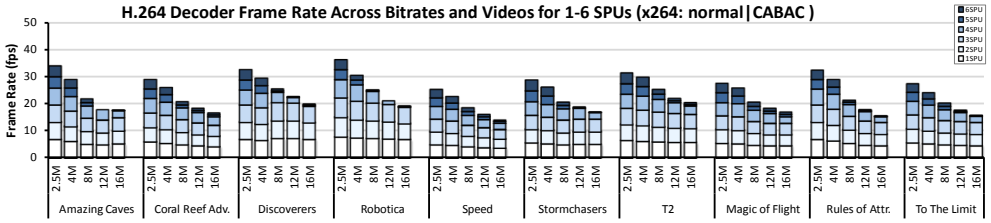
Decoder performance is measured on the Sony's Playstation 3, 3.2 GHz Cell Processor (limited by Sony for access to six of the CBE's eight SPUs) running Linux Fedora 9. Data was collected for ten videos, four encoder preset configurations, and five bit rates, decoding on 1-6 SPUs. Performance results are presented in Fig. 4.13. The presented charts indicate performance for the modified-normal and normal presets using CAVLC encoding as well as normal and hq presets using CABAC encoding for each video by bit rate. Each bar indicates the performance in terms of decoding frame rate for 1-6 SPUs with each SPU adding to the frame rate. Fig. 4.12 illustrates the contribution of various decoder components to the overall frame rate. The largest



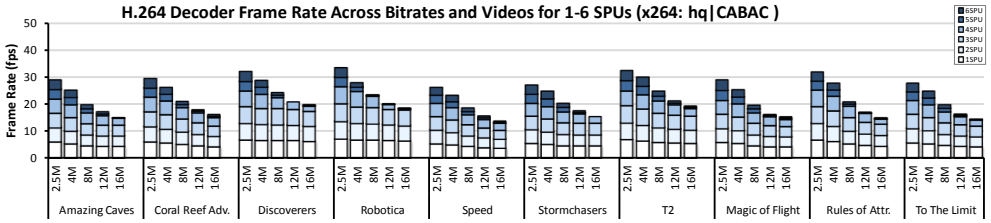
(a) Modified Normal CAVLC



(b) Normal CAVLC



(c) Normal CABAC



(d) HQ CABAC

Figure 4.13: Raw parallelized performance results.

contributors are MV decoding and deblocking, making them important for further optimization and SIMD vectorization.

Results presented by Baik et al. for Samsung's parallelized decoder [12] implemented on four SPUs show an average frame rate of 20.5fps across four 1080p test videos CAVLC encoded with B frames at 2.5Mbps. The presented implementation achieves an average 25.23fps or a 23% improvement when decoding similarly encoded video streams on four SPUs. Substantial further improvements are achieved due to

enhanced scalability using additional SPUs as seen in Fig. 4.13a. For videos encoded at the same bitrate, but using the higher quality and much more expensive CABAC encoder and *x264*'s *hq* encoding settings, the decoder still slightly outperforms Baik's implementation [12] even when limited to four cores. The implementation achieved a "best case" average framerate of 34.94fps on 2.5Mbps *modified-normal* CAVLC encoded video streams on six SPUs, and a "worst case" entropy decoder limited average framerate of 15.43fps on 16Mbps *hq* CABAC encoded video streams.

Decoder performance results suggest that additional cores will allow further performance improvement for most videos. In some cases, at higher bit rates the PPU entropy decoder becomes dominant, and no further gains are possible even with additional SPUs. This is evident in Fig. 4.13d, with the reduced or missing performance blocks for SPUs 5 and 6 at 8Mbps and higher. CABAC encoded videos are costlier to decode in the PPU, so maximum frame rates are more limited compared to CAVLC. When decoder performance is not completely limited by the entropy decoder, it is clear that the improvement gain from each additional SPU is slightly reduced from the previous SPU due to increased demand on the PPU and memory bandwidth requirements. However, the frame rate improvement with the addition of SPU 6 still averages 11.3% across all tested bitrates and 12.6% for bitrates below 12Mbps for CAVLC. The average improvement due to SPU6 for CABAC encoded videos is limited by the PPU/entropy decoder dominance beyond four SPUs at higher bitrates, but still averages 6.6% across all tested bitrates and 9.3% for bitrates below 12Mbps.

4.6 Power State Management Policies

4.6.1 Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) specification [67], seeks to standardize processor power management interfaces. The ACPI standard defines processor performance states P0 – Pn, where each P state is associated with a frequency-

Table 4.3: Video decoder configurations.

C Managed	Processor scaling through the PE scheduler is active, and each PE is automatically transitioned from C0 to C3 according to the sleep policy described in Section 4.6.5.
C+P Managed	The *C Managed* configuration is implemented with the addition of P state management according to the policies described in Section 4.6.6.
Full Speed	No processor scaling is implemented. All available PEs are utilized and running at P0. This configuration without C state management produces the *normal* power consumption used against all reported power figures.

Table 4.4: C State power consumption.

	$C0_{active}$	$C0_{idle}$	C3 [68]
P	1.0	0.4	0.2

Table 4.5: P State power consumption [81].

	P0	P1	P2	P3	P4	P5	P6	P7
f_{scale}	1.00	0.91	0.84	0.74	0.65	0.57	0.48	0.39
Volts (V)	1.12	1.09	1.06	1.03	0.99	0.96	0.93	0.90
P	1.00	0.86	0.75	0.62	0.51	0.42	0.33	0.25

voltage pair. P0 gives the maximum performance while higher numbered states give successively reduced speed and power consumption by lowering frequency and voltage points. State P*n* is the system's lowest performance state utilizing the smallest amount of energy.

ACPI also specifies four processor power states or sleep states, C0 – C3. All P states are a subset of state C0, the only C state in which instructions are executed. Higher numbered states represent successively deeper sleep with lower power and longer wakeup latencies.

Current state of the art processors offer sophisticated fine grained DPM support. The recent ACPI compliant multicore offerings from AMD and Intel are capable of supporting independent C states and clock frequencies for individual core and "uncore" architectural elements [58] [8] [82]. Future processors may be expected to support independent dynamic voltage scaling for each core as well.

4.6.2 Agile Performance Scaling

The quality of a video decoding power management scheme depends on achievable power consumption rates as well as the frequency and severity of any frame deadline misses. Optimal decoder performance is realized by selecting the slowest possible P state still meeting the frame decoding deadline. To achieve near optimal performance, the scalability of the parallelized video decoding implementation is exploited to match as closely as possible the ideal decoding rate for each frame.

The goal is to finish decoding each frame as close to its deadline as possible. The inter frame decoding time given in seconds, t_{frame}, is defined as the inverse of the desired frame rate. For example, at 25fps, the time available to decode each frame is $1/25 = 40ms$. From t_{frame}, a target row decoding time is calculated defining a *target decoding rate*. The target row time, t_{row}, is given by t_{frame}, the resolution dependent number of MB rows, n, and, t_{inter}, the *inter frame time* associated with any work occurring between finishing the last MB of the previous frame and the start of the first MB in the current frame:

$$t_{row} = \frac{t_{frame} - t_{inter}}{n}$$

For each new MB row, the *PE scheduler* evaluates decoding progress with respect to the target row decode rate. Fig. 4.14 gives an example of target decoding rates for individual frames. The number of PEs selected for use, m, is either incremented or decremented by the scheduler at the beginning of each row depending on whether measured decoding progress is behind or ahead of the target progress indicated by the dotted sawtooth in the figure. If measured progress is lagging, m is incremented, if measured progress is ahead of target progress, m is decremented. The implementation on the Sony PS3 with 6 SPUs available will not be increment m beyond 6.

Fig. 4.14 illustrates dependencies between PEs and shows how dynamic PE

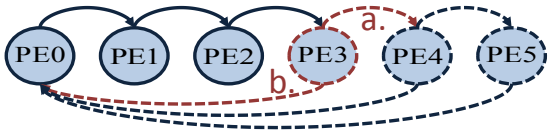


Figure 4.14: Processing element performance scaling.

scheduling occurs. In the parallelized decoder implementation, instead of PE3 continuously polling PE2, on which it depends for progress updates, PE2 actively pushes progress information to PE3. For this reason, when a PE is assigned an MB row, it is also assigned a target PE. When PEs 0 through 2 are busy and PE3 is next to receive an MB row, PE3's target depends on the value of m. In the example, if $m > 4$, relationship a in the figure is instantiated and the next row will be assigned to PE4. Otherwise dependency b is instantiated and the next row will go to PE0.

Fig. 4.16a illustrates the performance scaled parallel system's behavior. The value of m is represented with a '+' for each MB row. The scheduler's agility is evident in its immediate response to workload variations, very closely matching the target decoding rate. In frames 0 and 2, m is almost immediately maximized and remains at its maximum value because the system fundamentally lacks resources needed to meet the target decoding rate.

4.6.3 Dynamic Voltage and Frequency Management

DVFM, also known as Dynamic Voltage and Frequency Scaling (DVFS), is a common technique for minimizing power consumption in computing systems [71] [128]. *Dynamic* processor power consumption, P, is quadratically dependent on its operating voltage, V, and linearly dependent on clock frequency, f as given by the well known relationship $P \propto V^2 f$.

Reducing operating voltage has a profound effect on dynamic power making it a focus of DVFM schemes. Lowering clock frequency enables a lower operating voltage, substantially improving power consumption but increasing execution times. Effective

DVFM policies attempt to choose the lowest possible frequency still meeting any given time constraints. In video decoding, minimum power consumption is achieved by selecting a frequency for each frame such that frame decoding finishes at precisely the frame deadline. Running the processor at full speed typically results in *slack time* between the end of frame decoding and the frame deadline. Optimizing away slack time by slowing the clock frequency, which enables reduced operating voltage, is the key to successful DVFM for video decoding.

4.6.4 Sleep States

In addition to DVFM, which only addresses active power, *static* or leakage power may be dramatically reduced through the use of various low power idle or sleep states. Aggressive sleep states may completely disable system clocking and even disconnect affected systems from the supply voltage. More aggressive configurations can reduce overall system power consumption to very near zero, but require substantially more time and energy than less aggressive configurations. In general DPM policies are designed to put idle processors to sleep so that the power saved in the sleep state is greater than the power and time costs associated with entering and leaving the low power state.

4.6.5 Sleep State Selection

In this implementation, the system's C or sleep states are managed autonomously by each PE. Hardware activation of sleep modes is emulated with a simple time out. Only the deepest sleep state required by ACPI, C3, is considered. When the system has been idle for five times the round-trip transition latency from C1 to C3, the PE is placed into state C3.

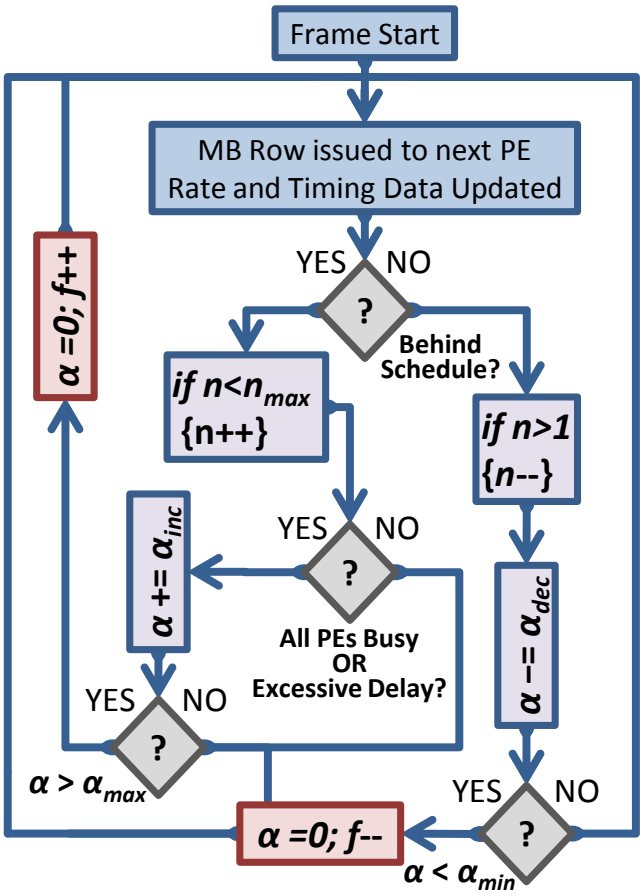
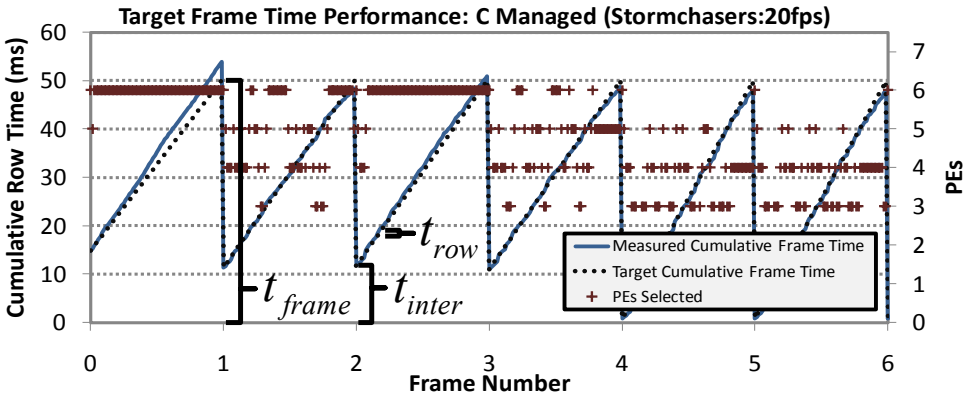


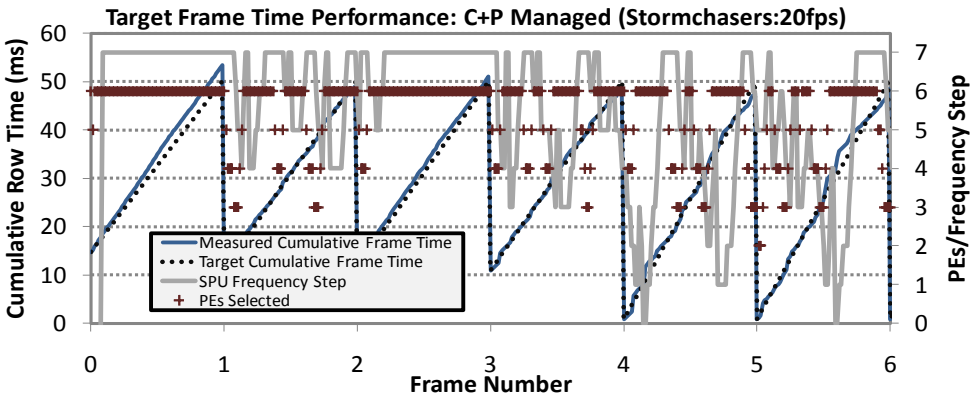
Figure 4.15: PE allocation and P (voltage-frequency pair) state selection. The flow chart illustrates rate controlled PE allocation and P state selection. The number of PEs, n, is either incremented or decremented depending on how decoding time matches the target time for each MB row. The operating frequency, f (in effect the current P state) is increased only when PE scaling is not possible.

4.6.6 DVFS State Selection

The *P state manager* makes frequency and voltage point selections as a second order response to target decoding rate matching performance. PE scaling decisions are designed to maintain the target rate directly, whereas P state decisions depend on the status of the PE scheduler. The implemented *slow as possible* P state management policy encourages moving to a lower power, i.e. higher numbered, P state when possible, selecting higher performance states only when absolutely necessary. In essence, while



(a) Fixed frequency PE scheduling: The number of PEs requested is always minimized to maintain the target decoding rate.



(b) Frequency and Voltage state management: The light solid line illustrates run-time frequency transitions. Reduced operating frequency forces the scheduler to *request more PEs* than were needed in the fixed frequency configuration.

Figure 4.16: PE scaling to meet target decoding times in a system with six PEs. Processing of each frame is represented with a slope beginning after t_{inter}, progressing at t_{row} per MB row, and finishing at t_{frame}. Measured row decoding time is plotted against the target time. The total number of PEs requested at the beginning of each MB row is indicated by the '+'.

the PE scheduler is tasked with meeting the target decoding rate, the P state manager is charged with running the system as slowly as possible.

The PE scaling and P state decisions are made at the beginning of each MB row as illustrated in Fig. 4.15. When the frame decode time is slower than the target time performance is augmented, but decrementing the P state (incrementing frequency) is only chosen for this purpose when no additional PEs are available. On the other hand, when the decoder is ahead of schedule, the P state is always incremented (frequency is

decremented), and number of PEs is decremented unless the decoder is considered to be on schedule.

Additionally, a tracking metric is implemented to dampen or accelerate P state transitions when necessary. The metric variable is incremented or decremented prior to P state selection. P state modifications are made only if the metric exceeds predefined high and low threshold values indicating the need to raise or lower the P state. As an example, the metric is used to dampen transitions to slower performance states while the decoder works the last few MB rows. This is achieved by reducing the dampening metric increment value. The effect is to reduce the likelihood of misapplied slowing at frame end, reducing deadline violation probability.

4.7 Power Management Results

4.7.1 Experimental Setup

Video Decoder

The performance scaling scheme and DPM policies have been implemented to run on the CBE's PPU as part of the decoding loop responsible for entropy decoding. Timing instrumentation has also been added to support the processor scaling and DPM schemes in the PPU. SPUs are instrumented with timing functions for reporting the duration activity while decoding each row.

The decoder has also been instrumented to accurately emulate frequency scaling in both the PPU and SPUs by executing *computed delays* immediately prior to any synchronization activity or idle periods. Delay durations are computed based on the active frequency scaling factor, f_{scale}, which takes a real value in the range $[0, 1]$. A scaling factor $f_{scale} = 1$ represents the fastest available clock frequency, while smaller values represent clock speed as a ratio of the maximum frequency $f = f_{target}/f_{max}$. The implemented factors are presented in Table 4.5. The executed delay time, t_{delay}, is calculated using time measured since the last synchronization activity or idle period, t_{active}, by

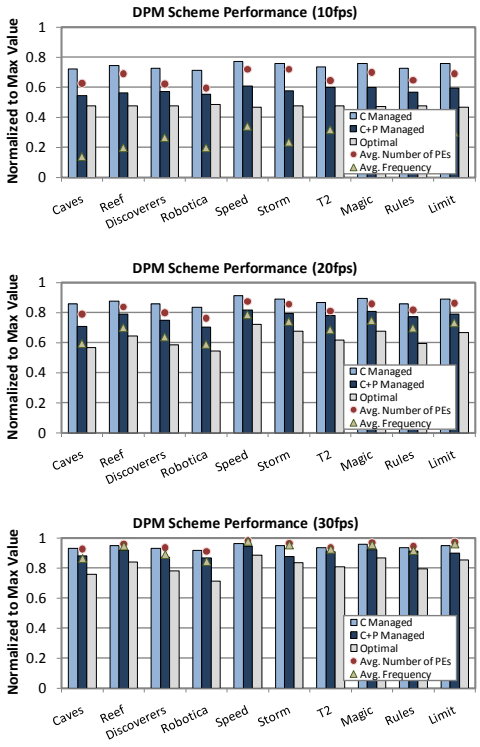


Figure 4.17: Power Performance at 10, 20, and 30fps target rates. *C Managed* mode includes only PE scaling and C or sleep state management, while *C+P Managed* mode additionally includes P or DVFM state management. The *normal* power configuration uses full speed + max PEs, and the optimal power figures are calculated based on idealized power state selections for just in time frame decoding.

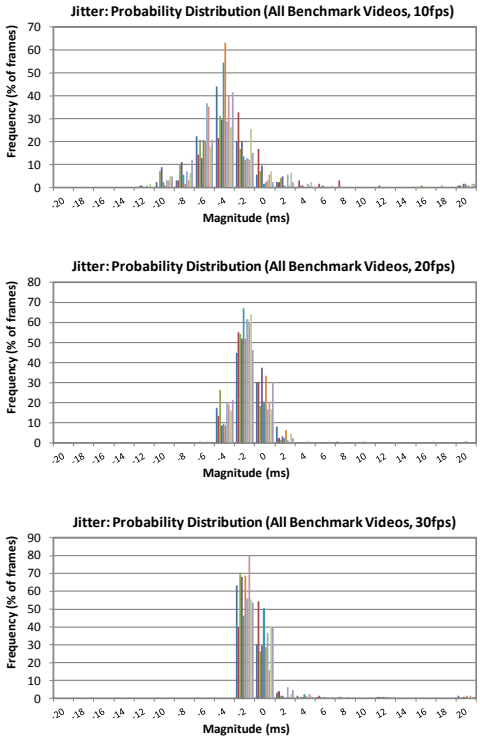


Figure 4.18: Jitter distribution around the target frame completion time indicated as 0ms at 10, 20, and 30fps target rates. Frequency and voltage are dramatically reduced without introducing significant deadline misses.

$$t_{delay} = \frac{t_{active}}{f_{scale}} - t_{active}$$

In addition to required frequency scaling delays, latencies associated with switching between P states and C states must be considered. For power management emulation, latency values have been chosen based on reported figures for the Intel Xeon 5600 series processor. The P state transition latency used in the system is $2\mu s$ [82].

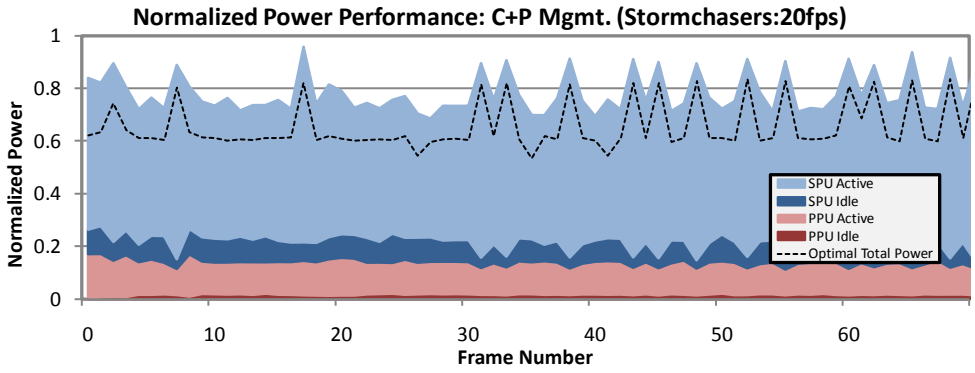


Figure 4.19: Power breakdown: Active and idle power calculated using the power model in Section 4.7.1 and data reported by the parallel decoder running on the CBE are shown for both PPU master and SPU processing elements. Calculated optimal performance is indicated by the dashed line.

The reported round trip C0-C3 latency used is $40\mu s$ [58]. Introducing appropriate computed delays into the decoder provides extremely precise behavioral emulation of a frequency scaled parallelized H.264 video decoder.

Power Model

The frequency scaled decoder provides accurate behavior and timing information for all PEs in the power managed parallel video decoding system. Active and idle time information for each PE are collected at MB row resolution and analyzed using the processor power model. Power ratios chosen for C and P state power consumption are given in Tables 4.4 and 4.5 respectively. All power ratios are given relative to the average processor power consumption given the processor is not idle and is operating in the highest performance state, P0. This is the *maximum operating power* for the processor defined to have a value of 1. All lower performance processor states' power consumption are defined in terms of the maximum operating power.

4.7.2 Power Performance Results

Processor Scaling and DVFM

The runtime behavior of the PE scheduler and P state managers are illustrated in Fig. 4.16. Fig. 4.16a in contrast with Fig. 4.16b illustrates the behavior of PE scaling in the absence and presence of frequency scaling. When the workload drops, the scheduler immediately responds by reducing the number of active PEs signaling the P state manager to reduce performance. Consequently, the PE scheduler tends to use the maximum number of cores while the system runs at a reduced frequency, approximating optimal system behavior predicted by Cho et al. [38].

Overall Power Performance

The charts in Fig. 4.17 illustrate power performance across all benchmarks at 10, 20, and 30fps. Power performance is given for the C and C+P configurations described in Table 4.3. Optimal performance has been calculated from the reported total work required across all PEs by the *Full Speed* decoder. To calculate optimal power, total work is distributed evenly across all PEs giving a best case decoding time. This time is scaled using the smallest available core frequency which will not violate the frame deadline. The total energy consumed in each processor is calculated from the scaled decoding time at the appropriate power rate from Table 4.5 and any remaining idle time at the C3 power rate given in Table 4.4. The energy consumed during t_{frame} gives the optimal power figure for each frame. A power breakdown for the C+P managed system is illustrated in Fig. 4.19 along with the target optimal power performance.

All power figures are normalized to the *Full Speed* decoder without C state management. Fig. 4.17 illustrates the increasing power advantage as available frame slack time increases at reduced frame rates. On average, the C+P managed configu-

ration comes within 11% of the optimal power performance, and out-performs the C managed configuration by 10%.

Deadline Accuracy

Experiments have shown that frame deadline misses, or *jitter*, may be the most important factor in perceived video quality [30]. Frequently large frame buffers are used to address the jitter problem but these can introduce undesirable latencies and large memory requirements posing problems for resource constrained embedded systems in particular. Buffering even a few video frames can be extremely costly. This is particularly true of high resolution frames such as HD. A single raw 1080p video frame requires 6.2MB of buffer memory.

Fig. 4.18 illustrates the deadline accuracy of the C+P managed decoder. The power management scheme is found to not introduce significant jitter. When the system is under reduced stress, as in the 10 and 20fps charts, the number of significant deadline misses is small. When the system faces a heavy workload, the number of significant misses will dramatically increase. However, all of the misses greater than 10ms experienced by the C+P managed decoder are *unavoidable*, meaning that the decoder cannot make the deadline even when running at the highest operating frequency and using the maximum number of PEs.

4.8 Conclusion

Parallelizing the *FFmpeg* H.264 decoder on the CBE requires immense effort due to the codec's notorious complexity, but there are substantial opportunities for parallelization and optimization. Primary implementation issues arise from SPU memory limitations and the large size and complexity of the source code and H.264 specification, synchronization requirements, and data dependencies. Significant scalable performance gains have been demonstrated from parallelization of the H.264 decoder and its implemen-

tation on the Cell Broadband Engine. We can expect to see an increasing number of cores in embedded applications and processors like the CBE, amplifying the need for scalability as efforts continue to improve performance in the face of physical barriers for monolithic processors.

A processor scaling DPM framework capable of extracting valuable power savings for parallelized video decoders has been presented. Agile performance scaling in the H.264 decoder enables precise decoding rate control which can be translated into effective power state management for near optimal power performance while successfully avoiding deadline misses. Future work includes addressing possible advantages of workload estimation, and applications for performance processor scaling in multithreading multicore processors.

Chapter 5

HIGH PERFORMANCE INSTRUCTION MAPPING FOR SCRATCHPAD MEMORIES

Performance and power consumption are critical design considerations in embedded systems. System memories may consume half of the limited power budget [86], which has motivated a tremendous amount of work aimed at improving memory performance and efficiency. scratchpad memories have grown dramatically in importance in recent years, particularly as performance requirements for embedded systems become more like those traditionally associated power hungry general purpose desktop computers.

Scratchpad memory (SPM) enhanced embedded processors provide fast and power efficient compiler/programmer managed storage for accessing instructions and memory. Performance is improved over a local cache by eliminating overhead and limiting unpredictability due to hardware cache management. scratchpad memories (SPMs) are software managed data and instruction storage structures integrated into a processor or MPSoC architecture much the same as system cache. The SPM enables performance and silicon area advantages over the system cache by shedding hardware required for cache management and relying on compile time and programmer directives to adequately manage its use. How best to utilize the SPM is a difficult problem which has been heavily explored over the past decade.

There are a number of approaches for selecting what to place into the SPM and when to place it there. In the problem considered here a code partition is defined on the SPM, and a large code set must be dynamically mapped into the available space by assigning multiple segments of code to the same address in memory. An example of this situation is the IBM Cell Broadband Engine (CBE) which provides eight Synergistic Processing Units (SPUs) each with a software managed 256KB Local Store SPM. In the CBE model, *all* code to be executed in an SPU must be mapped to the local SPM [127].

See Appendix B for additional details on the CBE. This model promises to become increasingly important as multicore SoCs make their way into multitasking handheld devices. Here each processing core may be assigned a compute intensive task such as video coding in a real-time resource constrained environment.

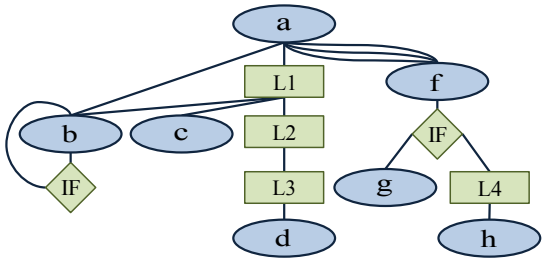
In this work a Code Overlay Generator (COG) Algorithm for identifying high performance overlay mappings designed to minimize overhead is introduced. Two extensions to the COG algorithm are presented and the performance of the implementations against a previously published algorithm and the automatic overlay mapping generator in the IBM Cell SPU compiler, *spu-gcc* are compared.

In the next section previous and related work are discussed. Details on code overlays are given in Section 5.2 and overlay performance and cost models are introduced in Section 5.3. The COG algorithm and extensions are presented in Section 5.4 followed by an analysis of expected performance compared to the previous algorithms in Sections 5.5 and 5.6. The experimental setup and results are presented in Section 5.7, followed by Conclusions.

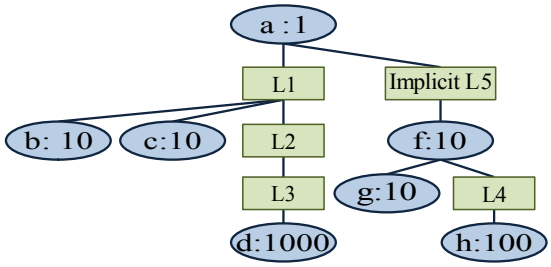
5.1 Previous Work

There are many good references highlighting the large amount of work related to optimal partitioning and assignment for SPM. Most of the available work targets energy minimization which generally corresponds to reducing the number of memory *misses* during program execution, as additional overhead incurred due to a miss results in greater energy consumption than with a memory *hit*. The scheme seeks to *minimize instruction misses*.

Existing work may be categorized into either instruction mapping, data mapping schemes, or both, as well as either static or dynamic allocation techniques. A code overlay mapping generator for dynamically mapping instructions to memory at runtime has been developed. The implementation does not require profiling information at com-



(a) Example GCCFG as might be generated from source code.



(b) Simplified GCCFG with function base costs shown.

Figure 5.1: Example GCCFG (a) before and (b) after reduction.

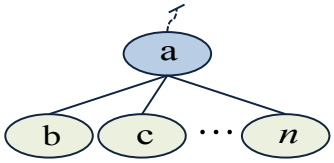


Figure 5.2: Flat call graph structure. Note that this structure appears twice, headed by functions *a* and *f*, in the graph of Figure 5.1b since loops may be interpreted as multiple instances of the same function node.

pile time. Also presented is an improved scheme for modeling relationships between code objects at runtime to better predict instruction memory misses based on structural analysis of the program. Good overviews of the large body SPM related efforts is given by Egger et al. [55], Verma et al. [152], and Janapsatya et al. [86].

Several previous efforts focus on temporal locality of code. Early work in this area investigates the concurrency of code modules based on branches of the execution call graph [48] [140], but do not consider the effects of control structures such as loops. Similar to the present work, Steinke et al. [141] views the problem in terms of minimizing memory accesses, and evaluates the structure of the program in terms of

basic blocks and functions to formulate an ILP problem. Udayakumaran et al. present an algorithm which looks at timestamps in code sections to determine temporal locality [149], Janapsataya et al. [86] introduces a concomitance metric which relies on profiling trace data, and Angiolinie et al. presents a dynamic programming algorithm which also requires trace data [10]. Egger et al. [55] implements a paged SPM management and prefetching scheme. These schemes rely on profiling information which can be impractical or inflexible particularly where program execution paths vary widely on different input data.

Verma et al. [152] and Pabalkar et al. [124] are most similar to the work presented here. Verma uses a first-fit heuristic to assign objects to the SPM, but their scheme results in a static mapping which omits objects that do not fit by placing them back in main memory. Pabalkar's algorithm faces a related problem when attempting to assign all code modules to the SPM due to potential deadlock which is not handled. More details on SDRM deadlock are provided in Section 5.5.2. Additionally, the existing overlay mapping generation algorithm included in the *spu-gcc* compiler provided for the CBE by IBM addresses the current problem [130], but the algorithm gives limited performance except under special conditions. Results are compared with the SDRM and *spu-gcc* solutions in this presentation.

5.2 Code Overlay

Code Overlay is a technique for mapping instruction code onto available memory in which the code would not otherwise fit. In designing an overlay mapping, available memory is partitioned into one or more fixed *regions* to which one or more code *segments* may be assigned. Every segment assigned to a given region is mapped to the same address in the SPM. The size of a region in memory is precisely that of its largest mapped segment. Code may be stacked into segments so that the start address of the second code element follows the end address of the first element and so on.

Table 5.1: Function control relationships.

function calls	one function calls another function
function calls in loop	function call within a loop
conditional function calls	a function conditionally calls function
recursive function calls	a function calls itself

Normally instructions may be assigned to segments at object file, function, or basic block resolution, and any number of elements may be assigned to one segment. In this work, overlays are created using function or object file resolution, where an object file may contain one or more functions. The object file resolution is a limitation of the GNU *gcc* linker, *ln*. When the code overlay scheme is implemented and the code is executed, each memory region always contains exactly one code segment at any given time during execution. When instructions are requested which are not currently present in the SPM, the appropriate segment is loaded into its assigned region overwriting the current segment. This event is referred to as an overlay *miss*. Each miss has an associated overhead which is defined as the amount of time expended to load the missing segment. This overhead is proportional to the size of the code segment. The cumulative overhead in terms of time associated with execution of a program using a specific overlay mapping is the mapping's *total cost*. Additional details and CBE implementation considerations have been provided in the previous chapter at Section 4.4.2.

5.3 Overlay Miss Model

By analyzing source code, we can identify key characteristics which will be important for producing an accurate model of program behavior to effectively reduce overlay misses during execution. Accomplishing this miss reduction requires identifying any functions in the code which might be mapped to the same memory address, or *overlayed*, and determining the relationships between them expected to result in overlay misses.

5.3.1 Graphical Representation of Code

In order to create an abstract graph representation of the input program, it is necessary to address four important control flow relationships between functions as shown in Table 5.1. An enhanced Control Flow Graph (CFG) referred to as a Global Call Control Flow Graph (GCCFG) as presented by Pabalkar et al. [124] is constructed. The GCCFG is similar to the CFG, but with control flow relationships explicitly represented as nodes. Figure 5.1 illustrates an example GCCFG which might be generated by identifying jumps or function calls, loops, and conditional statements in the source code. In this work, analysis is performed on *cfg* output generated by *gcc* using the *-fdump-tree-cfg* switch [60]. The *cfg* dump is parsed to collect control information at the granularity of basic blocks from which loops and function call information is collected. The implementation is currently limited to function level resolution by the existing overlay manager available for the CBE used in the experiments. The function level graph generated from *cfg* data is simplified in four steps:

1. First we ignore recursive function calls as indicated in the graph at function *b*. Since we are concerned with interference between functions, the effect of a recursive call is that the code necessary to run the called function is already in memory, resulting in no overlay miss.

2. In the second step, we will treat functions inside conditional statements including function pointers, which may represent multiple functions, as if they interfere with one another in much the same way as if they were called outside of any conditional control structure. The impact of this transformation on the model's accuracy is dependent on a large number of variables including the detailed behavior of the code on a given input. We observe that two conditionally executed functions called by the same parent can have an interference relationship simi-

lar to that of two functions called by the same parent outside of any conditional control structure. Without actually executing the code, it is not possible to fully anticipate this relationship at compile time. We do know that if each of two exclusive conditionally executed functions are called at any time during program execution then there will be interference between them limited by the number of times the parent function is called.

3. Next we identify and remove *loop augmenting edges* from the original GCCFG. These are edges between two functions which also have a loop node between them as in the edge (a, b) in Figure 5.1a. The effect of the loop augmenting edge is to increase the number of iterations in the loop by the number of augmenting edges. We do not know the size of the loop to begin with, so we will ignore the additional edges by subsuming them into the augmented loop node.

4. Finally, we generate *implicit loops* from the original GCCFG whenever we find multiple edges between two nodes, as seen between a and f in the figure. Since calling the same function multiple times is equivalent to calling the function from within a loop, we insert an implicit loop node to the graph between the calling and called functions and remove all but one edge.

Now we have produced the simplified GCCFG structure in Figure 5.1b. In the following sections we begin characterizing interference relationships and costs.

5.3.2 Counting Overlay Misses

For each function, we will identify *base*, *return*, and *total* interference values. The *base interference*, I_b, represents the number of times we expect a function to be called during program execution. Starting from the root node we assign I_b a value of one. Based on the assumption that no code is present in the SPM at the start of program execution, every function will have an overlay overhead of at least one miss on its first call.

Base interferences are assigned by traversing the graph depth first. When we visit a child, I_b remains the same as the parent node unless we pass through a loop between them. Each time we descend through a loop node, the current value of I_b is multiplied by a loop factor. A factor of 10 is used in Figure 5.1b. When returning upward through a loop node, I_b is divided by the loop factor. In this way, a base interference is assigned to each function node with geometrically increasing values as we descend into loops.

Definition 1 Base interference, $I_b(f)$ *is the number of times a function is expected to be called during program execution, i.e. the number of times the function must be in memory due to a call from another function.*

The *return interference*, I_r, for each function represents the number of times the function must be present in memory due to function call returns. If we consider the call graph in Figure 5.2, the order in which the functions must be present in memory is

$$< \mathbf{a}, b, \mathbf{a}, c, \mathbf{a}, ..., \mathbf{a}, n, \mathbf{a} >$$

Notice that the parent function, a, must be present in memory for all $n-1$ function call returns, where n is the number of functions in the graph.

The value of I_r for each function is the sum of the I_b value of its *immediate children*, which includes the children of any loop nodes but not the loop nodes themselves. As an example, functions a with immediate children b, c, d, f and f with immediate children g, h in Figure 5.1b have I_r values 1030 and 110 respectively. All other functions in the graph have $I_r(f) = 0$, as they do not call other functions.

Definition 2 Return interference, $I_r(f)$ *is the number of times a function is expected to call other functions, i.e. the number of times the function must be in memory due to a*

return from a function call:

$$I_r(f) = \sum_{\forall c} I_b(f_c) \ : f_c \textit{ is an immediate child of } f$$

Now we may calculate the *total interference*, I_t, for each function as the sum of its base interference and return interference values. Total interference represents the total number of times we expect each function must be present in memory during program execution. It is important to highlight that in the absence of conditionals, if we have the actual loop sizes recorded in the GCCFG, then $I_t(f)$ will precisely represent the *actual number of times* the function must be present in memory during program execution. This may also be interpreted as the number of overlay misses we expect if all functions share the same memory address in a single region.

Definition 3 Total interference, $I_t(f)$*: The total number of times the function,* f*, must be in memory:* $I_t(f) = I_b(f) + I_r(f)$

Taking the sum of total interferences for every function, f, in program, P, we find the number of overlay misses expected in the worst case:

$$\textit{interference}_{max} = \sum_{\forall i: f_i \in P} I_t(f_i)$$

Worst case overlay performance occurs when available memory restricts overlay mappings to a single region the size of the largest function in the program. This circumstance represents the smallest possible size for a valid mapping. With one function per segment, interference overhead occurs between every function during every function call and return. Again, in the absence of conditionals, and given the actual loop sizes in advance, $\textit{interference}_{max}$ represents the *actual worst case overhead* of the program in terms of overlay misses.

The actual DMA overhead of an overlay miss in terms of time can be calculated from DMA size. Therefore, we can estimate the actual worst case *overhead cost* due to

overlay misses in terms of time by summing the products of total cost and calculated DMA overhead for each function.

5.3.3 Interference Between Functions

In order to establish the quality of an overlay mapping, we must define the *interference* relationship between code segments. To simplify the analysis, we will restrict the contents of a code segment to a single function, and equate segment interference with function interference. We will explore the potential benefits of clustering functions into segments in Section 5.4.6.

Definition 4 Interference between functions, *inter*(u, v) *is the number of times two functions,* u*, and* v *are expected to replace each other in memory if both are mapped to the same region.*

We consider two types of interference relationships defined by the *lowest common ancestor* (LCA), between two functions, u and v. $LCA(u, v)$ is defined as the loop or function ancestor common to both u and v which has the greatest depth in the GCCFG.

$$LCA(u, v) = \text{lowest common ancestor between } u \text{ and } v$$

For example in Figure 5.1b, $LCA(c, g) = a$, $LCA(g, h) = f$. The two types of interference are as follows:

1. Neither u nor v are the LCA. In this case, neither function is a *descendant* of the other and the base interference of the LCA node determines the interference between the two functions. In Figure 5.1b, $inter(b, d) = I_b(L1) = 10$, and $inter(g, h) = I_b(f) = 10$.

2. Either u or v is the LCA. Here, the function which is not the LCA, say v is a *descendant* of the other function u. The interference between the two functions is determined by the sum of the base interference of the LCA node and

Algorithm 1 Interference cost algorithm.

FIND INTERFERENCE BETWEEN FUNCTIONS

1: F {initialized set of all functions}
2: $LCA(u, v)$ {initialized common ancestors $\forall u, v \in F$}
3: $I_b(f)$ {initialized base interferences $\forall f \in F$}
4: $shield(u, v)$ {initialized reference to the function which shields u from v}
5: $inter(u, v)$ {uninitialized function interferences}
6: **for** $\forall u, v : u, v \in F, u < v$ **do**
7: **if** $u = LCA(u, v)$ **then**
8: $inter(u, v) = I_b(u) + I_b(shield(u, v))$
9: **else if** $v = LCA(u, v)$ **then**
10: $inter(u, v) = I_b(v) + I_b(shield(v, u))$
11: **else**
12: $inter(u, v) = base(LCA(u, v))$
13: **end if**
14: **end for**

the base interference of the first child function of the LCA node on the path to the descendant function. In Figure 5.1b, $inter(a, d) = I_b(a) + I_b(d) = 1001$, and $inter(a, h) = I_b(a) + I_b(f) = 11$.

In cases where two functions have more than one lowest common ancestor, as is the case when one of the functions is called from several parts of the GCCFG, we consider the interference relationship between them to be dominated by the common ancestor which gives the largest interference.

5.3.4 Code Overlay Cost

Algorithm 1 computes interference relationships between all functions in the GCCFG. The loop starting at line 5 iterates through all edges of the *interference graph* defined as the fully connected graph with one node for each function, and each edge representing the interference between two functions.

In line 6 we check whether the LCA, s, of two nodes u, and v, is neither u nor v, in which case the interference cost between the nodes is the base cost of s. The base cost of the LCA represents the number of times one function would replace the other in

memory if they are mapped to the same region. In Figure 5.1b the interference between g and h is 10, the base cost of f. The algorithm has complexity $\mathrm{O}(nm)$.

Otherwise, in line 9 we know that either u or v is the common ancestor, and the interference between the two nodes is the sum of the base costs of the ancestor node and its first child on the path from ancestor to descendant. In Figure 5.1b, the interference cost between a and h is $\text{base}(a) + \text{base}(f) = 1 + 10$.

The first function on the path from the LCA to the descendant function determines the interference relationship because from the ancestors perspective any function call initiated within this child node is indistinguishable from the memory presence of the child itself. Considering functions a and f again, if they are in the same region, their interference is illustrated with the trace

$$< a, (f), a, (f), ... >$$

The two functions alternate in memory with each call to f and return to a. If we consider placing the entire GCCFG branch headed by f into a single region, the functions will swap one another out in memory as given by the trace

$$< a, (f, g, f, h, f, ..., f), a, (f, g, f, h, f, ..., f), ... >$$

Here the displacement between a and h is again once for every call to f and return to a. We say that f is *shielding* a from the interference effects of its descendants.

Definition 5 *A function,* u, shields *its parent function,* s, *from interference costs associated with all of its descendants,* T *such that when* s, *and* $T' \subseteq T$ *are mapped to the same region,* $shield(s, t) = u : \forall t \in T'$ *where* u *is the shielding function (graph node), and* $inter(s, T') \coloneqq inter(s, u)$.

The shielding property of interference relationships between function proves important in making decisions about assigning functions to regions and in predicting

the cost or quality of overlay regions and mappings as a whole which are discussed below in Section 5.4.1.

5.3.5 SDRM and spu-gcc Interference Definitions

In the SDRM algorithm presented in [124], interference costs are calculated similarly, but only base costs are considered. The interference used is the minimum of the two base costs for *callee-callee* relationships where neither node is the common ancestor, and the base cost of the descendant is used for *caller-callee* relationships. This difference is further addressed in Section 5.4.2. The cost is multiplied by the sum of the two function sizes since function size is understood to correlate with performance/energy overhead in the event of an overlay miss. The SDRM algorithm is analyzed in Section 5.5. Interference costs in the *spu-gcc* compiler are only considered between parent-child nodes, and are equivalent to the base cost of the child node [130]. The *spu-gcc* algorithm is briefly analyzed in Section 5.6.

5.4 COG Algorithm Framework

5.4.1 COG Overlay Cost Model

The *cost* of an overlay miss is defined as the actual associated DMA overhead in terms of time calculated from the size of the DMA. The cost associated with a given overlay mapping is the sum of the costs of its regions. We will first study the interferences between functions assigned to the same region in order to determine region costs. Functions assigned to separate regions do not interfere with one another, as they do not share the same address space in memory.

The calculation used to find interference between two functions has been described in Section 5.3.3. These interferences are the basis for calculating interference between functions and regions. The interference between a function, u, and an overlay

region, R, is given by:

$$\text{inter}_{fR}(u, R) = \max_{\forall v: u \neq LCA(u,v)} (inter(u, v)) + \sum_{\forall v: u = LCA(u,v)} inter(u, v) \tag{5.1}$$

In Equation 5.1 we consider two sources of interference in the GCCFG structure between a function, u, and all other functions assigned to the same region:

1. The first type of interference is associated with all functions in the region which are not descendants of the function under consideration, i.e. u is not the LCA. The first term on the right side of the equation gives the interference due to non-descendant functions in the region. Only the maximum interference is considered here since in general the number of times a function is overwritten between calls depends only on one common ancestor, s, due to its shielding u from all other non-descendant functions in the region. From Definition 5 we know that any other functions in the region are shielded from u. If there were another function in the region on the path from s to u, it would have an interference cost greater than or equal to $inter(s, u)$, and it would be shielding u from s.

2. The second term on the right side of Equation 5.3 gives the interference of u in R due to the descendants of u in R, i.e. u is the LCA. Again, by Definition 5, for each of the immediate children, t, of u, the interference between u and any or all of the descendants of t is exactly $inter(u, t)$. Consequently, the interference between u and all of its descendants is the sum of the interference costs between u and any of its immediate children with descendants in the region.

The expected *cost* of an overlay region is derived from the interference or number of overlay misses between its functions, and the *DMA overhead* associated with each overlay miss. DMA overhead is the actual *cost* of an overlay miss in terms of time spent retrieving the missing code segment. The time required to execute the DMA,

which will be referred to as simply *cost*, is a function of the size of the missing function, u:

$$\text{cost}(u) = \text{ time needed to DMA } size(u) \text{ Bytes} \tag{5.2}$$

The actual DMA overhead function giving cost in seconds as a function of DMA size is empirically determined by measuring DMA performance on the target architecture. This function as determined for the CBE is described as part of the simulation discussion in section 5.7.1. Expected overhead cost of a function assigned to a region is given by:

$$\text{cost}_{fR}(u, R) = \text{cost}(u) \cdot \text{inter}_{fR}(u, R) \tag{5.3}$$

The total expected overlay overhead cost associated with one region is the sum of the costs of all of its assigned functions given by:

$$\text{cost}_R(R) = \sum_{\forall u : u \in R} \text{cost}_{fR}(u, R) \tag{5.4}$$

Algorithm 2 is used to calculate function in region cost. Finally, the total expected overlay overhead associated with an overlay mapping, *OVL*, is the sum of the costs of all of its assigned regions:

$$\text{cost}_{mapping}(OVL) = \sum_{\forall R : R \in OVL} \text{cost}_R(R) \tag{5.5}$$

The value obtained from $\text{cost}_{\text{mapping}}(OVL)$ is used to classify overlay mappings according to expected performance, enabling classification of solutions in terms of expected performance.

5.4.2 COG Algorithm

The Code Overlay Generator (COG) Algorithm is designed to produce overlays which result in the smallest possible number of misses. Once interference costs have been

Algorithm 2 Function cost in region algorithm.

$LCA(i, j)$ {initialized common ancestors}
R {overlay region}
u {function to be evaluated in region R}
$F = R \backslash \{u\}$ {all functions in R other than u}
$F_R = \forall v$ s.t. $v \in R : v \neq u$ { all functions in R other than u}
F_c {immediate children of f}
$inter_d = 0$ {descendants' interference }
$inter_{nd} = 0$ {non-descendants' interference }
$inter\,(i, j)$ {initialized interference relationships}
for $\forall v : v \in F$ **do**
  if $u = LCA(u, v)$ **then**
    if $shield(u, v) \in F_c$ **then**
      $inter_d = inter_d + inter(u, v)$
      $F_c = F_c \backslash \{shield(u, v)\}$ {an immediate child cost is added at most once}
    end if
  else
    if $inter(u, v) > inter_{nd}$ **then**
      $inter_{nd} = inter(u, v)$
    end if
  end if
end for
return $cost = \mathrm{c}(u) \cdot (inter_d + inter_{nd})$

Algorithm 3 Code Overlay Generator Algorithm.

OVL {set of r empty overlay regions}
R {an overlay region in OVL}
R_{min} {overlay region with minimum cost}
$inter\,(i, j)$ {initialized interference costs}
$\vec{F}$ {functions sorted from largest to smallest total interference}
for $\forall f : f \in \vec{F}$ from largest to smallest total interference **do**
  $R_{min} = R$ s.t. $\mathrm{cost}_{fR}(f, R) : \forall R \in OVL$ is minimized
  add f to R_{min}
end for
return OVL

calculated and we have a method for empirically analyzing the quality of function to region mappings, we can construct an algorithm to generate high performance overlays.

The algorithm works on a fixed number, r, of regions and generates overlay mappings designed to minimize misses. It does not consider function sizes. As a result, we will not know the size of the generated overlay mapping until the algorithm is complete. To account for mapping size and find the best overlay mapping for a given

memory size, we first calculate the lower bound on the number of regions which will fit in memory. This bound is determined by selecting functions from a list sorted in descending order by size. Then starting with the largest function, we place them end to end until the sum of the sizes of *n+1* selected functions exceeds the available memory, *M*. We then begin generating overlay mappings starting at $r = n$, increment r for each iteration and solve again.

The solution with the lowest overlay cost is retained after each iteration. Since some mappings with more regions may have smaller sizes, we can find better solutions by continuing to search after solutions start to exceed the available memory until a mapping reaches a factor such as *2M* or the number of regions is equal to the number of functions. COG is presented as Algorithm 3.

For each function, *f*, to be assigned from the list of functions sorted by *total interference*, COG calculates $cost_{fR}(f, R)$ for each region in line 7. The current function is assigned to the region in which this cost is minimized. In the worst case the algorithm is run n times for overlay mappings with 1 to n regions, n functions are allocated each run and at most n comparisons are made for each function to find the best region for a computational complexity of $\mathrm{O}\left(n^3\right)$.

5.4.3 COG-Expansion

After running COG, we may have unused bytes remaining in the available SPM memory. In the event COG returns a solution with unused memory space at least as large as the smallest function, we can improve the overlay's performance by creating new regions and moving functions into them. The simple extension to the COG algorithm given as Algorithm 5.4.2, is called COG Expansion (COG-E). In line 8, the algorithm removes functions to new regions beginning with the function having the greatest total cost as long as they fit into remaining memory and the source region contains more than

Algorithm 4 Code Overlay Generator-Expansion Algorithm.

1: $remaining\_mem$ = unused instruction memory
2: OVL {overlay solution from COG}
3: R {an overlay region in OVL}
4: R_{min} {overlay region in with minimum cost}
5: $\vec{F}$ {functions sorted from largest to smallest total interference}
6: **while** $memory\_size - \text{size}(OVL) > min\_function\_size$ **do**
7: **for** $\forall f : f \in \vec{F}$ from largest to smallest total interference **do**
8: **if** size(f) < $remaining\_mem$ **then**
9: remove f from it's current region
10: create a new region and add f
11: $remaining\_mem = total\_mem - \text{size}(OVL)$
12: **end if**
13: **end for**
14: **for** $\forall f : f \in \vec{F}$ from largest to smallest total interference **do**
15: $R_{min} = R$ s.t. $cost_{fR}(f, R) : \forall R \in OVL$ is minimized
16: add f to R_{min}
17: **end for**
18: **end while**

one function. If a function does not fit into remaining memory we skip it and continue to the next function in descending order of total interference.

In order to further improve the solution, in line 15 we check for opportunities to reduce overlay cost by testing the functions again from greatest to least total interference to see if their cost in their currently assigned region is less than their cost in a new region as long as the move does not increase the size of the new region. This process can cause the overlay size to shrink since the largest functions from some regions may have been moved to larger regions. We can take advantage of this reduction in overlay size by running the algorithm again until remaining memory is too small to hold the smallest function. We can avoid increasing the computational complexity of the algorithm by limiting the outer loop to some small number of iterations.

Increasing the number of regions in this way is guaranteed to reduce overlay cost since removing a function from a region with multiple functions must reduce the region's interference cost if multiple functions are called in that region, and adding a function to an empty region incurs no cost. The complexity of the expansion algorithm

Algorithm 5 Code Overlay Generator-Compression Algorithm.

```
1: OVL {overlay solution from COG}
2: R {an overlay region in OVL}
3: R_largest {largest region in terms of memory size}
4: while size(OVL) > mem_available
   and num_regions(OVL) > 1| do
5:    for f = largest_function(R)∀R : R ∈ OVL\R_largest do
6:       R_min = R s.t. cost_fR(f, R) : ∀R ∈ OVL is minimized
7:       add f to R_min
8:    end for
9: end while
```

is determined by the second for loop (line 14). The complexity of the *for* loop and consequently the COG-E algorithm is $\mathrm{O}\left(n^2\right)$.

5.4.4 COG-Compression

As a second extension to the COG algorithm, consider COG Compression (COG-C). When COG returns an overlay mapping which is too large to fit in the SPM, we attempt to compress the size of the overlay until it will fit. This is done by evaluating regions other than the largest one and systematically moving their largest assigned function into a larger region. Each such function move can reduce the size of the overlay if all remaining functions are smaller than the largest function. Once the overlay is small enough to fit into available memory, we stop and return to the COG Algorithm where the compressed overlay is retained if it has a lower cost than the best solution so far. These steps are illustrated as Algorithm 5.4.3. The complexity of the algorithm is again limited by restricting the outer loop to a small constant size. The loop at line 5 runs for n iterations with at most n comparisons per iteration giving the COG-C algorithm a complexity of $\mathrm{O}(n^2)$.

5.4.5 COG, Unified Algorithm

In practice, the three COG algorithms presented can be run simultaneously to select the solution with best predicted overlay performance. Since COG is the first step of each

algorithm for each number of regions, the first step in the unified algorithm is to run the COG algorithm. When the COG solution is found to be smaller than available memory, we will run COG-E to find an improved solution which better utilizes memory. In the event that the COG solution is found to be larger than available memory, we run COG-C in an effort to find a mapping with reduced size and improved performance over the last valid COG solution with fewer regions. As is the case in the previously described algorithms, we retain the best solution at each step, and return the overall best solution after exhausting the region search space.

The computational complexity of the combined algorithm is again $O\left(n^3\right)$, since we find a solution for each number of regions in the search space only once, and execute at most one COG extension algorithm per iteration. Although this method will return the solution with the best predicted performance from among COG and its two extensions, it is not guaranteed to return the mapping which gives the best real world performance, as compile time evaluation of the solutions is limited by the accuracy of the performance model.

5.4.6 Clustering Functions into Segments

A function clustering algorithm was implemented to improve performance at low memory sizes compared with *spu-gcc*. As discussed in Section 5.6, the *spu-gcc* algorithm outperforms COG and SDRM when memory size restricts overlays to one region. By clustering functions into larger segments, performance improvements are possible, particularly for extremely memory constrained solutions.

The implemented clustering algorithm works on edges of the interference graph much like the SDRM implementation. The main difference is that edges are used to combine functions into one segment rather than separate them into regions. The interference graph is generated for the assigned functions of each region in the mapping, and interference edges are sorted from greatest to smallest cost. Starting with the most

expensive edge, we remove it from the sorted list and add its associated functions back to the region by combining them into a single segment if their combined size does not exceed the original region size or an empirically determined threshold value or we place them into two separate segments if the combined size is too large. If one of the functions is already assigned to a segment, then the other function is added to that segment using the same criteria such that the second function is either added to the existing segment, or a new segment is created. After all edges have been consumed, the new segmented region is returned and integrated into the overlay mapping.

5.5 SDRM Analysis

5.5.1 SDRM Cost Model

In the SDRM algorithm [124], interference costs are calculated similarly to the method presented here for COG, but only base costs are considered. The incorporation of return interference is an important new contribution in the presented model. In SDRM the interference used is the minimum of the two base costs for *callee-callee* relationships where neither node is the common ancestor, and the base cost of the descendant is used for *caller-callee* relationships. This value is multiplied by the sum of the two function sizes since function size is understood to correlate with performance/energy overhead in the event of an overlay miss. In addition, the SDRM model calculates region cost based on the sum of interferences between all functions assigned to the region without accounting for the effects of shielding on function interference. In order to make valid comparisons between the COG and SDRM models, the DMA cost model as described in Equation 5.2 is used when calculating SDRM region costs instead of using function sizes directly.

5.5.2 Deadlock

SDRM uses the interference costs described in Section 5.5.1 to construct an overlay mapping. The interference costs are annotated on edges between functions in an in-

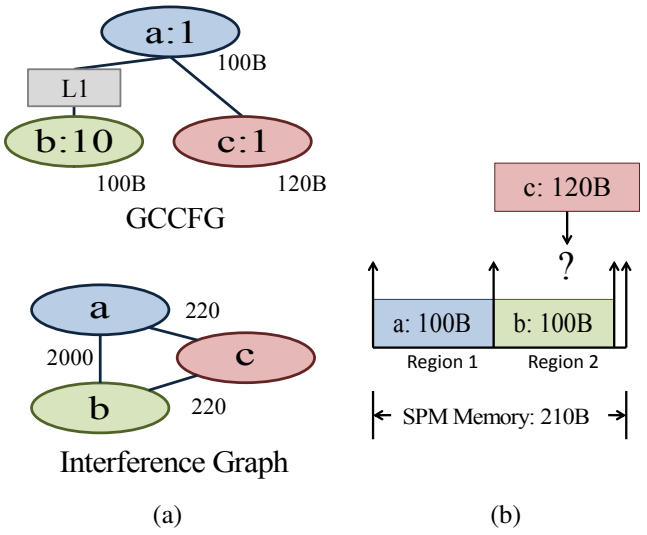


Figure 5.3: (a) SDRM deadlock example GCCFG. (b) SDRM deadlock example, functions *a* and *b* have been assigned first because they share the most expensive edge in the interference graph, function c cannot be placed in memory because its size is larger than the largest region plus remaining memory in the SPM.

terference graph. These edges are sorted from most expensive to least expensive. The most expensive edge is selected and each of its associated functions are added to newly created overlay regions, then the next edge is selected. If there is not enough unused memory to create a new region, the function is added to one of the existing regions such that the cost of adding the function is minimized. In the worst case, the SDRM algorithm makes n comparisons to select a region by traversing n^2 edges, for a computational complexity of $O(n^3)$.

The scheme can run into trouble as illustrated in the example GCCFG in Figure 5.3a. The interference graph shows that the most expensive edge occurs between functions *a* and *b*. Figure 5.3b illustrates the behavior of the SDRM algorithm for the given SPM size. The algorithm first selects the edge (a, b) adding functions *a* and *b* to new regions in the SPM. When *c* is selected, it is found to be larger than the size of either region plus the remaining SPM memory. The consequence is that the algorithm hangs without producing a solution.

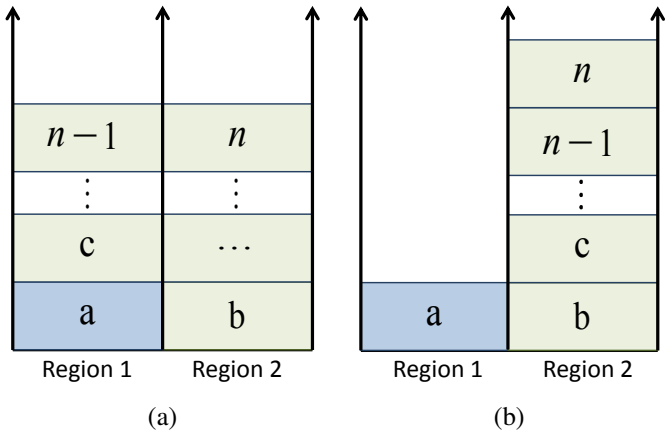


Figure 5.4: (a) Overlay generated by SDRM. (b) Overlay generated by COG.

5.5.3 Performance Case Study

Figure 5.2 illustrates a *flat* call graph structure commonly found in computer programs. After inspecting the call graphs for each of the benchmarks, it turns out that on average 67% of the functions are direct members of such structures even when counting only instances of the structure with at least four nodes. Given the data structure in the figure and assuming all functions are the same size, the interference graph generated by SDRM has the same weight on every edge. The resulting overlay is given in Figure 5.4a. COG generates the alternative mapping shown in Figure 5.4b. The difference derives from the fact that COG considers the return cost between function a and its children where the SDRM algorithm does not. The memory resident function traces for SDRM are $R_1 :< a, c, a, ..., a, n-1, a >$ and $R_2 :< b, d, ..., n >$. For COG the traces are $R_1 :< a >$ and $R_2 :< b, c, ..., n >$. It is clear that the return costs of a inflate the number of misses in the SDRM mapping. The cost of each mapping can be given in terms of misses for the SDRM and COG solutions with n functions and r regions as follows:

$$misses(SDRM) = n + \frac{n}{r} - 1 \tag{5.6}$$

$$misses(SDRM) = n + \frac{n}{2} - 1 = \frac{3n}{2} - 1 \approx \frac{3}{2}n : r = 2 \tag{5.7}$$

Table 5.2: Simulation results vs. SDRM.

	ldec.	ispl.	gsm
COG-E	n/a	39%	68%
COG-C	n/a	38%	24%
COG	n/a	38%	-18%
COG v. gcc	87%	70%	76%

$$\lim_{r \to n} \left(misses(SDRM) = n + \frac{n}{r} - 1 \right) = n \tag{5.8}$$

The COG mapping has a cost given by:

$$misses(COG) = 1 + r \cdot \frac{(n-1)}{r} = n \tag{5.9}$$

Equations 5.6 and 5.9 describe the number of overlay misses experienced by the SDRM and COG solutions respectively. Equations 5.7 and 5.8 indicate that in the case of the structure in Figure 5.2, the SDRM solution is at best equal to the COG solution described in Equation 5.9 as the number of regions approaches the number of functions (i.e. as available memory increases), and at worst 50% more expensive than COG in the case of a two region solution given in Equation 5.7. Increasing the number of regions improves the SDRM solution, but COG is optimal even when there are only two regions, as n is the smallest possible number of misses.

5.6 *spu-gcc* Analysis

The *spu-gcc* automatic overlay algorithm is described in [130]. It has a lower complexity, working on at most n^2 edges, of $\mathrm{O}\left(n^2\right)$. The algorithm clusters highly interfering neighboring functions into the same segment as long as the segment fits in the memory region. Interference costs in the *spu-gcc* compiler are considered only between parent-child nodes [130], and are equivalent to the base cost of the child node in the scheme presented here.

A key distinction between this algorithm and COG/SDRM is that it relies exclusively on code segments rather than regions to group functions into an overlay mapping.

The compiler algorithm always returns a mapping with exactly one region. Each segment may contain one or more functions, unlike SDRM and COG, both of which ignore the potential benefit of placing multiple functions into one segment.

Since COG and SDRM use a separate segment for each function, performance for any solution generated with a single region gives worst-case performance because every function call results in a call and return miss. For this reason the *spu-gcc* solution is expected to outperform COG and SDRM when the solution consists of just one region. However, large segments quickly become a handicap in the spu-gcc algorithm as the size of available memory is increased. Performance is hampered with larger segment sizes because any misses during program execution result in memory access overhead which is a function of segment size. As the region size grows further, performance begins to improve again, approaching the optimal with every function always in the SPM mapped to one segment containing the entire program. These effects will be highlighted in the discussion of experimental results.

In the experiments, *spu-gcc* overlays have been generated for the set of benchmarks using the *auto-overlay* option described in the IBM Cell Programmer's guide [77]. The benchmark source code was modified by adding a large data buffer which consumes sufficient LS space to force the compiler to generate multiple code segments in order to fit the instructions in memory. Larger buffer sizes result in smaller overlay mapping sizes. By examining the compiler output for a range of buffer sizes, it is possible to determine the configurations needed to generate *spu-gcc* overlay solutions that sweep the program size. The size of the automatically generated overlay mappings is found by parsing the compiler generated spu memory map file to find the largest segment in the compiler's one region solution.

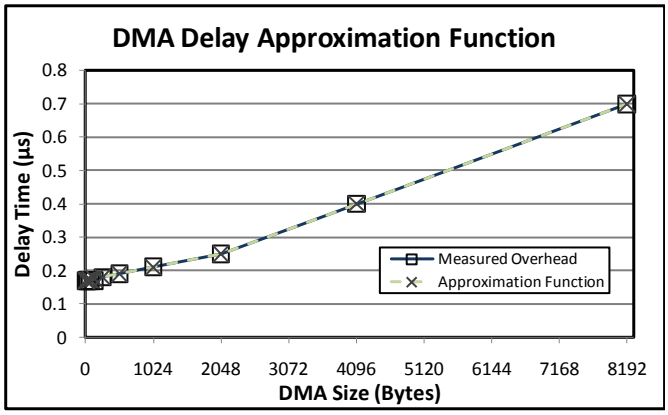


Figure 5.5: Cell Broadband Engine measured overlay overhead costs and approximation function in terms of delay as a function of DMA size.

Table 5.3: Simulation benchmarks.

benchmark	functions	max overlay(B)	min overlay(B)
ldecod	471	430,532	20,060
ispell	110	88,312	6,104
gsm	72	34,768	6,444
sha	8	1,702	717
rijndael	7	11,428	4,468
dijkstra	6	1,623	656
patricia	5	2,956	1,497

Table 5.4: CBE benchmarks.

benchmark	functions	max overlay(B)	min overlay(B)
cjpeg	210	129,320	3,736
ffmpeg	106	100,012	7,072
gsm-untoast	41	12,504	2,172

5.7 Experimental Results

5.7.1 Experimental Setup

Benchmarks and Memory Setup The benchmarks used for experimentation are presented in Tables 5.3 and 5.4. A distinct set of benchmarks is used for simulation based analysis (Table 5.3) and execution on the CBE (Table 5.4). For each benchmark memory sizes needed to test performance across a meaningful range of available SPM instruction memories are selected. Benchmarks with smaller code sizes have been swept from the smallest possible overlay mapping size, defined by the size of the largest function in the program, up to the total size of the program, or the sum of sizes of all

functions. This sweep is done in 15 steps for small benchmarks. For larger benchmarks the sweep is done in 10 steps and ranges from the smallest possible overlay mapping to one half of the full program size. These ranges can be found in Tables 5.3 and 5.4 along with the number of functions present in each benchmark.

Benchmarks with less than 10 functions are considered *small*. When selecting benchmarks for testing, every effort has been made to utilize available benchmarks with a large number of functions. Previous work, including SDRM [124], have only given results for benchmarks with very few functions. Results achieved using larger benchmarks are more interesting for two reasons. Benchmarks with 10 or fewer functions can actually be solved optimally in a reasonable amount of time using brute force and memoization methods, defeating the original motivation for a heuristic approach. Also, when analyzing results for small benchmarks, performance is often erratic and highly dependent on quantum effects of individual function to region assignments.

Simulation Overlay mapping performance has been evaluated for the benchmarks in Table 5.3 by simulating the memory access overhead of the CBE between main memory and the SPU Local Store due to code overlay misses. Inputs to the simulator are execution trace data, function sizes, and the overlay mapping to be evaluated.

Benchmark execution traces were generated by instrumenting the source code with print statements. The trace consists of function names in the order in which they must to be present in memory during program execution. Each function in the program has been modified to emit its name into the output trace upon entry and immediately following any function calls present in the function body. The resulting trace accounts for function calls and returns as described in Section 5.3.2, enabling accurate accounting of interference between functions.

The simulator maintains a table representing the state of the overlay map in memory as the trace is consumed. Miss overhead is recorded whenever a function

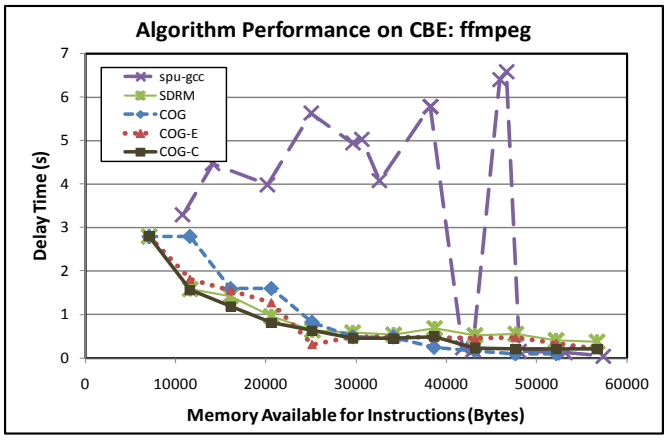
appears in the trace and does not reside in the memory table, and the overlay map state table is updated to indicate the appropriate code segment is present. It is important to note that rather than an approximation, the number of misses recorded for each function in the simulator is *identical* to the number which would have been observed during actual execution. DMA overhead cost of a benchmark function, f, in the simulation is calculated as a function of segment size according to real DMA overhead measurements taken on the testbench CBE which can be modeled to within an average error of 0.4% using the following equation:

$$DMA\_cost(f) = \begin{cases} \text{3.9E-5} \cdot size(f) + 0.17\mu s & :\text{size}(f) <= \text{2kB} \\ \text{7.3E-5} \cdot size(f) + 0.1\mu s & :\text{size}(f) > \text{2kB} \end{cases}$$

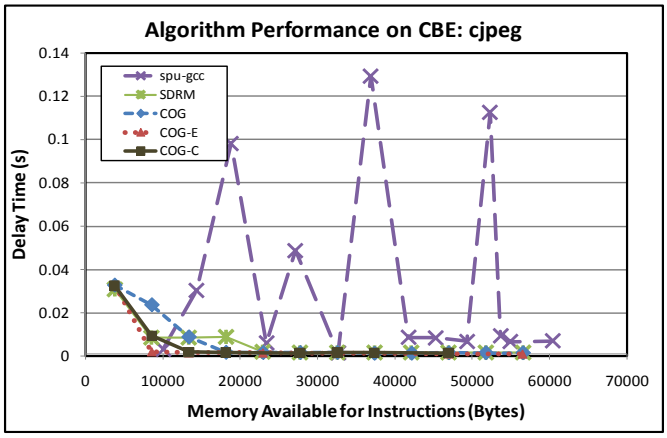
The approximated DMA cost curve is plotted against the measured curve in Figure 5.5. DMA overhead costs are accumulated as the simulator executes. The total overhead returned at the end of the input trace is the data point plotted in the performance evaluation charts seen in Figure 5.12 on page 116

IBM Cell Implementation The benchmarks in Table 5.4 have been modified to run using user defined overlay mappings with function level resolution on a single CBE SPU. The SPU compiler, *spu-gcc*, is capable of taking a user defined linker script as an input in order to map code into LS memory according to the user's overlay scheme [77]. When specifying the mapping of functions to segments and segments to regions, code objects must be specified in the script as individual object files. Consequently, in order to implement and evaluate arbitrary code overlay mappings at function level resolution, each function must be placed in a separate source file.

In order to obtain performance results on the CBE, linker scripts describing overlay solutions for each algorithm at each tested memory size are generated. Next each benchmark is compiled once for each linker script, resulting in a separate executable for each data point. The benchmarks are also instrumented to print total ex-



(a) Performance of each algorithm on *ffmpeg* (H.264 decoder) running on the CBE.



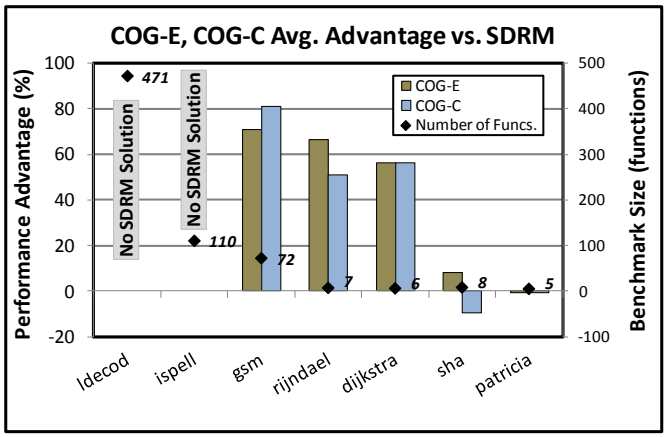
(b) Performance of each algorithm on *cjpeg* (jpeg encoder) running on the CBE.

Figure 5.6: Performance results.

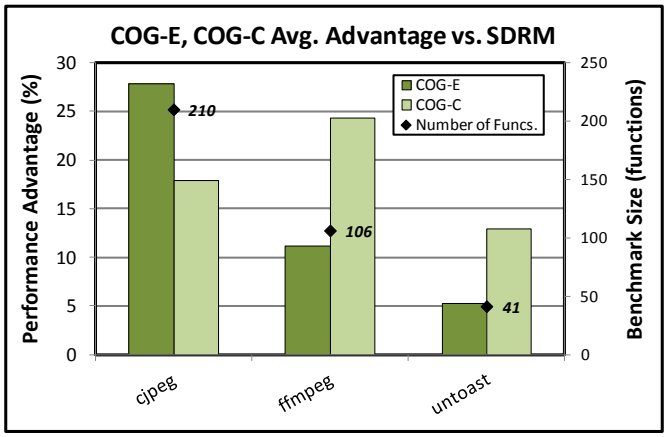
ecution time information after each run. The executables are then run multiple times and the resulting timing data is averaged to obtain the final data point as presented in Figures 5.6a and 5.6b.

5.7.2 Results

In Figures 5.6a and 5.6b, notice that the COG extension algorithms, COG-E and COG-C, both produce better performing overlays than the unextended COG algorithm and SDRM, particularly when memory is severely restricted. The limited performance of the unextended algorithm is due to the fact that it can produce solutions which do



(a) Performance advantage of COG over SDRM for each benchmark in simulation. SDRM does not produce a solution for *ldecod* and *ispell*.

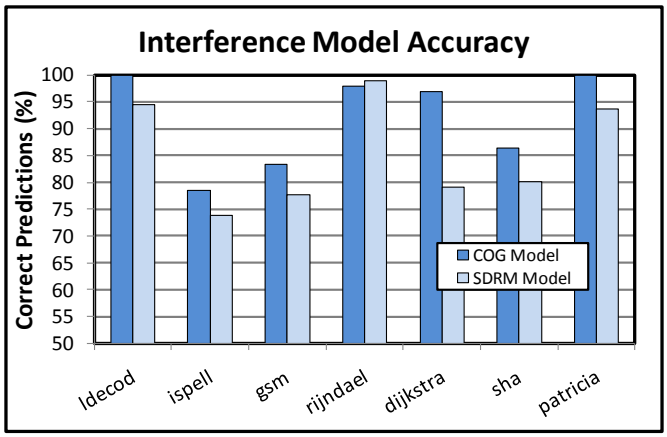


(b) Performance advantage of COG over SDRM for each benchmark configured to run on the CBE.

Figure 5.7: Overall performance and prediction results.

not fully utilize available memory as corrected by the COG-E algorithm, and the extension algorithms tend to produce solutions with more regions. The relationship between an overlay mapping's performance and number of regions is further evaluated in Section 5.7.2.

The overall performance results comparing SDRM and the COG extension algorithms are given in Figures 5.7a and 5.7b. The size of each benchmark is also indicated on the graph in terms of number of functions. There are several instances where the SDRM algorithm fails to generate a solution due to deadlock during overlay map-



(a) Accuracy of overlay cost model in predicting which mapping will exhibit the best performance.

Figure 5.8: Overall performance and prediction results.

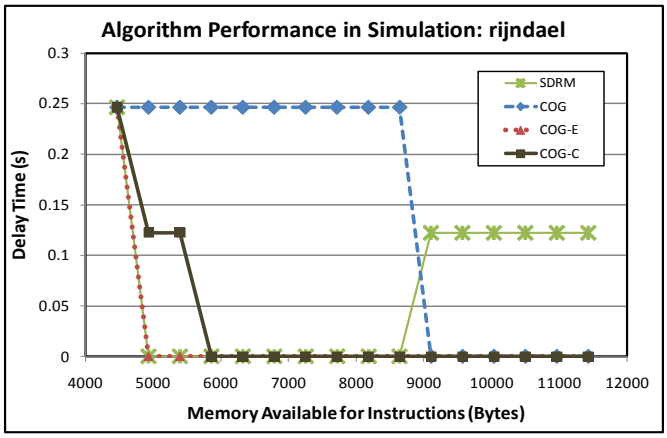


Figure 5.9: Simulation performance results for *rijndael*.

ping generation. In particular, for the problem space on *ldecod* and *ispell* in Figure 5.7a, the SDRM algorithm does not return a solution for any data point. For other benchmarks, as seen in the SDRM trace in Figure 5.12a, SDRM generates solutions for some memory sizes, typically on the larger end of the sweep. On average, the COG extension algorithms out perform SDRM by 38% in the simulated benchmarks, and by 16% in benchmarks executed on the CBE in terms of overlay overhead.

As mentioned above, performance analysis is more meaningful when the number of functions in the program is larger. For the smallest benchmarks, *rijndael*, *dijkstra*, *sha*, and *patricia*, quantum effects due to the small number of functions amplifies

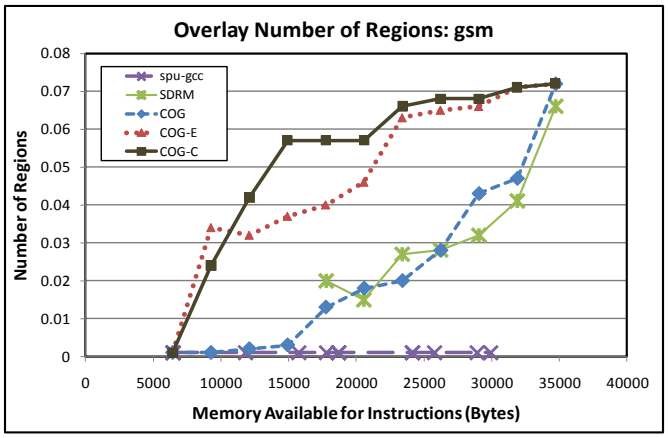


Figure 5.10: Number of regions in the generated mapping solution for each algorithm for various available memory sizes. SDRM fails to generate solutions for smaller memory sizes.

the impact of individual mapping decisions. This effect is clear in the overlay performance results as illustrated in Figure 5.9. The SDRM solution in the figure has been impacted in particular by a compile time decision which has a substantial negative impact on the performance of its solutions above 6kB of instruction memory, where we would normally expect performance to improve.

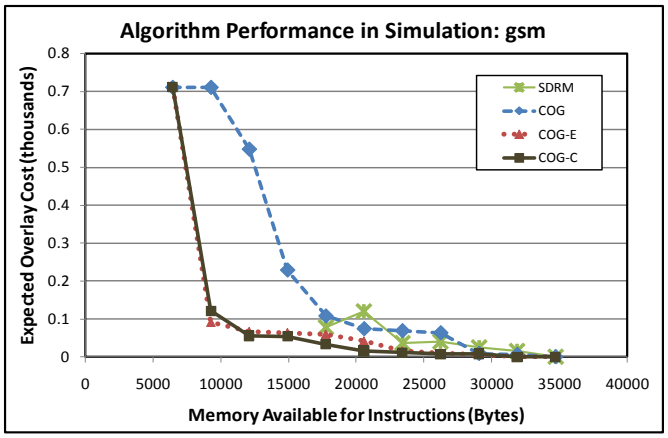
Number of Regions and Performance The solution generated for the smallest possible overlay mapping is always limited to one region, where the solution for the largest possible overlay mapping normally contains a region for each function. Figure 5.10 illustrates the number of regions in solutions generated by each algorithm for *gsm*. The three COG algorithms generate the same solution when available memory is restricted to the size of the largest function as indicated by the convergence of the three traces on the left. The mapping algorithms again produce identical solutions once memory size matches the total program size at the upper right in the graph since each function can be assigned to its own region. The SDRM solution is shown to have fewer regions at this point in the graph because in the SDRM model, functions whose common ancestor is *main* are assigned no interference cost in the interference graph and as

a result may end up sharing a region although space is available in memory to create new regions and separate them.

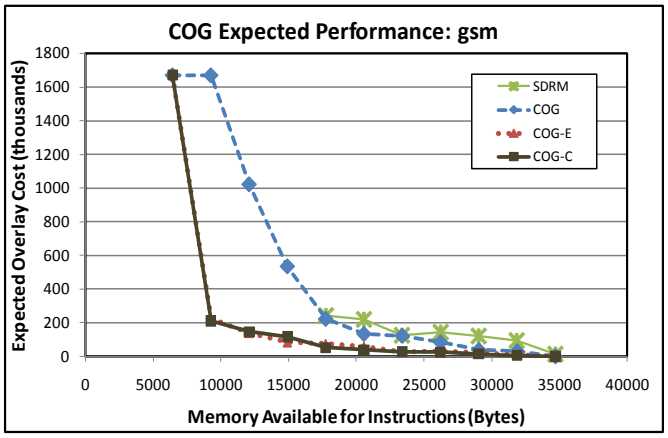
Mapping solutions with more regions are expected to give lower overhead costs, as only functions mapped to the same region can interfere with one another. From Figure 5.10 it can be observed that solutions with larger region counts generally do perform better as evident in the corresponding performance chart in Figure 5.12a. In particular, the *spu-gcc* solution which is limited to one region, experiences a substantial performance disadvantage as available memory enables solutions with more regions from COG and SDRM.

Performance Model Accuracy This assessment of performance model accuracy is based on the ability of each model to predict, given two overlay mappings, which mapping will actually give better performance. Performance predictions from the COG and SDRM models are plotted for each simulation benchmark and compared against the simulated performance results.

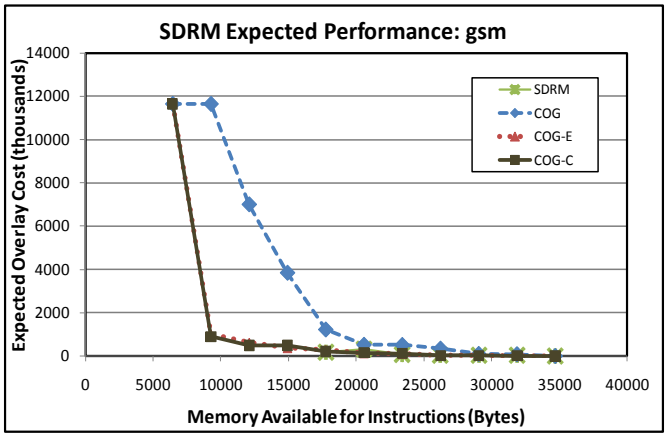
For the *gsm* benchmark predictions made by the COG model in Figure 5.11b and predictions using the SDRM model in Figure 5.11c are presented. The actual results are shown for comparison in Figure 5.11a. By inspection it can be found that, as expected, the SDRM cost model tends to give overly optimistic predictions for solutions generated by the SDRM algorithm. This is evident when comparing the SDRM trace in Figure 5.11b with the SDRM trace in Figure 5.11c. The ability of each model to predict performance is empirically tested by counting the number of times they have correctly predicted which overlay solution actually performed better independently for every pair of algorithms in COG, COG-E, COG-C, and SDRM. The results of these comparisons are presented in Figure 5.8. On average, across all benchmarks, COG correctly predicted overlay performance 6.5% more often than the SDRM performance model.



(a) Simulated performance results for *gsm*.

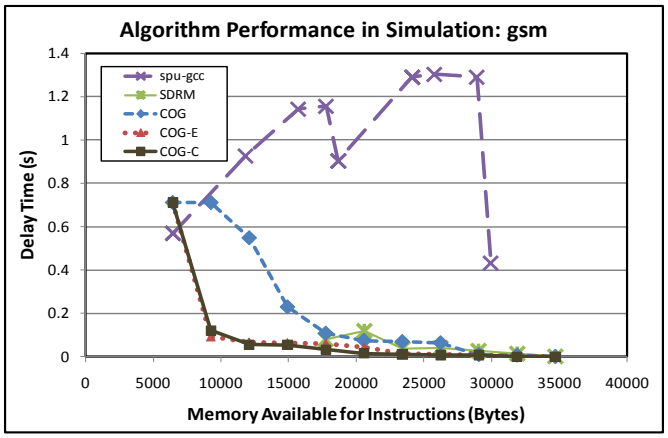


(b) COG cost model performance prediction for *gsm*.

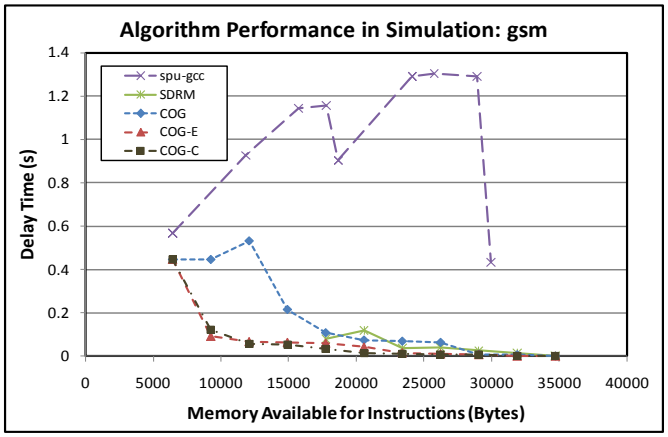


(c) SDRM cost model performance prediction for *gsm*.

Figure 5.11: Performance model comparison for *gsm*: (a) shows the simulated performance for the COG and SDRM solutions, (b) gives the performance predictions for each overlay solution using the COG model, and (c) shows the performance predictions using the SDRM model.



(a) Simulation performance results for *gsm* with one function per segment. SDRM fails to generate solutions for smaller memory sizes.



(b) Simulation performance results for *gsm* with functions clustering performed to create larger segments. SDRM fails to generate solutions for smaller memory sizes.

Figure 5.12: Simulation performance results: *gsm*.

Segment Clustering Performance Performance results for the segment clustering algorithm are shown for *gsm* in Figure 5.12a without segment clustering and in Figure 5.12b with segment clustering. The segment clustering algorithm gives us a solution which outperforms the *spu-gcc* algorithm by 21% at smaller memory sizes. The algorithm enables limited improvement for larger memory sizes for several reasons. As discussed in the *spu-gcc* analysis in Section 5.6, larger segment sizes tend to handicap performance due to increased DMA overhead. For that reason, performance improvements are not apparent for larger memory sizes, although the alternative overlay

generators still significantly outperform *spu-gcc* in those configurations thanks to regionalization of the code.

5.8 Conclusion

In this work, a code overlay generator designed to map instructions onto limited scratch-pad memories for improving performance in embedded systems has been presented. In addition, an overlay mapping cost model for identifying good solutions at compile time without the benefit of profiling information is described. The algorithm performs better than the previously published heuristic, while eliminating the deadlocking problem experienced in the previous work. The algorithm is also demonstrated to perform substantially better than the scheme provided with the IBM Cell Broadband Engine compiler, *spu-gcc*.

Algorithms for gathering functions into segments for improved performance have also been presented. Substantial overlay performance improvements are extracted in the presence of extremely limited memory constraints by including a step to cluster functions into segments. Gains are more limited with function clustering when more memory is available, and improvement in this area, in addition to exploring possibilities for reducing the computational complexity of the heuristic and code prefetching are subjects of future work.

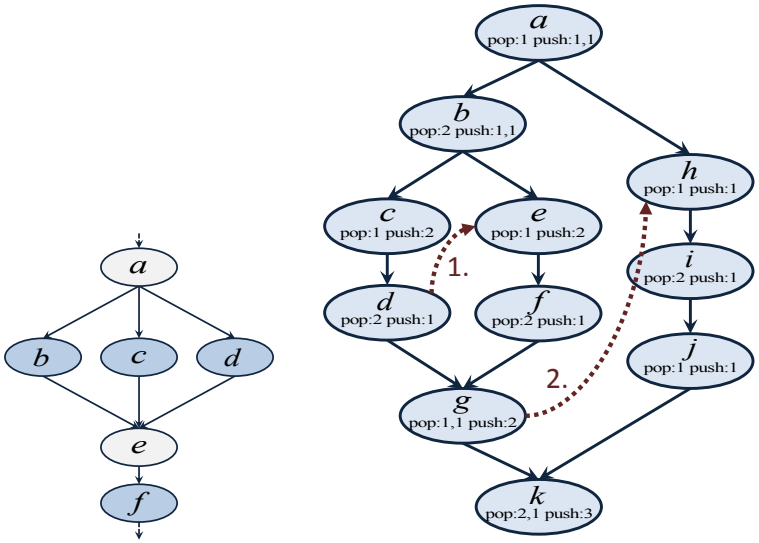
Chapter 6

LIGHTWEIGHT RUN-TIME SCHEDULING FOR MULTITASKING MULTICORE STREAM APPLICATIONS

The recent rise of multicore architectures has led to dramatic changes in the traditional view of application development. Most applications have historically been developed for monolithic cores with little consideration for the need to identify available parallelism. Smart compilers can extract and express such parallelism opportunities to a certain degree. However, in order to fully extract potential temporal and spatial parallelism from an application, a new programming model is required. After more than a decade of concerted effort aimed at compilers capable of automatically identifying and exploiting parallelism in a given application, a common impression is that such a tool would truly need to be "impossibly smart" as Gordon et al. suggested in 2002 [62].

Stream programming formats have garnered significant attention in recent years for specifying applications in multimedia, signal processing, networking and graphics domains. Streaming languages enforce exposure of spatial and temporal parallelism in a program, thereby enabling the compiler to effectively analyze program constructs to produce efficient mappings for multicore architectures. Still, numerous challenges must be overcome. The wide variation in the structure and instruction set architectures of available multicore processors implies that a distinct optimizing compiler is necessary for each target system. Even if several architectures are based around the same instruction set, in general a separate and specialized set of optimizations still must be developed for each configuration.

Given the fast moving world of embedded processor design where commercial processor development cycles can be as short as two years [137], providing optimized compilers for state of the art architectures becomes an enormous challenge. Another challenge stems from the lack of flexibility in optimized mappings produced with an



(a) Stream graph representation.

(b) PASS Generation: stream graph node ordering.

Figure 6.1: Stream graphs representation and PASS actor ordering.

off-line or compile-time approach. The generated design cannot easily account for run-time variations in resource availability which are important in multi-tasking environments.

In essence, a re-targetable stream program optimization scheme is extremely desirable due to the importance and proliferation of stream processing architectures such as the IBM Cell Broadband Engine (CBE) [127] and the Imagine Processor [90] [91] as well as graphics processing units (GPUs) from NVIDIA [64] and AMD/ATI [1].

This work addresses the problem of generating flexible streaming application implementations that can dynamically adopt to variable resource requirements at run-time. The solution presented here involves two steps. In the first step, static compile time analysis is used to generate a resource agnostic canonical schedule. The canonical schedule then serves as a basis for dynamic optimizations computed by the run-time scheduler with the objective of maximizing throughput under variable resource constraints.

6.1 The Stream Programming Model

The stream model is a good fit for data-centric applications such as digital signal processing (DSP) including video and audio coding and decoding as well as various communication standards and protocols, much of which today are common in *billions* of handheld devices such as the billion cell phones sold worldwide in 2007 alone [110] [150]. As data-centric and embedded software applications continue to grow in importance, parallelization for improved energy efficiency and performance is critical.

Typical streaming applications operate on a stream of similar data elements such as image pixels in video and image processing or digitized antenna samples in software radios. The data stream is transformed by one or more compute intensive work kernels, referred to here as *actors*. Actors either consume data from a stream or produce data to form a stream, communicating with one another through explicit channels, normally with the aid of first in first out (FIFO) buffers. The arrangement of actors and the communication channels between them describe a stream graph. Streaming applications tend to exhibit high levels of data parallelism and producer-consumer data locality. An extremely detailed historical perspective on stream computing dating back to the 1960's is presented by Stephens [142].

In the stream programming model, applications are represented as a dataflow. An actor may implement a work kernel which is executed repeatedly on a stream of data elements, consuming the stream while simultaneously generating a transformed output data stream. Actors and buffers are mapped to cores and memories respectively in the processor. Atomic data elements either produced or consumed by a given actor are referred to as *tokens*. The consumption of tokens from the FIFO by an actor is referred to as *popping*, writing to the FIFO is called *pushing*, and reading tokens from the FIFO without popping them from the buffer is referred to as *peeking*.

The stream program may be represented as a stream graph. The stream graph in Figure 6.1a contains 6 actors, of which *a* and *e* serve a specialized *split-join* function enabling multiplexing, demultiplexing, or duplication of the data stream. The communication memory buffers are implicit in the directed edges of the graph. When the execution of an individual actor is dependent solely on tokens from other actors, asynchronous execution in a distributed memory environment is possible. A stream program exposes independent code segments within the program as independent actors. Asynchronously executable actors denote *functional parallelism* present in the application. Additionally, when actors do not carry an internal state, i.e. execution of instance n of a *stateless* actor a, denoted a_n, is not dependent on execution $a_m : m \neq n$, multiple copies of the same actor can run simultaneously across various cores consuming a multiplexed input stream, taking advantage of *data parallelism*. Actors carrying internal states are referred to as *stateful*. Individual executions of stateful actors are not independent from one another, eliminating the possibility of executing multiple instances simultaneously.

Under the stream programming model, the onus of identifying task and data parallelization opportunities in a given algorithm rests on the programmer rather than the compiler. The compiler takes the stream representation with exposed parallelization opportunities and generates appropriate *mappings* and *schedules* based on available resources. Mappings refer to the assignment of each actor to run on one or more cores, and the schedule describes the number of times and order in which mapped actors are executed.

6.2 Previous Work

Stream languages lay the groundwork for programmers to work within the stream model and produce efficiently parallelizable code. Existing stream programming languages and compilers include *StreamIt* [147] and *Spindle* [46] providing basic frameworks for structured data flow programming, *Brooke* [27] and NVIDIA's *CUDA* [122]

for GPUs, *StreamC* [109] targeting the Imagine architecture, and *Spiral* [131] for mapping high performance linear transforms onto parallel architectures among others. The runtime scheduler presented here takes graph data generated by the StreamIt compiler front end with the StreamIt benchmarks used at its inputs.

Significant effort has been directed at developing and improving automated parallelization and mapping of various programs onto the growing number of data parallel and multicore architectures. Much of this effort has been focused heavily on "embarrassingly" data parallel applications such as those used in many scientific computing applications including many high resolution physical models or intractably large solution space search algorithms. Commercial parallelization frameworks are currently available with a focus on these highly data parallel applications. Examples include MATLAB's Parallel Computing Toolbox [44] or NVIDIA's CUDA [47]. These frameworks share some similarities with the stream language approach in that the programmer is in general required to identify available parallelism in the code whether through Message Passing Interface (MPI), or other intrinsics. However, in contrast with the stream programming model, reduced structural formalism leaves significant work to the compiler where deducing opportunities for partitioning and load balancing when mapping to arbitrary architectures can be intractable.

6.2.1 Architectures and Static Compilation Techniques

Approaches to the multicore mapping and scheduling problem can be broadly classified as static (offline or compile-time) and dynamic (online or run-time) or a combination of the two. Previous stream scheduling work includes several static compiler techniques on various dataflow architectures. In general, the static or compiler mapping and scheduling approach provides highly optimized solutions when advance knowledge of hardware configurations is available. Static scheduling algorithms tend to provide extremely high performance solutions, but at the expense of high computational

complexity and low reconfigurability at runtime. For this reason, existing techniques tend to target very specific architectures.

The *Imagine* stream processor is described as a co-processor with a general purpose processor serving as host [88]. Imagine explicitly targets data parallelism and SIMD optimizations in data streams and inter actor or kernel producer-consumer data locality. The architecture utilizes a stream register file (SRF) to store a large number of tokens which are loaded into ALU clusters for simultaneous execution, after which the results are returned to the SRF. Notably, the system is designed as a *distributed register file* architecture which requires careful communication scheduling, but significantly reduces power, area, and delay performance when contrasted with a traditional monolithic *central register file* architecture. Scheduling algorithms for Imagine focus on efficiently allocating the SRF, exploiting producer-consumer locality, and maximizing concurrency [89].

The *kernelC* compiler targeting Imagine implements very large instruction word (VLIW) control and data flow analysis, mapping and communication scheduling of kernels onto the processing elements. The StreamC compiler works on the stream graph to efficiently map required buffers onto the SRF statically such that buffers have sufficient space available to avoid additional memory accesses [108] [109]. Although the Imagine processor does a great job of exploiting producer-consumer locality of data associated with communication via inter kernel data streams mapped to the SRF, the sequential execution of all kernels in a complex stream graph can limit locality. This limitation is due to the fact that there may be many stages between execution of a kernel writing to the SRF and the kernel reading those results.

The *Merrimac* architecture [51] is essentially a scaled version of Imagine with a similar instruction set. Merrimac is designed to offer very low cost computational throughput by exploiting the low cost of adding large numbers of arithmetic units thanks to the effective expansion of silicone real estate, while addressing the high cost

of communication bandwidth which is in turn exacerbated by the expansive real estate. Again exploiting the locality of data in streaming applications means that given the right underlying architecture, compilers can map actors onto processing elements such that the majority of communication takes place through very short channels at substantially reduced cost in comparison with broadcast, cross-chip, or off-chip communication. Das et al. [52] uses operation or actor ordering to reduce off-chip communication and hide memory latencies. The *strip mining* technique, a form of loop unrolling for vector processors [159] [103], is exploited to reduce the local memory requirement or eliminate memory spilling by looping through a series of software kernels over suitably sized slices of the data stream.

Burger et al. [29] introduced a class of instruction set architectures called Explicit Data Graph Execution (EDGE) to address the failure of increasingly complex and cumbersome superscalar processors to achieve sufficient parallelism. EDGE's main characteristic is described as *direct instruction communication*, the notion of delivering the result of a computation directly from the producer to the intended consumer without intermediate register read and write operations.

The *TRIPS* processor is a dataflow processor using the EDGE ISA targeting instruction level parallelism with a two dimensional grid of ALUs. Using a *block-atomic execution* model, the compiler groups instructions into atomic *hyperblocks* which are fetched, executed, and committed together on the ALU grid. The TRIPS compiler attempts to create large hyperblocks with large amounts of instruction level parallelism. Unfortunately there are several factors limiting the compiler's ability to construct blocks large enough to fully utilize the ALUs. A limited number of memory reads and writes are allowed per hyperblock, and all hyperblocks are defined to be single entry. As a consequence, it can be difficult for the compiler to build hyperblocks nearing the 128 instruction limit. Unlike Imagine, this model does not require explicit scheduling for communication and instruction execution, instead relying on instruction

data predication executing many instructions simultaneously as long as ALU resources are available. Data predication makes run-time scheduling an inherent component of the TRIPS architecture as instructions are automatically executed as soon as their inputs are available, but the basic block to processing element and instruction to ALU assignments are done statically.

Raw is an early example of a fine grained stream architecture presented by Waingold et al. in 1997 [2] [155]. Raw has been designed to reduce hardware complexity through the use of many simple tiled processing elements and to help usher in the current era of billion-transistor computer chips. By exposing communication and memory details to the compiler, static partitioning, placement, routing, and scheduling optimizations at the instruction level significantly reduce the need for expensive hardware management.

The Raw compiler system performs an idealized partitioning of tasks based on the number of available tiles. Threads are then mapped to cores attempting to minimize communication latency, followed by generation of the required network routing and a global schedule. A static global schedule reduces the need for synchronization at runtime. Additionally, configurable logic enables custom instructions designed by the compiler framework to further optimize individual tiles for targeted performance improvements based on the thread to tile mapping. Exposure of inter PE communication details to the compiler in the Raw architecture has the advantage of reducing communication overhead and increasing predictability thanks to detailed offline scheduling, but the additional complexity vs. an asynchronous network communication architecture also suggests that runtime schedulers may have difficulty generating high performance schedules with sufficient turnaround.

The StreamIt compiler takes a variable grained approach to static stream scheduling. Thies et al. [148] introduces several general compiler optimizations for the StreamIt language along with analysis of their correctness. *Phased schedules* are introduced

as a compromise to achieve the reduced code size of single appearance schedules (SAS) [123] with the reduced latency and buffer requirements of a predicated *pull schedule*.

The StreamIt framework presented by Gordon et al. [62] targeting Raw generally follows the steps laid out for Raw above [155] with partitioning, mapping, and scheduling phases. By merging (*fusing*) or splitting (*fissing*) actors [1] in the stream graph appropriate granularity for the target architecture is achieved. Fusion and fission can be done vertically, in effect respectively lowering or raising the number of steps in a pipeline, or horizontally, respectively decreasing or increasing the number of parallel streams in a portion of the graph. Using a greedy algorithm to ensure load balancing, actors are either fused into larger actors until the number of actors shrinks to match the number of tiles in the architecture or fissed until the number of actors grows to match the number of tiles in the target architecture. Simulated annealing is used to produce a mapping with low communication overhead. Lastly, communication paths and timing are generated as required by the software exposed Raw architecture.

In their subsequent work, Gordon et al. [61] improve exploitation of data parallelism in their stream scheduling algorithm by coarsening granularity in the stream graph before fissing aggregated stateless actors and reducing the communication and synchronization costs associated with fissing actors without the coarsening step.

The Cell Broadband Engine (CBE) shares some fundamental characteristics with Raw. Both architectures use a distributed memory model with a scalable communication network, and in both models, memory access and inter processing element communication are software exposed. Unlike Raw, the individual processing elements, referred to as synergistic processing elements (SPEs) in the CBE architecture, have a relatively large scratchpad memory suitable for coarse grained stream graph partitioning. Additionally, the CBE network is based on an addressed ring rather than a

[1] Actors are referred to as *filters* in StreamIt.

statically scheduled grid. The CBE is also a heterogeneous architecture with a general purpose *control plane* processor referred to as the power processing element (PPE).

The *multicore streaming layer* (MSL) presented by Zhang et al. [160] provides an abstracted interface for stream compilers targeting the CBE with the goal of implementing required explicit communication primitives, simplifying buffer management, and reducing the complexity associated with high-level scheduling and load balancing optimizations. The framework is implemented for the CBE with the existing StreamIt compiler infrastructure and optimizations, but the system could be ported to alternate architectures with the benefit of enabling reuse of the stream compiler front end and optimizations for multiple target architectures.

Additional stream schedulers specifically targeting the CBE include *stream graph modulo scheduling* (SGMS), a StreamIt based scheme employing an ILP formulation for optimally unfolding the stream graph and balancing workloads across cores and a heuristic step assigning actors to pipeline stages from Kudlur et al. [95]. This work is extended by Choi et al. [42] with additional real-time and buffer constraints.

Che et al. [32] improves upon SGMS with a two-step compile time scheduler for StreamIt also targeting the CBE which considers code overlay and instruction mapping to the limited scratchpad memory. In the first step, an integer linear programming formulation is described to optimally partition workloads across processing elements including fusing and fissing of actors. Next, a greedy heuristic is used to efficiently map code onto available memory using code overlays.

Park et al. [126] presents a static technique referred to as *team scheduling* targeting the ELM Architecture [18] [50] which is characterized by a distributed memory with hierarchical data registers and software managed communication similar to Raw. The benchmarks are converted to *elk* [125], which is based on StreamIt adding multiple input/output streams and variable rate. Park's implementation attempts to improve

upon SGMS by handling situations where buffer requirements may not be stable as is the case with variable-rate streams and improved handling of feedback loops. Team scheduling works to group individual actors assigned to the same processing element such that there is no need to check for availability of input data within the *team* and synchronization occurs only after executing all executions of all actors in the team. Communication costs are amortized for individual teams by increasing the number of actor executions per synchronization. By decoupling synchronization for different parts of the stream graph, both latency and buffer requirements are reduced.

6.2.2 Dynamic Runtime Techniques

Dynamic scheduling for dataflow applications promises to improve performance in comparison to static scheduling in circumstances where detailed knowledge of the runtime environment is unavailable at compile-time, or when resource availability is expected to vary at runtime as with multitasking. Existing static approaches tend to be limited by the need for detailed knowledge about the target architecture and accurate work estimates for scheduled actors needed to implement effective load balancing. When considering handheld devices which are increasingly proving to be dominant computation tools in general, user interaction is critically important. As users start and stop applications while expecting hardware to perform well while executing multiple applications with changing priorities, static mapping and scheduling decisions become less effective as the possible system state space grows with the increasing number of available processing elements. Wiggers et al. [158] makes such an argument for runtime scheduling and demonstrate that, in particular for dataflow based applications, runtime scheduling can be managed at a high level of abstraction with predictable behavior and resource requirements.

In addition to the previously presented Multicore Streaming Layer, Zhang et al. [160] presents a dynamic scheduling scheme for the CBE. The authors argue that

dynamic scheduling is beneficial when the behavior of the stream program is less predictable, particularly in terms of predicted actor execution times which is important for accurate load balancing. Their approach relies exclusively on input predicated execution of actors in the stream graph, with the PPU assigning or subscribing work to available SPUs at run-time. SPU-SPU communication is not implemented according to the authors due to limited SPU scratchpad memory limitations. Such communication may also be ineffective in this scheme because actors are not given affinity to particular processing elements in effect requiring main memory storage of all generated data streams.

Although it targets scalar rather than streaming code, Bellens et al. [20] presents a runtime parallelization and scheduling framework *CellSs* which takes notations from the programmer identifying code segments to be executed on an SPE and based on a dependency graph, executes as many of those segments as possible concurrently. Tasks are statically assigned to SPEs, and runtime queues are maintained for each SPE with waiting tasks issued when SPEs are idle. The system will also reallocate tasks to an idle SPE at runtime if its queue becomes empty and work is available. The results indicate the system performs very well on deeply data parallel applications, but it is less effective on applications with complex dependency graphs.

Blagojevic et al. [24] presents *multigrain parallel scheduling* (MGPS), combining task-level and loop-level parallelization schemes for the CBE by making load balancing decisions at run-time. The implementation is based on an embarrassingly parallel DNA analysis tool configured for parallel execution using the Message Passing Interface (MPI) protocol. CBE performance is first enhanced through manual vectorization of key code segments. The MGPS scheme seeks to use task level parallelism within the MPI framework as long as there are enough concurrently executable tasks to keep the SPEs occupied. When task-level parallelism is insufficient, an intra-task loop-level scheme is used to spread the work of a single task across multiple SPEs. The scheme is limited by the compile-time assignment of all tasks to all processing

elements, as well as the additional overhead associated with runtime task to processing element assignment and limited use of inter SPE communication.

MGPS is improved in [25] with runtime profiling of important program execution phases to improve sub-task loop granularity decisions by the online scheduler for evolutionary performance improvement. This is done by assigning actor work to processing elements at the execution level such that although actor ordering and dependencies are addressed based on the actor level stream graph, work associated with stateless individual actors may be spread across two or more processing elements. The scheduler presented here is also designed to take advantage of low cost run-time profiling enabling generation of accurate evolutionary schedules and addressing limited compile-time knowledge about actor workloads as well as workloads which vary as a function of the input data. Similar to the previous implementation, the work focuses on two DNA analysis benchmark algorithms both exhibiting high levels of data or task level parallelism, but performance results on the CBE do not come within 60% of optimal using 8 SPEs.

The work that comes closest to the presented implementation is the *Flextream* algorithm from Hormati et al. [70]. Their algorithm is based on an offline ILP generating a parallelized solution assuming all resources are available and an online heuristic dynamically modifying the offline solution according to available resources. The requirement for detailed architecture configuration information and generation of a parallelized solution off-line ties the approach to a particular configuration of the target architecture, limiting its portability and reusability.

Further, as the offline interim solution is optimized for maximum resource availability, it can be expected that performance of the online heuristic is more limited with respect to the theoretically optimal solution when resources are most constrained and thus most unlike the reference schedule. In contrast the presented approach generates a canonical schedule offline for the stream program which is then dynamically paral-

lelized at run-time based on available resources, and which performs more efficiently when those resources are most constrained. Significantly, unlike the existing effort requiring estimates of actor execution times or even profiling data, the offline portion of the presented framework is fully independent of such requirements.

6.3 The Lightweight Runtime Scheduler Framework

The lightweight stream scheduler is designed to maximize run-time flexibility in the absence of information at compile time about the number of cores available in the target architecture or their memory sizes. Additionally, the run-time algorithm is designed to be as lightweight as possible enabling dynamic resource reallocation in the presence of dynamic multi-tasking while finding solutions with near-optimal load balancing and throughput. A high level view of the scheduler framework is presented in Figure 6.2.

The assumption is made that limited information is available about the target architecture at compile time. In this work the problem of compiling source code for specific instruction set architectures is not addressed. The target architecture is also assumed to be a multicore processor with many identical or similar cores and limited distributed memory similar to the CBE. Additionally, it is assumed the stream program may be modeled as a *Synchronous Data Flow* (SDF) [97] [87]. The SDF model is a data flow formalism in which permissible periodic sequential schedules may be statically determined because the number of data tokens consumed and generated in each actor are fixed and known at design time.

6.3.1 Offline Analysis

The offline portion of the scheduling algorithm aims to provide the run-time system with enough information to identify a schedule which will maximize parallelism at run-time. The stream language compiler relies on the application programmer to identify opportunities for task and data level parallelism as expressed in the stream data flow

graph. The offline algorithm takes the *canonical* graph representation of the stream program as generated early in the StreamIt compiler process. The canonical graph represents the programmer's view of the application before any transformations are performed on the graph.

Hormati et al. [70] takes the approach of finding a detailed offline solution for a specific target hardware assuming maximum availability of resources. Similar to compile time schedulers presented in [95] and [32], Hormati uses an Integer Linear Programming (ILP) technique which may be extremely time consuming or even intractable for large code sizes and large numbers of target cores even given just tens of actors or cores, limiting scalability. In [95] a 32 core solution for *vocoder* with 96 actors takes 2 minutes. In the present work the opposite approach is taken. A lightweight system generates a general periodic single core solution in a few milliseconds, and gives the run-time scheduler a basis for generating efficient multicore schedules.

Periodic Admissible Sequential Schedule (PASS)

The PASS [87] is a sequential single core schedule for a given SDF graph which can be executed periodically on an infinite stream of input data. A PASS for the given stream program defines the number of executions of each actor in the graph such that after executing the schedule once, tokens are consumed from the input stream, tokens are generated on the output stream, and the number of tokens in all FIFO buffers is bounded, i.e. the number of tokens in FIFOs does not grow to $\pm\infty$.

In the case of acyclic stream graphs it is possible to generate a PASS in which all FIFO buffers are empty before the first schedule execution (except for input FIFO buffers from source). In general a PASS may require non-empty FIFO buffers based on the sequence of actor executions. If an actor b from the PASS illustrated in Figure 6.3 on page 139 appears in the schedule before its input token has been generated by actor a, then a token must be present in the FIFO, as an initial buffer condition.

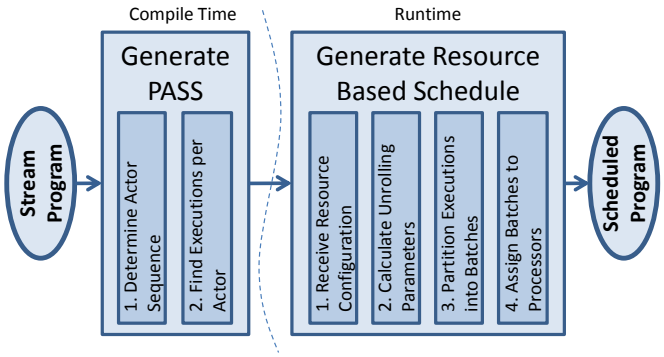


Figure 6.2: Scheduler framework.

This situation also occurs in the stream model when actors peek at tokens in their input FIFO without popping them from the FIFO. As we are interested in a dynamically re-targetable and parallelizable PASS schedule, we will deliberately consider and construct only sequences that do not require non-zero initial FIFO buffer conditions. Such a sequence can be easily generated by ordering actor executions so that no actor appears in the PASS prior to any other actor for which it has any dependency. An actor f is dependent on another actor a if there is a directed path from a to f in the stream graph as is the case in Figure 6.1a.

If the graph contains any backward edge which results in a cycle, then the initial buffer condition tokens are unavoidable. The current StreamIt benchmark programs do not include backward edges, and cycles are currently not handled in the presented system, although they may be addressed by clustering all actors in the cycle together as a single stateful actor, or by treating all actors in the loop as stateful thus enabling pipelining within the loop.

An in-order sequence of actors is easily computed by traversing the graph depth first, adding an actor to the sequence only once all of its parents have been added. As an example, we traverse the graph in Figure 6.1b on page 119 as indicated by the lettered labels, visiting the left-most child at each step in the depth first traversal. When we first encounter actor g, we have the sequence $< a, b, c, d >$. We find that a parent of

g, i.e. f, has not been added to the sequence. We do not add g to the sequence but instead return so that the next actor encountered is e as indicated by *Arrow* 1 in the figure. Now the algorithm returns $< e, f, g >$ but does not add k since its parent j is not in the sequence. The algorithm continues at the point indicated by *Arrow* 2 to complete the ordering of actors with $< h, i, j, k >$.

The sequence generated tends to maintain temporal locality of actors in *pipelines* consisting of linear sequences of actors. Since each actor in the pipeline is dependent only on the output of its predecessor, the depth first fully in-order sequence tends to minimize buffer memory requirements thanks to buffer reuse and communication requirements thanks to actor locality during sequential execution of the PASS.

In addition to determining a desirable sequence of actor executions we must also determine the number of executions required of each actor to satisfy the bounded buffer requirement. This can be solved as a system of linear equations as described by Jantsch [87]. The implementation presented here uses a simple method of finding the least common multiples (LCMs) of push and pop values on each edge giving the number of executions required for each associated actor as $LCM/push$ for the parent and LCM/pop for the child, then ensuring that the same execution number is associated with each edge of each actor. The final PASS is composed of all actors in their depth first in-order sequence with each actor repeated according to the calculated number of executions. For the example graph in Figure 6.1b, the calculated PASS comprised of individual actor *executions* is represented by the sequential schedule

$$PASS =< a_0, a_1, b_0, c_0, d_0, e_0, f_0, g_0, h_0, h_1, i_0, j_0, k_0 > \tag{6.1}$$

6.3.2 Run-time Scheduling

The run-time scheduling algorithm takes as its inputs the canonical stream graph representing the stream application, the PASS generated by the offline algorithm, and information about the target architecture including number of cores and core memory

size. The online algorithm attempts to optimally balance work among the available cores without exceeding available memory. An optimally balanced workload results in ideal throughput performance because it minimizes the amount of time any processing element will be idle due to pipeline stalls.

We define the theoretical optimal work per core, W_{opt}, for any schedule as the total work required to execute the sequential PASS once, W_{PASS}, divided by the number of cores, n

$$W_{opt} = \frac{W_{PASS}}{n} \tag{6.2}$$

If work is given in clock cycles, i.e. time, then the theoretical optimal throughput, T_{opt}, is calculated as

$$T_{opt} = \frac{1}{W_{opt}} \tag{6.3}$$

In designing a multicore schedule, there are two primary obstacles to achieving optimal throughput.

- The optimal throughput calculation assumes actor workloads can be distributed based on a continuous domain. In reality actors have discrete *atomic* workloads, and optimally balancing work across all cores may be impossible. In such cases the resulting throughput is limited by the core with the greatest workload, W_{max}. Without perfect load balancing, the maximum achievable throughput, T_{max} is then $1/W_{max}$, and since W_{max} must be larger than W_{opt} : $T_{max} < T_{optimal}$.
- Data communication can dramatically limit throughput. In the case of CBE and other similar processors communication overheads can be amortized through *double buffering*.

The first step in the online scheduling algorithm is to calculate initial schedule parameters based on the generated PASS and available resources. Here we calculate an *unrolling factor* and a theoretical optimum or target core workload, W_{opt}.

The unrolling factor, j, is defined as the number of PASS instances in the periodic parallel schedule of the stream graph [87]. In essence, the work load in j iterations of the PASS is distributed as evenly as possible across multiple cores to generate the multicore mapping. If an unrolling factor is used, W_{PASS} is multiplied by the unrolling factor to find the total work, W_{total}, to be distributed among available cores, and we replace Equation 6.2 with Equation 6.5.

$$W_{total} = j \cdot W_{PASS} \tag{6.4}$$

$$W_{opt} = \frac{W_{total}}{n} \tag{6.5}$$

When the PASS is unrolled, the number of executions for each actor in the PASS are multiplied by j. For example, an unrolling factor $j = 2$ implies that the number of executions of each actor in the PASS is doubled. Larger values of j imply more executions in the unrolled PASS. The PASS from Equation 6.1 with an unroll factor of 2 becomes the schedule

$$\begin{aligned} S =< & a_0, a_1, a_2, a_3, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, \\ & f_0, f_1, g_0, g_1, h_0, h_1, h_2, h_3, i_0, i_1, j_0, j_1, k_0, k_1 > \end{aligned} \tag{6.6}$$

PASS unrolling can dramatically improve opportunities for precise load balancing by increasing available *resolution*, k, defined as the ratio

$$k = \frac{W_{opt}}{W_{max\_atomic}} \tag{6.7}$$

where W_{max_atomic} is the maximum *atomic work unit*. Large values of k reflect high resolution, indicating that all atomic work units are small relative to W_{opt}. W_{max_atomic} is defined as the larger of the maximum workload over all stateless actors, W_{max_asl}, and the maximum workload over all stateful actors multiplied by the unrolling factor, $j \cdot W_{max\_asf}$:

$$W_{max\_atomic} = \max(W_{max\_asl}, j \cdot W_{max\_asf}) \tag{6.8}$$

Increasing the unrolling factor can linearly improve resolution leading to more precise load balancing dependent upon the following conditions:

- **Case I** $W_{max\_asf} = 0$: If there is no stateful work, j is limited only by available memory as discussed in Section 6.3.5. In the absence of memory limitations, resolution grows linearly and infinitely with the unrolling factor.

$$\lim_{j \to \infty} \left(k = \frac{W_{opt}}{W_{max\_atomic}} = \frac{\left(\frac{j \cdot W_{PASS}}{n}\right)}{W_{max\_asl}} \right) = \infty \tag{6.9}$$

- **Case II** $W_{max\_asf} > 0$: In the presence of stateful actors, the maximum useful unrolling factor derived from Equation 6.8 is given by

$$j_{max} = \left\lceil \frac{W_{max\_asl}}{W_{max\_asf}} \right\rceil \tag{6.10}$$

  If $j < j_{max}$, increasing j increases resolution as indicated in Equation 6.9 in Case I. However, when $j \geq j_{max}$, $W_{max\_atomic} \coloneqq j \cdot W_{max\_asf}$ and we have

$$k = \frac{W_{opt}}{W_{max\_atomic}} = \frac{\left(\frac{j \cdot W_{PASS}}{n}\right)}{j \cdot W_{max\_asf}} = \frac{\left(\frac{W_{PASS}}{n}\right)}{W_{max\_asf}} \tag{6.11}$$

  In this case increasing j has no further impact on resolution.

In effect, so long as $j \cdot W_{max\_asf} < W_{max\_asl}$ resulting in $W_{max\_atomic} \coloneqq W_{max\_asl}$, increasing the unrolling factor raises resolution and opportunities for load balancing. Once $j \cdot W_{max\_asf} \geq W_{max\_asl}$, increases to j have no further effect on resolution or load balancing efficiency while buffer memory requirements continue to grow.

Constrained Number of Useful Processors

The presence of stateful actors also imposes an absolute ceiling on theoretical performance. While programs without stateful actors can theoretically be infinitely unrolled

and distributed across an arbitrary number of processing elements, stateful work inherently limits the unrolling factor as described above, limiting the possible number of useful processors, n. Since atomic work can never be divided across processors, throughput can be at most $1/W_{max\_atomic}$ as n grows large, and Equation 6.5 is modified to give

$$W_{opt} = \max\left(\frac{W_{total}}{n}, W_{max\_atomic}\right) \tag{6.12}$$

From Equation 6.12, we find that the number of useful processors is limited by

$$n \leq W_{total}/W_{max\_atomic} \tag{6.13}$$

6.3.3 Actor Execution to Batch Assignment

In this step, all work in the unrolled PASS is partitioned into batches such that the work assigned to each batch, W_{batch}, is as close to W_{opt} as possible without exceeding it. The unrolled PASS is represented as a vector of actor executions as illustrated in Figure 6.3 for the stream graph in Figure 6.1b with $j = 1$. The lettered boxes in the figure represent atomic executions of actors. The amount of work associated with each actor is represented by the width of the box, indicating 1, 2, or 3 units of work. In this case, the PASS has a total work size, $W_{total} = 16$. We will partition the PASS for scheduling on $n = 4$ available cores giving $W_{opt} = W_{total}/n = 4$. Actor executions are assigned to the current batch in the order in which they appear in the PASS as seen in the figure. When adding the next actor execution would cause W_{batch} to exceed W_{opt}, a new batch is created to which the execution is added.

Once the number of batches matches the number of available cores, after B_3 is complete in the example, the n batches, $B_0 - B_3$, are sorted from *smallest to largest* according to their assigned work. In this case the order is $< B_0, B_1, B_2, B_3 >$. A new series of batches is started with B'_0 and a new target work size is calculated for each new batch in the B' series. For batch B'_n, $W'_{opt} = W_{opt} - work(B_n)$. In the example, for B'_0, $W'_{opt} = W_{opt} - work(B_0) = 1$. New B' batches are created in the same manner as the original n batches with the exception that a new W'_{opt} is calculated for each batch. The

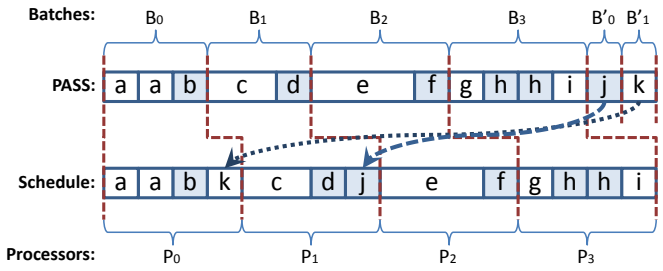


Figure 6.3: Execution to batch, and batch to processor assignment.

end result is that each member of B' compliments a member of B so that the sum of their assigned work, $work(B_n) + work(B'_n)$ is as close to W_{opt} as possible. This property will be exploited in the batch to processor assignment stage.

As batches are constructed, information about internal and external buffers and stage numbers is collected. Internal buffers are set aside as required, assigned to actors, and reused where possible. External buffers are set aside, along with source and destination information and space to accommodate double buffering. Batches are assigned stage numbers based on their earliest possible execution determined by inter batch communication dependencies. Batches are always assigned to the smallest possible stage number.

After actor execution to batch assignment is complete, a new atomic unit of work is defined. An actor *event* is comprised of all executions associated with a single actor which are assigned to the same batch. At run time, this set of actor executions will always be executed atomically as a single event.

6.3.4 Batch to Processor Assignment

In this step, batches are assigned to available processors. If the total number of batches, $|B|$, is less than or equal to the number of available cores, n, then each batch is assigned to a core and no further work is necessary. However, if $|B| > n$, we must merge batches until $|B| = n$.

While $|B| > n$, all batches are sorted from *smallest to largest* in terms of their assigned work. In the example in Figure 6.3, the sorted list is $< B'_0, B'_1, B_0, B_1, B_2, B_3 >$. Next batch $B_{(|B|-1)-n}$ is merged with $B_{|B|-n}$, i.e. B'_1 is merged with B_0.

Fundamentally, the $|B| - n$ batches with the smallest amount of assigned work are treated as the extra batches. The batch from among the remaining n batches with the smallest amount of assigned work is merged with the extra batch with the largest amount of work until $|B| = n$. Finally each of the n batches are assigned to one of the n cores.

6.3.5 Memory Constraints

Memory requirements associated with each core consist of the total buffer requirements for all *intra* processor FIFOs, buffers for incoming and outgoing *inter* processor FIFOs including double buffering requirements, instruction memory, and any stack and heap memory requirements. The largest memory requirement from among all cores as determined by the multicore schedule tells us whether the generated schedule will fit completely into distributed on-core memory or spilling of some buffers or instruction code over to main memory is necessary.

The online scheduling algorithm minimizes memory usage to the greatest extent possible. However, the PASS unrolling factor linearly increases buffer requirements as it grows. For instance, for $j = 2$, doubling the number of executions of all actors in a batch will double the amount of data generated during one execution of the batch. In effect buffer requirements are doubled.

If the generated schedule is found to violate on-core memory constraints, a new unrolling factor is computed and a new schedule is generated. Based on the linear relationship between the unrolling factor, j, and buffer requirements, the new unrolling factor, j', is set to a fraction of j proportional to the ratio of the size of available memory in a processing core, $M_{available}$, and the largest core memory requirement dictated by

Table 6.1: StreamIt Benchmarks

benchmark	actors	% stateful	total code size(B)
beamformer	57	75%	761
bitonic_sort	40	0%	136
channelvocoder	55	62%	3,815
dct	8	0%	239
des	53	8%	652
fft	17	0%	342
filterbank	85	38%	5,303
fm	43	33%	2,725
mpeg2_subset	23	4%	1,232
serpent_full	120	28%	7,248
tde-pp	29	3%	1,104
vocoder	114	60%	1,989

the generated schedule, M_{max}:

$$j' = j \cdot \frac{M_{available}}{M_{max}} \tag{6.14}$$

If Equation 6.14 gives $j' = j$ and $j' > 1$ then $j' := j' - 1$. As long as the memory constraint is violated and $j > 1$, j is set equal to j', and the schedule is reconstructed. If the unrolling factor has been reduced to one, and the memory constraint is still violated, buffers must be relocated to main memory until the memory constraint is met. Relocating, or *spilling* buffers or instructions to main memory introduces expensive communication costs, and will generally reduce performance. The spilling mechanism's implementation and performance have not been evaluated in this work.

6.4 Experimental Setup

6.4.1 Simulation

The offline and online portions of the lightweight scheduler have been implemented in C++ with instrumentation for collecting timing data on the scheduling algorithm. Dynamic multicore schedules are generated for the 12 "*asplos 06*" StreamIt benchmarks [7] described in Table 6.1 across ranges of available cores from 1 to 128, and core memory sizes from 8KB to 256KB. For the simulated performance results, the scheduling algorithm itself is run without threading on a dual core Pentium 4 processor running at 3.2GHz with 1.5GB RAM and 512kB cache.

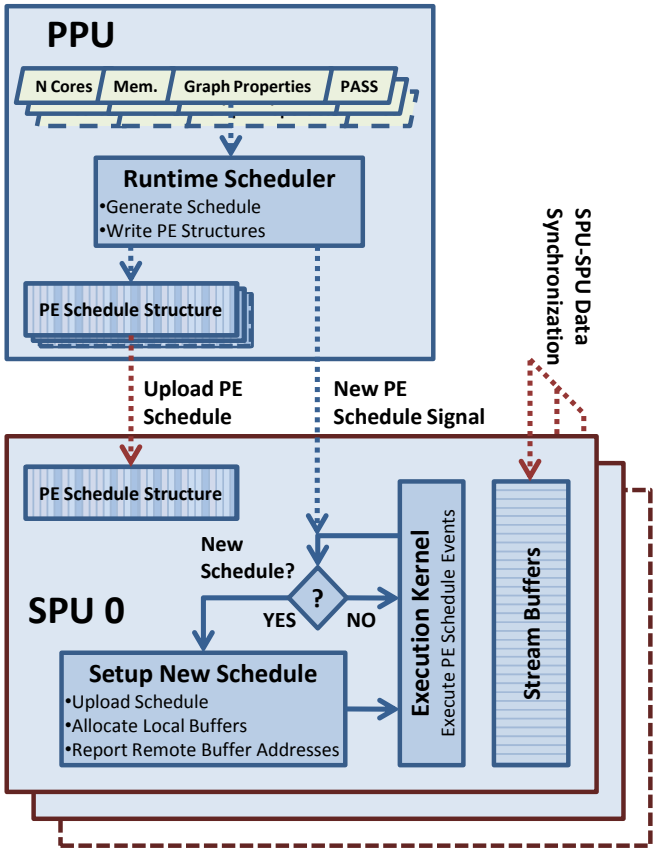


Figure 6.4: Lightweight runtime scheduler implementation on the CBE.

Simulated performance numbers are reported in terms of steady-state throughput and are computed based on the maximum of the maximum processor workload and communication costs in the presence of double buffering. Using double buffering, throughput performance is dominated by communication overhead any time the communication cost for any core is larger than the maximum workload from among all cores. Throughput numbers for generated schedules are compared against the theoretical optimal throughput in the absence of memory constraints. As such, the optimal performance figures reflect schedules generated using non-memory-restricted unrolling factors and buffer sizes.

6.4.2 Cell BE Implementation

The lightweight runtime stream scheduler is implemented on the CBE experimentally validating simulated scheduler performance. An overview of the CBE implementation is presented in Figure 6.4. The scheduler itself consists of the on-line scheduling algorithm running on the PPU and an execution framework running on each SPU. The PPU executes the online schedule generation algorithm as well as a data structure construction phase which formats the generated schedule for execution on each allocated SPU.

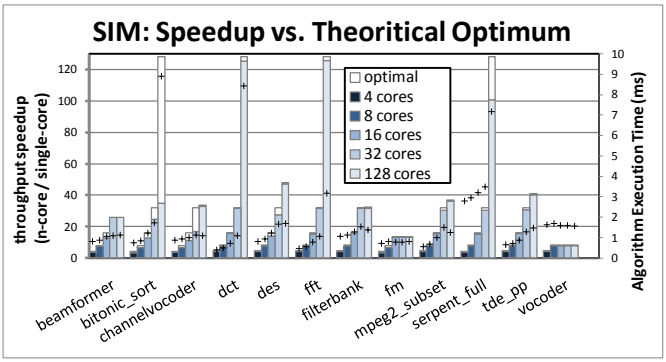
Each active SPU maintains an execution kernel whose state is controlled by the PPU through mailbox messages. The SPU kernel enters a wait mode as soon as the SPU thread is started. Once a start message is received from the PPU, the SPU reloads its initialization control block previously updated by the PPU with pointers to data structures defining the local SPU portion of the schedule. Once the schedule data structures are loaded into the local store (LS), the SPU constructs local buffers and associated headers in accordance with the uploaded schedule information. Once all buffers have been established, each SPU delivers the addresses of local remote buffers to the remote SPUs scheduled to use them.

The current SPU schedule execution scheme is fully asynchronous, with actors executing whenever sufficient data is available on their inputs. The SPU kernel loops through its assigned actor events in PASS order so that pipelines are always executed sequentially and in order. If the first actor in a pipeline executes, then data is guaranteed to be available for the next actor in the pipeline and so on. The PPU generated schedule also contains all of the stage assignment information needed to implement a stage phased execution.

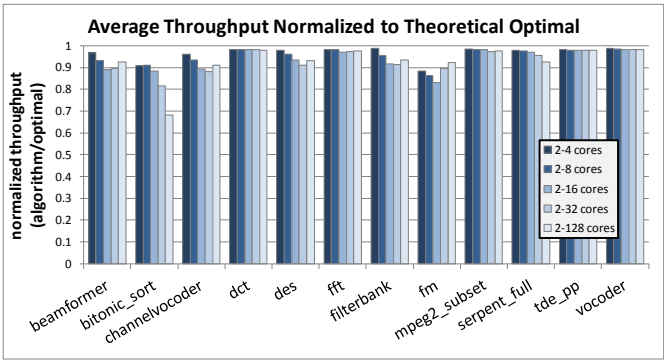
A pull-down data communication scheme is implemented for all SPU-SPU producer-consumer data communication. Actors producing data for use by remote

SPUs always write the data to a local buffer comprised of multiple identical buffer *components* for double buffering purposes. When a local buffer component is filled by the producer, a signal is transmitted to the consumer SPU updating an associated buffer component pointer. The remote consumer SPU initiates a DMA transferring data from the producer's local buffer. Once complete, the consumer transmits a signal back to the producer updating the associated buffer component pointer notifying the producer that the buffer component is again available for writing. All communication signals including data transfers are executed using non-blocking DMA operations. Prior to actor execution, if any required inputs have DMAs in flight, the associated tag status is checked. If the DMA is found to be complete, the input buffer data is marked as valid, and actor execution can begin.

The CBE implementation includes all PE synchronization, buffer management, and communication including producer-consumer buffer data transfers required to physically execute the benchmark programs. However, in the current implementation, file reader-writer functionality is not implemented. Physical execution of the actor or function code is also not implemented. Instead a delay function is incorporated emulating actor execution based on compiler work estimates or defined actor workloads. The delay function utilizes the CBE timebase as a time unit rather than clock cycles which have been used in the simulation. The timing utility used in the SPU relies on a decrementer register with resolution defined by the system *timebase*. The timebase decrementer is the highest resolution timer available in the SPU, but it represents a lower resolution timer than the system clock. For the Cell Blade QS20 system at the Georgia Institute of Technology, Sony-Toshiba-IBM Center of Competence used to collect results in this work, the system timebase is defined at 14.318 million ticks per second, or 223.5 clock cycles per tick based on the 3.2GHz system clock. Consequently, throughput figures are effectively scaled down according to this ratio. Additionally, because overhead costs associated with buffer management and actors' pushing and



(a) Online algorithm speedup over single core performance vs. optimal speedup. Speedup of bitonic_sort is limited by communication costs which cannot be hidden due to small work sizes.



(b) Average of schedule throughput performances ratios to optimal performance for each benchmark over 5 ranges of available target cores.

Figure 6.5: Overall simulated performance results.

popping data are not similarly scaled, their real cost is substantially mitigated in the presented performance results.

6.5 Results

6.5.1 Simulated Performance

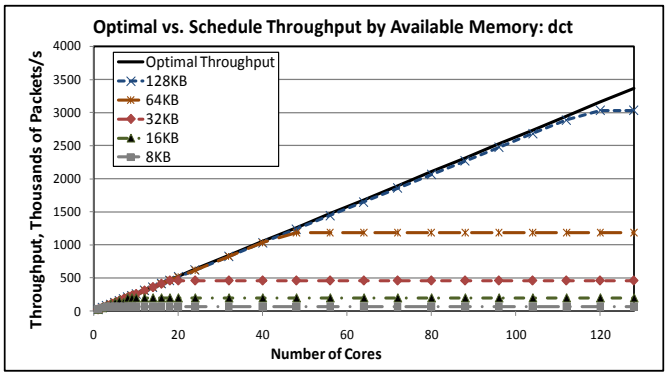
Simulated throughput speedup figures for schedules generated by the online algorithm are illustrated in Figure 6.5a for a core memory size of 256KB as in the CBE. Speedup is calculated as the ratio of schedule throughput to single-core throughput. Shaded bars indicate throughput achieved by the generated schedule for a given number of available processing cores. White bars indicate additional theoretical optimal performance not realized in the generated schedule.

Ideally, the optimal speedup should match the number of available cores. Given 128 cores, we should expect to see a theoretical speedup of 128x over single-core solutions as seen in Figure 6.5a for *bitonic_sort*, *fft*, and *serpent_full*. However, optimal speedup is limited by the maximum atomic work as described in Section 6.3.2.

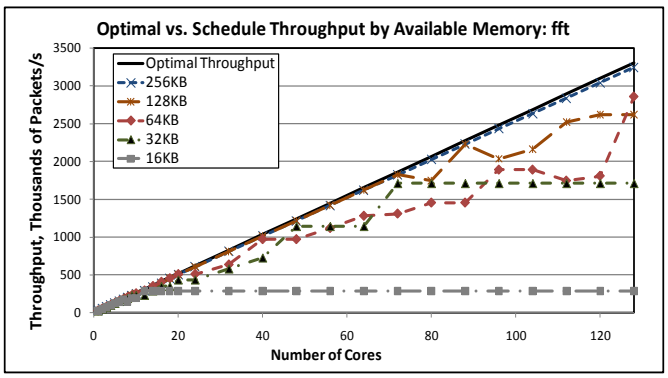
Schedules generated by the online scheduler give speedup figures closely matching the theoretical optimal for most configurations. The average ratio of schedule speedup to optimal speedup across all configurations shown in the figure is 92%. The solutions for *bitonic_sort* are limited by communication costs because all of its actors have very small work sizes and are unable to mask communication costs using double buffering, particularly when large unrolling factors are used.

Also shown in Figure 6.5a are execution times for the online scheduling algorithm indicated by a '+' for each configuration. Scheduler execution times are linearly dependent on the total number of actor executions in the generated schedule. In practice, execution times are more directly dependent on the number of actor events in the schedule because the overhead of allocating memory to create a new event or batch is much higher than the overhead associated with accumulating work for actor executions. Large numbers of actors or processing elements in particular contribute to the algorithm's execution time. These factors contribute to larger schedule generation times for *bitonic_sort*, *dct*, and *serpent_full* at 128 cores. The first two use large unrolling factors, 23 and 94 respectively. *Serpent_full* uses a small unrolling factor of three, but has the largest number of actors, many of which are executed up to 128 times in the unmodified PASS. Overall, the average online algorithm execution time for a 32 core configuration including stage assignment and buffer allocation is 1.35ms on the Pentium platform described in Section 6.4.1. This suggests an improvement over the average run-time of 15.9ms reported by Hormati [70] on a slightly different set of benchmarks with fewer actors.

Figure 6.5b shows the ratio of schedule throughput to optimal throughput aver-



(a) Average of schedule throughput performances ratios to optimal performance for each benchmark over 5 ranges of available target cores.



(b) Average of schedule throughput performances ratios to optimal performance for each benchmark over 5 ranges of available target cores.

Figure 6.6: Throughput performance vs. available memory in arbitrarily scalable benchmarks.

aged over ranges of available processing elements. We omit single core solutions from this average since they are always throughput optimal. Five of the benchmarks consistently have solutions within 5% of the optimal across all ranges of available cores. Due to better atomic resolution for smaller numbers of cores, particularly in cases where unrolling is not possible, the highest quality solutions tend to be found when the number of cores is most restricted, i.e. 1-16 cores. This is significant, as optimum performance is most critical when resources are most constrained.

Memory Constraints

Given a memory constraint of 8KB, the online scheduler finds solutions for half of the benchmarks, *beamformer*, *bitonic_sort*, *dct*, *des*, *fm*, and *vocoder* without the need to spill memory objects into main memory. Five of the remaining benchmarks have solutions that fit into a 16KB memory without spilling. The remaining benchmark, *tde_pp*, requires more storage due to several actors with very large push and pop sizes of up to 7KB. Consequently, some schedule cores still require spilling for memory sizes up to 48KB, although this is well within the workable range of existing systems such as the CBE.

For some benchmarks, particularly those without stateful actors, increasing the available memory increases opportunities for unrolling the PASS, enabling extremely well balanced solutions for large numbers of processing cores. The charts in Figure 6.6 illustrate the effect of increasing available memory on *dct* and *fft* throughput. Solutions are available for small memories, but throughput is limited due to restricted PASS unrolling. As additional memory is made available, unrolling factors increase, improving atomic work resolution and enabling better solutions. For both benchmarks, near optimal solutions are possible for an arbitrary number of cores given adequate memory resources.

6.5.2 CBE Performance

Overall performance results for the CBE are illustrated in Figure 6.7 with solutions generated for a 128kB memory size. As in the simulated results, throughput performance tends to track closely with the optimal values. The two benchmarks, *beamformer* and *channelvocoder*, with the most limited performance when using 16 SPUs have a maximum atomic work unit defined by large stateful actors, limiting opportunities for increasing partitioning resolution through PASS unrolling. A number of the remaining

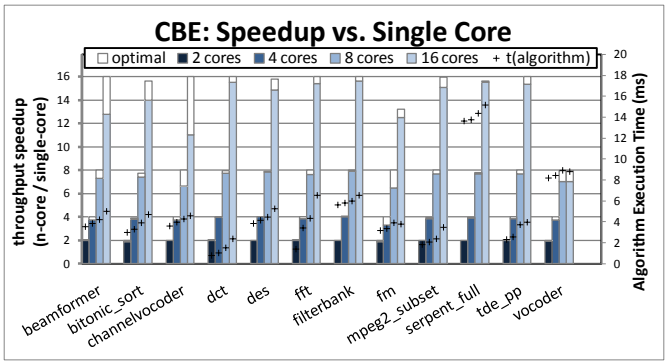


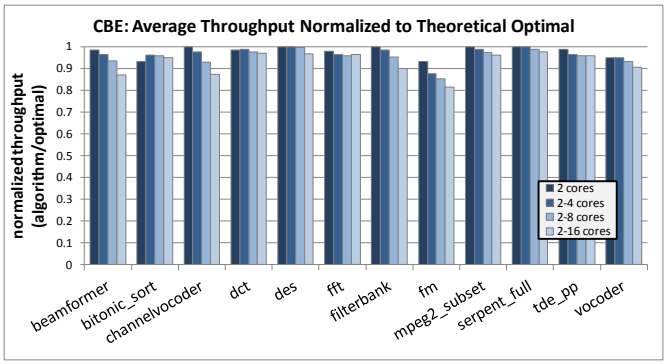
Figure 6.7: Speedup over single core performance measured on the CBE vs. optimal speedup.

benchmarks are also limited by stateful maximum atomic work, but the algorithm is still able to find sufficient opportunity for near optimal load balancing. As seen in the simulated results in Figure 6.5a, both of these benchmarks approach optimal performance again as the number of cores increases to 32, since stateful work also limits optimal performance. In generating CBE results, the online scheduling algorithm is run on the PPE which is slower than the Pentium processor used to generate results in the previous section. The average schedule generation time on the PPE was 3.9ms for these benchmarks based on the data points in Figure 6.7.

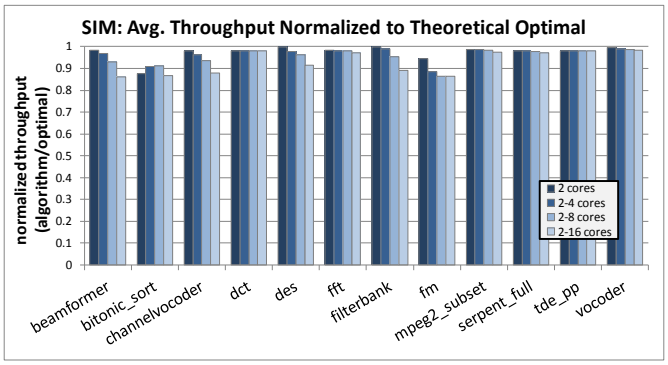
Measured schedule throughput performance tracks the simulation results very closely, serving to validate performance expectations for configurations beyond 16 cores in the simulated results. Figure 6.8 gives the normalized average performance figures for the CBE implementation as well as the corresponding figures collected through simulation. The average of absolute differences between the data points in the two graphs is 2.0%.

Schedule Latency

For stream graphs without stateful work, a trivial parallel solution giving optimal throughput speedup for an arbitrary number of PEs is available if we simply assign the entire PASS to each PE. The downside to such a solution is that the schedule's *latency*, defined



(a) Average of schedule throughput performances ratios to optimal performance for each benchmark over 4 ranges of available target cores.



(b) Equivalent simulated performance figures for simulator validation.

Figure 6.8: CBE average performance figures with simulated figures for comparison.

by the time between schedule initiation and completion of the first results, is fixed by latency in the single core schedule.

Schedule latency is determined by the time required for data tokens to propagate through the schedule. For the single core schedule, the entire PASS is executed, so that latency is equivalent to the sum of all actor work described by the PASS. When the PASS is spread across multiple PE's with actor execution level resolution, an actor, a, which is executed n times in the PASS as an example may have $n/2$ executions assigned to two different PEs. Since actor work divided across PEs must be stateless, both sets of actor executions may be executed in parallel and the time needed to complete work associated with actor a in the PASS partitioned schedule is half of that needed in the original PASS. In effect, spreading actor executions across PEs exploits data par-

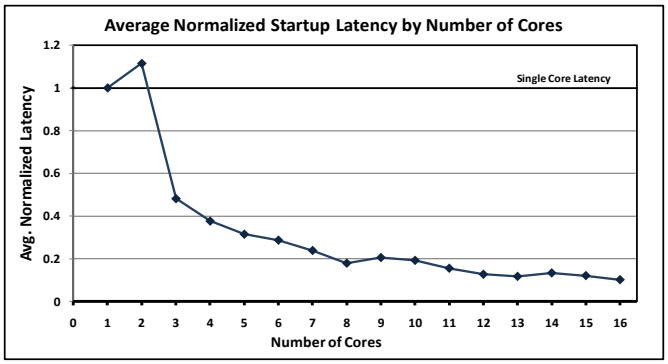
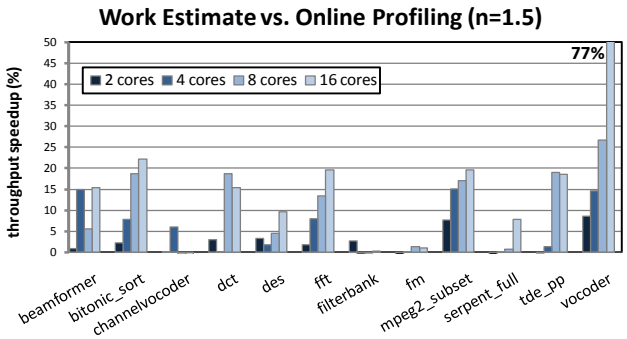


Figure 6.9: Latency: For each benchmark, latency is normalized to the single core value. Each data point in the chart represents the normalized latency for *n* processing elements averaged across all benchmarks.

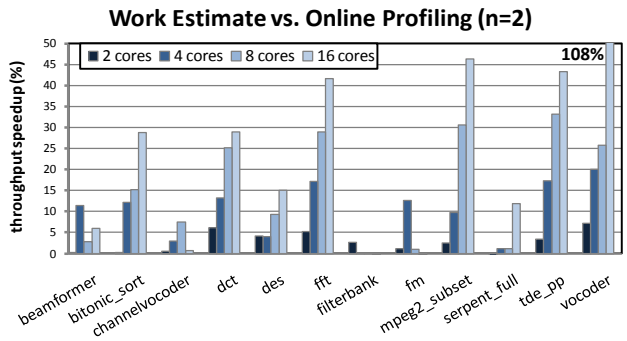
allelism to both reduce the maximum PE workload *increasing throughput*, and enable data parallel execution *reducing latency*.

Figure 6.9 illustrates measured latency normalized to the single core value for 1-16 PEs. Each data point is the average across all benchmarks. The effect of exploiting data parallelism on latency reduction can be seen in the figure for 3-16 PEs. In general, a large latency improvement occurs once the schedule is assigned three or more PEs. Latency continues to fall as the number of cores increases up to 16 cores. For much larger numbers of cores, the latency number is expected to rise again for arbitrarily scalable benchmarks, as large unrolling factors are needed to exploit the available PEs, eventually overcoming the latency benefits of data parallelism.

For the majority of benchmarks, the addition of a second core did not significantly impact latency. In three instances, *bitonic_sort*, *dct* and *des*, the addition of a second core actually increased latency as expressed in Figure 6.9. This effect results from increasing the unrolling factor in the scheduler followed by construction of a schedule which does not partition any actor work across PE boundaries. The result is a purely task parallel schedule with increased total work. Task parallelization or pipelining improves throughput but not latency. Pure task level partitioning also accounts for the lack of latency reduction in other benchmarks when adding a second core.



(a) Work randomization scaling factor: 1.5



(b) Work randomization scaling factor: 2

Figure 6.10: Online profiling performance advantage.

Online Profiling and Schedule Refactoring

Executed actor emulation delays are initially based on the workload estimates generated for each actor by the StreamIt compiler. In order to test the effectiveness of runtime profiling on schedule improvement, a randomization function has been added to the offline system enabling random scaling of the executed work estimates, W, within a range, $[W/n, W \cdot n]$, such that W is the distribution's median value. The scaled work estimate is retained separately from the work estimate data used to generate online schedules. The scaled value is used by the actor emulation delay function at run time resulting in performance degredation.

This method makes possible evaluating the effectiveness of runtime profiling as a means of improving performance in the face of inaccurate work estimates. Since the

online scheduling algorithm enables run-time refactoring of the schedule, the scheduler can overcome inaccurate offline work estimates which might otherwise significantly impact throughput performance. This is an important benefit of the run time scheduler over static scheduling.

The implemented runtime profiler uses the SPU decrementer to measure actor execution and overhead costs. This information is periodically transmitted to a data structure in main memory using non-blocking DMA transfers. When a new schedule is generated by the PPU, the actor work estimates are replaced with the collected profiling data prior to schedule generation. The resulting schedule is expected to produce improved throughput since the scheduler's load balancing effort will more closely match reality.

Representative performance increases are presented in Figure 6.10. The data points presented represents the average of five separate sets of randomly generated actor work values constructed in accordance with the distribution described above. Separate scaling values of $n = 1.5$ in Figure 6.10a and $n = 2$ in Figure 6.10b have been used. In the first figure, the overall average improvement after updating the schedule based on profiling data is 7%. The performance improvement tends to increase with the number of PEs used in the schedule. Since actual throughput is dependent on whichever PE has the largest workload, increasing numbers of PEs expose the system to more chances of poor performance in the face of inaccurate work estimates. At 16 PEs, the average throughput increase is 16% for $n = 1.5$. In the case of $n = 2$, where work estimates range from half to twice the actual workload, the average improvement is 11%, 27% when using 16 PEs. These values represent average cases, but performance in individual cases can be arbitrarily bad due to bad partitioning decisions made by the scheduler using inaccurate actor workload data, a condition exemplified by *vocoder* in Figure 6.10.

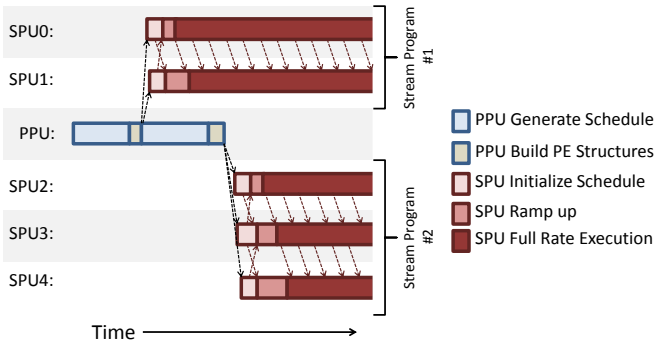
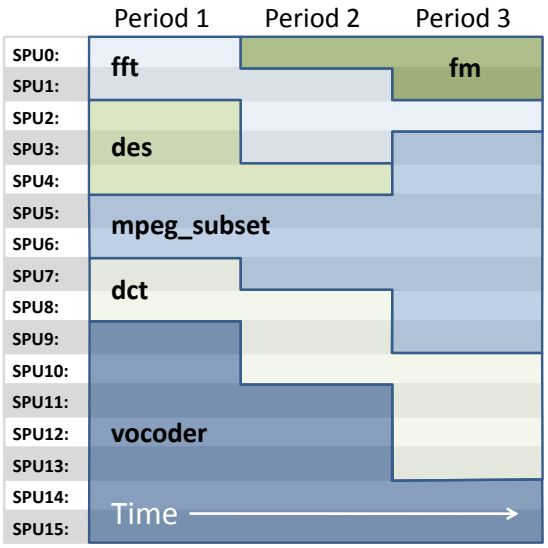


Figure 6.11: Multitasking setup and execution. The PPU generates a schedule for each stream program. Assigned SPUs are notified that a new schedule is ready. The structures are uploaded and initializations are completed while the PPU continues with the next program.

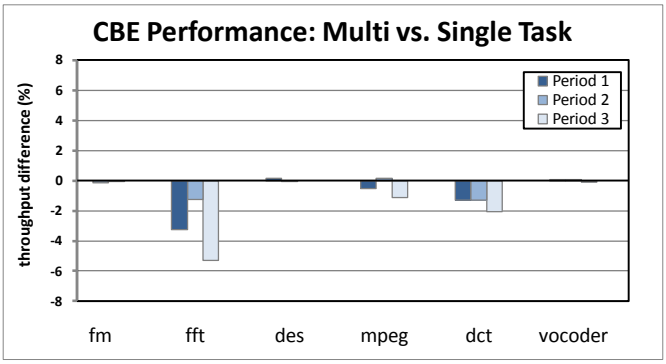
Multitasking performance

Multitasking applications on the multicore architecture is an important motivation for development of a runtime scheduler. Given applications such as future handheld devices with many core embedded processors, users will dynamically determine which applications are running at any given time. Due to the large number of possible operating conditions in terms of available resources presented to each process, offline optimization of every configuration is undesirable or even intractable. The current CBE implementation supports multitasking by concurrently running an arbitrary number of benchmarks simultaneously limited only by the number of available processing elements.

Figure 6.11 provides a notional overview of system behavior when simultaneously executing multiple stream programs. Once a schedule is generated, the assigned PE's are notified that a new schedule is available. Each SPE uploads its control block containing pointers to it's dedicated schedule structures. The SPE specific structures are uploaded, buffers are allocated and the execution kernel is initialized. SPUs with



(a) In a multitasking scenario, a dynamic set of applications is running on the available PEs. Periodically, the resources allocations for each application are updated, resulting in a new application to PE mappings.



(b) Multitasking throughput performance on the CBE.

Figure 6.12: Multitasking scenario and throughput performance on the CBE.

external buffer dependencies exchange pointers to associated buffer headers located in each SPU's local store, and begin executing actors as data becomes available.

A multitasking execution case study is described in Figure 6.12a. The figure illustrates a time series of run-time schedule mappings with up to five applications running simultaneously. The PE resources assigned to each application are periodically reassessed, resulting in a new mapping of applications to PEs. In addition, the number of tasks changes over time as a new application, *fm*, is added in period two while one of the original applications, *des*, is terminated before period three.

Throughput performance for the simultaneously executing multicore benchmark schedules compared to running the same schedule as a single task on the CBE is illustrated in Figure 6.12b. In general, the single task performance is matched in the multitasking environment. Three of the benchmarks experience a noticeable decline in performance in the multitasking environment. The presented data indicates an average 0.5% decline is seen in *mpeg2_subset*, a 1.6% decline in *dct*, and an average decline of 3.3% in *fft*. These three benchmarks happen to have the largest buffer communication requirements, suggesting that increased contention for the CBE's on-chip communication network in the multitasking environment is likely the primary contributer to their reduced throughput performance.

6.6 Conclusion

In this work, a lightweight run-time stream program scheduling scheme has been presented. The offline algorithm generates a simple canonical sequence and single core schedule with limited knowledge of the target architecture's resources. The run-time scheduler has been shown to quickly generate schedules with near optimal throughput performance based on dynamically available resources such as an allocation of processing cores and their memory constraint. Such a system is invaluable for multitasking multicore processors representing the next generation of handheld embedded computing systems.

Throughput performance has been simulated for multicore systems up to 128 cores over a range of memory sizes. An implementation of the online scheduler and runtime framework for the CBE serves to validate the simulation results as well as offer insights into online profiling, latency, and multitasking throughput performance.

Future efforts will focus on more sophisticated memory management techniques such as a detailed analysis of memory spilling, and code and data overlay techniques as well as improved buffer management and actor ordering within batches. Addition-

ally, additional work is needed to address the geometry of available PEs and how best to assign work to each element considering locality for communication and on-chip network contention.

Chapter 7

CONCLUSION

Embedded computing systems operating in unique and autonomous environments are particularly sensitive to power and performance constraints. In addition, user expectations regarding everything from the type of tasks and services these systems can perform to battery life to context and user awareness continue to rise precipitously. The increasing ubiquity of such systems in areas such as remote military operations under a massive fully burdened cost of energy introduced in Chapter 1 serves to illustrate the degree of urgency with which improved efficiency solutions are sought.

In this dissertation, several techniques in power/performance optimization have been explored for substantially alleviating pressure on embedded systems to perform in the face of tight resource constraints. Conclusions and areas for further investigation have been included at the end of each chapter. From a broadened and more comprehensive perspective, conclusions, assessments of this work's contributions, and directions for further research are presented in the following sections.

7.1 Power Aware System Management

The presented power aware system enabling techniques, including exchanging QoS for power in advanced bistable display technology presented in Chapter 2 and the state of the art H.264 video codec in Chapter 4, have been demonstrated to provide effective power management modes via the controlled introduction of quality distortion in the user experience. The real power savings enabled through these power optimization schemes provide performance options and operating modes genuinely empowering users with valuable control over the target system's behavior. Users, given the ability to express their expectations for system performance by prioritizing power consumption

vs. performance metrics, are rewarded with improved user experience and improved utilization of the embedded computational system.

The video quality vs. power scheme as presented relies heavily on accurate workload prediction in order to accurately select frequency voltage states to maximize power savings. This estimate and profiling were done offline in the presented work. Online estimation of the workload is an important area of research for video processing in general and represents an area where further investigation is warranted. The video block dropping scheme may also directly benefit from the agile core scaling and power management scheme presented for H.264 in Chapter 5, which effectively avoids the need for workload prediction all together while substantially alleviating buffering requirements typical in such systems. Combining QoS states in the power aware scheme with multicore processor and power scaling schemes is left for future work.

7.2 Multiprocessor Paradigm

The presented multicore power and performance enhancements ultimately contribute to the feasibility of many core embedded MPSoCs. These enhancements substantially address several of the difficulties associated with implementing applications for such architectural models. Several lessons can be carried forward as embedded multiprocessor systems move toward increasingly parallel architectures and workloads with increasing emphasis on power efficient implementations.

Scalable parallel implementation of an extremely complex application, as with the parallel H.264 video decoder presented in Chapter 4, has brought forth numerous insights into difficulties associated with parallelizing an extremely large code base as well as highlighting fundamental differences between traditional sequential execution focused programming styles and emerging multicore centric methods such as stream programming. A solid understanding of these characteristics is essential to successful automated parallelization efforts. Additional investigation of improved data and stream

centric implementations of the H.264 codec using the stream programming model, which have yet to be realized, and their mappings for MPSoCs are warranted. Future efforts will also benefit from further investigation of the H.264 encoder for which preliminary results have been detailed in Appendix C.

Effectively automating implementation of applications for multicore architectures is an important step toward successful adaptation of next generation embedded MPSoCs. The two methods presented, automated instruction mapping for distributed processing element local memories presented in Chapter 5 and dynamic scheduling for multitasking multicore data-centric streaming applications presented in Chapter 6 have been shown to realize near optimal solutions for difficult mapping problems. In the future, combining instruction mapping with multicore stream scheduling can further improve scheduler performance by reducing memory constraints in certain applications within highly restricted memory environments.

REFERENCES

[1] Advanced Micro Devices. Technical Overview: ATI Stream Computing. Online.

[2] A. Agarwal. Raw computation. *Scientific American*, 1999.

[3] E. Akyol and M. van der Schaar. Complexity model based proactive dynamic voltage scaling for video decoding systems. *Multimedia, IEEE Transactions on*, 9(7):1475 –1492, 2007.

[4] M. Alvanos, G. Tzenakis, D. S. Nikolopoulos, and A. Bilas. Parallelization and performance of an h.264 video encoder on the cell b.e. In *Proceedings of the Fifth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, 2009.

[5] M. Alvarez, A. Ramirez, X. Martorell, E. Ayguade, and M. Valero. Scalability of Macroblock-level Parallelism for H.264 Decoding. In *ACACES*. Technical University of Catalonia (UPC), 2008.

[6] M. Alvarez, E. Salamí, A. Ramirez, and M. Valero. A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC. In *HiPEAC*. Universitat Politecnica de Catalunya, 2005.

[7] S. Amarasinghe. Streamit downloads. Online, 2007. As of February 2011.

[8] AMD Corp. Six-Core AMD OpteronŹ Processor Features. Online.

[9] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[10] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 259–267, Washington, DC, 2004.

[11] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 298 – 309, 2005.

[12] H. Baik, K.-H. Sihn, Y. il Kim, S. Bae, N. Han, and H. J. Song. Analysis and Parallelization of H.264 decoder on Cell Broadband Engine Architecture. In *Signal Processing and Information Technology*, pages 791–795. Samsung Electron. Co., Ltd., Suwon, Korea, 2007.

[13] M. A. Baker and K. S. Chatha. A lightweight run-time scheduler for multitasking multicore stream applications. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 297 –304, 2010.

[14] M. A. Baker, P. Dalale, K. S. Chatha, and S. B. Vrudhula. A scalable parallel h.264 decoder on the cell broadband engine architecture. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 353–362, New York, NY, USA, 2009. ACM.

[15] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha. A performance model and code overlay generator for scratchpad enhanced embedded processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 287–296, New York, NY, USA, 2010. ACM.

[16] M. A. Baker, V. Parameswaran, K. S. Chatha, and B. Li. Power reduction via macroblock prioritization for power aware h.264 video applications. In *CODES+ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 261–266, New York, NY, USA, 2008. ACM.

[17] M. A. Baker, A. Shrivastava, and K. S. Chatha. Smart driver for power reduction in next generation bistable electrophoretic display technology. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 197–202, New York, NY, USA, 2007. ACM.

[18] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An energy-efficient processor architecture for embedded systems. *IEEE Comput. Archit. Lett.*, 7:29–32, 2008.

[19] F. Bellard. FFmpeg. http://www.ffmpeg.org/.

[20] P. Bellens, J. Perez, R. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, page 5, 2006.

[21] Berkeley View Lab. Sony/toshiba/ibm cell processor. Online. As of July 2010.

[22] T. Bert, H. D. Smet, F. Beunis, and K. Neyts. Complete electrical and optical simulation of electronic paper. *Displays*, 27(2):50 – 55, 2006.

[23] L. Blackwell. Lcd specs: Not so swift. *PC World*, 2005.

[24] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *Proceedings of the 12th*

ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '07, pages 90–100, New York, NY, USA, 2007. ACM.

[25] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Comput.*, 33(10-11):700–719, 2007.

[26] M. T. Brundage. Future trends and thrusts for army manportable power sources. In *US Army RDECOM CERDEC. 2007 Joint Service Power Expo*, 2007. As of July 2010.

[27] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[28] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, page 288, Washington, DC, USA, 1995. IEEE Computer Society.

[29] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37:44–55, 2004.

[30] P. Calyam and C. gun Lee. Characterizing voice and video traffic behavior over the internet. In *In: Proceedings of the international symposium on computer and information sciences (ISCIS)*, 2005.

[31] W. Che and K. S. Chatha. Design of an automatic target recognition algorithm on the ibm cell broadband engine. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 21 –28, 2010.

[32] W. Che, A. Panda, and K. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1118 –1123, 2010.

[33] T.-C. Chen, Y.-W. Huang, and L.-G. Chen. Analysis and design of macroblock pipelining for h.264/avc vlsi architecture. In *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, volume 2, pages II – 273–6 Vol.2, May 2004.

[34] Y.-K. Chen, E. Q. Li, X. Zhou, and S. Ge. Implementation of H.264 Encoder and Decoder on Personal Computers. *Journal of Visual Communication and Image Representation*, 17(2):509–532, 2006.

[35] Y.-K. Chen, X. Tian, S. Ge, and M. Girkar. Towards efficient multi-level threading of h.264 encoder on intel hyper-threading architectures. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 63, 2004.

[36] W.-C. Cheng, Y. Hou, and M. Pedram. Power minimization in a backlit tft-lcd display by concurrent brightness and contrast scaling. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10252, Washington, DC, USA, 2004. IEEE Computer Society.

[37] C. C. Chi, B. Juurlink, and C. Meenderinck. Evaluation of parallel h.264 decoding strategies for the cell broadband engine. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 105–114, New York, NY, USA, 2010. ACM.

[38] S. Cho and R. G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21:342–353, 2010.

[39] Y. Cho, S. Kim, J. Lee, and H. Shin. Parallelizing the h.264 decoder on the cell be architecture. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 49–58, New York, NY, USA, 2010. ACM.

[40] I. Choi, H. Shim, and N. Chang. Low-power color tft lcd display for hand-held embedded systems. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 112–117, New York, NY, USA, 2002. ACM.

[41] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 732–737, New York, NY, USA, 2002. ACM.

[42] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream compilation for real-time embedded multicore systems. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[43] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient Parallelization of H.264 Decoding With Macro Block Level Scheduling. In *IEEE International Conference on Multimedia*, pages 1874–1877. University of California, Berkeley, USA, 2007.

[44] R. Choy, A. Edelman, and C. M. Of. Parallel matlab: Doing it right. In *Proceedings of the IEEE*, pages 331–341, 2005.

[45] B. Comiskey, J. D. Albert, H. Yoshizawa, and J. Jacobson. An electrophoretic ink for all-printed reflective electronic displays. *Nature*, pages 253–255, May 1998.

[46] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: A dsl approach to specifying streaming applications. In F. Pfenning and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39815-8_1.

[47] N. Corporation. Nvidia cuda c programming guide version 3.2. Online, 2011. As of February 2011.

[48] R. Cytron and P. G. Loewner. An automatic overlay generator. *IBM Journal of Research and Development*, 30:603–608, Nov. 1986.

[49] A. Dalisa. Electrophoretic display technology. *Electron Devices, IEEE Transactions on*, 24(7):827 – 834, July 1977.

[50] W. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *Computer*, 41(7):27 –32, 2008.

[51] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 35–, Washington, DC, USA, 2003. IEEE Computer Society.

[52] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 33–42, New York, NY, USA, 2006. ACM.

[53] DRS Technologies. Handheld Terminal Unit (HTU-EK). Online, Aug. 2005.

[54] E. T. F. Editors. Migration of the cell broadband engine to 45nm soi. Online, 2008. As of July 2010.

[55] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *International Conference On Embedded Software*, pages 321–330, Seoul, Korea, 2006.

[56] S. Electronics. Lcd driver ic s6b0723a. Specification.

[57] F. Gatti, A. Acquaviva, L. Benini, and B. Ricco'. Low power control techniques for tft lcd displays. In *CASES '02: Proceedings of the 2002 international conference on*

Compilers, architecture, and synthesis for embedded systems, pages 218–224, New York, NY, USA, 2002. ACM.

[58] J. D. Gelas. Dynamic Power Management: A Quantitative Approach. Online: AnandTech.com, 2010.

[59] T. G. George. Video codec basics. Online, 2007. As of July 2010.

[60] gnu.org. GCC online documentation. http://gcc.gnu.org/onlinedocs/.

[61] M. I. Gordon, W. Thies, and S. 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.

[62] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. *SIGARCH Comput. Archit. News*, 30:291–303, 2002.

[63] H.264/AVC. Reference software. http://iphome.hhi.de/suehring/tml/.

[64] T. R. Halfhill. Looking Beyond Graphics: NVIDIAŠs Next-Generation CUDA Compute and Graphics Architecture, Code-Named Fermi, Adds Muscle for Parallel Processing. White paper, NVIDIA Sponsored, Sept. 2009.

[65] M. S. Hasan, M. LaMacchia, L. Muzzelo, R. Gunsaulis, L. R. Housewright, and J. Miller. Designing the joint tactical radio system (jtrs) handheld, manpack, and small form fit (hms) radios for interoperable networking and waveform applications. In *Military Communications Conference, 2007. MILCOM 2007. IEEE*, pages 1 –6, 2007.

[66] R. Henning and C. Chakrabarti. A Quality/Energy Tradeoff Approach for IDCT Computation in MPEG-2 Video Decoding. In *IEEE Workshop on Signal Processing Systems*. SUNY, Binghamton, 2000.

[67] Hewlett-Packard Corp., Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced Configuration and Power Interface Specification, 2010. Revision 4.0a.

[68] A. Hoban. Designing Real-Time Solutions on Embedded Intelő Architecture Processors. Online, May 2010.

[69] M. Hopper and V. Novotny. An electrophoretic display, its properties, model, and addressing. *Electron Devices, IEEE Transactions on*, 26(8):1148 – 1152, 1979.

[70] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, Washington, DC, USA, 2009. IEEE Computer Society.

[71] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8 –11, 1994.

[72] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/AVC Baseline Profile Decoder Complexity Analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):704–716, 2003.

[73] S. Hua, G. Qu, and S. Bhattacharyya. An Energy Reduction Technique for Multimedia Application With Tolerance to Deadline Misses. In *DAC Proceedings*, pages 131–136. UMD, 2003.

[74] Y. Huang, A. V. Tran, and Y. Wang. A Workload Prediction Model for Decoding MPEG Video and its Application to Workload-Scalable Transcoding. In *Proceedings of the 15th international conference on Multimedia*, pages 952–961. National University of Singapore, 2007.

[75] H. Hwang, T. Oh, and S. H. Hyunuk Jung. Conversion of Reference C Code to Dataflow Model: H.264 Encoder Case Study. In *IEEE*. Seoul National University KOREA, 2006.

[76] IBM. *Cell Broadband Engine Architecture*. IBM Systems and Technology Group, 2007.

[77] IBM. *Software Development Kit for Multicore Acceleration Version 3.1 Programmer's Guide*. IBM Systems and Technology Group, 2008.

[78] M. Ihmig. Porting linux 2.6.9 to the PXA270 based development platform. research collaboration between Intel and CMU, May 2005.

[79] S. Inoue, H. Kawai, S. Kanbe, T. Saeki, and T. Shimoda. High-resolution microencapsulated electrophoretic display (epd) driven by poly-si tfts with four-level grayscale. *Electron Devices, IEEE Transactions on*, 49(9):1532 – 1539, 2002.

[80] Intel. Intel PXA27x processor family developer's manual. Apr. 2004.

[81] Intel Corp. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor. Online, 2004.

[82] Intel Corp. Intel Xeon Processor 5600 Series, Datasheet Volume I. Online, 2010.

[83] S. Irani, G. Singh, S. Shukla, and R. Gupta. An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(12):1349 – 1361, 2005.

[84] T. R. Jacobs, V. A. Chouliaras, and D. J. Mulvaney. Thread-Parallel MPEG-4 and H.264 Coders for System-on-Chip Multi-Processor Architectures. In *International Symposium on Parallel Computing in Electrical Engineering*, pages 363–368. Loughborough University, UK, 2006.

[85] A. Jagmohan, B. Paulovicks, V. Sheinin, and H. Yeo. H.264 Video Encoding Algorithm on Cell Broadband Engine. Department of Multimedia Technologies, IBM TJ Watson Research, Oct. 2005.

[86] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proc. Asia and South Pacific Design Automation Conference*, pages 612–617, Yokohama, Japan, 2006.

[87] A. Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[88] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, Sept. 2002.

[89] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. In *in Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 82–92, 2001.

[90] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *Micro, IEEE*, 21(2):35 –46, Mar. 2001.

[91] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Dally. A programmable 512 gops stream processor for signal, image, and video processing. *Solid-State Circuits, IEEE Journal of*, 43(1):202 –213, 2008.

[92] J. G. Kim, J. W. Kim, and C. C. J. Ku. Corruption Model of Loss Propagation for Relative Prioritized Packet Video. In *SPIE Proceedings*. USC, 2000.

[93] J. S. Kim, J. G. Kim, K. O. Kang, and J. Kim. A Distortion Control Scheme for Allocating Constant Distortion in FD-CD Video Transcoder. In *IEEE International Conference on Multimedia and Expo*, pages 161–164. Hanbat National University, 2004.

[94] C. Krasic, J. Walpole, and W. Feng. Quality-Adaptive Media Streaming by Priority Drop. In *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 112–121. OHSU, 2003.

[95] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, 2008.

[96] B. Lee, E. Nurvitadhi, R. Dixit, C. Yu, and M. Kim. Dynamic voltage scaling techniques for power efficient video decoding. *J. Syst. Archit.*, 51(10-11):633–652, 2005.

[97] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, 1987.

[98] E. Q. Li and Y.-K. Chen. Implementation of H.264 Encoder on General-Purpose Processors with Hyper-Threading Technology. In *SPIE*. Intel China Research Center, Beijing, 2004.

[99] J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77 – 87, 2006.

[100] C.-J. Lian, S.-Y. Chien, C.-P. Lin, P.-C. Tseng, and L.-G. Chen. Power-aware multimedia: concepts and design perspectives. *Circuits and Systems Magazine, IEEE*, 7(2):26 –34, 2007.

[101] Linear Technology Corporation. LTspice Users Guide . Online, Aug. 2005.

[102] C. T. Lopez. 'Beans, Bullets and BTUs' Define Army Energy Security. Official Army Homepage, July 2009.

[103] D. B. Loveman. Program improvement by source-to-source transformation. *J. ACM*, 24:121–145, 1977.

[104] E. Ltd. Video quality measurement tool v. 1.52. Videos used with permission, http://www.elecard.com/download/clips.php.

[105] A. Major, Y. Yi, I. Nousias, M. Milward, S. Khawam, and T. Arslan. H.264 decoder implementation on a dynamically reconfigurable instruction cell based architecture. In *SOC Conference, 2006 IEEE International*, pages 49 –52, 2006.

[106] B. Marks. Power consumption of multiplexed liquid-crystal displays. *Electron Devices, IEEE Transactions on*, 29(8):1218 – 1222, 1982.

[107] B. Marks. Power reduction in liquid-crystal display modules. *Electron Devices, IEEE Transactions on*, 29(12):1884 – 1886, Dec. 1982.

[108] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication scheduling. *SIGARCH Comput. Archit. News*, 28:82–92, 2000.

[109] P. R. Mattson. *A programming system for the imagine media processor*. PhD thesis, Stanford University, Stanford, CA, USA, 2002. AAI3040045.

[110] C. Medford. Microsoft/yahoo is mobile equal of google. Online, 2008. As of February 2011.

[111] C. Meenderinck, A. Azevedo, M. Alvarez, B. Juurlink, and A. Ramirez. Parallel scalability of h.264. *Parallel Scalability of H.264. 1st Workshop on Programmability Issues for Multi-Core Com-puters (MULTIPROG)*, 2008.

[112] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel scalability of video decoders. *J. Signal Process. Syst.*, 57(2):173–194, 2009.

[113] M. Mesa, A. Ramirez, A. Azevedo, C. Meenderinck, B. Juurlink, and M. Valero. Scalability of macroblock-level parallelism for h.264 decoding. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 236 –243, 2009.

[114] M. Mesarina and Y. Turner. Reduced energy decoding of mpeg streams. *Multimedia Syst.*, 9(2):202–213, 2003.

[115] Microsoft Corporation. WMV HD Content Showcase. http://www.microsoft.com/windows/windowsmedia/musicandvideo/hdvideo/contentshowcase.aspx.

[116] Milpower. Ba-5590 specification. Online. As of July 2010.

[117] G. E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[118] W. T. Moye. ENIAC: The Army-Sponsored Revolution. Online, Jan. 1996.

[119] MSU. Video quality measurement tool v. 1.52. http://compression.ru/video/.

[120] N. Blachford. Cell Architecture Explained. Online. As of July 2010.

[121] NEC. Nl2432hc17-01b qvga lcd for mobile applications with touch panel. Specification.

[122] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[123] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 497–502, Piscataway, NJ, USA, 2006. IEEE Press.

[124] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories. In *High Performance Computing*, Lecture Notes in Computer Science, Berlin, 2008.

[125] J. Park. Elk programming language. Online, 2007. As of February 2011.

[126] J. Park and W. J. Dally. Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 1–10, New York, NY, USA, 2010. ACM.

[127] D. Pham et al. Overview of the Architecture, Circuit Design, and Physical Implementation of a First-generation Cell Processor. In *IEEE Journal of Solid-State Circuits*, volume 41, pages 179–196. IBM, 2006.

[128] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001. ACM.

[129] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259, New York, NY, USA, 2001. ACM.

[130] Power Architecture editors. *An introduction to compiling for the Cell Broadband Engine architecture*. IBM, developerWorks, 2006.

[131] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[132] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression Video Coding for Next-generation Multimedia*. Wiley, The Robert Gordon University, Aberdeen, UK, 2003.

[133] A. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical Parallelization of an H.264/AVC Video Encoder. In *International Symposium on Parallel Computing in Electrical Engineering*, pages 363–368. Technical University of Valencia, 2006.

[134] M. Roitzsch. Slice-balancing h.264 video encoding for improved scalability of multicore decoding. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 269–278, New York, NY, USA, 2007. ACM.

[135] Samsung. Samsung DDR3 SDRAM specifications. Online, 2010. As of July 2010.

[136] F. H. Seitner, R. M. Schreier, M. Bleyer, and M. Gelautz. Evaluation of data-parallel splitting approaches for h.264 decoding. In *MoMM '08: Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pages 40–49, New York, NY, USA, 2008. ACM.

[137] S. Shankland. Intel steps up chip cadence. CNET News, Apr. 2006.

[138] Smart Design Technology. SAH1513 ultra high sensitivity Atheros GPS module specification. Online. As of July 2010.

[139] D. Son. Dynamic voltage scaling on mpeg decoding. In *ICPADS '01: Proceedings of the Eighth International Conference on Parallel and Distributed Systems*, page 633, Washington, DC, USA, 2001. IEEE Computer Society.

[140] T. R. Spacek. A proposal to establish a pseudo virtual memory via writable overlays. *Communications of the ACM*, 15:421–426, June 1972.

[141] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, New York, NY, USA, 2002. ACM.

[142] R. Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997.

[143] F. Strubbe, F. Beunis, and K. Neyts. Determination of the effective charge of individual colloidal particles. *Journal of Colloid and Interface Science*, 301(1):302–309, Sept. 2006.

[144] S. Sun, D. Wang, and S. Chen. A Highly Efficient Parallel Algorithm for H.264 Encoder Based on Macro-Block Region Partition. In *HPCC*. National University of Defense Technology, Changsha, China, 2007.

[145] X. P. Systems. Xtreme power systems 2.4ghz transmitter specifications. Online. As of July 2010.

[146] H. Takao, M. Miyasaka, H. Kawai, H. Hara, A. Miyazaki, T. Kodaira, S. Tam, S. Inoue, and T. Shimoda. Flexible semiconductor devices: fingerprint sensor and electrophoretic display on plastic [tft based]. In *Solid-State Device Research conference, 2004. ESSDERC 2004. Proceeding of the 34th European*, pages 309 – 312, 2004.

[147] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[148] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe. Streamit: A compiler for streaming applications. Technical report, MIT Laboratory for Computer Science, 2001.

[149] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5:472–511, May 2006.

[150] United Nations-UISP. Urgent need to prepare developing countries for surge in e-wastes. Online, 2010. As of February 2011.

[151] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proceedings of the SPIE, Image and Video Communications and Processing*, pages 1874–1877, 2003.

[152] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14:802–815, Aug. 2006.

[153] S. Vermael, K. Neyts, G. Stojmenovik, F. Beunis, and L. J. M. Schlangen. A 1-dimensional simulation tool for electrophoretic displays. In *23rd International Display Research Conference*, 2003.

[154] VideoLAN. x264 - a free h264/AVC encoder. http://www.videolan.org/developers/x264.html.

[155] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86 –93, Sept. 1997.

[156] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

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

[158] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Monotonicity and run-time scheduling. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 177–186, New York, NY, USA, 2009. ACM.

[159] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[160] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, 2008.

[161] Z. Zhao and P. Liang. Data Partition for Wavefront Parallelization of H.264 Video Encoder. In *ISCAS*. University of California, Riverside, 2006.

APPENDIX A

MOTIVATIONAL CASE STUDY

A.1 Universal Tactical Handheld Device needs assessment

The following tasks are derived from the Network Centric concept of Army operations usually described as Command Control Communications Computing Intelligence Surveillance Reconnaissance (C4ISR). The tasks are designed to fit into the Network Centric model, have relevance to the current tactical Army and be feasibly implementable in an embedded multicore system.

1. Command and Control

    a) Situational Awareness: Combat leaders make decisions based on their knowledge of the current situation. Generally accurate understanding of the situation enables leaders to correctly assess the capabilities of their arrayed forces and to more accurately anticipate the enemy's actions.

        i. GPS positioning and mapping for self and friendly forces.

        ii. Integration of intelligence information on reported and observed enemy locations and activities.

    b) Graphical Orders: Generate, transmit, display graphical orders as well as capability for briefing and rehearsing the plan.

2. Communication

    a) Software Defined Radio: A major effort currently underway within the US Department of Defense is the standardization and implementation of widely compatible communications waveforms and the physical radios capable of implementing and using any of them without the need for hardware modification. This is done using software defined radio (SDR) as part of the Joint Tactical Radio System (JTRS) program.

b) Information Security: Encryption and decryption for all data and voice transmission.

c) Multiple simultaneous networks: Simultaneous communication with higher and lower echelons, relay of sensor data, and possibly feed from external sources such as theater and national collection assets.

3. Computing: A combination of sensing and computation enhances awareness. Strong computational capabilities can rapidly turn raw sensor data into valuable information about the environment and the enemy.

   a) Data collected from a microphone array can quickly determine precise information about the direction and even location of hostile actions such as small arms and mortar fire. Audio processing may be used to identify the types of sources for sound events including voice identification of individuals and weapons' signatures.

   b) Image and video analysis can quickly aid in the identification of targets, enhance surveillance, or use facial characteristics to identify individuals based on database information.

4. Intelligence Surveillance and Reconnaissance

   a) Database capability: Store, manage and receive updates to intelligence databases. Intelligently handle system queries and minimize missed opportunities due to failure at piecing together relevant pieces of information into a coherent picture.

A.2 Use-case Scenario

A use-case scenario consistent with current operations in Afghanistan is given here. Various elements of the description are illustrated in Figure A.1. The hypothetical

soldier worn "Squad Reconnaissance Device" (SRD) is employed in response to enemy activity, in this case an attack on a local national police station seen in the figure (a). The device is embedded organically into the force at the platoon or squad level. A reconnaissance squad (b) in support of the deploying quick reaction force (QRF) is tasked with investigating the attack which has been reported by Afghan police and detonation sensors. The reconnaissance squad deploys a Raven UAV which can quickly reach the site of the attack and begin reporting with surveillance video.

The squad receives the video feed via the SRD (c) which is also capable of performing analysis such as automatic target recognition (ATR) on the incoming video stream. The squad is able to control the UAV and collect information they need to assess the activity in the target area and assist the deploying QRF as they plan and execute their response. In addition to the QRF, the surveillance feed is transmitted to a fire support platoon (d) who is able to cross reference the video data and intelligence updates with requests for fire from the tactical operations center and the QRF. The handheld device can communicate over multiple waveforms via the software defined radio system, and retransmits the UAV signal over an alternate protocol.

Analysis is performed on the incoming video stream to help track and locate dangerous objects and individuals at the scene of the attack such as identifying a potential RPG (e), recording, and retransmitting information about its reported location and updating the global intelligence database. The SRD also acts as a client of the global database, integrating intelligence reports, and providing a map view of the battle field with data overlays and analysis capabilities similar to ATR, such as line of sight analysis on the terrain (f).

A high level overview of the system architecture is given in Figure A.2 The available system components are designed to meet specific mission needs including voice and data transmission across multiple networks and protocols, display interface for video, mapping, and accessing intelligence information. Removable storage has

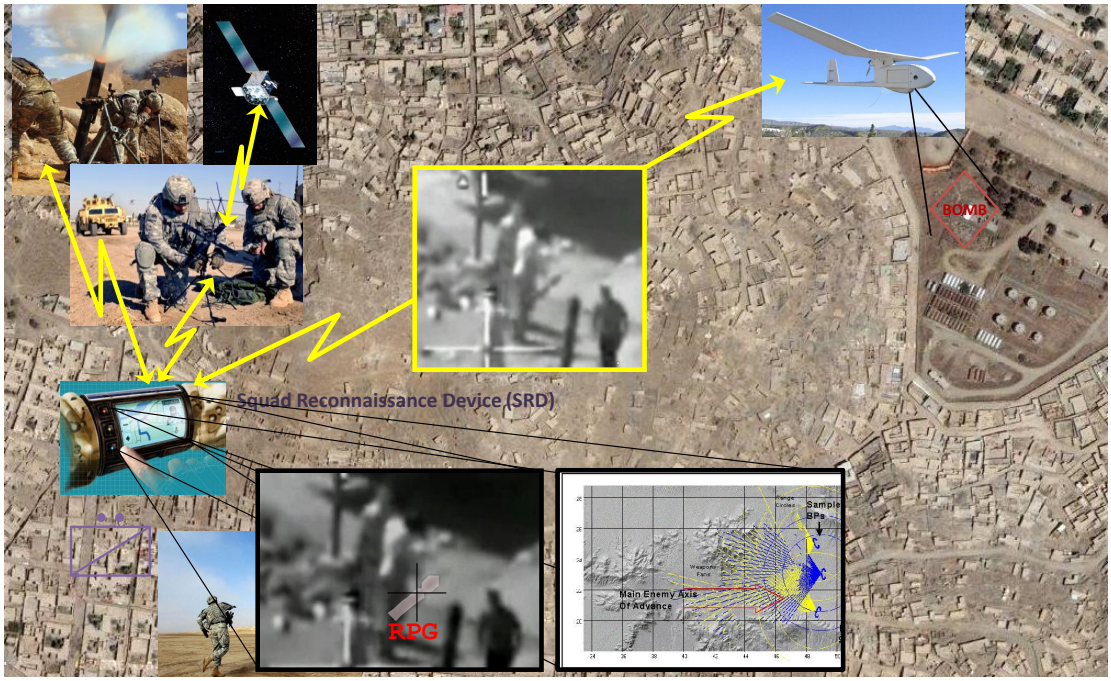


Figure A.1: System use-case, see Section A.2 for details.

been included for retaining information related to the mission, or uploading new maps and other databases. A GPS module is necessary for accurate situational awareness and reporting.

A multicore solution is important primarily due to the power constraint. A single core processor with the power and required throughput and multitasking capability will be extremely expensive in terms of power consumption if it could be built at all. An ASIC solution for many of the DSP tasks needed for our system will be power efficient, but would alleviate the flexibility we gain by implementing an array of processors suitable to more general applications. As an example, the SDR waveforms and video CODECs needed may be mission specific or experience multiple upgrades over the life of the system. Hardware implementation of these algorithms will likely make upgrading expensive or impossible. The Multicore SoC architecture is described in more detail in the next section.

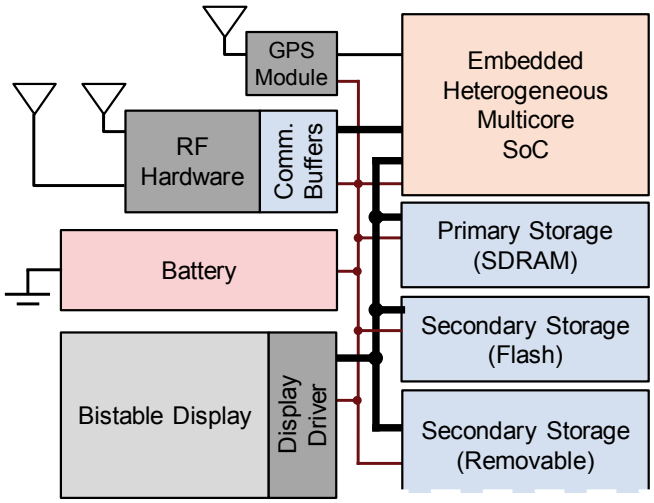


Figure A.2: System architectural overview.

A.3 System Baseline

The fundamental requirement for the proposed architecture is taken from the Joint Tactical Radio System (JTRS) Handheld Small Form Fit (HMS) requirement [65]. This system is a handheld tactical Software Defined Radio (SDR) under development by the JTRS Joint Project Office. A key design parameter is the requirement that the system be compatible with the standard radio battery currently available in the inventory [116]. The BA-5390 Li/MnO_2 is a current version. This battery provides 300 Wh energy capacity [26] which is used as a baseline for our design. The battery is currently larger than would be feasible for a wrist-mounted unit, but upcoming battery and power supply technologies promise to provide relief in this area [26]. The handheld radio system is required to run for 10 hours on one of these standard batteries.

Additionally, the radio is designed to transmit with a signal strength of between 0.0001 and 20 Watts. In this analysis it is assumed that the system will be capable of transmitting on up to four channels with varying power and duration requirements. Assuming an average continuous 20W radio transmission over 10 hours,

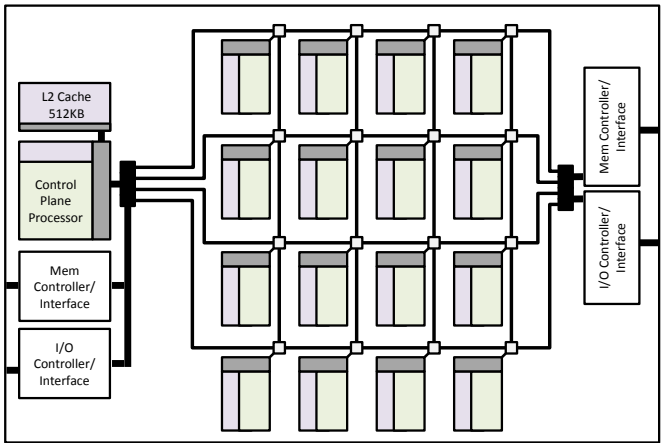


Figure A.3: CBE-like architecture: Distributed memory heterogeneous networked multicore.

the remaining power budget from the BA-5390's 300Wh is 100Wh, or 10W average continuous power.

The proposed system architecture is modeled on the heterogeneous multicore IBM Cell Broadband Engine. The CBE-like system architecture is shown in Figure A.3. Performance figures used in this case study are predominantly taken from the CBE itself. The use case and power constraints determine several factors in the system's design. The CBE's PPU serves as a control processor, and provides general purpose CPU functionality which is substantially backwards compatible with PowerPC software. The PPU also offers good performance in control intensive applications over its eight number crunching highly data parallel SPU accelerators.

A.4 Power Analysis

The current 45nm version of the CBE has seen power consumption improvements of 65% at 4GHz over the original 90nm design's power consumption commonly reported as $P_{CBE} \approx$ [60W,80W] [54]. The first generation IBM Cell BE SPU has also been reported to consume about $P_{SPU} \approx$ 5Wat 4GHz [120] [21]. Based on the reported 65% power improvement of 4th generation CBE in 45nm technology over the first generation 90nm technology [54], we can estimate power consumption in the latest gener-

ation 45nm CBE to be $P_{CBE} \approx$ [21W,28W]and the 45nm SPU power consumption $P_{SPU} \approx$ 1.75W.

In order to address the 10W power budget, we can first scale these power figures by throttling the system frequency to achieve a linear reduction in power performance. In the proposed system, the 4GHz system clock will be scaled back to 0.8GHz. The chosen frequency is based on iterative observation of its effect on the expected power performance and the total expected power consumption of the resulting multicore system with respect to the use-case loading and the 10W power budget.

We make the simplifying assumption that any potential power reduction from voltage scaling along with the reduced operating frequency will be offset by increased leakage power due to the increased chip area and number of transistors in the proposed system which is anticipated to include 16 accelerator cores, twice the number in the CBE. With the additional frequency scaling factor we find that

$$f_{new}/f_{old} = 0.8\text{GHz}/4\text{GHz} = 0.2 \tag{A.1}$$

$$P'_{CBE} \approx P_{CBE} \cdot 0.2 = [4.2\text{W},5.6\text{W}] \tag{A.2}$$

$$P'_{SPU} \approx P_{SPU} \cdot 0.2 = 0.35\text{W} \tag{A.3}$$

Few specifics are available about the power performance of the CBE's Element Interconnect Bus (EIB) and IO operations, but we can infer a typical number from the power performance of the total system and the total power consumption in the processing elements. We will assume that higher complexity dual thread PPU consumes power at twice the SPU's rate, and calculate the total power due to processing elements as $2P_{SPU} + 8P_{SPU} = 10 \cdot 0.35W = 3.5W$giving the range below for the EIB in the CBE's 9 core architecture:

$$P'_{EIB} \approx P'_{CBE} - 3.5\text{W} = [1.7\text{W},2.1\text{W}] \tag{A.4}$$

Table A.1: Frequency scaled CBE performance characteristics and Samsung DDR3 SDRAM specifications. Scaling factor CBE is 0.8GHz/4GHz=0.2.

Component	Performance at 4GHz (GFLOPS)	Scaled Perf. at 0.8GHz (GFLOPS)	Bandwidth at 4GHz (GBps)	Scaled Bandwidth at 0.8GHz (GBps)
SPU/MFC	32	6.4	25.6	5.12
Memory / I/O Access			89.6	17.92
Network on Chip			300	60
SDRAM 800MHz [135]			12	12 (not scaled)

Throughput and bandwidth figures are also scaled by the factor given by Equation A.1. The resulting values are given in Table A.1.

A.5 System Performance Requirements

(a) Control of a remote vehicle (UAV).

Low bandwidth communication, 128kbps using the Spektrum DSM2 commercial r/c aircraft controller as a model [145].

(b) Collect video feed from remote vehicle.

H.264 decoding requires an estimated 1.1GFLOPS for CIF 352x288 video [72]. We use use VGA 640x480 video giving reasonable size and resolution for a hand-held device. VGA has approximately three times the number of pixels or *area* as CIF, so we estimate that there is three times the amount of work required or 3.3GFLOPS to decode the VGA video. We consider a data rate with reasonable quality for the H.264 encoded video stream to be 0.13MBps (1Mbps).

Based on our parallelized H.264 decoder implemented on the CBE [14], 6 SPEs were required to decode a full HD (1920x1080) video at 30fps. The clock speed in that system is 3.2GHz which means the theoretical maximum computational throughput for the SPU in that system is

$$(3.2\text{MHz}/4\text{MHz}) \cdot 32\text{GFLOPS} = 25.6\text{GFLOPS}$$

The number of GFLOPS we expect to require for decoding full HD video is

$$\frac{1920 \times 1080}{352 \times 288} 1.1\text{GFLOPS} = 22.5\text{GFLOPS}$$

22.5GFLOPS across 7 cores in the existing implementation is 3.2GFLOPS per core or 12.5% of the 25.6GLOPS theoretical maximum for each SPU. This figure is suspiciously low, but it may not be unrealistic for a few reasons: a) the throughput is not expected to approach the theoretical maximum due to both the inability of the software to perfectly utilize the ALUs and particularly b) considering the control intensive nature of the H.264 CODEC, c) there are many synchronization pauses during macroblock decoding due to the nature of the interdependencies in the video CODEC and stream.

As indicated earlier, the VGA video theoretically requires 3.3GFLOPS to decode, and the frequency scaled SPU is theoretically capable of 6.4GFLOPS as given in Table A.1 Assuming the same ratio of throughput vs. the theoretical limit when decoding the VGA video in the frequency scaled system as with decoding full HD video in the real system, each frequency scaled SPU can handle 6.4GFLOPS $\cdot$ 0.12 = 0.8GFLOPS of the total video decoding requirement. From this figure, we estimate that 5 SPUs are required for decoding the VGA video at 30fps as indicated in Table A.2.

(c) Use image analysis to positively identify an object or individual from a reconnaissance video stream.

In this case an Automatic Target Recognition (ATR) algorithm is used to identify a specific object in the video feed. An example of an ATR algorithm implemented on the IBM Cell can analyze a 512x512 input image in 70ms using 6 SPUs and the PPU [31], approximately equivalent to 14fps for our VGA UAV video stream. After frequency scaling the performance in this case is scaled by the ratio of the frequency of the system used in [31] and our scaled frequency or

0.8GHz/3.2GHz = 0.25, and we expect the system using 7 cores to be capable of returning results for $0.25 \cdot 14$fps = 3.5fps.

(d) Use positioning information to locate self and others with integrated intelligence on the enemy from multiple sources including the remote vehicle.

Integrated GPS device provides self position information. We expect that one processing element will be required to collect and map friendly and enemy locations received the data network. Low bandwidth positioning information and map generation is required.

(e) Relay video reconnaissance information to higher echelon.

System will retransmit the encoded 0.13MBps VGA video feed from the UAV feed by re-encoding the unmodified incoming stream via the software defined radio described next.

(f) Software Defined Radio (SDR) Requirements.

The SDR hardware requirements are taken from Hasan et. al. [65] where the presented system is designed to handle the worst case performance requirements from among 5 JTRS waveforms without exceeding 50% of the system's computational capability for two channels simultaneously. In Table A.2 the required FLOPS are given as twice the requirement given for the system by Hasan because we expect to support four channels.

(g) Communicate via voice with other elements on the same voice network.

Low bandwidth voice communication included in SDR analysis.

(h) Encrypt/decrypt all incoming and outgoing data streams.

Encryption is included in SDR analysis. In reality this is most likely to be done with dedicated hardware for security reasons.

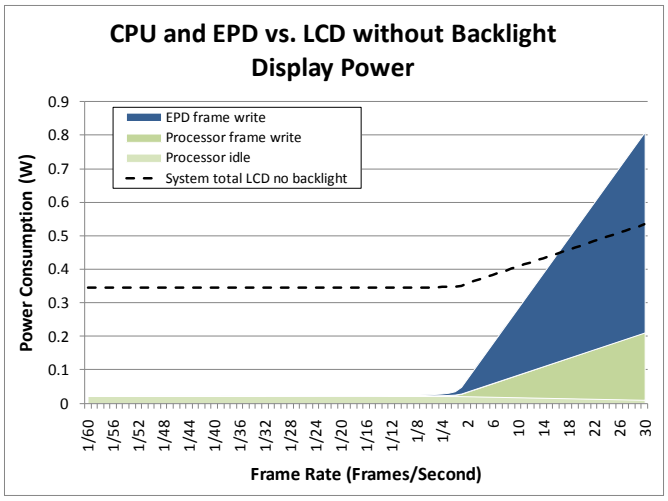


Figure A.4: EPD vs. LCD power performance without backlight.

(i) Display full frame rate video.

Expected power performance of the VGA Electrophoretic Display (EPD) has been computed based on the cost of switching the screen according to the model from [17]. The model does not account for the cost of the display driver circuitry which has been calculated by extrapolating from the power consumption measured on the Glencoe PXA270 Smartphone Development Board with a QVGA LCD display. The power consumption measured in the LCD driver is scaled from a drive voltage of 3.3V for the LCD to 30V for the EPD. The EPD display and driver are found to consume 0.6W at 30Hz. In Figure A.4, the main power components illustrated are processor power based on measurements of the Glencoe's PXA270, and the EPD power consumption for various video decoding frame rates. Processor idle power is negligible in this chart. The dashed line indicates power consumption in a comparable LCD display excluding the backlight which actually consumes dramatically more power than what is indicated for the LCD alone in the figure.

Table A.2: Estimated use case system requirements.

Requirement	Stream Bandwidth (MBps)	Computational Requirement (MFLOPS)	Processing Elements Required	RAM Reqt. (MB)	Approx. Avg. Power (W)
SoC Scheduler			1		0.35
a. Remote UAV Cont.	0.016		1		0.35
b. VGA Vid. Dec.	63[a]	3,300	5[b]	256[c]	1.75
c. ATR Algorithm	1.1[d]		7[e]	2563	2.45
h. SDR (4 Ch.)[f]		1.1 (MIPS)	2	120	0.7
g. Mapping/Intel			1		0.35
I/O and NoC					2.1[g]
Display					0.6[h]
GPS Receiver					0.165 [138]
SDRAM					1.5 [135]
FLASH Memory					0.3[17]
RF Hardware					20

[a] $0.13MBps$ incoming stream $+2 \cdot 0.46MBbps$ decoded raw image stream.

[b] From calculations in Section b.

[c] Based on existing CBE Implementation.

[d] ATR data rate calculated as SD Video data rate ($640B \cdot 480B \cdot 3.5fps$) for Y plane of decoded video only.

[e] Requirement is based on ATR implementation in [31].

[f] Figures based on proposed system in [65], Hasan's system supports two channels, so the figures are doubled to support four channels in our system.

[g] From Equation A.4. We are using the total communication power from the scaled CBE processor, which should be an upper-bound since the total bandwidth requirement in our system appears to be two orders of magnitude smaller than the CBE theoretical throughput numbers in Table A.1.

[h] Data from Figure A.4 assuming the system is displaying 30fps video.

A.6 Requirements Analysis

System performance requirements are consolidated in Table A.2 Our main goal is to determine the number of processing elements required in our system, and a system configuration which can handle the requirements of the use-case scenario. Due to the 10W power constraint, we are unable to simply design a hugely powerful system to accomplish all of the necessary tasks. In the first step we estimate the total number of processing cores which would be required. Based on the data and assumptions about the tasks, it appeared that the number is on the order of 10-20. The figures in the table are the result of an iterative analysis to determine the necessary clock frequency which will bring power consumption in the CBE-like SoC within our threshold while retaining sufficient computational power. An 800MHz clock was ultimately chosen which is comparable to that used in existing embedded microprocessors, such as Marvell's

Monahan processor used in Hasan's analysis [65], and the state of the art low power embedded processors used in most high end handheld devices such as smart phones. It also seems sensible to target 16 accelerator cores in the design as this number has advantages in terms of logic, analysis, and scalability. In the end, the 800MHz target operating frequency, and the workload partitioning described in Table A.2 indicates that 16 cores are required in the target architecture.

The off-chip system bus is not thoroughly analyzed here. The throughput numbers for memory access and on chip communication have been extrapolated from the CBE, first by scaling performance down with the clock frequency, then by assuming that it does not improve with the doubling of the number of accelerator cores and the regular NoC topology.

The system memory was chosen based on the power and performance of available technology. Memory bandwidth and power figures for Samsung's 1GB 800MHz DDR3 SDRAM are provided in Tables A.1 and A.2. We have not completed a thorough analysis to demonstrate that 1GB of memory is not excessive or under-sized. Based on the performance of existing CBE implementations which have 256MB main memory and 256MB video RAM, we have doubled that figure in our system to 1GB of SDRAM.

Power figures for the various components are presented with explanatory notes in Table A.2. Power figures are based either on commercial specifications for equivalent components or on the expected number of processing cores required to execute them on the system. Figure A.5 gives a breakdown of system power consumption by system component. The first bar shows system power consumption without optimizations, and using an LCD with backlight in place of the EPD display. The second bar shows the performance after replacing the LCD with an EPD. The third bar indicates possible power performance improvement due to the EPD smart driver concept presented in Chapter 2. Notice that here the system power consumption has dropped from 30.3W to 30.0W. The video decoding scheme presented in Chapter 3 can reduce SoC power

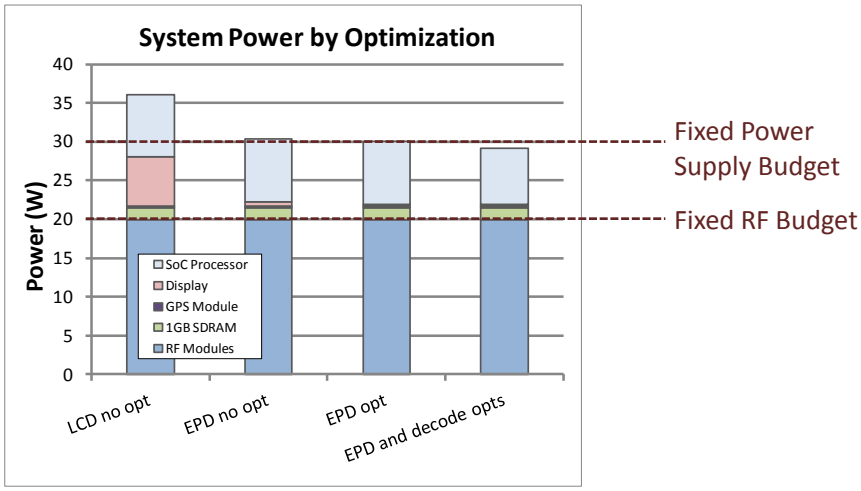


Figure A.5: System and component power performance by optimization.

consumption a further 0.8W bringing the total system power to 29.2W to achieve our baseline power budget requirement.

The improvements to system performance enabled by the other three contributions presented in this dissertation, the scalable parallel H.264 implementation in Chapter 4, the overlay generator in Chapter 5, and the lightweight runtime stream scheduler in Chapter 6 are not directly measurable here. We must consider the effect of necessarily running the H.264 video decoder on a single processing core which is impossible without the parallel scheme.

It is important to point out that there are currently no commercially available implementations of such a decoding scheme, which makes it a major contribution to the system in this analysis. If the video decoder were not parallelizable, we would need a much more powerful core dedicated to this specific task. Based on the analysis presented here, under the current CBE-like architecture a single accelerator core would need to be clocked at 5 times the speed used in our design, or 4GHz. This is 25% faster than the commercial system our design is based on, resulting in power numbers similar to those presented in our analysis in the previous section prior to scaling down the operating frequency, which clearly violates our power budget. Without an overlay

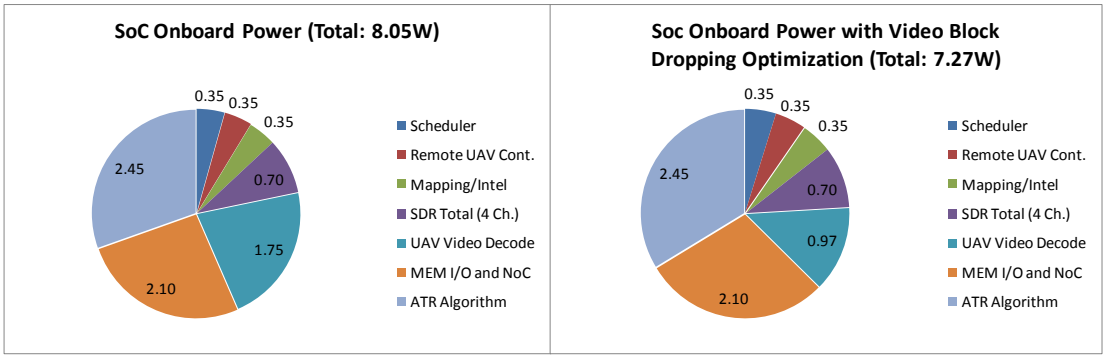


Figure A.6: SoC power breakdown by component/task. The chart on the left shows power consumption before implementing the H.264 decoder block dropping scheme, and the chart on the right shows performance with the scheme.

management technique like the one presented in Chapter 5 these figures will only further degrade, as the code overlay contributes up to 50% of the achievable frame-rate in the decoder implementation from Chapter 4. The runtime scheduler presented in Chapter 6 is not inherently required to the operation of this system, but without the runtime scheduler the system is essentially stuck with the presented software suite and processor allocations. Whereas it is imperative that the system be capable of adjusting to other scenarios and other use cases.

A.7 Conclusion

Automatically identifying and extracting SIMD opportunities from algorithms in order to improve utilization of wide issue architectures like the CBE's SPU continues to be a difficult problem. Our experience with the SIMDizing *xlc* compiler for CBE from IBM showed only very small gains where large gains should be possible when modifying the code by hand.

Memory bandwidth has traditionally been an issue which has been part of the impetus for moving to multicore architectures. Processors like the CBE seem capable of handling typical applications such as the example CBE implementations discussed in this analysis (H.264 decode, ATR) and the use-case scenario presented here. Future

systems may have many more cores, capable of handling many more concurrent applications, likely testing bandwidth limits. In addition chip to chip interconnects currently used between the processor and memory will always be slow and relatively power hungry. It seems desirable to include main memory or more on-chip memory in the SoC to address such issues. Die area is also a problem. Techniques such as 3D stacks can help, but they exacerbate thermal issues which essentially require better power management.

Scheduling and mapping tasks onto the available cores, considering limitations of the regular topology of the on chip network to avoid bottlenecks is also important. Such tools for dynamically managing a variable set of applications with sometimes widely variable resource needs are not currently available. The runtime stream scheduler goes part of the way toward addressing this problem.

The proposed system is not necessarily inherently well suited for applications with real-time deadlines and constraints. In particular, SDR and video decoding have real-time constraints. The scratchpad memories used in the CBE like accelerator cores helps to manage this problem by making memory accesses more predictable, but multiple applications sharing cores in the same network may make communication latencies less predictable as well.

APPENDIX B

THE CELL BROADBAND ENGINE

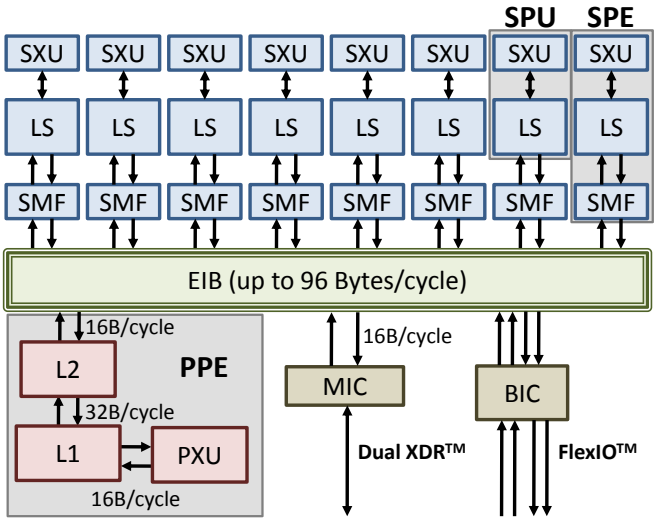


Figure B.1: IBM Cell Architecture [127]. PPE: Power Processor Element, PPU/PXU: Power Processor Unit, EIB: Element Interconnect Bus, SPU/E: Synergistic Processor Unit/Element, LS: Local Store, SMF: Synergistic Memory Flow Control, LS: Local Store, SXU: SPU Core, MIC: Memory Interface Controller, BIC: Bus Interface Controller.

The Cell Broadband Engine (CBE) is an innovative chip multiprocessor which strives to achieve "supercomputer power" in an "everyday" processor. An architectural diagram is provided in Fig. B.1. The CBE consists of a dual threaded general purpose control/OS processor built on the Power PC architecture. The Power Processing Unit (PPU) is connected to eight simpler PEs referred to as Synergistic Processing Units (SPUs) via the Element Interconnect Bus (EIB). The SPUs have extensive support for Single Instruction Multiple Data (SIMD) operations up to single precision floating point, and are able to access main memory directly through a Memory Flow Controller (MFC) which manages Direct Memory Access (DMA) activity in the SPU. Each SPU includes a 128x128 bit register file and a scratchpad memory Local Store (LS) of 256KB which is shared between data and instructions [76].

As is the case with multicore and distributed memory architectures in general, efficiently implementing parallel applications on the CBE requires good knowledge of the architecture and its associated development tools. IBM has provided very good

documentation and compiler tools for use by software developers [77] [130], but programmers targeting such an architecture face several specific challenges.

An important decision made by the CBE's architects during the design process was to reduce the complexity of both the PPU and SPUs by eliminating complex hardware algorithms associated with speculative instruction execution and branch prediction. Additionally, as the name suggests, the scratchpad memory in each SPU local store does not offer any cache replacement functionality. As a consequence, the runtime contents of the LS must be carefully managed by the programmer or the compile-time framework [127]. Compared to traditional cache based programming models where the programmer need not be concerned with memory allocation or data placement, this requirement places a significant burden on the programmer or compiler.

A similarly compounding addition to the programmer's workload is the need to partition the program into appropriate subcomponents in a load balanced manner across the available processing elements. Such a partitioning requires accurate knowledge about the execution times of various code segments as well as a high level of understanding of the implemented problem. The programmer or compilation framework must identify sequential and parallel portions of code which can be translated into functional and data level partitioning work partitioning and efficient mapping of those components onto the available SPUs.

APPENDIX C

H.264 ENCODER PARALLELIZATION ANALYSIS

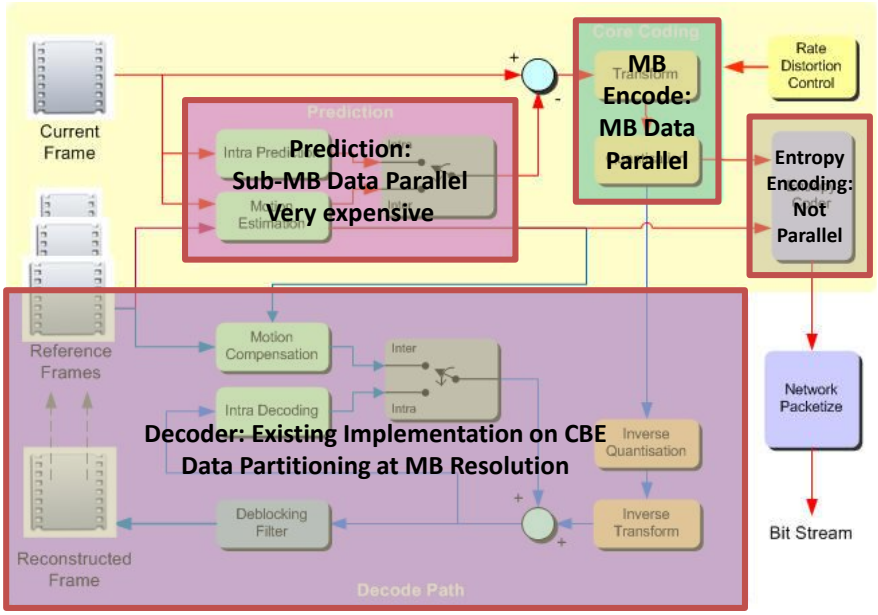


Figure C.1: The H.264 encoder [59].

C.1 Background

In order to analyze techniques for developing a parallel implementation of the H.264 encoder, we first need to look at its components. Figure C.1 illustrates the components and data path in the encoder. Beginning with a series of input images or frames, the prediction sub-path executes a search from among the possible modes and sources of prediction data for each 16x16 pixel macroblock (MB) in the incoming video frame. Essentially the goal of the prediction module is to find the best match for the present MB from among all of the available (previously handled) image data. Next the difference is taken between the "predicted" MB and the actual MB to be reconstructed later. The smaller the difference, or *residual*, between the two the less information needs to be transmitted in the encoded stream and the more efficient the image compression. These residual values for each MB are transformed from the spatial to the frequency domain and then each data point is quantized discarding some information in the process.

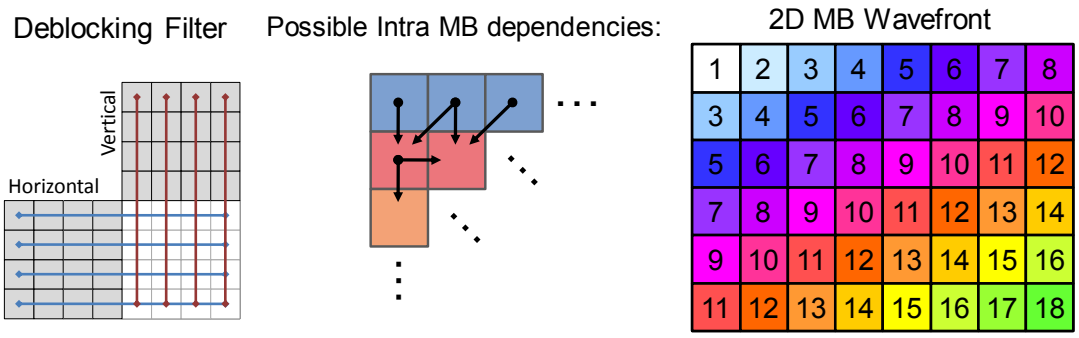


Figure C.2: Left, MB deblocking dependencies; center, Intra MB prediction dependencies; and right, 2D wavefront, like colored MBs can be decoded independently from one another.

Since the MB encoding process is lossy, it is necessary to implement the decoding process as part of the encoder shown as the decode path in Figure C.1. This is done to ensure that the reference information used by the actual video decoder is the same reference information used to make prediction decisions in the encoder. We have previously implemented an H.264 decoder on the IBM Cell [14]. Our decoder implementation is based on the 2D wavefront [151] where each processing core is assigned a row of MBs at a time, and synchronization between the processing elements accounts for intra and deblocking dependencies. The basic scheme is illustrated in Figure C.2 and Figure C.3 and is fully applicable to the decoding path in Figure C.1.

The two main considerations in developing a *scalable* encoder implementation for the MPSoC are, a) how will the work be partitioned so that work is easily reallocated for different numbers of available cores, and b) does the work partitioning maximize utilization of the available cores so that their workloads are well balanced. In order to achieve acceptable load balancing and scalability, the focus here is on a data partitioning approach rather than functional partitioning. The primary reason for this is the limited realizable scalability generally available in functional partitioning. From experience we know that assigning a wide variety of tasks requiring a large code base in order to achieve well balanced data partitioning is feasible based on our previous H.264

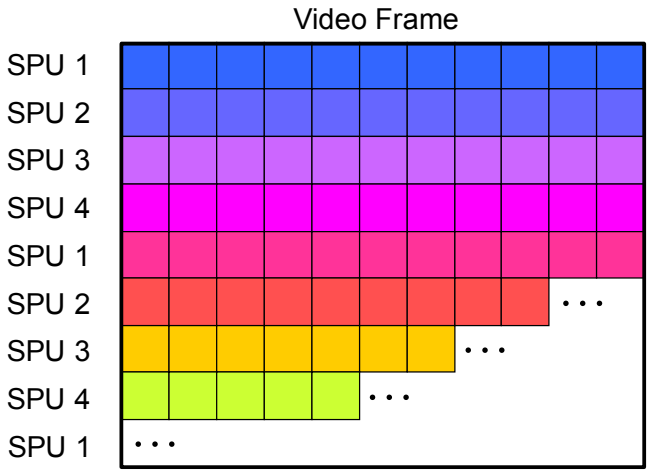


Figure C.3: Data partitioning in our scalable parallel H.264 decoder implementation. The SPU is the accelerator core from the IBM Cell Broadband Engine.

implementation. Efficient code overlay enables effective code management despite the limited SPU scratchpad memory.

We have an idea about how to parallelize the decoder portion of the encoder diagram. We would like to know how to parallelize the remaining portions of the encoder. As is the case with entropy decoding, entropy encoding is not parallelizable below frame level. For this reason, it may be assumed best left to a dedicated processor. This block must achieve throughput at least as great as the remainder of the system, otherwise it becomes the performance bottleneck. The "core coding" transform and quantization blocks are not easily partitioned across multiple cores, as in the decoder, since they also operate at the MB level. The prediction block on the other hand is highly parallelizable. The prediction algorithms are essentially search engines, testing some set of the possible prediction modes and motion vectors in order to find the best match for the current block. There are no dependencies within the search space meaning that it can be easily partitioned across multiple threads once the search space information is available. The prediction algorithms are also very time consuming given that they are searching for a match for a 16x16 matrix of pixels in an area of 80x80 or more pixels in one or more reference frames, while at the same time interpolating the reference data

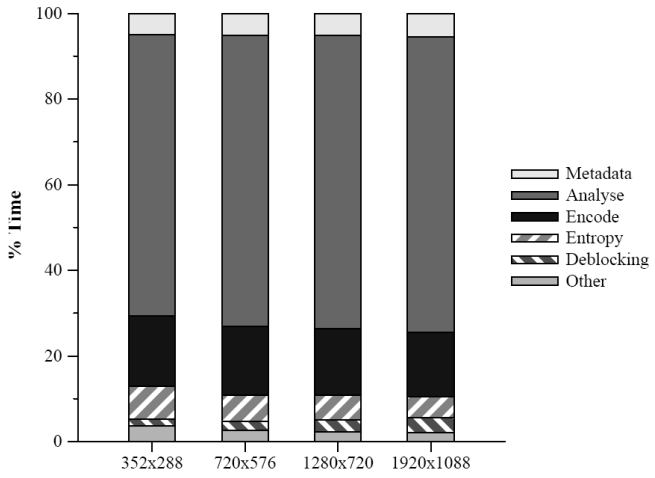


Figure C.4: Encoder work breakdown as presented by Alvanos et al. [4].

points down to the quarter pixel, effectively increasing the motion vector search space by a factor of 16 over the area mentioned above.

A drawback to the ease with which the prediction algorithms can be multi-threaded using data partitioning is the fact that the MB under consideration faces precisely the same inter and intra dependencies as those faced in the decoder. In fact the encoder faces tighter restrictions because in general it can examine *all intra modes* for *all MBs* in the stream. For this reason, existing parallel encoder implementations operate at the frame level, which tend to be unsuitable for embedded applications due to enormous latencies and memory requirements.

The predictor cannot check the quality of the intra prediction modes until the pixels from which the MB might be intra predicted are available. For that to happen, they must have already been encoded and decoded. This indicates a tight coupling between the encoder and decoder suggesting that they cannot be separated as with encoding a full frame using many threads, then switching over and decoding the frame with the same threads. It also suggests that using half of the available threads for encoding and the other half for decoding will result in undesirable synchronization and communication complexity and overhead.

Based on the above analysis, it appears that in order to achieve scalability and avoid unwieldy synchronization problems, a dedicated thread or set of threads should be involved in the encoding and decoding of each MB. The fact that intra dependencies in the encoder are the same or more restrictive leads us to require the 2D wavefront based parallelization scheme as in the decoder. The question remains with regard to whether and how to partition the prediction work. Alvanos [4] has produced the data in Figure C.4 based on the performance of the open source x264 encoder from VideoLAN [154]. The bars in the figure show the relative time needed to execute various components in the encoder. If we ignore the entropy encoding segment, assuming that it will run on a dedicated core, we note from the figure that the prediction algorithms consume approximately 70% of the remaining runtime. This figure suggests that encoding an MB can be sped up by something approaching that amount if we implement the prediction algorithms across multiple threads for each MB.

C.2 Sequential Workload Partitioning Analysis

We would like to know whether it is better to partition the work by dividing the image at MB row resolution or to partition the work by dividing the prediction work within each MB among multiple processors. The first step is to determine the theoretical speedup which is possible just from partitioning the image into MB rows and assigning a row to each processor as we have already done in our decoder implementation. By making the simplifying assumption that each MB requires a fixed amount of encoding time, we can estimate frame decoding times in terms of the number of MBs per frame, M, as in the analysis in [113]. Optimal frame decoding time is the ratio of MBs in the frame given by the frame's width, W, multiplied by its height, H, so that $W \times H = M$, and the number of processors used to decode the frame, N. Thus we have the minimum number time slots needed to decode the video frame is M/N.

Figure C.5, focusing on the "completed" blocks on the upper left corner of the

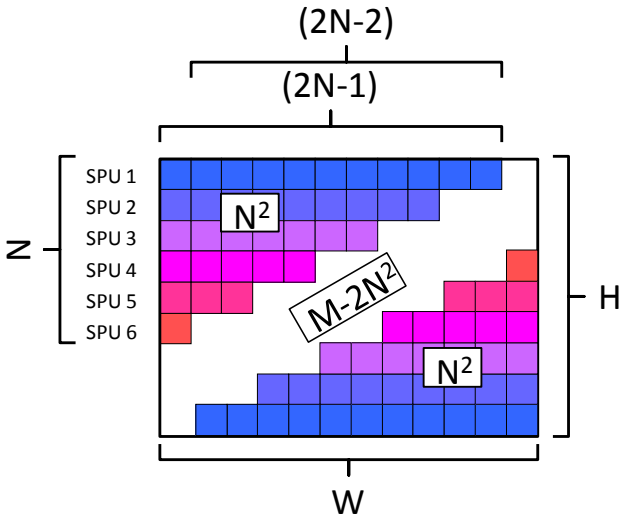


Figure C.5: 6 SPUs decoding a video frame. N: Number of processing elements. M: Total number of MBs per frame.

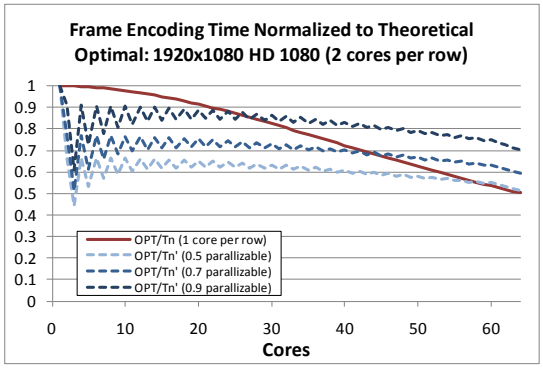
illustrated frame, shows the IBM Cell with 6 SPUs working on a video at the moment when the 6th SPU starts decoding. Up to this point, fewer than 6 SPUs have been working, after this point, all 6 SPUs are utilized until the last row is started, after which each SPU becomes idle again after finishing its row. This "ramp-up" work has taken $2N - 1$ time slots, where N is the number of processors. The number of MBs completed in the ramp-up phase is given by

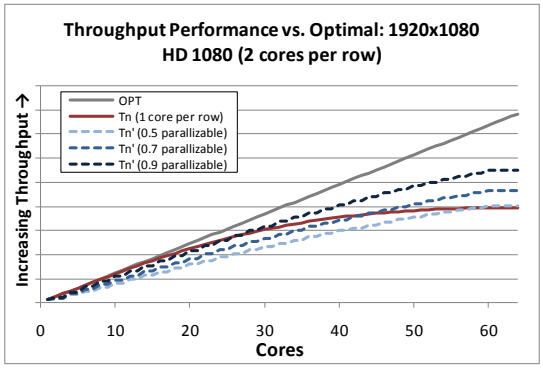
$$\frac{(2N-2)\cdot N}{2} + N = N^2 \tag{C.1}$$

Once the last row of MBs is started, this final N^2 MBs are again completed in $2N - 1$ time slots. The remaining $M - 2N^2$, MBs shown in white in the center of the frame in Figure C.5, are encoded optimally, unlike the ramp-up and ramp-down MBs, in $(M - 2N^2)/N$ timeslots. This gives us a total decoding time in MB timeslots, T_N, for a frame with M MBs encoding on N processing elements of

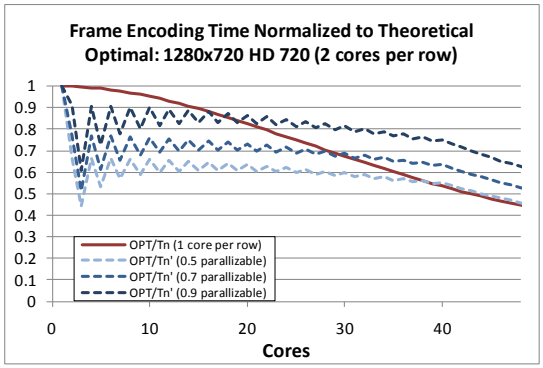
$$T_N = 2(2N - 1) + \frac{M-2N^2}{N} = 2N + \frac{M}{N} - 2 \tag{C.2}$$

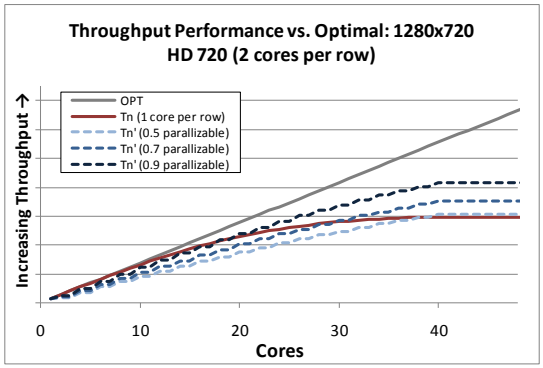
We would like to compare the performance of a system using one processor for each MB row to an implementation which assigns multiple processing elements to



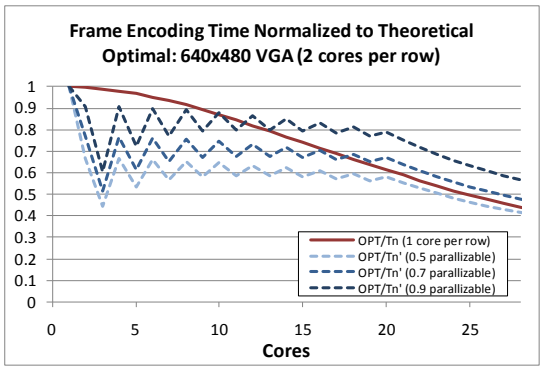


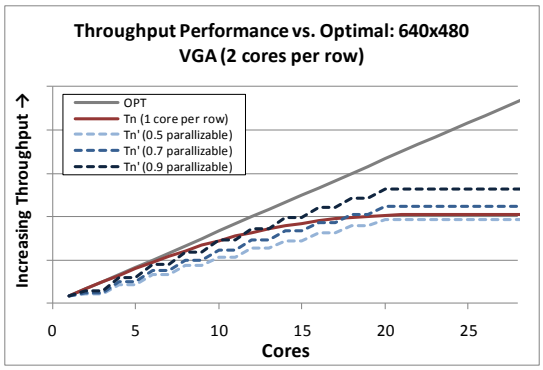
(a) Encoder performance across configurations encoding full HD 1080p.





(b) Encoder performance across configurations encoding HD 720p video.





(c) Encoder performance across configurations encoding VGA video.

Figure C.6: .

Comparative encoder throughput performance, using a single core for each row or multiple cores to reduce execution time of the sequential portion of MB encoder. Results are given for multiple video resolutions: (a) 1080p, (b) 720p, and (c) VGA.

each row in order to parallelize the prediction algorithms. To do this, we first choose the number of processors to be assigned to each MB row, N_{row}. This gives us a new notional value for N, $N' = \lfloor N/N_{row} \rfloor$, since assigning multiple processors to each row

means that fewer rows can be encoded at once. Replacing N with N' in Equation C.2 gives us the number of timeslots this implementation will take, prior to considering speedup from parallelization of the prediction algorithms. This Figure can then be scaled according to the expected parallelization speedup. The speedup is applied only to the parallelizable fraction of the encoder work, p. The scaled performance result is given in timeslots, $T_N^{'}$ as:

$$T_N^{'} = \frac{p(T_{N'})}{N'} + (1 - p)(T_{N'}) \tag{C.3}$$

By plotting T_N and $T_N^{'}$ over various configurations including number of cores used, video resolutions, and p values, we can get a feel for how to partition the encoding work. Results of this analysis are shown in Figure C.6.

C.3 Results

In all cases, throughput performance levels out once $N > \lceil (W + 1)/2 \rceil$. This occurs for N>61 in the case of full HD 1080p video where $W = 120$ as can be seen in Figure C.6a. For each figure, two charts are given. The left chart shows the ratio of the absolute theoretical optimal decoding time, M/N, to the frame decoding time for different encoder configurations. In each of these charts, the configurations are one processor assigned to each MB row, and two processors assigned to each MB row with three possible values of p from 0.6 to 0.9 based on the performance data previously presented in Figure C.4. The charts on the right show throughput values, $1/T$, for the same configurations as well as the theoretical optimum. Data for $P_{row} > 2$ are not presented here because they exhibited degrading performance as P_{row} grows larger.

C.4 Conclusion

It is observed from the graphs that assigning a single core per row is preferable to assigning multiple processors to each row for smaller numbers of available processors. For the smaller VGA video this number is around $N = 18$ whereas in HD 1080p it

occurs near $N = 44$ for $p = 0.7$ Smaller p values can significantly reduce performance for configurations with $N > 1$ due to reduced opportunity for parallelization in the prediction algorithms.