Compiler Optimizations for Multithreaded Multicore Network Processors

A Thesis
Presented to
The Academic Faculty

by

Xiaotong Zhuang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August 2006

Compiler Optimizations for Multithreaded Multicore Network Processors

Approved by:

Date Approved July. 6 2006

To myself, my wife and my parents.

Work hard, be good.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the tireless efforts of my advisor, Professor Santosh Pande. He has taught me how to work as an independent researcher and has been a steady source of wise discussions, cordial encouragement and support. He will be a great example throughout my professional life and I would like to express my sincere thanks to him.

I am also indebted to other members in my thesis committee including Professor Professor Rajiv Gupta, Professor Hsien-Hsin Lee, Professor Calton Pu, Professor Karsten Schwan for their comments and help to improve this dissertation.

I am extraordinarily grateful to all my friends and colleagues in the Systems Group and CERCS at College of Computing. This exceptional group of people created a collaborative environment that is friendly, well-organized, and stimulating. A number of colleagues and students, past and present, such as Ada Gavrilovska, Josh Fryman, Kenneth Mackenzie, Weidong Shi, have helped to setup the development environment for the IXP Network Processor. It is a great pleasure to thank them here for their commitment.

I also thank my many friends both in and out of Georgia Institute of Technology like Tao Zhang, Kun Zhang, ChokSheak Lau etc. they have made my time in graduate school a most enjoyable and memorable experience. I thank the lovely Ping Yu for her love, constant encouragement, limitless support, and patience with me while I completed this dissertation. I can no longer imagine life without her. Most of all I thank my parents for their love, sacrifice, and devotion. They have provided the foundation for all that I have accomplished and all that I ever will; I am forever in their debt. I dedicate this thesis to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

With advances in optical transmission technologies that offer high bandwidths and sustained growth of Internet traffic, network speed has reached a phenomenal level putting high pressure on network devices. Meanwhile, the ever-increasing requirements of network applications demand more functionalities to be fulfilled along with the packet transmission.

Network processors are new types of processors with multiple threads and multiple processor cores on the same chip. The multicore design offers enormous parallel processing power to handle high speed packet streams, which are mostly independent. To expedite packet processing and provide enough programmability, network processors are designed with special features like simplified ISA and pipeline, banked register file, special functional units etc. The multithreaded chip multiprocessor design together with other architectural peculiarities raise new challenges for compiler optimizations. Moreover, the unique demands of network applications, such as real time scheduling and packet scheduling, put a heavy burden on the programmers and compiler writers. In this work, we study several compiler optimization techniques on the popular Intel IXP network processor.

Due to very high clocking speeds, the memory gap on this network processor is huge, making registers extremely precious. Moreover, the register file is split into two banks, and for any ALU instruction, the two source operands must come from different banks. We present and compare three different approaches to do register allocation and bank assignment. We introduce the notion of register conflict graph (RCG), which captures the dual-bank constraint. Generally, the problem is to break odd cycles on the RCG with minimal performance loss. This work was built as a post-pass optimizer for Intel's IXP 1200 series.

We also address the problem of sharing registers across multiple threads in order to maximize the utilization of hardware resources. Context switches on the IXP network

processor only happen when long latency operations are encountered. As a result, context switches are highly frequent. Therefore, the designer of the IXP network processor decided to make context switches extremely lightweight, i.e. only the program counter(PC) is stored together with the context. Since registers are not saved and restored during context switches, it becomes difficult to share registers across threads. For a conventional processor, each thread can assume that it can use the entire register file, because registers are always part of the context. However, with lightweight context switch, each thread must take a separate piece of the register file, making register usage inefficient.

We found that even if registers are not protected across context switches, some registers can still be shared across threads as long as those registers are not used across context switch boundaries. This leads to big savings on the precious register resource, which in turn contributes to speedup as some memory accesses are converted into register references. In the development tools Intel used to ship, there was no support for this type of active register allocation. Therefore, we designed a compiler framework to systematically estimate the register requirements across threads and then balance register allocation by designating private and shared registers at different program points in a thread.

Furthermore, using compiler analysis to share registers across threads could be overly conservative due to the lack of runtime information. For example, some registers might be unused if context switches occur in a particular sequence. Thus, we propose to use compiler to provide information regarding which registers are dead at each context switch point. Then, the hardware dynamically converts register accesses to memory accesses.

Programs executing on network processors typically have runtime constraints. Scheduling of multiple threads sharing a CPU must be orchestrated by the OS and the hardware using certain sharing policies. Real time applications demand a real time aware OS kernel to meet their specified deadlines. However, due to stringent performance requirements on network processors, which process packets from very high speed network traffic, neither OS nor hardware mechanisms is typically feasible/available. Because network processors target very high-speed lines and pushing such solutions into the hardware would be simply unacceptable. On the other hand, the operating system poses a huge overhead, which is generally

not affordable on a network processor. In this research, we demonstrate that a compiler approach could achieve some of the OS scheduling and real time scheduling functionalities without introducing a hefty overhead.

CHAPTER I

INTRODUCTION

The speed of the network is increasing at a dramatic pace. With advances in optical transmission technologies and sustained growth of Internet traffic, network speed has reached a phenomenal level putting high pressure on network devices. Lots of research efforts have focused on supporting higher line speed and a larger number of line interfaces [53]. Recent industrial trend indicates that current high-speed routers can target OC-192 (10 Gb/s) or OC-768 (40 Gb/s) line rates. However, it is anticipated that the next generation routers would support OC-768 or even OC-3072 (160 Gb/s) line rates with hundreds of interfaces.

On the other hand, the ever-increasing requirements of network applications demand more functionalities to be fulfilled along with the packet transmission. The explosive development of the Internet has motivated packet processing at higher protocol layers. Typical service disciplines such as stateless and stateful classification of packets, QoS support, real-time constraints, traffic shaping, intrusion detection, security support, active networks, etc. put a heavy computation burden on network devices and add more complexity to their designs. Besides, due to the high variation of advanced packet processing and service provisioning, a highly flexible infrastructure is desirable that could easily adapt to and accommodate new implementations without reconstructing the entire system.

1.1 The Emergence of Network Processors

Network processors have historically evolved from routers and so we first take a look at their evolution. Traditionally, there are two ways to build routers, however both have their disadvantages in face of higher line speed and the need for programmability.

1.1.1 Routers built with GPP

Early routers were built on top of general-purpose processors with dedicated routing software to process and forward packets among network interfaces. Software-based routers are suitable for low-speed networks because the overhead of general-purpose processors severely limits the packet speed they can handle. For example, even at OC-192 line speed, we have to deal with a packet stream of up to 32M packet/s, or 32ns/packet. Software-based solution cannot keep pace with this line speed even for simple packet forwarding. The wide deployments of virtual private networks (VPNs) and secure IP (IPSec) leads to expensive encryption/decryption operations being performed on the packets during routing. [10] shows that on a real 600Mhz Alpha 21264 workstation dedicated for cipher encryption, the standard encryption algorithm 3DES can only deliver 7.32MB/s, not even enough to saturate a trailing edge 100Mbs Ethernet link. On the contrary, a hardware implementation can improve the throughput of 3DES by over 700%. Although relying on software implementation makes them highly programmable, (i.e. the routing software can be easily updated and enhanced), the tremendous amount of overhead sets a stringent performance and scalability limit, therefore restricts such routers to small scale periphery networks.

1.1.2 ASIC-based Routers

ASIC-based routers are built with ASIC chips manufactured by equipment vendors. These chips are hardwired embedded chips to maximize the performance for common packet processing modules. The standard architecture design principle of targeting the speedup of most common operations is widely applied to ASIC-based routers. Ethernet switches for layer-2 processing are deployed with little extra functionalities but can handle most local traffic with an astonishing speed. Later, layer-3 switch/routers were on the market and achieved a great success. However the high performance of such products is at the cost of sacrificing variability and advanced features. Most of them focus on the *de facto* IP protocol while largely ignoring other complicated routing algorithms and service disciplines as mentioned earlier. Secondly, ASIC-based routers have a long design cycle. These products have no programmability or little programmability. Every functional update that requires redesigning must take at least 12 months, significantly hindering the deployment cycle.

1.1.3 Network Processor and Its Market Growth

Network processors are new types of processors dedicated to network processing. In other words, they are Application Specific Instruction Processors (ASIP) geared towards both fast speed and flexibility. A network processor contains multiple packet processors and a number of special hardware units. In this manner, the speed of network processors can be comparable to their ASIC counterparts. At the same time, their programmability allows a wide range of applications being implemented.

Notice that it is almost impossible for software based packet processing to attain OC-192 or higher processing rates which is commonplace in core and edge networks. For network processors, OC-192 has been reached by most vendors. On the other hand, value-added services, such as QoS and VPN are desirable in enterprise networks, access points, etc. However these services are constantly changing; new features are frequently being added, therefore such services should be deployed with high flexibility. In short, network processors combine the benefits of programmability of software-based routers and the high performance of ASIC-based routers.

However, network processors are in stark contrast to either GPP or ASIC based routers, since the tradeoff between flexibility and speed still exists. Due to this reason, network processors are particularly designed in terms of their specialized hardware components, ISAs, programming interface and compiler support.

Despite of the downturn in network and communication sector, the network processor market is growing rapidly in recent years. A forecast made by technology analyst firm Semico Research Corp [20] (as shown in Figure 1) predicted that the total revenue of the network processor market will reach $ 600 million in 2007. Not only we observe major network processor vendors like AMCC, Intel, IBM, Motorola, Agere etc. fiercely competing for this fast growing market, there are also a number of companies such as Teja, Network Speed, FutureSoft, etc. embarking on the software development and system integration services for network processors. Network processors provide a broad range of functionalities and lay basis for a wide variety of applications to be implemented with strong hardware support.

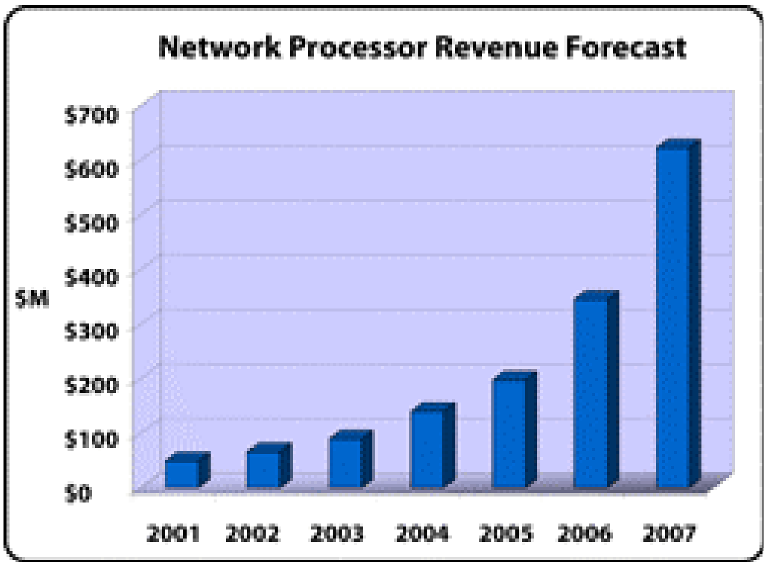


Figure 1: Network processor revenue trend

With many network processor vendors and products on the market, in this chapter, we intend to summarize their characteristics from several aspects. We will first address the application domain of network processors, then the architectural and compiler supports on typical network processors.

1.2 *Functionalities of Network Processors and Their Target Applications*

Network processors provide a broad range of functionalities and lay basis for a wide variety of network applications to be implemented with strong hardware support. With many network processor vendors and products on the market, in this section we intend to summarize their characteristics. We will also address the application domain of network processors.

1.2.1 Basic Packet Classification and Forwarding

Processing packets depending on their contents and forwarding them after classification are the basic functionalities that must be supported on network processors. Network processors are typically equipped with specialized hardware to accelerate such operations because they are highly optimized for these common cases [63, 52, 41]. For example, IP lookup engines, hash units can help IP packet forwarding. For network processors, ample flexibility is available to process packets belonging to a wide range of protocols. Also packet classification and forwarding get more complicated as higher layers of the protocol stack are concerned.

1.2.2 Traffic Management

Traffic management involves routing packets following certain rules or algorithms. Packet transmission can be orchestrated by their QoS requirements, packet scheduling algorithm, or real-time constraints [15] amongst other things. For example, packets can be first categorized into flows. Different flows may have their distinct QoS properties. It is likely that multimedia traffic such as voice and video will constitute an increasingly larger portion of the future Internet traffic. Unlike normal data packets, voice and video are quite different in that loss of a limited number of packets can be tolerated. These requirements can be translated into QoS specifications for each flow and handled accordingly in network processors. Packet scheduling based on scheduling algorithms like Priority Class based approaches, First Come First Serve, Fair Queueing/Weighted Fair Queueing (FQ/WFQ) have been extensively studied in network domain. The goals of packet scheduling include sharing the bandwidth properly, enforcing a predetermined policy etc. Normally, packets are assigned priorities after being processed. At the output port, the outgoing packets are ordered according to their priorities. Packet transmission will follow that order. Moreover, routers can be thought as one type of real-time systems that process packets in real-time. Real-time constraints are tightly tied to QoS specification for packet flows. Packets are distinguished and categorized according to their real-time properties instead of being transmitted with best effort. In this manner, more important (urgent) packets are handled more promptly. Typical real-time scheduling algorithms are Rate Monotonic (RM), Earliest Deadline First (EDF), Minimal Laxity First (MLF), Dynamic Window-Constrained Scheduling (DWCS), etc. All of these applications involve eliciting information (such as priorities) from the packets, classifying them, queuing them and finally outputting them at the egress points.

1.2.3 Advanced Features

Network processors can be easily reprogrammed, which allows almost all kinds of applications to be implemented as long as the code can fit into the processor code store. There are a variety of applications that can be ported to network processors and actually some of them are a part of active industrial efforts. For example, network security systems have attracted lots of attentions in recent years due to the widespread of computer viruses and network intrusion activities. Encryption/decryption algorithms are computational intensive, but can be more efficiently performed on network processor with certain amount of specialized hardware. Other examples are pattern-matching virus signatures, content inspection [67], etc. Attempting such processing at packet level is more beneficial since the system has no chance of getting infected; moreover critical information about sources of attack is only possible at packet level. Techniques such as IPTraceBack are very useful to trace the route back; one cannot simply rely on information in packet headers since addresses are often spoofed. Network processors extend the packet processing towards higher-layer protocols, where the ability to capture and analyze payload at layer 4 or higher is desirable. For instance, the so-called "web server switches" can utilize higher-layer information to direct packet streams to a number of web servers in order to achieve load balancing or early determination of required service. However, as we move up the protocol stack, a higher degree of variety is encountered compared with relatively simple and stable protocols found in layer 2 and layer 3. The processing must be catered towards application needs, which might be highly custom-oriented and volatile. To some extent, network processors fit such needs very well.

More generally, network processors offer system developers an alternative means to construct part of the software that originally reside on general-purpose processors but with a tight liaison to the communication networks. For example, middleware for distributed network systems can be partly implemented on network processors to improve their efficiency. Moreover, access control software, firewalls and VPN are possible candidates for a network processor centralized infrastructure. The above application scenarios indicate special needs for a special type of processing fulfilled by programmable network processors. Next, we discuss the architecture, which caters to such needs.

1.3 Architecture Overview

Achieving high processing speed as well as providing enough programmability requires careful architectural design. The architecture of network processors considers many special properties for packet processing and takes advantage of them.

1.3.1 Exploiting Parallelism Through Multitheaded Multicore Design

Obviously, under high packet rate and even a moderate amount of processing workload for each packet, the total amount of computation power substantially exceeds the capability of a single processor. Due to the strong independence among network packets, exploiting parallelism at packet level is a natural way to improve the overall throughput. Thus, network processors typically incorporate multiple processor cores and multiple threads to handle independent packets concurrently. For example, the Intel IXP 2800 processor contains 16 packet processing engines and 8 threads running on each engine; IBM PowerNP4GS3 has 8 programmable units and 4 threads on each unit. It is highly likely that future industrial products might assemble more packet processors and threads as this trend has been observed from recent generations of network processor products (e.g. IXP 1200 only has 6 packet processors). Meanwhile, with multiple threads on the same processor running similar code, their code can be shared across threads on the same engine, reducing total code size.

1.3.2 Concise ISA and Simplified Pipeline on Packet Processing Engines

Typically, multiple packet processing engines are placed on the same die to reduce the communication overhead. Such chip multi-processor architecture takes advantage of higher level of integration provided by the modern IC technology. However, the pipeline of each packet processing engine is actually quite simple. Normally a concise RISC instruction set is adopted. In this way, the underlying hardware can be made very simple, greatly increasing the operating frequency. Besides, due to the number of packet processing engines, the complexity of each individual engine is kept small to reduce the overall hardware cost. New generations of general purpose processors spend a lot of architectural resources to increase the instruction level parallelism. On the contrary, packet processing has already enjoyed sufficient amount of parallelism in the packet stream. Therefore, their design philosophy focuses on increasing the number of processors to exploit packet level parallelism instead of complicating each processor to get more parallelism from the processing of one packet. As an example, the packet processing engines on the Intel IXP processor are all in-order issued. For IXP2800, only 6 stages are in the pipeline. Note that, although this design philosophy renders it possible to pack a higher number of packet processors on one chip, the simplified pipeline also lowers the performance and lengthens the processing time for each packet. Meanwhile, although multiple threads can be deployed to make better use of each processor, multi-threading support is typically very weak. For example, we have not seen simultaneous multi-threading being implemented on any network processors. Context switch is very simple and fast and inter-thread management is left to the user.

1.3.3 Special Units

Typically network processors have specialized ASIC units to speedup common packet processing operations. In other words, network processors are domain specific processors, therefore many common, non-programmable operations can be extracted and can benefit from specialized hardware units. For example, the hash units and IP lookup engine are helpful for packet processing; also encryption/decryption engines are included to boost security oriented applications; other examples are the pattern-processing engine (PPE) and Checksum/CRC engine on Agere PayloadPlus, Queue Management Unit and Table Lookup Units on Motorola C-5e.

Some network processors even include powerful co-processors to assist with packet processors, such as the Routing Switch Processor (RSP), Fast Pattern Processor (FPP) on Agere PayloadPlus; DPPU co-processor on IBM PowerNP; Fabric Processor (FP) on Motorola C-5e etc.

1.3.4 Other Peculiarities

Network processors typically have many specialties in their ISA and architectural designs, which offer plenty of research topics especially for compiler design and optimization. Since this research is primarily based on the Intel IXP processor family, we would like to enumerate a few of them as follows.

The ISA contains special instructions such as the ones for hash units, scratchpad operations, controlling transmission state machines that are specially designed for IXP. Most ALU instructions can be completed in 1 cycle, while long latency instructions could trigger context switches to hide the latency.

The register file consists of general purpose registers, transfer registers for SRAM and transfer registers for DRAM. The general purpose registers are split into two banks. The two source operands of an ALU instruction must come from different banks. Transfer registers are for accessing SRAM or DRAM, such that the processor can continue processing with general purpose registers while memory accesses read/write transfer registers. Each transfer register number actually refers to two registers, one read transfer register and one write transfer register. Access to SDRAM memory is restricted to 8-byte boundaries and access to SRAM is restricted to 4-byte boundaries. Each thread can access all registers on the processor, however part of the register file can be accessed faster and rest of them must be accessed with indirect addressing mode.

Context switch is lightweight and non-preemptive. During context switches, nothing except PC is saved. In this manner, each context switch only takes 1 cycle. A thread gives up the processor only through explicit instructions.

1.4 Compiler Support

Since network processors are designed to allow the execution of user programs, a development tool set must be provided for programmers. Although programs running on network processors cannot be as large as the ones for general purpose processors, sufficient compiler supports are still very important because: 1) the code size on network processors may become too large to be hand-optimizable. For example, IBM PowerNP4GS3 has 128KB code store, Intel IXP 2800 has 4K instructions (40 bit per instruction) and Motorola C-5e has 8KB code store on each processor. Thus, programming network processors with assembly can be very time-consuming and error-prone; 2) as mentioned earlier, the architectures of network processors are quite distinct including many special features. These features even differ significantly from product to product. Therefore it becomes very difficult to ask the programmer to explore the best code for a particular architectural feature. As will be addressed in later sections, our experience on the Intel IXP network processor shows that introducing compiler optimizations for some architecture features generates very impressive performance results.

Noticeably, several network processors have provided their developing tool suites, which support programming with high level languages. For instance, Intel IXP processor can be programmed with a subset of ANSI-C and the C code is then compiled with their Micro-C compiler. The company is also developing a more advanced compiler currently. Motorola C-5e is shipped with a full suite of software developing tools including a GNU-based C compiler and debugger, simulator etc. The Agere PayloadPlus can be programmed with high-level domain-specific language. On the other hand, some products like Cisco Toaster 2 and IBM Power NP lack a program interface of high level language. Interestingly, as surveyed in [71] popular network processor products on the market tend to have good programming tool support. They found the 3 products mentioned above by Intel, Motorola and Agere have gained more market shares recently.

1.5 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes the details of the architecture of our target system and compiler challenges; Chapter 3 presents compiler optimizations for dual-bank register allocation; Chapter 4 is about how to sharing registers across multiple threads; Chapter 5 extends it to a dynamic approach based on both compiler and architectural support; Chapter 6 shows a novel way of managing runtime constraints with compiler techniques; Finally, Chapter 7 and Chapter 8 are for related work and conclusion.

CHAPTER II

COMPILER OPTIMIZATION FOR NETWORK PROCESSORS

This chapter will provide an introduction to the architecture of the network processor platform used in this work. We will also demonstrate the challenges of improving the performance of this network processor.

2.1 Introduction to the Intel IXP network Processor

In this research, we target a highly flexible and very popular network processor family-Intel's IXP. The IXP network processor family is designed to help meet the requirements of network products, ranging from cost-effective entry-level devices to high-performance solutions. It integrates a number of high-speed processor cores with programmable multithreaded packet processing processors called microengines, expanded instruction stores, and enhanced functionality, which enable manufacturers to flexibly meet a variety of requirements including faster line speeds, multi-protocol support, enhanced feature sets, data handling reliability, and lower cost.

There are three product lines for the IXP network processor that are being manufactured–Intel IXP4xx, IXP12xx [34, 35] and IXP2xxx [36, 37]. The IXP4xx products aim for low-end applications, such as small-to-medium enterprises, and networked embedded systems. IXP12xx series target wire-speed processing for OC-3 to OC-12 multiservice network applications. The newest ones are the IXP2xxx network processors, which are the most versatile and powerful, designed for network access, edge, and core applications from T1/E1 to OC-192 (10Gb/s).

Most of our work was done based on the IXP12xx network processor series, since it was introduced and donated to Georgia Tech during the time we started our research. As a matter of fact, later products–the IXP2xxx series–share a lot of similarities with

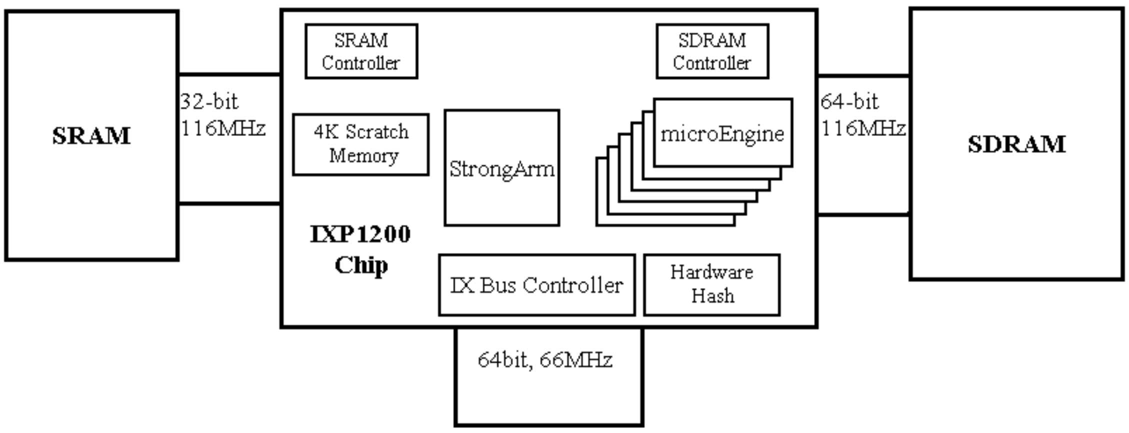


Figure 2: IXP 1200 Network processor diagram

the IXP12xx processors. Next, we will describe the architectural design of the IXP1200 network processor and briefly about the IXP28xx network processor, which are the two representative processors of the product series.

The growing demand for higher-performance network equipments raises the performance bar for silicon components. The IXP1200 network processor is a key component of Intel IXA (Intel eXchange Architecture) and is specifically designed for network control tasks, including Layer 3 processing of packets (or ATM cells) in real time. IXP1200 network processors are designed for wire-speed deep packet inspection and forwarding, while supporting multiple protocols required by todays networks.

Figure 2 shows the block diagram of the IXP1200 network processor. The processor consists of a StrongArm core, which functions as a traditional microprocessor. Connected to this core are six RISC *Microengines* or *Processing Units (PUs)*, which are responsible for managing the network data. These microengines are four-way hardware threaded yielding 24 system threads per IXP 1200. Both the Strong Arm core and the six microengines run at the same clock rate (232 MHz). Furthermore, all the microengines are connected to a high-speed bus (IX bus, 80 MHz) for receiving and transmitting packet data. On the other side of the IX bus are external MAC devices (media access controllers). The MAC devices can be either multi-port 10/100BT Ethernet MAC, Gigabit Ethernet MAC, or ATM MAC.

We have two types of IXP boards. One type of board (evaluation board) has two multi-port Ethernet MAC devices connecting to the IX bus, one supporting eight 10/100BT ports and the other supporting two Gigabit ports. The evaluation board has 4MB SRAM (32-bit bus) and 32MB SDRAM (64-bit bus). The other type of board (Radisys board) has much more memory space than the evaluation board (8MB SRAM and 256MB SDRAM), but supports only four 10/100BT ports. Both the StrongArm core and the microengines can access all the address space of the SRAM and the SDRAM. As performance requirements increase, meeting the power budget for a product design also becomes a significant challenge. IXP1200 network processors typically dissipate 5 watts of power or less.

Briefly, the IXP architecture has the following properties.

1. Highly parallelizable, totally 24 threads. Context switch is extremely lightweight (zero overhead). In fact, only the program counter (PC) is saved and restored.

2. Layered storage distribution. There is a 4 KB micro-code space for each microengine. Code are stored in on-chip memory. 128 GPR (general purpose register) and 32 Read/Write transmission registers are present for each microengine. For parallel fetching, the registers are separated into two banks. Multiple shared memory resources in the form of on-chip Scratchpad (4KB), SRAM, and SDRAM.

3. Compact instruction set, which includes about 50 miscellaneous instructions. All the ALU instructions can be finished in one cycle, this contains only plus, minus, boolean operations with or without shift. Other listed instructions are branch, memory read/write, field operation, hash table operations, context switch, etc. However, no integer division or multiplication instruction is supported. There is also no FPU.

4. Miscellaneous hardware accelerated functionalities such as SRAM lock, hardware 48bit hash, and hardware memory freelist. As we have mentioned earlier, the on-chip memory on IXP is quite limited, however, the off-chip memory is spacious which can be fetched with very long delay.

Figure 3 shows the data path in one of the microengines. We can see there are three

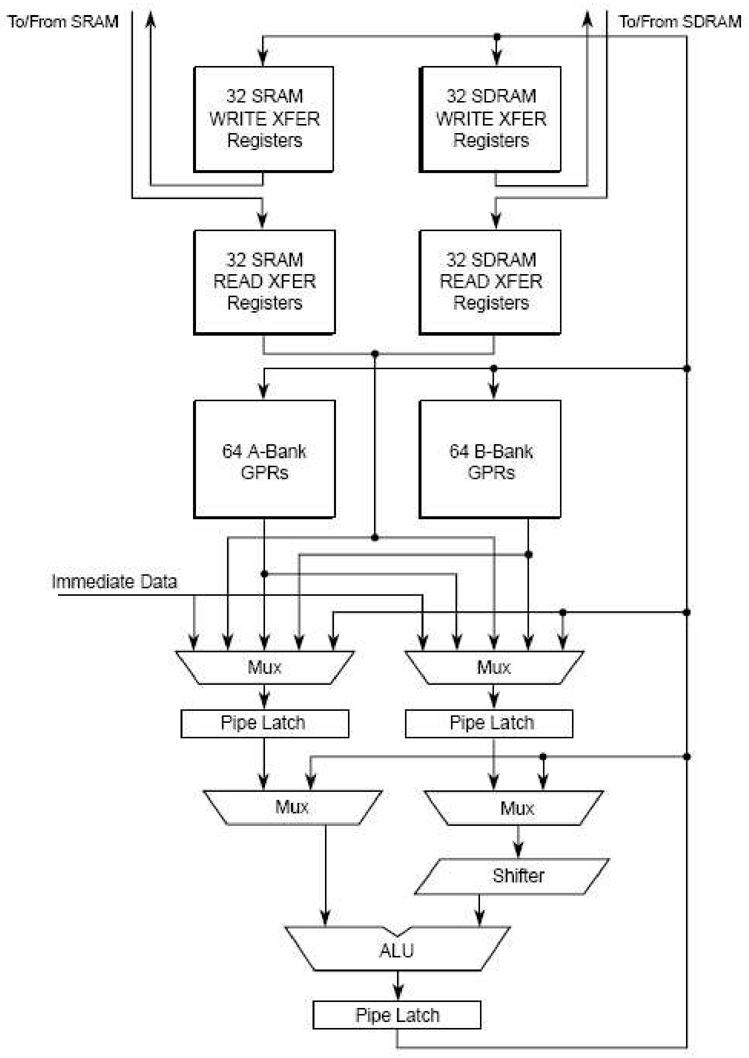


Figure 3: Data path in a micro-engine

types of registers in the register file–general purpose registers (GPRs), SRAM transfer registers and SDRAM transfer registers. A total of 256 General purpose registers (GPRs) are physically split into two banks–bank A and bank B. The ALU unit inside the processor core has two input ports. At any time, either bank A or bank B is connected to one of the two input ports. On the other hand, the output from the ALU unit is accessible from both banks. This design can potentially support a large number of GPRs and can control the cirtical path delay due to the increased register file connections. To shorten the execution time for ALU instructions, operands are fetched in parallel. Due to the above design of data paths coupled with the parallel fetch of operands, restrictions are imposed on operand

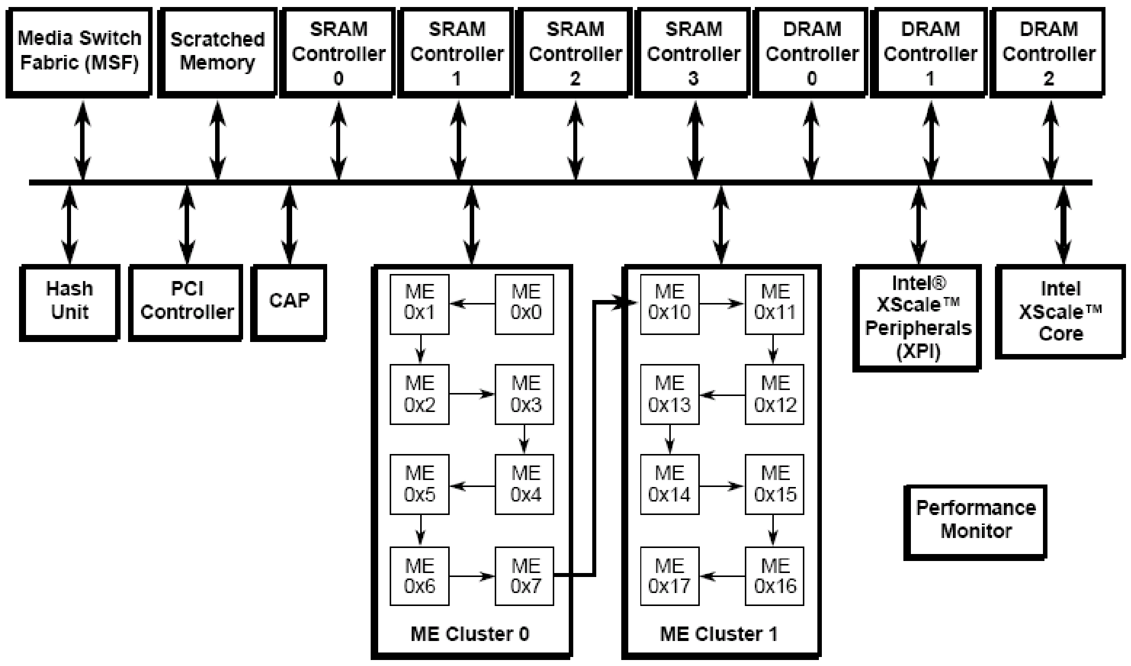


Figure 4: IXP 2800 Network processor diagram

register residency. Each ALU instruction can have two register-resident source operands which must come from different register banks. For instance, an instruction x=y+z requires registers y and z to be in separate banks. However, the destination register x can be in either bank A or bank B.

Both SRAM and SDRAM registers are split into read and write registers. To store a data to SRAM or SDRAM, the data must go through the write registers. Similarly, to read a data from SRAM or SDRAM, the data must go through read registers. The total number of transfer registers is 128, which is equally divided into SRAM read, SRAM write, SDRAM read and SDRAM write registers.

IXP2xxx series is a recent member of the IXP family. More functionalities are added including more threads and processor cores. Figure 4 shows the high level diagram of the IXP2800 processor. The processor can reach 1.4 GHZ clock rate and can process 28 million OC-192 packets per second over SONET. The RISC architecture allows a very concise instruction set and all ALU instructions (including plus, minus, shift, XOR, AND etc) take 1 cycle. The total number of microengines is increased to 16, organized into 4 clusters, and each microengine now contains 8 threads. Code on each microengine can take 4K instructions, and each instruction is lengthened from 32 bits to 40 bits. The total number of GPRs on each microengine is doubled to 256 (128 per bank), however the number of registers per thread is still 32. On the other hand, the number of transfer registers is quadrupled (to 512), perhaps because the memory/processor speedup is more phenomenal. Moreover, a special type of register called next neighbor register is introduced to facilitate inter-processor communication. Essentially, all microengines are sequentially numbered. Data written into a next neighbor register can be read by its next neighbor microengine.

The network processor provides more heterogeneous memory components with different capacities and latencies. Each microengine has its local memory, which is only around 2KB. Also, the scratchpad, SRAM and SDRAM are available with increasing latencies. Each bit of the scratchpad can be operated individually; SRAM memory cell can be locked/unlocked.

2.2 Compiler Challenges

Although network speed is still being improved with more advanced technologies, we observe that such new technologies are slowly deployed due to the over-supply of network bandwidth on the backbone networks [1] and consumers' preference to products with better performance/price ratio. Consequently, equipment vendors no longer view performance or throughput as the sole goal in developing new products. The emergence of network processors indicates such shifting of focus from high speed ASICs to easily programmable network processor products. Likewise, major network processors have undergone several generations till now, from early products with less programmability, unfriendly user interfaces and software supports to newer generations that have improved significantly in their ease of programming and application development.

The motivation of compiler design and optimization for network processors comes from several aspects. First, the programming interface has become an important factor which could affect the popularity of the product and a company's revenue. Therefore, a high level language and friendly programming environment are preferred. As concluded in [71], Intel's IXP network processor has shown the fastest revenue growth in 2003 mainly due to their flexibility and strong software support. We believe that ease of programming will continue to play an important role as the performance of competing network processors offer more and more features and larger program to be installed on network processors.

However, network processor architecture has many specially designed features as addresses in the previous section. Most of these features are not seen on general-purpose processors. Also, the architecture of network processors varies greatly among different equipment providers. Therefore it becomes very difficult to construct good compilers that can efficiently compile source code in high level language to machine code. This is also one of the reasons that early network processors rely heavily or even entirely on assembly programming, such that the programmer can directly control the quality of the code at assembly level. Even now, some of the mainstream products only support assembly level coding. However it is difficult to perform architecture specific optimizations at assembly level. Most importantly, lower level programming has induced lots of complaints from programmers, esp. they now have to deal with a much larger code base on new generation network processors. Thus, compiler development has caught significant attentions from the network processor vendors.

Actually, compiler optimization for network processor has become an emerging research topic [29, 66, 43, 47, 16]. There are several challenges facing us. One is to define a high level language with domain specific features that could help the programmers to write code expediently. We believe that exposing certain architecture features in the programming language can help compiler optimizations, but should be limited to a reasonable scale such that their negative effects are avoided. For example, on the IXP network processor, the memory hierarchy includes local on-chip memory (available on IXP2xxx series), global on-chip SRAM (a.k.a. scratchpad), off-chip SRAM and DRAM etc. We could allow user specification of which data structure should be put on which memory component, such that latency-critical data are put closer to the processor. However, when low-latency memory components are not enough to hold all latency-critical data, some of them must be moved to off-chip. In such scenario, we would say compiler optimization combined with user specification is likely to achieve the best result.

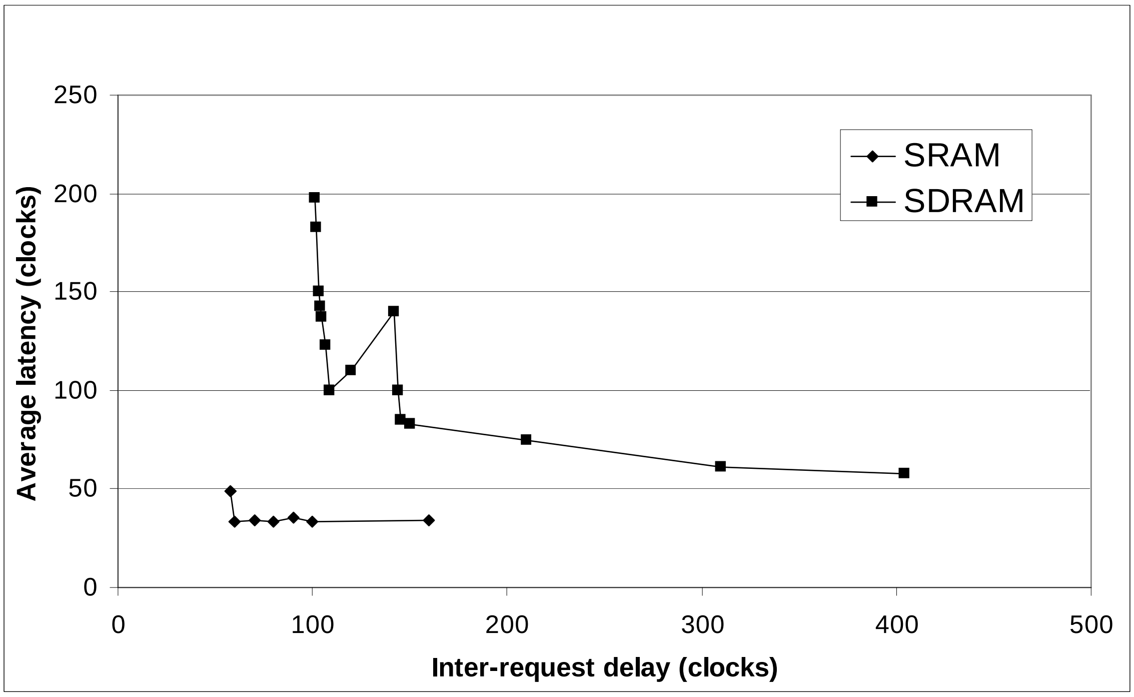


Figure 5: IXP1200 SRAM and SDRAM latency distribution

Notice that, the memory/processor speedup gap is huge on network processors due in large part to their high clocking frequency and lack of caches. Figure 5 shows the latency distribution of SRAM and SDRAM for the IXP1200 network processor. The X-axis stands for the delay (cycles) between consecutive requests. The Y-axis is the latency to complete the memory access. This figure demonstrates that the latency varies a lot for both SRAM and SDRAM accesses. Also, the inter-request delay could affect the latency. A small inter-request delay lengthens memory access latency due to more competition in the memory subsystem. Once the inter-request delay is large enough to avoid contentions, the curve plateaus. On average, SRAM latency is around 40-50 cycles, while SDRAM latency is in the range of 50 to 200 cycles. For new generations of the IXP network processors, the latency (in terms of number of cycles) is even larger. SRAM latency is over 150 cycles on IXP2800, while SDRAM latency is over 300 cycles. These data tell us that fully utilizing registers is extremely important to bridge the processor/memory gap on network processors. This motivates our work on register allocations as will be presented in the following chapters.

Besides language definition, compiler optimization catered towards special hardware needs is another important topic. The goal of compiler optimization is to generate code that is far superior to hand-crafted code by making the best out of what the hardware provides. As will be addressed in the next section, we have looked at interesting problems on the Intel IXP processor, such as the dual bank register assignment problem, inter-thread register allocation etc. Since many of the network processor features are not seen on other processors, we frequently need to model the problems in a new way and solve them individually. The relatively small footprint of the code on network processors makes the performance very sensitive to the effectiveness of compiler optimizations. Meanwhile, due to the small problem size, aggressive compilation techniques can be applied to obtain good results. For example, some papers have attempted Integer Linear Programming (ILP) [29] to search for the optimal solution. They have shown that the compilation time is still acceptable with ILP solvers.

Due to the unique application needs and lack of OS and sophisticated hardware, new problems open up on network processors that sometime are out of the scope of a traditional compiler. One possibility is to use compile-time information to guide runtime constraints. Notice that network processors typically lack strong OS support, which could harm their runtime performance. With multiple threads running on the same processor, issues like proper resource sharing, processor scheduling, I/O control, etc. could emerge just as on general purpose processors. Therefore, using compiler information to provide certain level of runtime support could be a viable way to improve the performance of multi-threading. Moreover, the overhead of compiler is typically tolerable compared with OS overhead if one were installed on the network processor.

2.3 Register Allocation

Register allocation is one of the phases during compilation. The compiler attempts to allocate as many variables (or virtual registers) to registers as possible. This is because registers can be assessed much faster than the memory, and registers are always a scarce resource. The register allocator must determine which values should be in registers at each program point.

There have been a wealth of research in literature about register allocation such as [24, 31, 42, 19, 2, 11, 12, 13, 17, 28, 8]. Register allocation could be viewed as a graph-coloring problem. Such allocator was first designed and implemented by Chaitin in 1981 [12, 13]. A graph-coloring register allocator first builds an *Interference Graph.* Each variable is represented by a node on the interence grpah. An edge connecting two nodes indicates that the two nodes cannot be put into the same register, since they have been used simultaneously (or co-live) in at least one program point. The problem then becomes how to color the interference graph with N colors, where N is the total number of registers that are available. To do so, the graph is reduced in size in repeating steps until no further simplication is possible. If the graph is not colorable, some variables must be put into memory, then the graph is re-colored until a solution is obtained.

Later, several work [8, 28] improved the graph-coloring algorithm proposed by Chaitin. Briggs et. el. [8] suggested an optimistic coloring approach. Instead of spilling variables into memory immediately when the graph cannot be simplified, they put those variables onto the stack temporarily, hoping that they can be colored during the coloring phase. More recently, George et. el [28] further improves the alogrithm with an iterated approach that can better integrate multiple phases.

Chow and Hennessy's allocator [17] uses a frequency sensitive heuristic to guide the order of coloring and splitting. The allocator can take advantage of frequency information supplied by the client to prioritize decisions such as coalescing and spilling. This addition should improve code quality, especially for programs with tight loops.

Lueh et. el. [49, 48] proposed fusion-based register allocation, which breaks up register allocation into a per-region basis. The simplest region is a basic block. Inside each region, spilling is performed such that the resulting interference graph is simplifiable. A transparent live range is one that is live on entry and exit to a region and is not used within the region. Sometimes, transparent live ranges are spilled for the graph to be simplifiable, although the actually spilling of transparent live ranges is delayed.

In Kim's dissertation [42], several innovative techniques are proposed for frequency sensitive region-based register allocation to improve the code quality generated by the region-based approach. In this manner, the compilation time can be reduced by almost half of its global counterpart, while maintaining the execution time within 5% of the global approach.

Live range splitting is a technique to reduce the spill amount which has been employed by many optimizing compilers [19, 7]. Long live ranges are split into shorter ones by inserting copies and load stores at appropriate places in the code.

Although graph coloring is recognized as a good technique for register allocation, for large interference graphs, the memory demands could be quite high. Gupta et. el. [31] presented an algorithm that uses the notion of clique separators to reduce the space overhead. It first partitions the code into code segments using clique separators. Each code partition is colored independently. The colorings of all partitions are combined to obtain a register allocation for the entire program.

Appel and George [2] partition the register allocation problem for the Pentium into two subproblems: optimal placement of spill code followed by register coalescing; they use ILP solver for the former, such that the number of spills is optimal. The coalescence phase afterwards does not add more spills. Also, only involving ILP solver in the first subproblem could greatly reduce compilation time.

We have a recent work on register allocation [78], which proposes to use a new encoding scheme to increase the number of architected registers exposed to the programmer. Instead of encoding the absolute register number, differential encoding encodes the difference between two consecutive register numbers accessed along the path. Also, register allocation is combined with differential encoding to reduce its overhead.

For embedded processors, the backend varies a lot due to the fact that most of them are specially designed for particular purposes. Therefore, the register allocation phase must take into consideration the underlying hardware provision to maximize the performance. Researches in this area [24] is still very active.

Register allocation is one of the problems we have looked at on the IXP network processor. It is important to maximize the use of registers on this network processor for several reasons.

First, the design philosophy of network processors is quite different from general purpose processors. Since there is already plenty of packet-level parallelism to be exploited, instead of trying to extract parallelism from a single program by employing complicated hardware mechanisms, it is more profitable to just deploy a large number of very simple processors to take advantage of the packet level parallelism. Therefore, the design of the network processor typically simplifies each processor but increases the number of processors. For IXP, the register file is greatly simplified. Each bank only has one port. Second, as mentioned earlier, the memory access latency could be huge. Therefore, putting variables into the registers could greatly improve performance by cutting down the access latency. Third, functions are typically inlined to avoid function invocation overhead. This could further increase the register pressure.

CHAPTER III

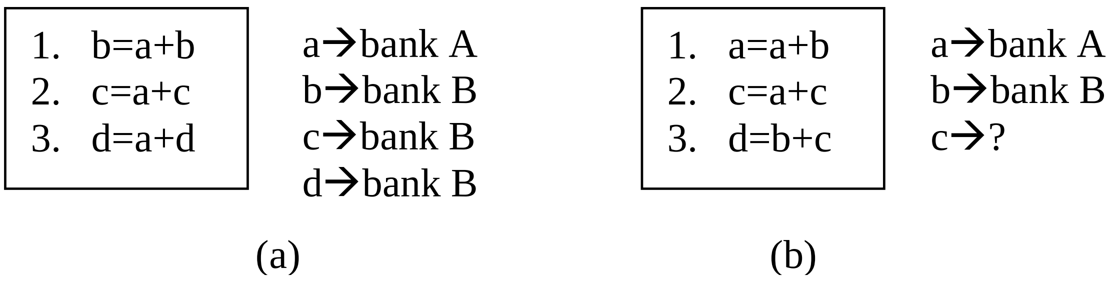
INTRA-THREAD REGISTER ALLOCATION

In this chapter, we will look at the register allocation problem with regard to a single thread. As mentioned earlier, the register architecture of the IXP network processor is very special. In particular, the general purpose registers (GPRs) are split into two banks. This gives rise to the dual-bank register assignment problem.

3.1 Dual-bank Register Assignment Problem

The design of dual-bank register file raises the problem of register allocation with consideration to the bank assignments. Without such consideration, register allocation is handicapped, and in some cases, even impossible. Figure 6 gives two examples to illustrate the dual-bank assignment problem. In these examples, we assume that the total number of physical registers is four, two in each bank. In Figure 6.a, without the above register bank restrictions, each of the variables can be assigned one physical register. However, the three instructions require variable a to be in the opposite bank of variable b, c and d. Thus, if variable a is in bank A, the other three variables must be in bank B, which is not possible (since there are only two physical registers in bank B). Figure 6.b shows variables a and b must be in opposite banks. Similarly, variables a and c, variables b and c must be in opposite banks too. This creates the problem that even if physical registers are enough, we still cannot satisfy all bank constraints, since the first two instructions require variable b and c to be in the same bank, which contradicts with the last instruction. Obviously, the first example shows that the dual-bank constraint may cause *imbalance of register* requirements to the two banks. The second example shows that if the bank constraints form a *cyclical conflict*, then no bank assignment is possible without resolving the conflict.

The current implementation of the IXP assembler takes passive approaches to the bank assignment problems. The first problem will generate a “not enough registers” message



```
1. b=a+b      a→bank A
2. c=a+c      b→bank B
3. d=a+d      c→bank B
              d→bank B
```

```
1. a=a+b      a→bank A
2. c=a+c      b→bank B
3. d=b+c      c→?
```

Figure 6: Example of the Dual-bank assignment problem

to the programmer, while the second problem will cause an error messages prompting the programmer to fix the unresolvable conflict. Obviously, it is difficult for the user to make the right decisions. As we will illustrate later, there are non-obvious trade-offs involved that can actually achieve low-overheads for both register spill and code growth. These tradeoffs are not easily perceivable by the users. With the development of high-level language support for the IXP processor, it is no long appropriate to ask users to resolve the conflicts (and also there is no need to provide "user understandable code" at assembly level), after the code has been transformed from the high-level language. A compiler solution is desirable to automatically assign both registers and banks.

The *dual-bank register allocation problem* is to determine the physical register allocation together with the bank assignment of that physical register for each virtual register. It aims to reduce the overhead due to additional spill code and other extra instructions which can degrade the performance and cause code growth. Speeding up the execution is at the highest priority; therefore reducing the number of spills becomes most important due to the long memory latency.

3.2 Register Conflict subGraph and No-conflict Rule

To represent the register constraints for the dual-bank assignment problem, in this section, we introduce the concept of *Register Conflict subGraph (RCG).* We build upon the standard representation of *Interference Graph* used in coloring based allocators. *Interference Graph (IG)* represents each *Live Range* as a node in the graph and an edge between two nodes means that the two live ranges interfere with each other or are co-live at some program point. We call the edges in the interference graph *Interference Edges.* To distinguish the interference graph before and after the register allocation, we define *Virtual Interference Graph (VIG)* and *Physical interference graph (PIG).* The live ranges (nodes) on VIG are *Virtual Live Ranges*, which are associated with virtual registers. Similarly, live ranges on the PIG are *Physical Live Ranges*, which correspond to physical registers[1]. Some edges in the interference graph are further distinguished as *Conflict Edges*, which are defined as follows.

Definition 1. *Conflict Edge*
If two live ranges interfere in the same ALU instruction as two source operands, the interference edge connecting them is called a conflict edge. They are said to conflict with each other.

Obviously, we have the following claim: *A conflict edge must be an interference edge.*

Definition 2. *Register Conflict subGraph (RCG)*
The register conflict graph is a subgraph of the interference graph consisting only of conflict edges and all nodes.

Similarly, we have the definition of *Virtual Register Conflict subGraph (VRCG)* and *Physical Register Conflict subGraph (PRCG)* based on the underlying interference graph.

Figure 7 shows two RCGs corresponding to the examples in Figure 6. Figure 7.a is the RCG for Figure 6.a. Note that all interference edges are conflict edges for this example. We can observe that there are edges from a to b, c and d, which means, a and b,c,d form two separate groups that should be put into different banks. In Figure 7.b, the RCG forms an odd-cycle, which means that the nodes cannot be separated into two groups such that the nodes in the same group do not conflict with each other.

Given a RCG, we can judge if the nodes (either virtual or physical live ranges) in the graph can be categorized into two groups with each group being assigned to one of the register banks. If this is possible, we say the RCG is *bank conflict-free*, otherwise it leads

[1]We define the virtual live ranges to be maximal du-ud chain (or a web) on the VIG, that can be separately allocated. For PIG, physical live ranges correspond to the physical register names, which can be the union of several virtual live ranges that are assigned the same physical register.

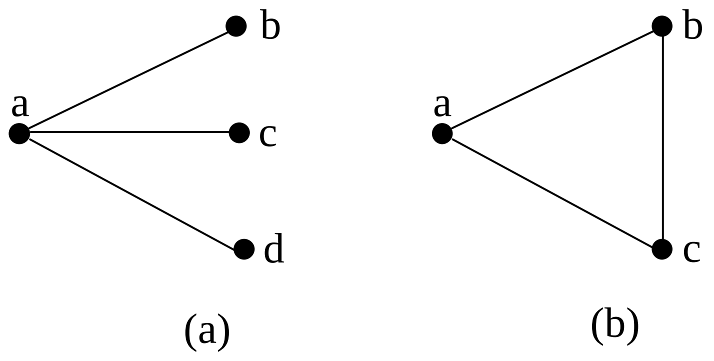


Figure 7: Examples of the RCG

to a bank conflict that should be resolved. The bank conflict property is also applicable to the IG, i.e. we say an IG has bank conflict if its RCG exhibits a bank conflict. The problem of determining whether a RCG is conflict-free is equivalent to determining if the RCG is *bipartite*. The following *No-conflict rule* gives the necessary and sufficient condition to judge whether a RCG has conflicts.

No-conflict Rule. *The RCG is conflict-free if and only if it contains no odd-length cycle.*
Proof: This problem is equivalent to determining if the RCG is a bipartite graph. According to the property of the bipartite graph, no odd-length cycle should exist. This is also a sufficient condition.

Thus, the example in Figure 6.b has a conflict, since its RCG (shown in Figure 7.b) is a length-3 cycle, but the example in Figure 6.a is conflict-free due to the absence of any cycle.

Note that, the No-conflict rule applies to both VRCG and PRCG; however, it gives no guarantee for balancing register assignment in each bank group. From the perspective of register allocation, the number of total physical registers available for each bank is fixed. Typically, the number of physical registers available to bank A equals that of bank B which means register allocator should make an effort towards balancing allocated registers equally between the two banks. We define such a problem as *Balanced Dual-bank Register Assignment*. Accordingly, the RCG with equal number of nodes in each bank group and conflict-free is called *Balanced Conflict-free RCG*. Therefore, the example in Figure 6.a shows a conflict-free but not a balanced conflict-free RCG.

3.3 Phase Ordering

Register bank assignment is closely related to the register allocator that performs virtual to physical register mapping. There are three approaches we can take to perform register allocation together with bank assignment.

The first method is to assign register banks before virtual registers are mapped to physical registers. We call it *pre-RA bank assignment* approach. All conflicts are resolved on the VRCG before the register allocator takes over. The second approach is to do register allocation first and then assign the banks to physical registers, at the same time resolve conflicts on the PRCG. This is called *post-RA bank assignment.* Finally, a combined register allocation approach can consider the register bank assignment at the same time as register allocation.

3.4 Pre-RA Bank Assignment Approach

We can designate the bank assignment for each virtual register before the register allocation (mapping of virtual to physical registers) is done. According to the No-conflict rule, if there are odd cycles of conflict edges in the VRCG, the VRCG has conflicts. The following claim says that sometimes the conflicts may continue to exist after the register allocation.

Claim. *If the VRCG doesn't meet the No-conflict rule, the conflicts will persists after a Chaitin style register allocation assuming no spill is generated.*

In general, any graph coloring allocator coupled with other phases meets the above claim. These include, for example, Chaitin's [12, 13], Briggs [8], and Appel & George's [28] register allocators. The claim can be easily verified. During the register allocation, the only possibility is to map *multiple* virtual registers to the *same* physical register through coalescence or coloring. If an odd cycle exists on the VRCG, no edge on the cycle can be collapsed since the interfering nodes cannot be coalesced during the register allocation.

The pre-RA bank assignment approach regards the register allocation pass as a black box and assigns the banks to the virtual registers first. After the bank assignment is done,

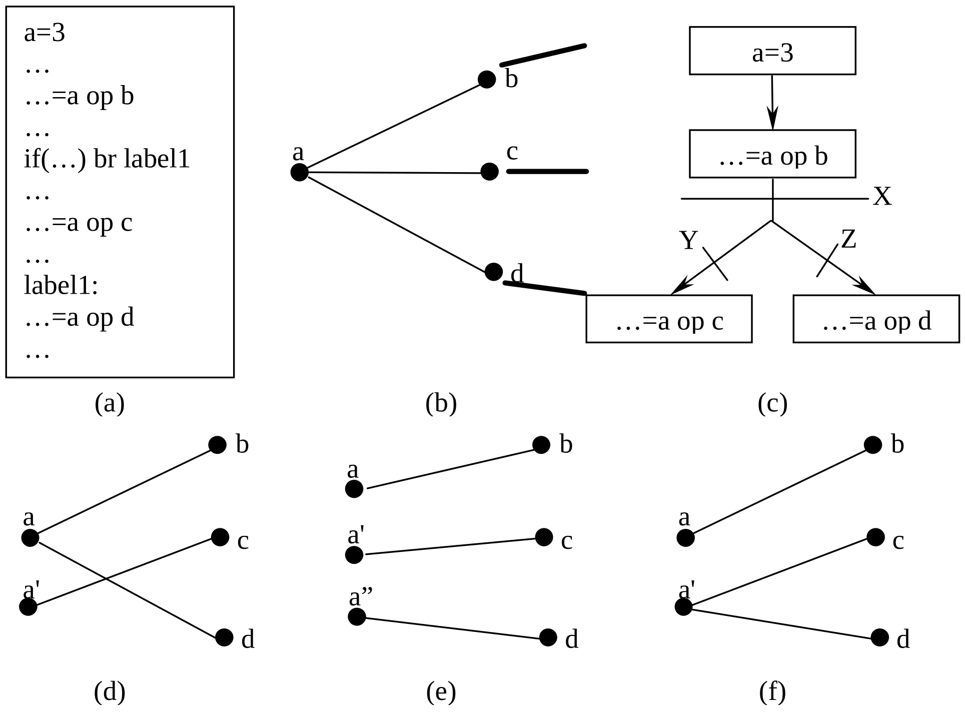


Figure 8: Edge breaking through live range splitting on RCG

virtual registers are grouped according to their bank assignments. The register allocation is then done separately for each bank.

To avoid conflicts, all odd-length cycles should be removed to make the graph bipartite. A straightforward way to remove odd-length cycles is to break the cycle at some node point in the RCG by splitting the live range of the node.

Figure 8 shows an example of breaking a cycle in the RCG. Figure 8.a gives the code segment in which the live range of variable a consists of 1 definition and 3 uses across a conditional branch. Figure 8.b shows the RCG. Variable a conflicts with b, c and d. Figure 8.c shows the live range using the control flow graph (i.e. du/ud chain of variable a). In Figure 8.b, if we want to break the cycles passing edge (a,c), (because (a,c) is a part of an odd cycle) we can simply split the live range at point Y (Figure 8.c), which means after point Y, the live range is renamed, for example to a’. Figure 8.d shows the RCG after splitting. At the split point, we need to insert a move instruction a’=a, so the live range gets separated. One instruction is added in the code and one extra node is added on the RCG. If we want to break the cycles spanning both edges (a,c) and (a,d), one choice is to split at both points Y and Z, i.e. renaming the live range to a' after point Y and renaming the live range to a" after point Z (Figure 8.e). This requires two move instructions and two additional nodes on the RCG. However, we can split the live range at point X, which leads to the RCG in Figure 8.f. We rename the live range at point X to a'. In this case, the cost is only one move insertion and one additional node on the RCG but we cannot break the cycles that pass both (a, c) and (a,d).

The problem of making RCG bipartite (breaking all odd cycles) with minimal cost is shown to be NP-complete by reducing a graph problem called *Maximal Bipartite Subgraph* [57] to it.

Theorem 1. *The problem of making RCG bipartite (break all odd cycles) with minimal cost is NP-complete.*

Proof: Firstly, it is trivial to show the problem is polynomial-time verifiable. Next, we reduce the maximal bipartite subgraph problem to it. The maximal bipartite subgraph problem is to find the minimal number of edges to be deleted to make a given graph bipartite. Suppose, we are given an instance of this problem-an undirected graph G(V,E). We construct a program code with a two-level CFG. The first level basic blocks (BBs) represent nodes on G called node BBs. The second level BBs represent edges on G called edge BBs. Each edge BB is connected to the two node BBs on the edge. In each node BB, there is one instruction that makes an assignment to a variable. In each edge BB, the two variables from its node BBs conflict as source operands of an ALU instruction. Now, the constructed graph has a RCG equivalent to G. In addition, breaking an edge on G is equivalent to splitting the live range in the corresponding edge BB by inserting a move instruction before the ALU instruction. Therefore, we have reduced the maximal bipartite subgraph problem to the problem of making RCG bipartite with minimal cost. Figure 9 shows an example.

There has been substantial work done on the Maximal Bipartite Subgraph problem in graph theory [59, 38]. Several approximation algorithms have been proposed. For example,

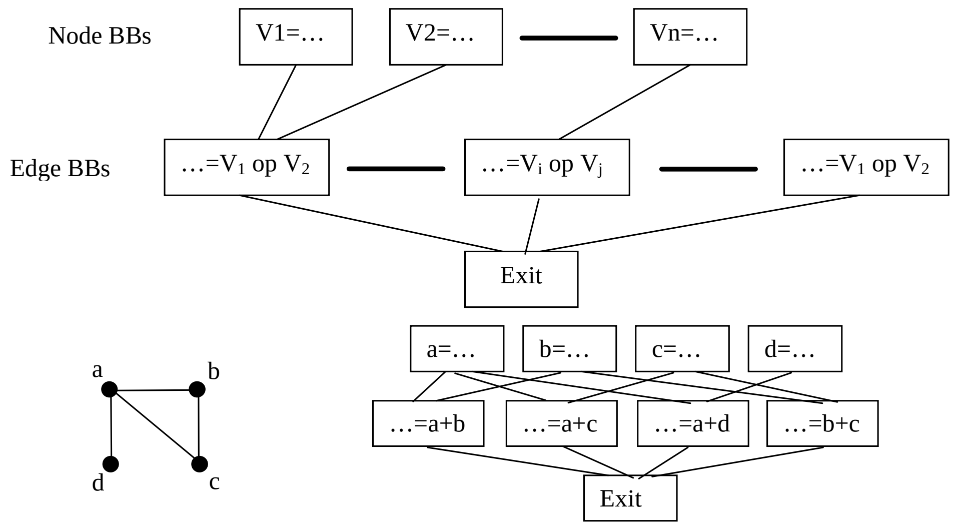


Figure 9: Illustration for proving that making RCG bipartite is NP-complete

[59] gives an algorithm with complexity of $O(n^4)$. However, these approaches assume that the underlying graph does not have length-3 cycles which are not true in our case. Besides, they target the minimal number of edges removed to make the graph bipartite, while we must consider live range splitting on the nodes and the cost is not uniform for node splitting. Finally, their analysis gives a rather loose lower bound with regard to the quality of the solution. Actually the solution of our algorithm is far beyond the lower bound calculated by these papers. Specifically, our heuristic algorithm below takes into consideration the special properties of RCG that can greatly reduce the complexity of the algorithm while maintaining quality of the solution.

Before presenting the heuristic algorithm, we briefly discuss the detection of odd cycles in the RCG. Here, we define that if two cycles have at least one edge that is different, then the two cycles are said to be different. For any algorithm that can resolve all the conflicts in the RCG, it needs to break all the odd-length cycles in the graph. However, a simple estimation tells us that finding all odd-length cycles will take exponential time to finish. A brute-force algorithm must try up to $\sum_{k=1}^{(n-1)/2} C_n^{2k+1}$ possibilities to find out all odd cycles, where n is the total number of nodes in the RCG. This sum has a complexity of $O(2^n)$,since we know $\sum_{k=0}^{n} C_n^k$, which is almost twice of $\sum_{k=1}^{(n-1)/2} C_n^{2k+1}$.

However, in real programs most of the odd cycles are small (we will give some data in

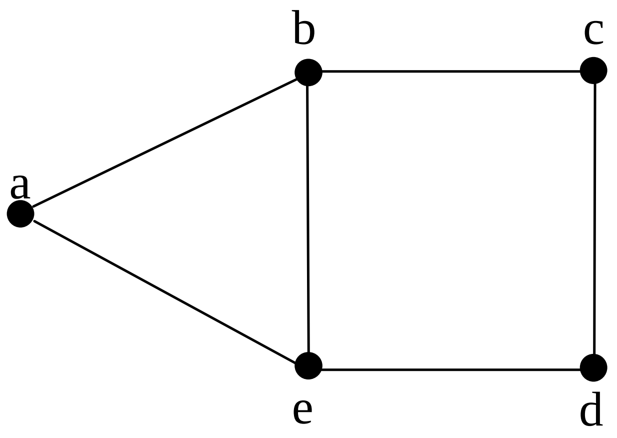


Figure 10: Duplication in counting odd cycles

later sections). Another observation is that the brute force searching may include duplications. In Figure 10, two odd cycles can be found i.e. abe and abcde. However, if edge (a,b) is removed, both cycles are broken. If edge (b,e) is removed, then cycle abcde still remains. This example shows that if two cycles share edges, breaking one of them can cause the other to disappear automatically.

Our heuristic algorithm starts with the shortest odd cycles, after breaking shorter ones, we move on to find longer cycles and break them, until all cycles are gone. As shown in Figure 10, the removal of shorter cycles may break longer cycles as well. The main data structure for odd cycle detection is called *Breadth-First Hierarchy* as described below.

3.4.1 Breadth-First Hierarchy

Given a RCG, we build a *breadth-first hierarchy* from one of the root nodes following the breadth-first searching algorithm. The procedure is in Figure 11:

The root node r is the only first level node. After visiting the root node, we visit and mark its directly connected neighbors, which are marked as second level nodes. Next, we visit neighbors of second level nodes that are unmarked and assign them level three and so on. It is easy to notice that the level of a node is its minimal distance to the root plus one. Besides, the time complexity to construct the breadth-first hierarchy is $O(n^2)$. Next, we give two lemmas about the breadth-first hierarchy.

```
Input: root r
Output: The breadth-first hierarchy
Level_num: number of levels in the hierarchy;
Node_level_set[]:array of sets;
Algorithm:
Function Build_Breadth_First_Hierarchy(node r)
Begin
   Node_level_set[1]$\leftarrow${r}; mark r;
   Level_num=1;
   While Node_level_set[Level_num] not empty do
      For each node p in Node_level_set[Level_num] do
         Add all p's directly connected neighbors that have not
         been marked to Node_level_set[Level_num+1];
         Mark all newly added nodes.
      od
      Level_num++
   Endw
   Return Level_num, Node_level_set[]
End
```

Figure 11: Building the breadth-first hierarchy

Lemma 1. *If an edge e connects node p and node q on the RCG, then* $|node\_level(p) - node\_level(q)| \leq 1$*, where node_level gives the level of the node on the breadth first hierarchy.*

Proof: Without loss of generality, assume $node\_level(p) > node\_level(q)$, if there is an edge from node q to node p, then p should be on level $node\_level(q)+1$, which proves the Lemma.

Lemma 2. *The RCG is conflict-free, iff any edge* $(p, q) \geq |node\_level(p) - node\_level(q)| = 1$.

Proof: From Lemma 1, we only need to show $node\_level(p) <> node\_level(q)$. If this condition is not satisfied, (i.e. if $node\_level(p) = node\_level(q)$), we find a path from root r to p called path(r,p) and a path from r to q called path(r,q), each with $node\_level(p)$ nodes. Let s be the latest common node for the two paths, i.e. path(s, p) and path(s,q) only share node s. We then have an odd cycle $s \to p \to q \to s$. On the contrary, we can separate the nodes into two groups; all nodes in odd level form one group, and all nodes in even level form another group. Then there are no edges within these two groups and thus, the graph is bipartite and conflict-free.

Lemma 1 shows that the edges on the RCG only appear between nodes from adjacent levels on the hierarchy. Lemma 2 implies that no two nodes on the same level being connected is equivelant to the RCG being conflict-free. In other words, if a RCG is not conflict-free, there must be an edge that connects two nodes on the same level. We call such an edge *Parallel Edge.* Next, we present Lemma 3 that will be used later in our heuristic algorithm to detect odd cycles with length k, where k is an odd number greater than 3. This lemma can help to build a fast algorithm to detect all such cycles.

Lemma 3. *The smallest odd cycle is of length k (k is an odd number), iff there is no parallel edge on any breadth-first hierarchy up to level* $(k-1)/2$.

Proof: Briefly, the proof is similar to that of lemma 2. If there is a parallel edge on a level less than $(k-1)/2$, we will find an odd cycle that is less than k.

Thus, to find all odd cycles of length k (assuming non-existence of shorter odd cycles) in the RCG, we build n breadth-first hierarchies with each of the n nodes as root nodes. All the hierarchies only need to be built up to $(k-1)/2$ level and checked for parallel edges. Since the algorithm runs faster when k is small and most odd cycles are actually small, we find this odd cycle detection algorithm finishes quickly in our implementation.

3.4.2 Live Range Splitting Patterns

Live range splitting patterns represent the possibilities a live range can be split. For each live range, we can find out all splitting patterns. For example, in Figure 8, we observe 4 possibilities to split the live range, i.e. X, Y, Z or (Y, Z). For each splitting pattern, we can calculate its cost. Although the number of splitting pattern may grow exponentially, in practice only a few live ranges contain a large number of uses as source operands. This is probably due to the nature of the applications running on the network processors. Also, the live range defined as connected du/ud chains can achieve value separation, which leads to smaller number of splitting patterns due to reduced number of uses associated with each live range. In implementation, we specify a limit of 1000 patterns for each live range, otherwise the live range is forced to be split as if it is involved in every cycle. In our evaluation, we show that this almost invariably happens.

```
Input: VRCG, CFG
Output: VRCG (no conflict),CFG, set of split live ranges

Algorithm:
Function Pre_RA_Bank_Assignment
Begin
   Construct patterns (calculate costs) for all live ranges, store to Pattern_set
   For m=3 to n, step 2
      Detect all odd cycles with cycle length m, store to Cycle_set;
      while (Cycle_set <> empty) do
         For each pattern p in Pattern_set do
            w=cost of using p
            bn=number of cycle p can break in Cycle_set
            The priority of pattern p is bn/w
         od
         The pattern with highest priority is applied on VRCG, CFG
         Remove broken cycles from Cycle_set
         Update Pattern_set if necessary
      Od
   Endfor
   Return VRCG, CFG and the set of split live ranges
End
```

Figure 12: Pre-RA bank assignment heuristics

3.4.3 The Pre-Register Bank Assignment Heuristic Algorithm

As mentioned before, our heuristic algorithm breaks odd cycles from the shortest, i.e. size three and goes to longer cycles as it proceeds. The algorithm is listed in Figure 12. It executes for several iterations, where each iteration breaks all cycles of length m. m takes odd integers from 3 to n. During each round, two sets are built. The *Cycle_set* stores all cycles with length m. The *Pattern_set* stores all patterns. Then we examine each pattern to see how many cycles it can break in the *Cycle_set*, the priority function for applying a pattern is calculated as the number of cycles it can break divided by the cost (the number of moves inserted). The pattern with highest priority is chosen. After a pattern is applied to CFG and VRCG, all broken cycles are removed from *Cycle_set* and the *Pattern_set* is also updated, since new live ranges are added and the old live ranges might be altered. The algorithm picks the most favorable pattern and proceeds with that pattern. The following claim guarantees the algorithm will eventually remove all odd cycles.

Claim. *Assume k is an odd number greater than 3, then after all length $< k$ odd cycles are broken, the breaking of length-k cycles will not create shorter odd cycles.*

This is obvious, since during the whole process, nodes are not merged to form new cycles on RCG. The complexity of the algorithm in the main loops is roughly $O(n \times P \times M)$, where n is the number of nodes on the original graph, P is the maximal number of patterns in the *Pattern_set* and M is the maximal size of the *Cycle_set*. Since most odd-cycles are short and breaking shorter cycles may break longer ones as well, the outer-most for loop in Figure 12 normally finishes early. From Figure 12, we notice that the update of *Cycle_set* and *Pattern_set* can be done with marginal computation.

The drawback of this approach is the difficulty to control the register pressure in each group, which may lead to imbalanced pressure between the two banks during register allocation. For example, assume that the virtual registers are grouped equally when they are passed to the register allocator. However, it then turns out that one of the groups needs more physical registers to avoid spilling, while the other group has free registers. As it is hard to judge the physical register and spill code that will be generated before the register allocation, the pre-register allocation approach may result in imbalanced spill. In other words, it may increase the overall spill cost. However, after the RCG becomes conflict-free, this problem can be alleviated by making the RCG near-balanced before passing it to the register allocator.

3.4.4 Near-Balancing the RCG

After live range splitting, it is quite likely that the RCG is no longer a connected graph. By identifying the connected components of the RCG, we can near-balance the number of nodes in the two banks through separate bank assignment to each connected component of the RCG. Suppose the RCG has m connected subgraphs: $G_1, G_2 \ldots G_m$, each with a subset of the nodes and edges of the RCG. Since the m subgraphs are all conflict-free, i.e. bipartite, we can separate each G_i into GA_i and GB_i, such that no conflict edge is inside GA_i and GB_i (This can be done to construct a breadth-first hierarchy and separate odd level and even level nodes). Let $A_i = |GA_i|, B_i = |GB_i|$ and the number of total nodes is $n = \sum A_i + \sum B_i$. We want to minimize $|\frac{n}{2} - \sum C_i|$, where $C_i = A_i$ or B_i. In our implementation, we apply exhaustive search, which takes $O(2^p)$, since p is typically less than 10. For larger p, a fully polynomial time algorithm can be derived from the *subset sum algorithm* [64], which closely approximates the optimum in polynomial time.

3.5 Post-RA Bank Assignment

Although the pre-RA bank assignment can avoid conflicts during the register allocation, it creates imbalanced register requirements (higher chromatic number for one of the register banks). On the contrary, the post-RA bank assignment approach allocates register with well-known register allocation algorithms to minimize the spills in the first place. Our bank assignment algorithm is invoked to resolve bank conflicts and balance physical register distribution across banks. The post-RA bank assignment algorithm will not increase spill code. Although some physical live ranges are split after moves are inserted, the cost is much lower than spills. As we know, the register allocator can map different virtual registers to the same physical register. Therefore, physical live ranges are typically larger than virtual ones. The post-RA bank assignment problem shares many properties with the pre-RA bank assignment problem. Therefore, some of the techniques can be borrowed. However, there are clear differences between these two approaches. Firstly, we cannot simply rename a live range, because each physical live range is allocated a physical register. If a live range is split, we must find an available physical register to hold the new live range. Secondly, the PRCG must be balanced and conflicts removed.

3.5.1 Cost for Splitting Patterns

The cost of splitting patterns for physical live ranges are calculated differently. Especially the cost to split at a certain point is not just equal to the inserted move instruction. In building the *Pattern_set* for the heuristic algorithm, we categorize the cost for a pattern into 3 types:

1. If the register pressure in the renamed program segment (for example, in Figure 8.c, from point Y to instruction a op c) is less than the number of available physical

```
1. …
2. …
3. …=a op b
4. …
5. …
```

```
1. a  =a ⊕ X
2. X = a ⊕ X
3. …=X op b //X=$a_{orig}$, a=$a_{orig}$⊕$X_{orig}$
4. X = a ⊕ X  //X=$X_{orig}$, a=$a_{orig}$⊕$X_{orig}$
5. a  = a ⊕ X  //X=$X_{orig}$, a=$a_{orig}$
```

(a)

```
1. …
2. …
3. a  =a op b
4. …
5. …
```

```
1. b  =b ⊕ X
2. X = b ⊕ X
3. …=a op X //X=$b_{orig}$, b=$b_{orig}$⊕$X_{orig}$
4. X = b ⊕ X  //X=$X_{orig}$, b=$b_{orig}$⊕$X_{orig}$
5. b  = b ⊕ X  //X=$X_{orig}$, b=$b_{orig}$
```

(b)

Figure 13: In-place bank exchange

registers, then the splitting cost is set to the cost for move insertion.

2. If condition in (1) is not true, we look around near the splitting point to find the chance of *rematerialization*, and calculate the cost accordingly.

3. Finally, we count the cost of doing *in-place bank exchange.*

3.5.1.1 Rematerialization

Rematerialization has been used by [8] to free a register through recomputing the value in-place before it is needed to avoid carrying the value in the register. We check for rematerialization before we analyze the splitting patterns; thus the register pressure in some region of the program can be reduced.

3.5.1.2 In-place Bank Exchange

After all the above endeavors fail, we have the last resort to solve the bank conflicts using this technique. *In-place bank exchange* requires no additional registers, however it requires 4 ALU instructions to remove one conflict edge on the RCG. Although it can be expensive in terms of space in contrast to a register spill, in-place bank exchange saves runtime cycles and most importantly, guarantees that odd cycles can be broken in the worst case without incurring spills.

Figure 13 illustrates two cases to break a conflict edge between live range a and b. In Figure 13.a, we assume that the destination operand of instruction (3) is not a. We insert two XOR instructions before and two XOR instructions after the ALU instruction in line 3. The register X is an occupied physical register that is in the opposite bank to variable a, thus it is also in the opposite bank to variable b. The first two exchanges put variable a in X and then the instruction in line 3 is conflict-free. The last two XOR instructions restore the values of a and X. The conflict edge between a and b can be removed, because the two variables can be assigned physical registers in the same bank now. Figure 13.b shows the case when a is the destination operand. We can exchange b and X to remove the conflict. In-place bank exchanges are special splitting patterns that are provided for all edges. Therefore all odd cycles can be broken in the worst case with these patterns.

3.5.2 Balancing Register Numbers in Two Banks

In this section, we discuss the balancing of registers in the two banks. After the removal of all odd cycles, we can apply the same bank assignment approach as the pre-RA algorithm to obtain a near-balanced RCG. After that, we have to reassign some of the live ranges to the opposite bank, which could induce conflicts. The induced conflicts have to be resolved at the cost of inserted instructions.

Suppose the live ranges on the RCG have been grouped into two groups, $BankA$ and $BankB$. Also assume, $|BankA| > |BankB|$. We attempt to pick one of the nodes in $BankA$ and move it to $BankB$ and estimate the cost of resolving the conflicts due to this move by again using the minimal cost splitting pattern. This procedure is repeated until the number of nodes in the two banks are equal. If the difference between $|BankA|$ and $|BankB|$ is small (as it usually is), we can attempt all combinations of moves for $(|BankA| - |BankB|)/2$ nodes from $BankA$ to $BankB$, which gives better solution than moving nodes one by one from $BankA$ to $BankB$, but takes slightly longer time to finish.

```
Input: PRCG, CFG
Output: PRCG (balanced and conflictless), CFG

Algorithm:
Function Post_RA_Bank_Assignment
Begin
   Register_allocation
   Construct patterns for all live ranges, including in-place exchange
      patterns, store to Pattern_set
   Foreach pattern p in Pattern_set do
      Calculate the cost for p (with available register, rematerialization or
         In-place exchange)
   od

   Break all odd cycles //the same as in Pre_RA_Bank_Assignment

   Near-balancing the two bank groups
   If |BankA|<>|BankB| then
      Balance the bank by moving nodes between them.
   Endif

   Return PRCG, CFG
End
```

Figure 14: Post-RA bank assignment heuristics

3.5.3 Heuristic Algorithm for Post-RA Bank Assignment

We discuss the algorithm for post-RA bank assignment algorithm in this section. In Figure 14, the procedure Post_RA_Bank_Assignment consists of 4 parts: register allocation, constructing and calculating the cost for pattern, breaking odd cycles and balancing the two bank groups. After register allocation is done (assuming a monolithic register file), the *Pattern_set* is constructed. Cost calculation is more complicated than the pre-RA bank assignment. Next, odd cycles are broken as in the previous section. Bank balancing has two steps. The near-balancing algorithm in the previous section is applied first, because it incurs no cost. Then, the approach described in the previous subsection applied to achieve a perfect balance.

3.6 Combined Register Allocation Approach

As mentioned before, it is possible to combine register allocation with register bank assignment. However, as many register allocation algorithms have been proposed in literature, we do not want to delve into all possibilities. In this section, we briefly introduce our combined algorithm based on a Briggs-style register allocator [8]. In contrast to the original algorithm, we have several modifications.

The allocator takes nodes from the interference graph and pushes them to one of the stacks.

1. Two stacks are maintained for the two banks.

2. During "coalesce", coalescence is performed only when the two nodes do not interfere with each other. In addition, coalescence should not create new odd cycles.

3. In the "simplify" stage, we push each node on the IG to one of the stacks. Nodes can be marked as "spill" or "conflict". Nodes are pushed in the following order.

   a) We pick a node and push it to a stack that causes no conflict (with nodes still on the IG) and no spilling (with neighbor number less than the number of registers in one bank).

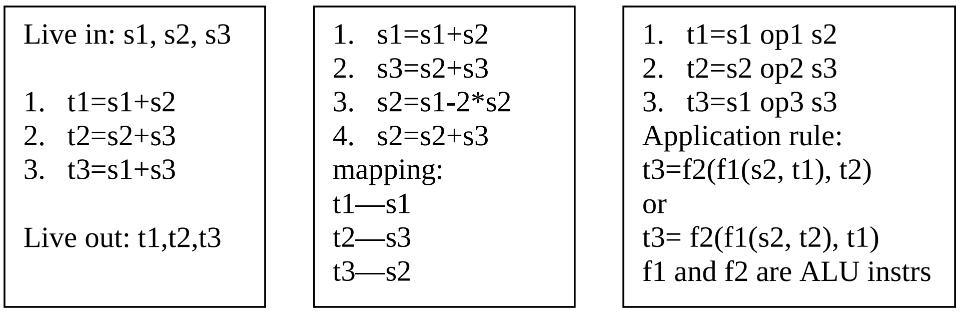
   b) If a) fails, find a node that does not need spilling (but has bank conflicts) and with minimal cost to resolve the conflict, push it to one of the stacks.

   c) If both a) and b) fail, find a node that must be spilled but with minimal spill cost as calculated by Brigg's algorithm and push it to one of the stacks.

4. During "select", we pop nodes from the two stacks one by one in the order they are pushed to the stacks.

   a) Give the node a color (the color must belong to the bank of that node) that is different from all colored neighbors on the IG, and that has no conflict with nodes already on the IG.

   b) If the color is available, but there is conflict, we resolve the conflict as in the previous section.



```
Live in: s1, s2, s3

1.  t1=s1+s2
2.  t2=s2+s3
3.  t3=s1+s3

Live out: t1,t2,t3
```

(a)

```
1.  s1=s1+s2
2.  s3=s2+s3
3.  s2=s1-2*s2
4.  s2=s2+s3
mapping:
t1—s1
t2—s3
t3—s2
```

(b)

```
1.  t1=s1 op1 s2
2.  t2=s2 op2 s3
3.  t3=s1 op3 s3
Application rule:
t3=f2(f1(s2, t1), t2)
or
t3= f2(f1(s2, t2), t1)
f1 and f2 are ALU instrs
```

(c)

Figure 15: Example for removal of conflict involving triangle cycles

c) If both a) and b) fail, the node must be spilled.

One difficulty in the combined approach is that we do not know the bank assignments of the remaining nodes on the IG during "simplify". In the worst case, we have to assume all neighbors will be in the same bank, so only half of the registers are available for coloring. This causes a lot of nodes to be marked as "spill".

3.7 *Some Enabling Techniques*

This section discusses two types of optimizations that can help with the aforementioned approaches. As shown later, these approaches do not require additional virtual or physical registers but can help to reduce the number of odd cycles on the RCG.

3.7.1 Removal of Length-3 Odd Cycle Conflicts

Figure 15 shows an example on how to remove conflicts involving odd cycles of length three. Recall that we attempt to break odd cycles in the order of increasing lengths (refer to Figure 12 and Figure 14) and thus, applying this transformation could break cycles of higher order as well. Figure 15.a is a code segment where three operands s1, s2, s3 form an odd cycle of length 3 on the RCG. Note that, all three operands are live in but not live out, while the 3 destination registers are the only live-outs. Figure 15.a shows the code transformation that removes the conflict edge $< s1, s3 >$. Also notice that the instruction in line 3 is supported by the IXP, which does a left shift before minus. Figure 15.c gives a general form of the code segment and the rule for this transformation to be legal. f1

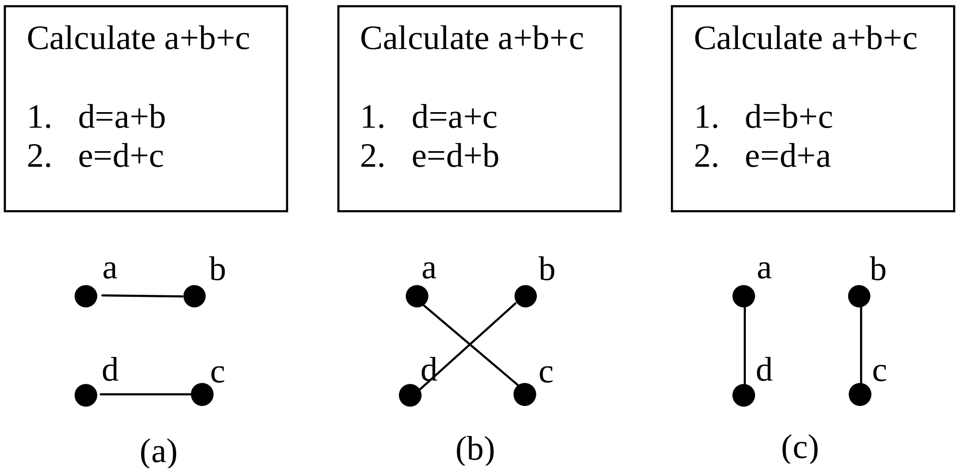


Figure 16: Example for application of algebraic laws

and f2 must be supported instructions. “plus”, “minus” are most commonly seen operators that find this transformation useful. This optimization is applicable to pre-RA and post-RA bank assignment. But for post-RA bank assignment, it merges live ranges, which may result in new odd cycles. In our implementation we have a simple check before the code transformation is applied in post-RA setting.

3.7.2 Application of Algebraic Laws

Algebraic laws such as associativity, distributivity can be applied to change the edge connectivities on the RCG, so as to reduce conflicts on the graph. These optimizations can be invoked on the RCG to break some of the cycles.

In Figure 16, three cases are shown to calculate a+b+c. With associativity, we can calculate a+b first or a+c first or b+c first. Therefore, on the RCG, 3 kinds of connectivities are possible, given that an ALU instruction can have at most two register operands. The choice largely depends on the number of odd cycles each one would create. In our implementation, we focus on the number of triangles that can go through these edges. It can be easily counted using breadth-first hierarchy based on the first two levels of nodes. The calculation order with the least number of length-3 odd cycles is chosen. More generally, most 2-operand ALU instructions satisfying associativity can be transformed with this law.

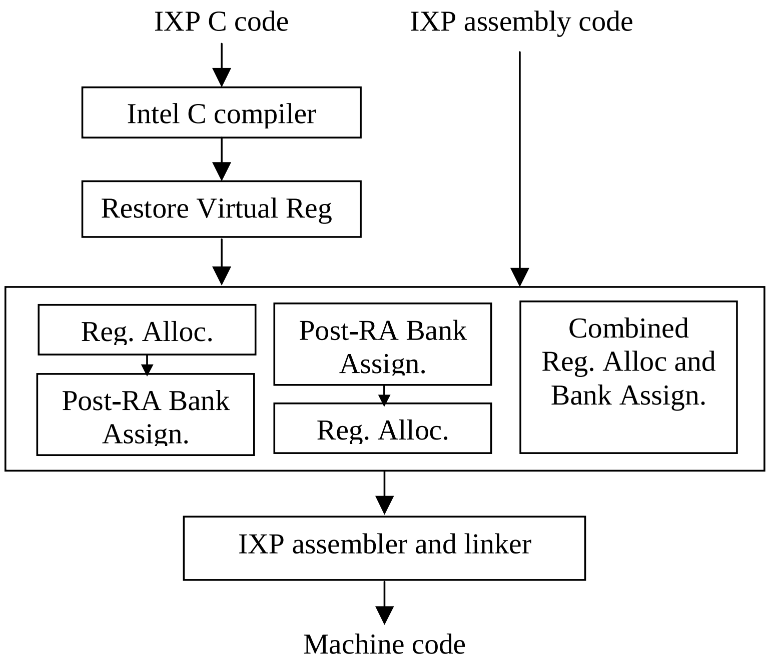


Figure 17: Compilation flowchart

3.8 Experimental Results

We evaluate the algorithms with the Intel-provided IXP1200 Developer Workbench 2.01. The IXP1200 workbench supports cycle-accurate simulation for IXP microengines and other peripherals with high fidelity. It provides both assembler and a C compiler supporting a subset of ANSI C.

We experimented with 8 benchmark programs to see the effectiveness of the three approaches. These benchmarks are collected from Commbench [72], Netbench [51], and a packet scheduling algorithm from [75]. The benchmark programs are rewritten in IXP C code and a few of them are directly written in assembly (micro-code). For the assembly code generated by the C compiler, we restore all virtual registers. Figure 17 shows the flowchart of the compilation process. The register allocation and bank assignment pass have three modules, i.e. Post-RA, Pre-RA and combined-RA. The general register alloator is the one proposed by Briggs et.al. [8]. Our pass builds the CFG, IG and RCG from the assembly code, after simple translation of the assembly directives. The IXP assembly consists of only 40 RISC instructions, which makes the translation easy. For one thread, the number of

Table 1: Benchmark applications

	Code Size	#live ranges	#interference edges	#conflict edges
Drr	108	11	55	25
Fir2dim	447	36	120	71
Frag	271	26	133	65
Kmp	123	13	53	27
Lzw	126	18	105	36
Md5	913	142	630	246
Wraps (receive)	875	145	643	236
Wraps (send)	921	135	464	193

total physical registers is 32 (the benchmarks are assumed to be run on only one thread). Therefore, 16 registers are available in each bank.

Table 1 shows the properties of the benchmark programs. The code size is the number of instructions after code generation. The number of live ranges and interference edges are listed in 3rd and 4th column. On average, the degree of the live ranges on the interference graph is about 9. The last column shows the number of conflict edges. Conflict edges are much less than interference edges. Only a small fraction of instructions become conflict edges, because only ALU instructions with two source GPR operands establish conflict edges in the RCG. Among these instructions, some conflict edges are identical.

Table 2 shows the cycle length distribution. We separate columns into two categories. Table 2.a shows the distribution of cycle length before the two enabling techniques in Section 3.7 are applied. Table 2.b are the distribution after these techniques are used. We apply the enabling techniques before register allocation and bank assignment. From Table 2, we find most cycles are of length 3. Cycles with length greater than or equal to 7 are rare. The techniques in Section 3.7 seem to have limited effects, especially for larger benchmarks.

Table 3 gives the number of instructions inserted to apply the patterns (we do not included instructions for spills, this will be counted in the next table) to break odd cycles and balance the banks. They include ”move insertion” and ”in-place exchange” etc. for post-RA bank assignments. The results signify that post-RA adds more instructions than the other two. This is due to the more ambitious conflict breaking attempted by this stage

Table 2: Cycle length distribution

	Without algebraic law & triangle conflict removal		
	Length-3	Length-5	Length ? 7
Drr	7	1	0
Fir2dim	48	2	0
Frag	51	3	1
Kmp	8	0	0
Lzw	49	4	0
Md5	836	13	2
Wraps (receive)	1132	19	3
Wraps (send)	842	10	0

(a)

	With algebraic law & triangle conflict removal		
	Length-3	Length-5	Length ? 7
Drr	7	1	0
Fir2dim	48	2	0
Frag	49	3	1
Kmp	8	0	0
Lzw	44	4	0
Md5	827	12	1
Wraps (receive)	1101	17	3
Wraps (send)	840	10	0

(b)

Table 3: Comparison for number of inserted instructions

	Pre-RA	Post-RA	Combined-RA
Drr	3	5	5
Fir2dim	10	18	11
Frag	8	20	8
Kmp	3	8	4
Lzw	4	10	3
Md5	38	59	19
Wraps (receive)	35	62	23
Wraps (send)	29	55	21

Table 4: Comparison for number of spills

	Pre-RA	Post-RA	Combined-RA
Drr	2	0	4
Fir2dim	5	0	7
Frag	3	0	8
Kmp	2	0	2
Lzw	2	0	5
Md5	30	23	35
Wraps (receive)	45	38	56
Wraps (send)	52	38	57

adding many instructions. Since virtual registers have been allocated physical registers, more odd cycles may result. Nodes on the physical RCG are more costly to split, because they represent several nodes for virtual live ranges. A combined-RA approach tends to generate fewer additional instructions, however, as we will see later, more spills are created.

Table 4 gives the number of spills generated by each approach. The combined-RA is worst, since the graph coloring works poorly when two stacks are assumed. Many nodes are marked as spill when pushing to the stack because the number of neighbors they have on the graph is larger than the number of physical registers in one register bank, but actually their neighbors on the graph may finally go to the opposite bank, which is not known at the time they are pushed onto the stack. Post-RA is the best in reducing number of spills (since it assumes one bank when doing RA), which compensates the increased number of additional instructions due to the high cost of spills.

The compilation time for all benchmarks is within 1 second on a Pentium 4 machine. Obviously, the combined-RA approach is polynomial time algorithm. For pre-RA and post-RA algorithm, the majority of the compilation time is spent on cycle breaking. As mentioned in Section 3.4.3, the complexity of the cycle breaking is $O(n \times P \times M)$. However, if the outmost loop can finish early, the complexity is close to $O(P \times M)$. Since we have set the bound for M, the complexity is further controlled by $O(P) \times Max(M)$. Finally, P is the maximum among the numbers of different odd-length cycles. Normally, this is the number of length-3 cycles on the RCG, which should be in $O(n^3)$.

In conclusion, our Post-RA bank assignment is successful in breaking odd cycles and balancing banks without increasing spills. The extra instructions cannot offset the benefits of spill reduction. Pre-RA generates more spills than Post-RA but less than the current version of the combined-RA.

CHAPTER IV

INTER-THREAD REGISTER ALLOCATION

The previous chapter focuses on the register allocation problem for a single thread. Note that, the entire register file of each micro-engine is shared by all threads. Therefore, it is possible to allocate registers also across threads.

4.1 Special Features Regarding Multithreading

As shown in Figure 18, typically, each microengine *(PU)* gets packets from its input queues, processes it and then writes to its output queues, or the input queues of the next PU in the next pipeline stage. With pipeline processing, typically, some PUs are in charge of getting packets from the input ports; some handle packet processing and some are for output ports.

Our optimization focuses on the code across different threads of the same PU. Some of the important features are as follows:

1. *Shared register file but typically non-overlapped partitions.* The general purpose register (GPR) file is shared by the 4 threads. Each thread has access to all registers; however without optimization, each thread is normally allocated non-overlapping part of the register file. The main reason that the register file should be partitioned results from light-weighted context switch as discussed below.

2. *Non-preemptable thread execution.* There is no operating system, no control present over the threads sharing the CPU. A thread gives up the CPU only when it blocks on I/O or other long latency operation or executes a context switch (ctx_switch) instruction voluntarily[1].

3. *Light-weighted context switch.* Context switch is cheap (only PC is saved), this is also the reason registers are normally allocated in a non-overlapped fashion from the

[1]ctx_switch instruction can be inserted by the programmer to achieve fair sharing of the CPU.

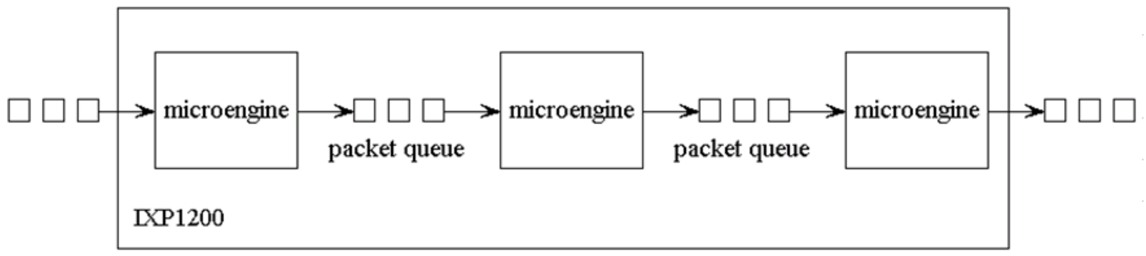


Figure 18: Pipeline

register file. If a register is allocated to two threads, after context switch, the content in that register may be modified by the other thread. Since registers are neither automatically saved nor restored during a context switch such possibilities exist and this is where it becomes a compiler problem to manage registers.

4. *Cheap ALU, expensive memory access.* No cache is available for memory accesses; at least 20 cycles are needed for each load/store instruction. Context switches are typically followed to hide the long latency of memory accesses. In contrast, all ALU instructions can be completed in 1 cycle. Large memory latency makes the overall performance sensitive to spills even though they may be few in number.

The above features of the IXP network processor are driven by design philosophy to simplify hardware so as to increase the clock rate and execution speed. For instance, context switch is kept very simple and fast (1 cycle latency). Therefore, only program counter (pc) is saved but no registers are saved because it can cause long delay in context switch which may offset the benefits of CPU sharing. On the other hand, since all the hardware details are exposed, compiler can make prudent decisions regarding register sharing etc. Next, we propose the multi-threaded register allocation problem.

4.2 The Register Allocation Problem

As mentioned above, although the register file can be accessed by all threads, it has to be partitioned without overlap across threads because no register is saved/restored during context switches. Here, we argue that some registers can be safely shared by all threads through compiler analysis since thread switch is predictable

Thread 1

```
     1.  a=…
     2.  ctx_switc
         h
     3.  if(…)br
         L1
     4.  b=…
     5.  …=a+b
     6.  c=…
     7.  br L2
L1:
     8.  c=…
     9.  …=a+c
     10. b=…
L2:
```

Thread 2

```
1.  ctx_switch
2.  d=…
3.  …=d+…
4.  store…
```

(a)

Thread 1

```
1.  r1=…
2.  ctx_switch
3.  if(…)br L1
4.  r2=…
5.  …=r1+r2
6.  r3=…
7.  br L2
L1:
8.  r3=…
9.  …=r1+r3
10. r2=…
L2:
11. …=r2+r3
12. load…
```

Thread 2

```
1.  ctx_switch
2.  r2=…
3.  …=r2+…
4.  store…
```

(b)

Thread 1

```
1.  r1=…
2.  ctx_switch
3.  if(…)br L1
4.  r2=…
5.  …=r1+r2
6.  r1(r3)=…
7.  br L2
L1:
8.  r2(r3)=…
9.  …=r1+r2(r3)
10. r1(r3)=r2(r3)*
11. r2=…
L2:
12. …=r2+r1(r3)
13. load…
```

(c)

Figure 19: Example of register sharing and move insertion

The example in Figure 19 illustrates the problem and possible ways to solve it. In Figure 19.a shows code for the two threads. Assume all variables are dead after their last use in the code. In thread 1, a code segment contains 12 instructions, including two context switch instructions–ctx_switch gives up CPU voluntarily and a load causes context switch to wait for I/O operation. Any pair of the 3 variables interferes with each other (co-live at some program points), so in Figure 19.b, they are assigned 3 different physical registers. Notice that variable a is live across ctx_switch instruction, so it must be allocated to a physical register that is not used by any other thread, because when thread 1 is context switched at this point, other thread should not modify the physical register of variable a, which means only thread 1 should use the register. On the contrary, variable b and c are only used between two context switch instructions. In other word, when thread 1 is switched out, both b and c must be dead. Therefore, it is safe to reuse the physical registers allocated to b and c in other threads. Thread 2 has 4 instructions, with two context switch instructions. d is only live between two context switch instructions, therefore d can share a physical registers with other threads. Simply, r2 is shared - used for b in thread 1 and d in thread 2, because the code guarantees that when context is switched to thread 2, r2 contains a dead value (b) for thread 1. Similarly, when context is switched to thread 1, r2 contains a dead value (d) in thread 2. This example shows the benefit of sharing registers and lowering total register requirements from four to three. We now illustrate that through another technique (live range splitting), one can reduce the total register requirement further.

Three registers seem necessary for thread 1, however we notice that at any program point, only two variables are co-live. This prompts our technique of splitting one of the variables and inserting a move instruction at certain point. This is demonstrated in Figure 19.c. In instruction 6, r3 is replaced by r1, while from instruction 8 to 9, r3 is replaced by r2. Instruction 10 copies r2 to r1, so in instruction 12, we have a consistent replacement ($r3 \rightarrow r1$). We have managed to reduce total register requirements down to two now.

The above example illustrates the potential benefits of register sharing across threads and live range splitting. To further justify that multi-threaded register allocation is important, and a compiler solution is feasible, we list some properties of the programs that run on the networks to support this argument.

1. For IXP1200, the hardware provides seemingly enough registers. 128 general purpose registers (GPRs) can be used for each PU. However, for each thread, only 32 GPRs are available if no GPR is shared across threads. Register sharing on IXP is a software-only solution, unlike some SMTs (Simultaneous Multi-threading) where it is hardware managed. Compiler designates and allocates a register either as a shared or private one.

2. Since there is no operating system to manage threads, memory access, context switch etc. are all explicit and thus context switch is predictable at compilation time.

3. As shown in our experiments, context switch instructions are typically less than 10% of the total instructions and many variables are not live across context switch instructions.

4. PUs are assigned with different tasks. Packets are processed in pipeline fashion. Currently, task assignment cannot be done automatically. Although in most cases,

the same task is assigned to threads on the same microengine. This actually leads to low utilization of the CPU, because it is hard to chop tasks properly so that they all take roughly 1/4 of the computation power of the PU. Therefore, we should assume tasks might be different for threads on the same PU.

Item 1 indicates that registers may not be sufficient on the network processor. Item 2 and 3 support the feasibility of a compiler solution to optimize register allocation. Finally, item 4 prompts two kinds of problems, i.e. symmetric vs. asymmetric register allocation, which will be defined in next section.

4.3 Preliminaries

4.3.1 System Model

We study a multithreaded network processor that can run multiple threads on a single processing unit. The threads on one PU share the computation power of the PU and register files etc. Formally, the model is as follows:

1. There are totally N_{reg} registers that can be used by N_{thd} threads sharing a single PU.
2. Explicit context switch. A thread does not give up the CPU once it starts execution on it, until a context switch instruction is met. Context switch can happen due to explicit instruction or long latency instructions like a load or a store.
3. Context switch is very cheap (only PC is saved) and it is intended to hide long latency operations.
4. The purpose of multithreading on the same PU is mainly for latency hiding and concurrency. When one thread is stalled due to I/O or other long latency operations, other thread can take the CPU. Therefore, code on different threads are almost independent. Thread communication or synchronization rarely happens, however, our current solutions still works under such circumstances. As a future work, knowledge about thread communication or synchronization might be exploited to improve the register allocator.

5. All registers are accessible by all threads, but registers used by one thread at the point of context switch should not be used anywhere by other threads (later, we will define these registers as *private registers*), because this might cause unexpected modification to registers and lead to unsafe code.

6. Move instruction is much cheaper than spill.

7. Code on different threads of the same PU can be different.

4.3.2 Problem Classification

As mentioned earlier, programs executing on different threads can be identical. We call the register allocation problem under such circumstances *Symmetric Register Allocation (SRA)*. On the contrary, *Asymmetric Register Allocation (ARA)* assumes different programs for different threads. Mixing threads with different computation requirements can achieve better CPU utilization. Since SRA is a sub-problem of ARA, in this paper, we develop our approaches based on ARA. Notice that, although currently most real programs are for SRA, we are not intentionally complicating the problem, because our algorithms are equally necessary and important to SRA, as will be illustrated later, SRA only reduces searching space during inter-thread register allocation, while all techniques in this paper are applicable to both problems. Our goal is to develop general techniques that can apply without undue restrictions.

4.3.3 Objectives

The number of total available registers is limited. Therefore, in a multithreaded network processor model, we aim to (for ARA) balance register allocation among all threads, so that more registers are allocated to the thread with higher register pressure, and register allocation is catered to the requirements of different threads in the system. Furthermore, designating a larger number of *shared registers* can help all threads to internally adjust their register pressures without causing spills.

In case there are not enough registers available for all threads, we attempt to split the live ranges inside a thread by using move instructions. Also, our objective is to minimize the number of move instructions inserted. The results show that move insertion is cheap and effective.

4.3.4 Problem Formulation

To formalize the problem, we define several concepts.

Definition 3. PR_i

Number of private registers for thread i, these are physical registers only (exclusively) used by thread i.

Definition 4. SR_i

Number of shared registers needed by thread i, these are physical registers used by thread i, but other thread may use them as well.

Definition 5. R_i

Number of total physical registers needed by thread i, equals $PR_i + SR_i$.

Definition 6. SGR

Number of globally shared registers needed, it is the maximum of shared register demands of each thread, since shared registers can be used by all threads, this is the maximum of all SRs.

Definition 7. N_{reg}

Total number of physical registers available on a PU.

For a thread, PR is the number of physical registers that are exclusively allocated to it or the number of physical registers that can be live across context switch instructions, while SR is the number of allocated physical registers that are dead during context switches, which means they can be shared across threads. For example, in Figure 19.b, for thread 1, $PR_1 = 1$, $SR_1 = 2$, for thread 2, $PR_2 = 0$, $SR_2 = 1$, therefore, SGR=2.

The relationship and restrictions among these variables are illustrated as the following conditions:

1. $SGR = Max(SR_1, SR_2 ... SR_{N_{thd}})$

2. $\sum_i PR_i + SGR \leq N_{reg}$

3. $PR_i + SR_i = R_i$

4. For SRA, all PR_i's and SR_i's are equal.

Given these restrictions, we need to assign registers in a way that the overall register need is satisfied and spills are minimized.

4.4 Construction of the Interference Graph

4.4.1 Non-Switch Region

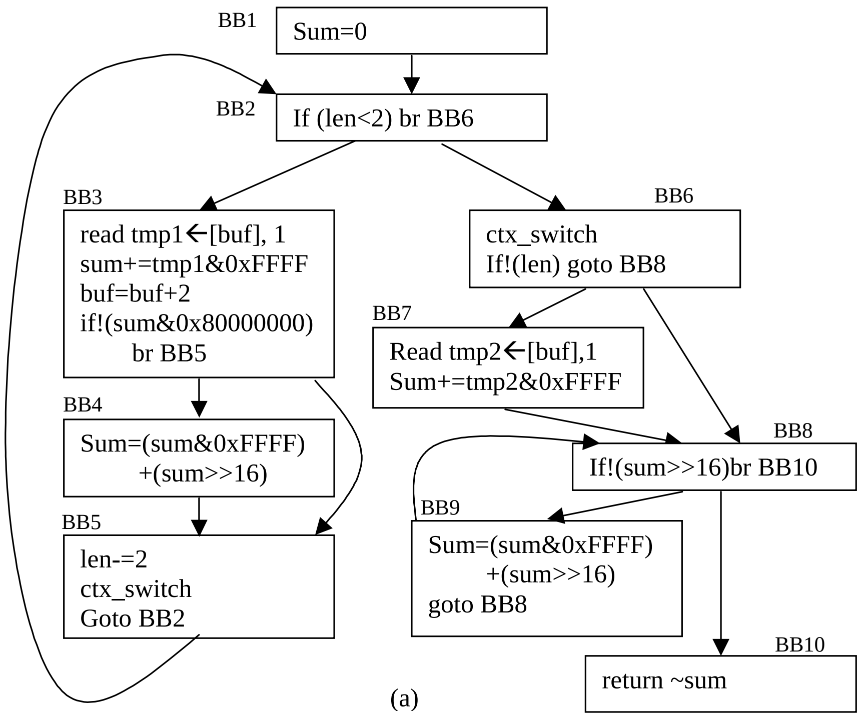
Definition 8. *Non-Switch Region (NSR)*
A non-switch region is a maximal connected sub-graph of the CFG without any internal context switch instructions. It contains connected parts from several basic blocks. The boundaries of the NSR are either context switch instructions or program entry/exit points.

Definition 9. *Context Switch Boundary (CSB)*
The program point of a context switch instruction. A CSB separates the basic block it resides, thus becomes the boundary of NSR(s).

A NSR can be constructed by starting from an individual instruction and grown it until all nearby instructions are either context switch instruction or program entry/exit points.

To illustrate, Figure 20.a shows the CFG and NSR for a code segment from benchmark "frag" in the Commbench suite [72]. This code segment is from one of the functions to calculate the IP checksum. The CFG consists of 10 basic blocks. Noticeably, there are four context switch instructions, i.e. the read instructions in BB3 and BB7, the explicit ctx_switch instructions in BB5 and BB6. The ctx_switch instructions are inserted by the programmer to avoid CPU monopoly.

Figure 20.b shows the NSRs. After terminating the CFG at the points of context switch instructions (boundaries), we get 3 NSRs. These NSRs are bound by either program entry/exit points or context switch instructions (CSBs). We can assume all terminating are inside basic blocks, therefore some basic blocks are split, like BB5 is split into BB5.a



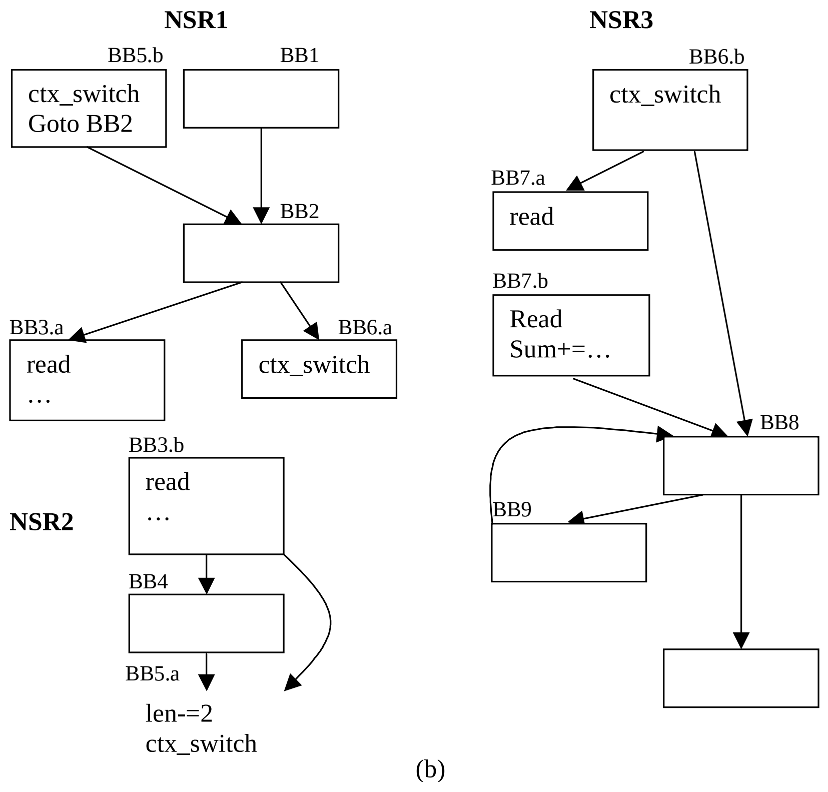


Figure 20: Program CFG and the constructed NSR

in NSR2 and BB5.b in NSR1. Sometimes, two parts of a separated basic block still belong to the same NSR like the BB7 in Figure 20. For the example in Figure 19, thread 1 has two NSRs, instruction 1 and 2 are in NSR1 and instructions 2 to 12 constitute NSR2. For thread 2, all instructions form one NSR.

4.4.2 Interference Graphs

After building the NSR, we then build the interference graph, which will guide register requirement estimation and register allocation. We need to distinguish two kinds of interferences and introduce some other definitions for the interference graph.

Definition 10. *Node*

Live range of a virtual register or variable[2]

Definition 11. *Boundary Node*

Node that is live across the CSB, which may interfere with other boundary nodes.

Definition 12. *Internal Node*

Node that is not live across any CSB.

Definition 13. *Boundary Interference*

If two boundary nodes are co-live across the same CSB, they are said to be boundary interfering with each other.

Definition 14. *Internal Interference*

If two nodes (internal or boundary nodes) interfere (co-live at a program point) within a NSR.

Definition 15. *Boundary Interference Graph (BIG)*

A graph consists of all boundary nodes and edges only representing boundary interference.

Definition 16. *Internal Interference Graph (IIG)*

For each NSR, we have an IIG, which only includes the internal nodes live within this NSR and their interference edges.

[2]Here, we assume each live range represents one variable.

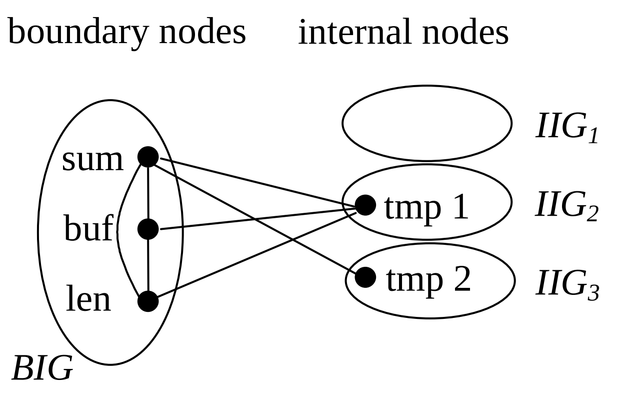


Figure 21: Global interference graph for the example

Definition 17. *Global Interference Graph (GIG)*
The global interference graph includes both boundary nodes and internal nodes. An edge is added if any two nodes (internal or boundary) interfere with each other.

The GIG of the code for the example in Figure 20 is drawn in Figure 21. We assume both len and buf are live at the entry point as the length and the buffer pointer of the packet are calculated. Also, we assume all variables are dead after their last use in the code. From Figure 20.b, we can see both variable tmp1 and tmp2 are only live within an NSR, so they are internal nodes. Other variables are live across CSB boundaries. They are boundary nodes. For memory read, since all data is first loaded into *transfer registers*, the destination register is not assumed to be live across the memory read i.e. the CSB. At BB1, sum, buf and len interfere with each other internally (they also interfere at CSB), thus, the 3 nodes form a clique on the GIG. tmp1 interferes with sum, buf and len in BB3.b, but at the live point of tmp2 in BB7.b, both buf and len are dead. Thus, sum, buf and len form a BIG; the IIG_1 for NSR1 is empty; the IIG_2 for NSR2 includes only tmp1, the IIG_3 for NSR3 includes only tmp2.

Obviously, we have the following claims for each thread.

Claim. *To avoid spills, the GIG should be colored with R colors and the BIG should be colored with PR colors. Each IIG, as a part of the GIG, should be colored with no more than R colors.*

Claim. *Internal nodes on different IIGs are not connected i.e. they do not interfere with*

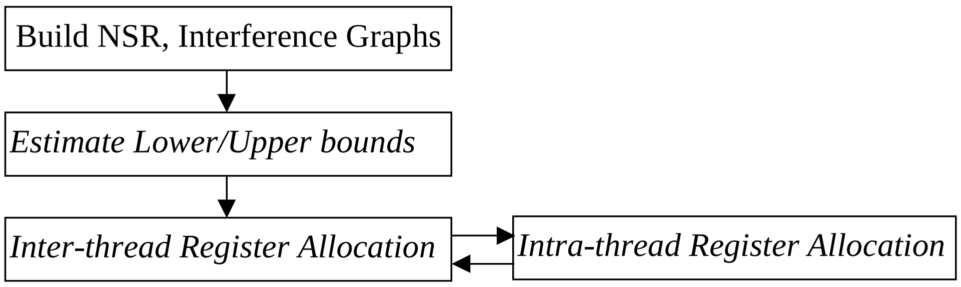

Build NSR, Interference Graphs
Estimate Lower/Upper bounds
Inter-thread Register Allocation
Intra-thread Register Allocation

Figure 22: Overall framework

each other.

Notice that, NSRs and interference graphs can be constructed inter-procedurally. CFGs and NSRs of different functions are connected with edges linking function calls and return points.

4.5 Overall Framework

Figure 22 shows our framework to perform register allocation. Our first step is to build NSR and interference graphs, we then try to estimate the lower and upper bound of PR and R for each thread. Starting from the upper bound the inter-thread register allocator reduces the overall register requirement gradually until it is within N_{reg}. During this process, when the inter-thread register allocator intends to reduce PR or SR, it calls the intra-thread allocators for all threads. The inter-thread allocator goes towards the direction of the smallest cost increase. The framework allows the intra-thread register allocator to be built separately from the inter-thread register allocator.

4.6 Register Number Estimation

As the first step toward assigning registers to multiple threads, we need to estimate the number of registers each thread needs based on the interference graph. The estimation helps guide the distribution of registers to threads at the beginning. Here, we are concerned with finding the bounds for R and PR as defined below. We do not estimate bounds for SR, since the number of SR is always equal to R-PR.

Definition 18. *MinPR, MaxPR*

Minimal, maximal number of PR

Definition 19. *MinR,MaxR*

Minimal, maximal number of R

4.6.1 Lower Bound Estimation

The lower bound is the minimum number of registers a thread needs. First we can get an estimation for the minimum number of private registers (MinPR) one thread needs. A rough estimation is $MinPR \geq RegPCSB_{max} =$ Max(number of co-live registers at CSBs).

It is obvious that if at a CSB point, there are $RegPCSB_{max}$ nodes (variables) co-live, we need at least this number of private registers since they cannot be shared during context switch. In other words, the minimal number of private registers needed is at least equal to the maximal number of nodes co-live at the CSB boundaries.

The following lemma says that this bound can be reached if enough move instructions are inserted. Also, we will explain more about move instruction insertion later.

Lemma 4. *Regardless of shared registers, MinPR can be made equal to $RegPCSB_{max}$ by inserting move instructions.*

Proof: If we are given private registers $PR_1, PR_2 \ldots PR_{RegPCSB_{max}}$, and at a certain CSB, there are $V_1, V_2 \ldots V_n$, totally n variables live across, $RegPCSB_{max} \geq n$. Simply, inserting n move instructions $PR_1 = V_1, PR_2 = V_2, \ldots PR_n = V_n$ before the CSB and n move instructions $V_1 = PR_1, V_2 = PR_2, \ldots V_n = PR_n$ after the CSB can make the code equivalent to the original and the number of private registers needed is no more than $RegPCSB_{max}$.

However, in reality, move instruction still costs 1 cycle in our model, although it is much cheaper than spill, we still need to keep the number of inserted move instructions small. Similarly, we can estimate the MinR needed.

$MinR \geq RegP_{max} =$ Max(number of co-live registers at program points)

This lower bound is also achievable given enough move instructions. The proof is similar to the one above.

4.6.2 Upper Bound Estimation

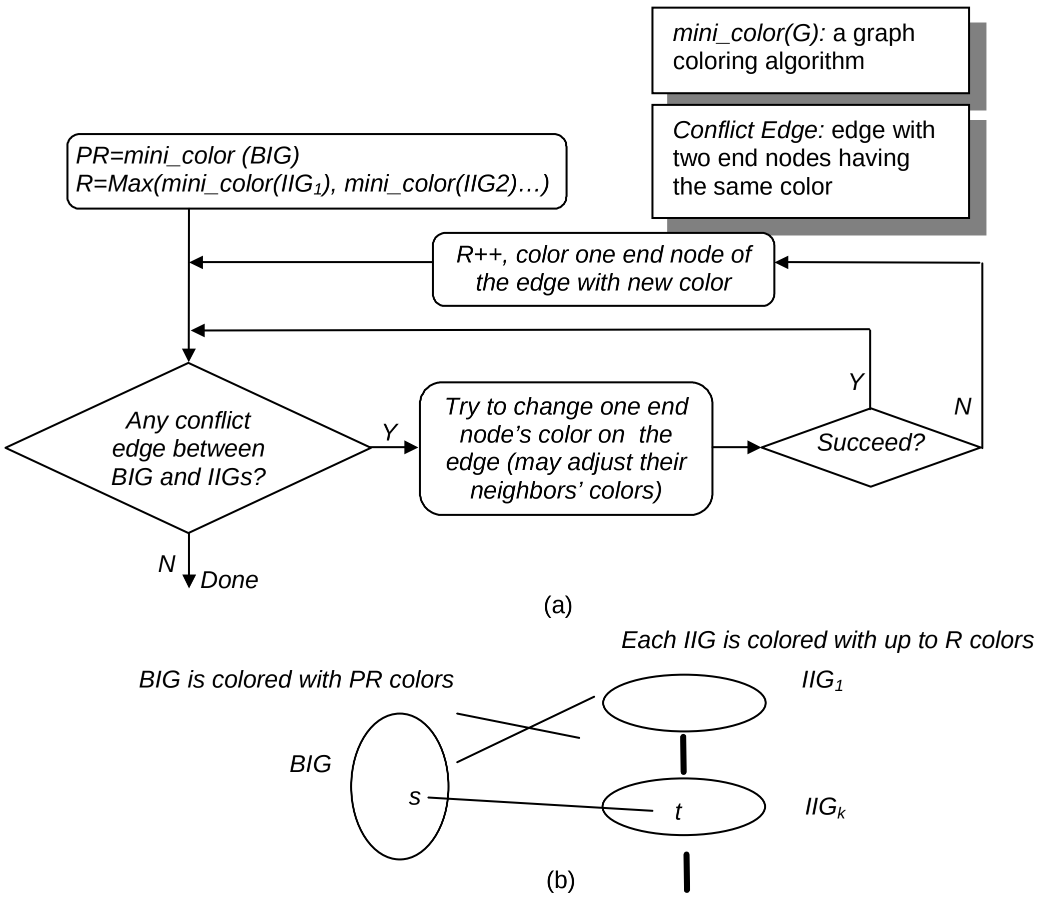
The upper bound gives the maximal number of registers required without any extra move instructions inserted. According to the first claim in section 4.4.2, the best estimation for MaxPR and MaxR is the minimal number of colors required to color BIG and GIG. However, for GIG the coloring problem is slightly different from the traditional graph coloring. In summary, the problem is to find a coloring scheme for a thread which satisfies:

1. All boundary nodes are colored with at most MaxPR colors
2. All nodes are MaxR colorable
3. Any two interfering nodes are colored differently

For the GIG in Figure 21, all boundary nodes can be minimally colored with 3 colors; thus, MaxPR=3. And, all nodes can be minimally colored with 4 colors (there is one 4-node clique), so MaxR=4 $\rightarrow$ SR=1.

Actually, there is a tradeoff between MaxPR and MaxR estimation. Reducing MaxPR may induce a larger MaxR. To minimize MaxPR, we can first remove all internal nodes and color the BIG minimally, then insert back the internal nodes and color the graph assuming all boundary nodes have fixed color. To find the tighest (minimal) value of MaxR, we should ignore the condition 1 above, i.e. we could assume that all nodes are indistinguishable and we could simply color the GIG as usual using any coloring allocator. Such a coloring would then minimize MaxR but may give a higher MaxPR.

We take an approach slightly different from the first one, i.e. we minimize the MaxPR first. This approach is motivated by the fact that increasing PR causes direct increase in the total number of registers, while increasing SR only affects the total number of registers when this SR is the maximum among all threads. Based on the second claim mentioned in Section 4.4.2, (i.e. IIGs are not connected with each other) we can color IIGs and BIG separately and then merge them together to keep a tight control on colorability. After merging, edges added between BIG and IIGs may cause conflicts. For example, in Figure 21,when IIGs and BIG are colored separately, variable sum may get the same color as tmp1, leading



mini_color(G): a graph coloring algorithm
Conflict Edge: edge with two end nodes having the same color
PR=mini_color (BIG)
R=Max(mini_color(IIG1), mini_color(IIG2)…)
R++, color one end node of the edge with new color
Y
N
Any conflict edge between BIG and IIGs?
Y
Try to change one end node's color on the edge (may adjust their neighbors' colors)
Succeed?
N
Done
(a)
Each IIG is colored with up to R colors
BIG is colored with PR colors
IIG1
BIG
s
t
IIGk
(b)

Figure 23: Estimate the maximal register requirements

to color conflict when the edge between them is added during the merge. A general algorithm to color the whole graph altogether may take much more time, since the graph can be big (it includes all live ranges in the program. Some code in our experiment contains hundreds of nodes). Our approach is similar to the fusion-based or region-based register allocation [11], except that our regions are chosen as the IIGs and BIG. The algorithm (Figure 23.a) first builds BIG and IIGs from the GIG and colors each of them independently. In other words, the BIG is colored with color number from 1 to PR, while each IIG is colored with color number from 1 up to R. Some IIGs may be colored with less than R colors, but an IIG can be colored with at most R colors.

The next step tries to merge each IIG with the BIG. The edges between IIG and BIG can cause problems if the two end nodes of an edge have the same color. Such edges are called *Conflict Edges*. The loop in Figure 23.a shows how to resolve all the conflict edges.

We illustrate the procedure in Figure 23.b. Suppose boundary node s and internal node t is colored with the same color. If s's color can be changed to another color within color number 1 to PR or t's color can be changed to another color within color number 1 to R, then one of them can be changed to another color to remove this conflict edge. If that fails, we heuristically try to change their neighbors' colors to see if the two nodes can be recolored after that. After all these attempts fail, we have to increase R and t is re-colored with the new color. The algorithm gives MaxPR and MaxR finally. The complexity of the algorithm is $\sum O(mini\_color(IIG_i)) + O(mini\_color(BIG)) + O($#Edge between BIG and IIGs). In contrast, the complexity to color the whole graph is $O(mini\_color(GIG))$. This means the algorithm is also quite fast to try out a given coloring for a thread.

4.7 Interthread Register Allocation

4.7.1 Our Approach

One of the difficulties in register allocation for multiple threads is that we do not know exactly how many registers each thread needs. Trying all combinations to find out the best register allocation will cause tremendous amount of compilation time and will be infeasible to build into any practical system. Our approach is to first get an estimation (range) of how many registers are needed by individual thread via the algorithm proposed in the previous section. From this starting point, we use a greedy heuristic algorithm to approach a sub-optimal solution by reducing the total number of required physical registers gradually. The algorithm also encapsulates the intra-thread register allocator, so that it can be developed independently.

4.7.2 The Register Allocation Algorithm

After getting the estimated upper bounds $MaxPR_i$ and $MaxR_i$ for each thread, Let $SR_i = MaxR_i - MaxPR_i$ and $PR_i = MaxPR_i$. We can check the following condition:

$$\sum_i PR_i + Max(S_1, S_2 ... S_{N_{thd}}) \leq N_{reg} (**)$$

If this holds, we can assign $SGR = Max(S_1, S_2 ... S_{N_{thd}})$ as the number of globally shared

```
INPUT:   $N_{thd}$, $N_{reg}$, CFGs of all threads
OUTPUT:  all $PR_i$ and $SR_i$, SGR, CFGs after register allocation

/*Intra-thread register allocator, returns move cost*/
Intra_thd_allocator(CFG, GIG, PR, SR);

ALGORITHM: Inter_thd_reg_allocation
    1.  Build_GIG()
    2.
    3.  Estimate_reg_requirement()
    4.
    5.  While(sum($PR_i$)+max($SR_1$,$SR_2$…$SR_{Nthd}$)>$N_{reg}$)
    6.      Foreach $PR_i$>Min$PR_i$ and $PR_i$+$SR_i$>Min$R_i$ do
    7.          cost_$PR_i$=Register allocation cost after reducing $PR_i$
        by 1.
    8.      od
    9.
    10.     max_SR=max($SR_1$, $SR_2$…$SR_{Nthd}$)
    11.     cost_SR= Register allocation cost after reducing all SRs
    12.                 that equal max_SR by 1, if all such SRs can
    13.                 be reduced(by checking PR+SR>MinR)
    14.     Find the min one among cost_SR, cost_$PR_1$,cost_$PR_2$…
    15.     Choose the one with minimal cost, modify PRs and SRs.
    16. Endw
    17.
    18. Actually modify the CFGs based on new PRs and SRs
    19. SGR= Max($SR_i$)
    20. Return all $PR_i$ and $SR_i$, SGR, all CFGs
```

Figure 24: Algorithm for inter-thread register allocation

registers and $MaxPR_i$ as the number of private registers for each thread to satisfy all register requirements. If the above condition (**) cannot hold good, the register requirement is too high. We must either reduce the PR(s) or SR(s) to satisfy (**).

From (**), we can see, there are two ways to reduce the left side value. Either we can reduce one of the PR_i, which will result in direct reduction of the left-side value, or to reduce SR_i. We should reduce the one(s) with the maximal value. In case multiple SR_i's have the same maximal value, we should consider reducing one of the PR_i if that costs less. The inter-thread register allocation algorithm is shown in Figure 24.

The algorithm first builds GIG and gets the estimations for each thread. If the needed registers are enough (less than N_{reg}), the program simply allocates register and return. Otherwise, it enters a loop to gradually reduce the number of overall register requirement through a greedy algorithm, i.e. every time we choose a direction that can achieve the minimal cost. To reduce the register requirement (i.e. the left side of (**)) by 1, we have many choices. Either we can reduce one of the PRs by 1 or reduce all the maximal SR(s) by 1 to cut down $Max(SR_1, SR_2...SR_{N_{thd}})$. Every time the PR in one thread is reduced, we check if it is larger than the lower bound. Also, the lower bound of $R_i = PR_i + SR_i >= MinR_i$ is verified when either PR_i or SR_i is reduced.

The function *Intra_thd_allocator* is an intra-thread register allocator. It accepts the PR and SR, then tries to return an allocation using PR and SR number of registers. This function is called when we calculate register allocation cost for each thread and when we finally modify the CFGs. It returns the allocation cost. Actually, the interference graph and coloring scheme given by the function *Estimate_reg_requirement* can be passed to the intra-thread register allocator as a starting point. However, to provide more flexibility, we leave this to the implementation of *Intra_thd_allocacor*.

The complexity of our heuristic algorithm is $O(N_{reg} \times N_{thd}) \times O(Intra\_thd\_allocator)$, which largely depends on the complexity of the intra-thread register allocator. Our register allocation algorithm generates satisfactory solution for all benchmark programs within almost negligible compilation time.

4.8 Intrathread Register Allocation

The intra-thread register allocator attempts to allocate up to PR number of physical registers to boundary nodes and up to R=PR+SR physical registers to all nodes.

4.8.1 Move Insertion and Live Range Splitting

Our intra-register allocation is based on live range splitting and move instruction insertion. Live range splitting has been used in register allocation [17] to spill part of the live range to memory. In this paper, we attempt to split the live ranges by inserting move instructions to reduce the chromatic number. Lemma 1 has shown that through live range splitting MinPR can be reached. Figure 25 gives another example. In Figure 25.a, live ranges A B and C interfere with each other at three different CSB points. The lower bound lemma in Section 4.4 gives MinPR=2, but the interference graph must be colored with 3 colors, because A,B, and C form a clique. In Figure 25.b we split the live range of variable A into A1 and A2 by inserting move instruction at the split point. The resulting interference

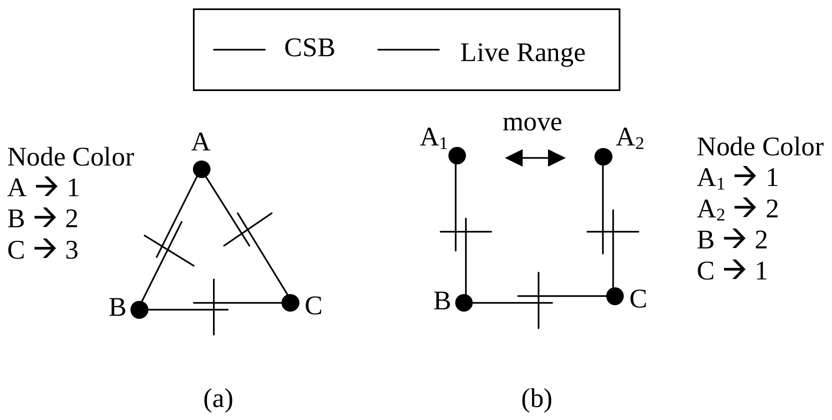


Figure 25: Live range splitting via move insertion

graph can be colored with 2 colors which is equal to MinPR. Notice that, this is also the way we reduce the number of registers required in the first example (Figure 19.c).

In our intra-thread allocation algorithm, we focus on live range splitting through move insertion because spill is too expensive on network processor and our experiments show MinPR (MinR) is much smaller than MaxPR (MaxR). This provides us room to reduce chromatic number toward the lower bound by inserting move instructions.

4.8.2 Intra-thread Register Allocation Algorithm

Our register allocator works incrementally, i.e. it records the *context* (interference graph with split nodes and the position of move instructions) of the last 2 invocations and modifies the context to satisfy the new PR and SR values. Notice that the inter-thread allocation algorithm in Figure 24 calls *Intra_thd_allocator* multiple times. In each step, either it accepts the previous context and reduces PR or SR by 1 or it rejects the previous modification and starts from the previous previous context and reduces PR or SR by 1. Incremental modification can save time for otherwise repetitive work. Further, based on the records of the two contexts, we can assume that each time the allocator is invoked, it attempts to reduce either PR or SR by 1 from one of the recorded contexts. We name these two kinds of invocation as *Reduce-PR invocation* and *Reduce-SR invocation.*

4.8.2.1 Reduce-PR Invocation

In this type of invocation, the allocator wants to reduce the PR by one from its last invocation. In other words, the last accepted context can color all boundary nodes with PR colors and this invocation wants to color it with PR-1 colors.

In this stage, we assume all move instructions are inserted near the CSB. With this assumption, we do not need to alter the colors of internal nodes. Normally, changing the color of both internal and boundary nodes might induce more move instructions (in this case, we must split the live range to recolor an internal nodes) and increase the cost accordingly. Later, we will show that some of the move instructions at the CSB can be eliminated by merging them with move instructions inside the NSR. This actually relocates the move instructions from the CSB boundary. Before discussing our algorithm, we first define *Neighbor Color Number (NCN).*

Definition 20. *Neighbor Color Number (NCN)*
The number of colors used by the neighbors of a given node in a colored graph.

The algorithm in Figure 26 uses function $NCN(t, BIG)$ to get the neighbor color number of node t on the BIG. The algorithm also works in a greedy manner. It tries each color c in PR colors and checks the cost to eliminate that color. Then, the color with least elimination cost is selected to be eliminated and all needed move instructions are inserted. Function $Set\_color\_node(c, BIG)$ returns the set of nodes on BIG with color c. We need to change every node in this set to a different color in PR.

First, we check the NCN of t that has color c on the BIG. If this number is less than PR-1 (which means there is at least one color available in PR not used by its neighbors), we can change t to another color. Since we have changed t's color on BIG and t may internally interfere with other internal nodes or boundary nodes (two boundary nodes can interfere only inside NSR but not on the CSB), we need to check if there is a color conflict. The function $Cut\_if\_conflict(t, c, c')$ attempts to insert move instructions to disconnect such edges. Figure 27 shows how the disconnection is done and the corresponding changes on the GIG. In Figure 27.a, s is originally colored with color c'; after node t is changed to color

```
INPUT:   PR, SR
OUTPUT:  cost (number of inserted move instructions)

Static context_pre, context_pre_pre

  1.  FUNCTION Reduce_PR(context):cost
  2.  Begin
  3.     Foreach color c in PR do
  4.        Cost=0
  5.        Foreach node t in Set_color_node(c,BIG) do
  6.           If NCN(t,BIG)<PR-1 then
  7.              Change t to another color c' in PR other than c.
  8.              Cost+=min(Cut_if_conflict(t,c,c')) for all possible c'
  9.           Else
 10.              Cost+=min(NSR_exclusion_cost(t,c,c')) for each
 11.                       color c' in PR other than c
 12.              Add newly split node with color c to
      Set_color_node(c,BIG)
 13.                       if it is boundary node
 14.           Endif
 15.        od
 16.        Eliminate_unnecessary_move()
 17.        Record to min_cost if this cost is smaller and record the
      context.
 18.     od
 19.     Keep the minimal cost context and return min_cost
 20.  End

 21.  FUNCTION Reduce_SR(context):cost
 22.  Begin
 23.     Foreach color c in SR
 24.        Cost=0
 25.        Foreach NSR_i color c is used do
 26.           Foreach internal node t in Set_color_node(c,IIG_i) do
 27.              If NCN(t, GIG)<R-1 then
 28.                 Color t with a color other than c.
 29.              Else
 30.                 Cost+=min(live_range_exclusion_cost(t,c,c'))
 31.                        For each color c' in R other than c
 32.                 Add newly split node with color c to
      Set_color_node(c,IIGi)
 33.              Endif
 34.           od
 35.        od
 36.        Eliminate_unnecessary_move()
 37.        Record to min_cost if this cost is smaller and record the
      context.
 38.     od
 39.     Keep the minimal cost context and return min_cost
 40.  End

 41.  FUNCTION Intra_thd_allocator(PR,SR):cost
 42.  Begin
 43.     According to the accepted context, pick stored either context_pre
 44.        or context_pre_pre => context.
 45.     If(PR is reduced) return Reduce_PR(context)
 46.     Else if (SR is reduced) return Reduce_SR(context)
 47.     Else return cost for the context //no change
 48.  End
```

Figure 26: Algorithm for intra-thread register allocation

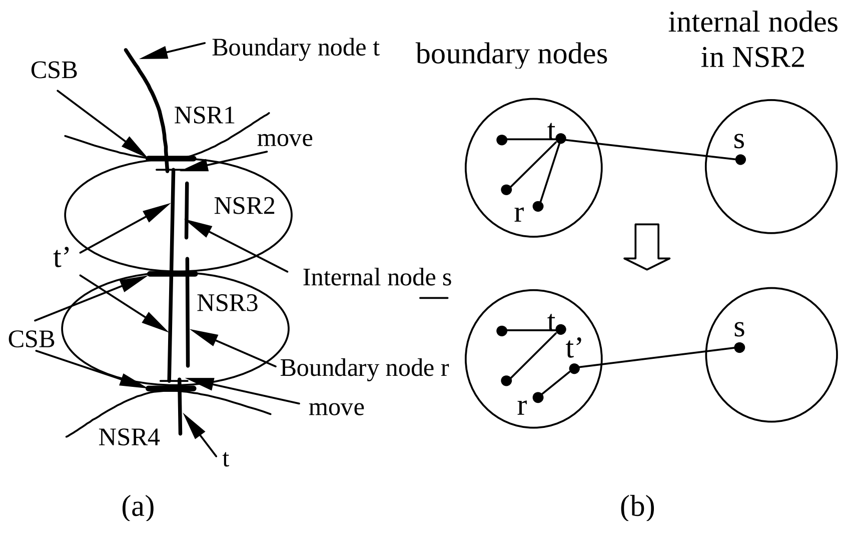


Figure 27: Node splitting to change the color of node t

c' from color c, it conflicts with internal node s. We insert a move at the CSB, so live range t is split. The part of the live range t in NSR2 becomes t', and this part can keep color c, so it does not conflict with s, while, on the BIG, t is changed to color c'. Figure 27.b shows the changes on the GIG. The edge between t and s gets eliminated after t' splits from t. t' keeps the original color of t, so in the IIG, it is compatible with s, while on the BIG, the color of t is changed. In the algorithm, we try every candidate color for t and pick the one with minimal cost.

If this step fails, i.e. NCN(t ,BIG)=PR-1, the algorithm calls function NSR_exclusion_cost (t,c,c') to get the cost of changing t to another color c' and to exclude all the NSRs with conflict nodes. NSR_exclusion_cost looks at each NSR where t is live to see if there is any node with color c' in it. If so, the NSR is excluded by splitting the live range of t in that NSR and by inserting move instructions. In our approach, the NSRs are split in whole, i.e. either the live range in that NSR is kept with color c' (if no conflict) or the live range is split (after splitting, t' in that NSR keeps color c).

Figure 28 shows how NSR exclusion is done. Boundary node t cannot change to color c' because the boundary node r and the internal node s are using color c'. The conflict NSRs are NSR2 and NSR3, where s and r are live. So, these two NSRs are excluded from the live range of the original boundary node t. On the GIG, we see t' is split from t, and t now can

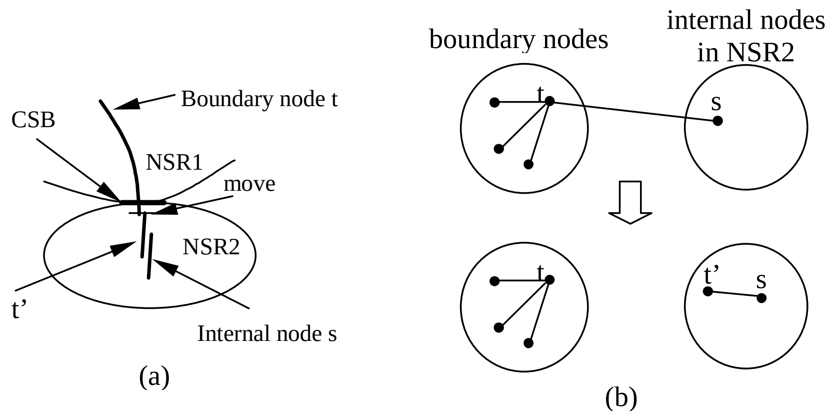


Figure 28: NSR exclusion to reduce PR

be colored with c'. t' keeps color c and it is still compatible with s and r. Notice that, after splitting, the edge originally connected from r to t is connected to t'. Therefore, the NCN of t is reduced and t can be recolored with c'. The algorithm tries each color other than c to recolor t and finds the minimal value to finally color t. Also notice that, after this step, t' is colored with c and, if it is a boundary node, we should add t' to Set_color_node(c,BIG) and we will color it with some other color during the later iterations. Set_color_node(c,BIG) will not increase infinitely, since further splitting t' will finally generate internal nodes.

4.8.2.2 Reduce-SR Invocation

To reduce SR, we check with each color c in SR to see which one should be reduced with minimal cost. The cost is calculated by adding up costs in every NSR where this color is used. Also notice that in this step, all boundary nodes are assumed to have fixed colors so that the phase will not affect the PR number.

The algorithm tries to recolor node with color c in a NSR to other color. If the node on the GIG has NCN less than R-1, we can just pick that color and color the node without any cost. Otherwise, live range splitting is needed.

Live range splitting is illustrated in Figure 29. In Figure 29.a, the example has 3 basic blocks. Live range t is recolored with color c', however, live range s also uses color c'. Our algorithm then splits t at the boundary where the two live ranges overlap. After splitting, t' can still use color c, and t is now changes to c'. We assign the color with minimal cost

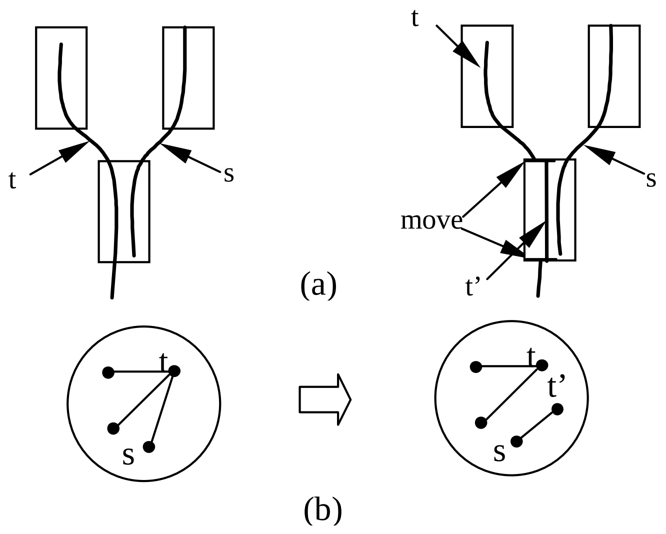


Figure 29: Excluding a live range within NSR to reduce SR

to node t. After the splitting, node t' is push into Set_color_node(c,IIGi), because now it bears color c.

This process will finally stop. After each splitting, the live range with color c is reduced. Since the value $R - 1 \geq RegP_{max}$ (according to the lower bound estimation in Section 4.6 and the algorithm in Figure 23), in the extreme case, each live range is a single program point, there will be at most $RegP_{max}$ nodes co-live and live range with color c can always be recolored.

4.8.2.3 Eliminate Unnecessary Moves

During the attempt to reduce PR, we assume that all move instructions are inserted near the CSB boundary and during Reduce_SR, some move instructions are inserted inside the NSR. At this point, we can merge some of the internal move instructions with those at the boundary. For two consecutive moves, the first move instruction to the live range is unnecessary if the color at the entrance to the first move is also acceptable in the region between the two move instructions. We can safely eliminate the first move and this actually relaxes the restriction in Reduce_PR to bind moves to the CSB.

4.9 The SRA Problem

For the SRA problem, given PRs are equal and SRs are also equal. The restriction can be rewritten in a simple form:

$$N_{thd} \times PR + SR \leq N_{reg}$$

Thus, the inter-thread register allocation algorithm can also be simplified. There are only two possibilities to reduce the register requirements. Due to the shrunk solution space, for algorithm in Figure 24, we can actually traverse all the possible PRs and SRs to find the best solution.

4.10 Experimental Results

The evaluation of our algorithm is done with the Intel-provided simulation environment-IXP1200 Developer Benchmark 2.01. The IXP1200 workbench supports cycle-accurate simulation for IXP microengines and other peripheries with high fidelity.

In this section, we experiment with 11 benchmark programs and some of their combinations to see the effectiveness of the register allocator. These benchmarks are collected from Commbench [72], Netbench [51], Intel provided example code and a packet scheduling algorithm from [75]. To evaluate our algorithm, the benchmark programs are rewritten in IXP C code (a subset of standard C) and a few of them are directly written in assembly (microcode). For those written in assembly code, we restore the virtual registers so that our register allocator can work on the live ranges from scratch. Our pass builds the CFG and interference graph from the assembly code, after simple translation of the assembly directives. The assembly code is then passed to the assembler to generate machine code. The IXP assembly consists of only 40 RISC instructions which makes the translation easy. The assembler simply exits if too many registers are required. However, after our pass, the register requirements are always satisfied, so the machine code can be generated properly. Table 5 shows the properties of the benchmark programs. The code size is number of instructions after code generation. The cycle counts are measured as follows: for some programs like L2l3forward, it cannot run to a stop in finite time, since these programs all

Table 5: Benchmark applications

	Code Size	Cycle/ iteration	#CTX insns w/o spills	#live ranges
Crc	78	52280	6	14
Drr	108	207037	12	11
Fir2dim	447	159149	32	36
Frag	271	28620	30	26
Kmp	123	148059	12	13
L2l3fwd (Rec)	635	1253.34	55	131
L2l3fwd (Send)	690	721.28	88	115
Lzw	126	43163	12	18
Md5	913	3983292	56	142
Wraps(receive)	875	2048.37	85	145
Wraps(send)	921	1264.87	103	135

(a)

	RegP_{max}	$\text{RegPCSB}_{\text{max}}$	MaxR	MaxPR	#NSR	Ave.NSR Size
Crc	6	5	9	8	4	19.5
Drr	5	4	8	6	7	15.43
Fir2dim	21	17	27	20	19	23.53
Frag	16	12	22	18	18	15.06
Kmp	7	5	10	7	5	24.6
L2l3fwd (Rec)	30	28	35	34	28	22.67
L2l3fwd (Send)	24	21	31	25	33	20.91
Lzw	13	10	15	11	9	14
Md5	41	37	60	46	31	29.45
Wraps(receive)	45	39	59	47	32	27.34
Wraps(send)	49	40	65	50	37	24.89

(b)

runs in a while loop to accept and process packets, the cycle counts are average number per iteration of the main loop. We list CTX instructions (context switch instructions, which includes load/store, voluntary context switch and other I/O operations that can cause context switch) each benchmark has. Roughly, about 10% instructions are CTX instructions. The CTX instructions here do not include spill instructions, as we have removed all spills and reconstructed original live ranges (we did this based on the source code and the annotations embedded in the generated assembly code by the Intel IXP compiler). The number of live ranges (nodes on the GIG) is listed in the last column of Table 5.a. These numbers come from the restored virtual registers.

In Table 5.b, column 1 and 2 are maximal register pressures in the program ($RegP_{max}$) and maximal register pressure at the CSBs ($RegPCSB_{max}$). These are the lower bound estimation for register requirements of the threads. Column 3 and 4 are the upper bound estimation for R and PR based on the algorithm in Figure 23. The last two columns give statistics for the numbers of NSRs and their average sizes. One observation is that normally larger NSR leads to bigger difference between the maximal and minimal value of P and PR. Because more internal nodes can exist in larger NSRs, the register pressure for GIG should exceed the BIG with larger margin.

Figure 30 evaluates our inter-thread register allocation algorithm for SRA. The same evaluation for ARA is combined in Table 3. For each benchmark program, we show two relevant bars. The first bar is the number of registers allocated to the benchmark assuming only single thread is available. We use a Chaitin [12] style register allocator for comparison with our shared register allocator. The second is the number of registers (private plus shared registers) assigned with our inter-thread register allocation algorithm. The same benchmark is assumed to execute on four threads. The algorithm continues until the cost returned is non-zero, which means we want to test how many PRs and SRs are needed without any move instruction insertions with the inter-thread allocation algorithm. If no shared registers are used and each thread runs the single-thread register allocator, many registers are wasted. Compared to the case with multi-threaded register requirements, the average total register saving for all benchmarks is 24%.

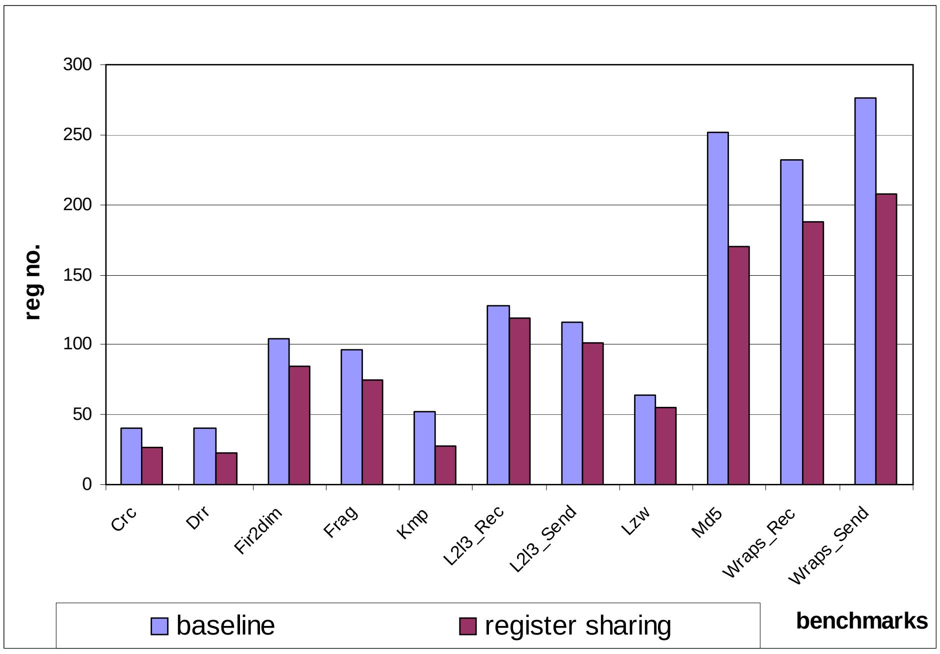


Figure 30: Original vs. SRA register allocation

In Table 6, we collect data for the extreme case with our register allocation algorithm, i.e. the maximal number of move instructions that will be inserted, if only the minimal number of registers is allocated. This means our algorithm must split many live ranges to reach the minimal number of registers. The move insertion overhead in the extreme case is mostly within 10% of the total number of instructions for the benchmarks. This cost is affordable compared to the overhead due to register spill if the register number is out of range with the single thread register allocation algorithm.

Finally, Table 7 evaluates our register allocation algorithm for ARA with 3 scenarios. Notice that all tasks are periodic, independently sharing the CPU and executing forever. Thus, we measure the performance improvements of each thread in terms of the percentage reduction of cycles per iteration. The first scenario puts two Md5 programs on thread 0 and 1, two fir2dim on thread 2 and 3. This can be a processing module between the receiving and sending module. Our data show the number of PRs and SRs assigned, the number of live ranges after the register allocation (#Live Ranges), context switch instruction number reduction and cycle change. The column "#CTX Reg Spill" is the original code generated by the Intel compiler that allocates registers with spilling and without register sharing

Table 6: Minimal case move insertion

	PR	SR	# Move
Crc	5	1	10
Drr	4	1	11
Fir2dim	17	4	19
Frag	12	4	15
Kmp	5	2	8
L2l3_Rec	28	2	23
L2l3_Send	21	3	30
Lzw	10	3	7
Md5	37	4	45
Wraps(receive)	39	6	47
Wraps(send)	40	9	50

across threads (only allocate 32 registers for each thread). And, "#CTX Reg Sharing" is the number with our allocator (actually no change compared with Table 5, because we avoid spills). The same is true for cycle count ("#Cycle Reg Spill" and "#Cycle Reg Sharing"). The fir2dim actually runs slower due to inserted moves. But this is profitable due to the big saving from Md5. Thus, the allocator is able to boost the performance of critical thread (Md5) by slightly slowing down the less performance critical one (fir2dim). The second scenario consists of L2l3fwd receive and send on thread 0 and 1, and Md5 on thread 2 and 3. This can be a complete processing modules serving on one sending and one receiving ports. The results still show the spills are saved for Md5 with minor costs for moves on L2l3fwd threads. The last scenario runs wraps receive and send on thread 0 and 1, fir2dim and frag on thread 2 and 3. The allocator balances register allocation to satisfy the wraps thread. Due to a high register pressure, wraps receive and send can run much slower (due to spills) if registers are not allocated properly. Our results show that over 20% speedup is achieved for wraps, whereas only slight slowdown is incurred for the other two benchmarks, which is in accordance with our optimization objective of boosting performance critical threads.

Table 7: Static and dynamic results for different ARA scenarios

	Benchmark (Thread#)	PR	SR	#Live Ranges	#Move Inserted	#CTX Reg Spill	#CTX Reg. sharing
Scenario 1	Md5(0,1)	41	10	152	20	79	56
	Fir2dim(2,3)	18	10	38	4	32	32
Scenario 2	L2l3fwd_rec(0)	28	5	133	24	55	55
	L2l3fwd_send(1)	21	5	120	18	88	88
	Md5(2,3)	37	5	165	38	79	56
Scenario 3	Wraps_rec(0)	42	11	161	40	123	85
	Wraps_send(1)	44	11	153	36	141	103
	Fir2dim(2)	17	11	40	6	32	32
	Frag(3)	14	11	28	4	30	30

(a)

	Benchmark (Thread#)	#CTX Reduction	#Cycle Reg Spill	#Cycle Reg. sharing	#Cycle Reduction
Scenario 1	Md5(0,1)	29.11%	5028375	4039294	19.67%
	Fir2dim(2,3)	0	159149	163515	-2.74%
Scenario 2	L2l3fwd_rec(0)	0	1253.34	1273.06	-1.57%
	L2l3fwd_send(1)	0	721.28	749.18	-3.87%
	Md5(2,3)	29.11%	5028375	4113871	18.19%
Scenario 3	Wraps_rec(0)	30.89%	2773.13	2134.52	23.03%
	Wraps_send(1)	26.95%	1709.36	1334.71	21.92%
	Fir2dim(2)	0	159149	164201	-3.17%
	Frag(3)	0	28620	28979	-1.25%

(b)

CHAPTER V

INTER-THREAD REGISTER ALLOCATION WITH HARDWARE SUPPORT

The inter-thread register allocation work presented in the previous chapter shows promising results. However, it would be interesting to perform register sharing across threads in a more dynamic manner. In this chapter, we intend to look at the problem from a different angle. Some hardware support is assumed to be available to help the compiler analysis. A hybrid (hardware/compiler) design is studied that can greatly improve register sharing across threads.

5.1 Register Utilization and Idle Cycles

As a matter of fact, allocating registers across threads without overlapping is not efficient in utilizing all registers. Conceivably, if the register pressure is low in some code segments of one thread, other threads cannot benefit from that. In our experiments, we find out that the register utilization for the benchmarks is quite low. Table 8 shows the register utilization for 8 benchmarks. We compute register utilization using the following formula, which determines how long a register is occupied (in terms of cycles) on average.

Table 8: Register Utilization Pre-Optimization

Benchmark	*Register Utilization*
ipfdwr (1 PU)	22.27%
ipfdwr (4 PU)	23.02%
md4 (1 PU)	17.94%
md4 (4 PU)	12.67%
nat (1 PU)	25.74%
nat (4 PU)	25.35%
url (1 PU)	12.95%
url (4 PU)	10.60%

Table 9: Number of Idle Cycles for each Benchmark

Benchmark	*# Idle Cycles*
ipfdwr (1 PU)	71
ipfdwr (4 PU)	88210
md4 (1 PU)	620388
md4 (4 PU)	9574894
nat (1 PU)	101284
nat (4 PU)	106866
url (1 PU)	1006534
url (4 PU)	7728240

$$\frac{\sum_{r \in regset}^{numRegs} \sum_{i=0}^{numCycles} liveAt(r,i)}{numRegs * numCycles}$$

Due to the unevenness of register pressure in each thread, although register utilization is low on average, spills still cannot be avoided. Due to the long latency of memory operations, spills are very damaging to the performance as observed from our benchmark applications. When all PUs are waiting for memory operations, the processing power is wasted. Thus, the lack of a cache, the high cost of memory accesses, and the symmetric programs typically executed on the IXP create a situation of long periods of idle activity. Table 9 shows the number of cycles that each benchmark spends in the idle state. One can see that a large number of idle cycles are spent by a given PU due to the above reason.

Therefore, these data motivate us to allow registers to be allocated flexibly across threads, which not only improves the utilization of the register file, but also transforms more memory operations into register operations, reducing the number of idle cycles. Moreover, the work presented in the previous section only allocates registers statically across threads without considering runtime information. Next, we will discuss why a dynamic approach could be more effective.

5.2 Motivation for a Dynamic Approach

In summary, the static (compiler) approach assumes that shared registers can be accessed by all threads safely, while private registers can only be accessed by one particular thread.

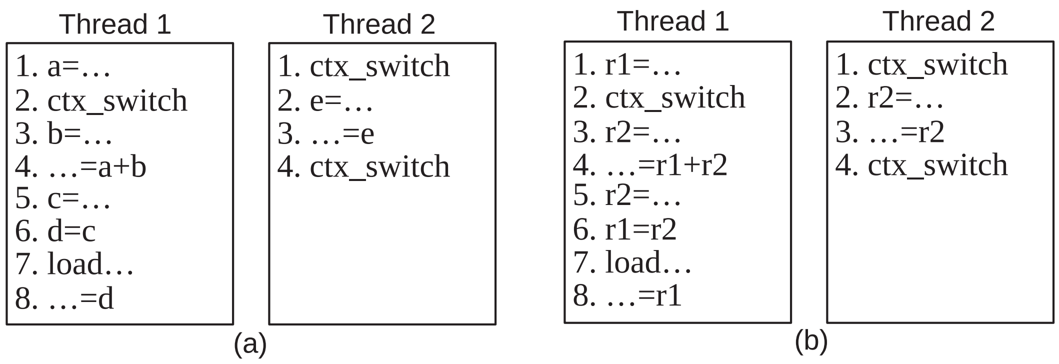


Figure 31: Example for inter-thread register allocation

In other words, private registers on different threads must be separately allocated.

Although the compiler approach can significantly speedup the application on the network processor over the traditional network processor technique of partitioning the register file into equal sets of private registers, it still makes conservative assumptions regarding the co-liveness of live ranges on different threads. One of the major assumptions is that a context switch in one thread can jump to any context switch points of other threads. For example, in Figure 31.a, we assume context switches from both the second and seventh instructions in thread 1 can jump to the second instruction in thread 2 (i.e. thread 2 takes a context switch at the first instruction to thread 1, then thread 1 context switches back and thread 2 continues from the second instruction.). This is a conservative assumption, which greatly limits register allocation. However, it might happen that at runtime, context switches are never conducted in this manner. If context switch does not go from either instruction 2 or 7 in thread 1 to instruction 2 in thread 2, we can safely allocate the private register of thread 1 ($r1$) to thread 2. Even if such context switches occur occasionally, a dynamic approach might be able to find a private register of other threads that are not used.

In summary, statically allocating private vs. share registers is not flexible in that private registers allocated to live ranges on different threads may not actually co-live at runtime. In other words, with runtime knowledge, we can possibly share some of the private registers on different threads.

A dynamic strategy also allows us to allocate registers for aliased values. Static alias analysis [55, 3] is a conservative means to determine whether a set of names which contain

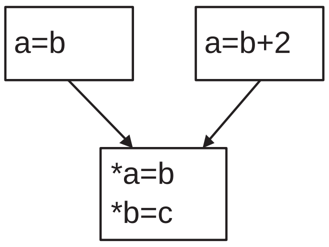


Figure 32: Example for alias analysis

Table 10: Memory Access Patterns for handwritten benchmarks

Benchmark	*# PU*	*Double-Loads*	*Total Mem Access*
ipfwdr	1	349931 (64.5%)	542762 (2.26%)
ipfwdr	4	1319746 (64.1%)	2058690 (4.31%)
md4	1	115367 (11.1%)	1040293 (4.59%)
md4	4	152655 (6.47%)	2359585 (6.86%)
nat	1	373967 (58.3%)	641675 (2.70%)
nat	4	548310 (55.0%)	997007 (2.08%)
url	1	479226 (49.5%)	967742 (4.40%)

memory locations will ever intersect. If their intersection is an empty set, it is safe to assign each location to a register, otherwise they must be accessed through memory. As will be proposed shortly, our strategy of delaying part of the register allocation till run-time exposes the literal locations to us. Figure 32 shows an example of aliasing problem. In this problem, the write to memory location a may be unnecessary if a holds the same value as b. However, there is no way to statically determine this information. At run-time, we know the actual values of a and b so it is safe to replace the memory operation with a fast move instruction. Early research on superscalar machines [45] has revealed that a significant number of store instructions are redundant. Our experiments show that there is great potential to remove redundant load and stores to expediate the program execution. Table 10 shows that a large portion of memory accesses are comprised of redundant loads.

5.3 Our Appraoch

In this section, we propose our approach of dynamically allocating registers across threads. We first statically identify dead registers at each context switch point and store this information for runtime use. The hardware captures opportunities to put memory contents to

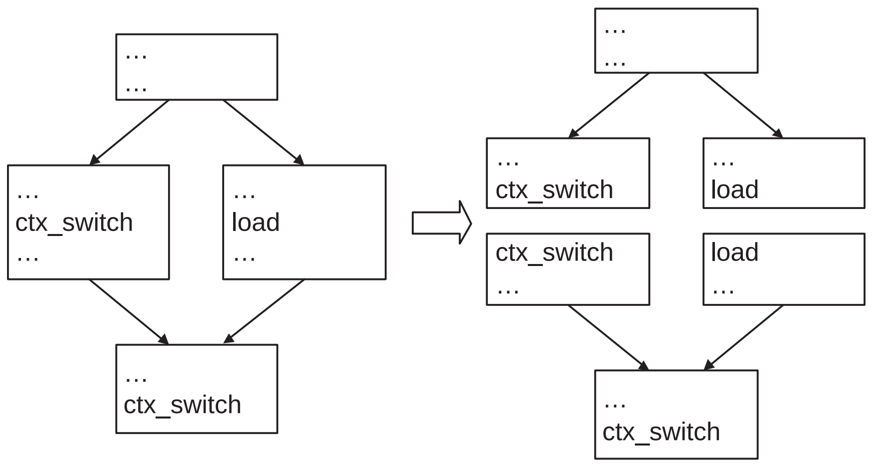


Figure 33: Example for non-switch region

dead registers so as to reduce the number of load/stores by dynamically converting them to move instructions.

5.3.1 Static Analysis

Our algorithm takes the source code (IXP micro-code) as input and outputs an annotated version of the file. We assume the existing register allocator uses a simple allocation strategy, which divides the register evenly among threads.

First, we create a control flow graph (CFG) which divides the blocks into non-switch regions. As mentioned earlier, non-switch regions are maximal connected components on the CFG, which are bounded with either context switch points or program entry/exit points. Figure 33 shows an example of a CFG that has been divided into non-switch regions. Basically the CFG is split at the context switch points.

We then use the following dataflow equations to discover the dead registers,

$$\begin{aligned} \text{Dead-Out}(BB) &= \bigcap\nolimits_{s \in \text{Succ}(BB)} \text{Dead-In}(s) \\ \text{Dead-In}(BB) &= \text{Dead-Out}(BB) - \text{Use}(BB) - \text{Def}(BB) \\ \text{Dead-In}(EXIT) &= U \end{aligned}$$

Table 11: Register Allocation Patterns in Handwritten Benchmarks

Benchmark	*Never-Used*	*Avg. Dead-Rest*
ipfwdr	8	13.57
md4	9	12.44
nat	8	15.09
url	9	15.12

It is important to note that `Dead-In` and `Dead-Out` are not the inverse of the traditional `Live-In` and `Live-Out`. A dead register is made live by a use or a def. This is because it is unsafe to use a dead register across a new definition of it, even if that definition is never used. For example,

```
alu_add     A15, A16, B15  // A15=A16+A15

alu_add     A16, A15, B16  // A16=A15+A16

...

alu_add     A15, A16, B15  // A15=A16+A15
```

Here the first operand is the destination operand. By traditional dataflow analysis, `A15` is dead after its use in the second line. However, it's cleary unsafe to treat `A15` as if it is dead for the rest of the program.

Table 11 shows a significant number of the registers (6%) are never allocated. Dead-Rest is the number of registers which are dead for the remainder of the program. We compute Dead-Rest at the edge of every non-switch region.

We then annotate every context-switch instruction with a bit vector containing all of the current dead-rest registers. The bit vector is 128 bits long, which requires extending the length of memory instruction. If the program uses only thread relative register references, then we can reduce the bit vector size to 32 bits.

Notice the program might use memory operations to communicate between PUs. In such cases, we first determine which addresses are used for inter-engine communication and which are used for spill values. Then, we extend the option field of every memory operation to note whether a memory operation is local or global. Actually, the inter PU communication can be done mostly through *Next Neighbor Registers*, which are available on new generations of the IXP processors [36].

5.3.2 Dynamic Analysis

Our dynamic analysis is to map memory addresses to dead registers. We then replace any instructions which use the mapped memory address with fast register-move instructions. For example, if the address 128 is the target of a store instruction and GPRs A15-17 are dead, we can rewrite the following instruction sequence. Here, \$0 is a transfer register; SRAM access must go through transfer registers.

```
alu_add     $0, A3, B4
sram_write  $0, 128, ..., CTX_SWAP
...
sram_read   $0, 128, ..., CTX_SWAP
alu_add     A1, B2,  $0
```

as

```
alu_add     A16, A3, B4
alu_b       A15, A16
...
alu_b       A17, A15
alu_add     A1,  B2, A17
```

Now suppose that only GPR A15 is dead. We greedily allocate the dead register to next address that requires it, the write transfer register `$0`. Now there are no remaining registers to allocate for the address 128. The following instruction sequence will be executed by our hardware,

```
alu_add     A15, A3, B4
alu_b       $0,  A15
sram_write  $0,  128, ..., CTX_SWAP
```

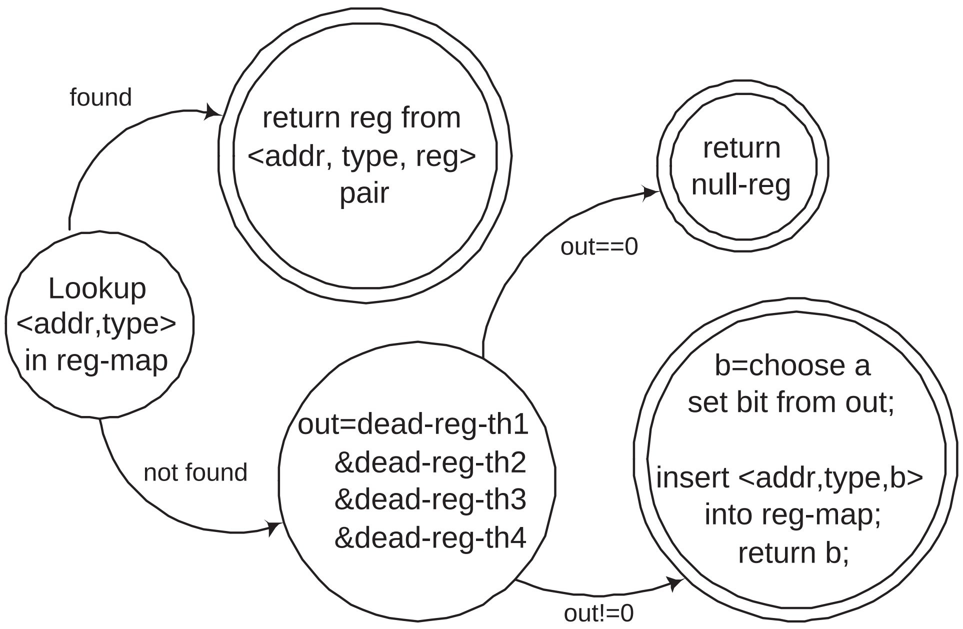

found
return reg from <addr, type, reg> pair
return null-reg
out==0
Lookup <addr,type> in reg-map
b=choose a set bit from out;
insert <addr,type,b> into reg-map; return b;
out=dead-reg-th1 &dead-reg-th2 &dead-reg-th3 &dead-reg-th4
not found
out!=0

Figure 34: Finite state machine for Dynamic Allocation

```
...
sram_read   $0,  128, ..., CTX_SWAP
alu_add     A1,  B2, $0
```

This transformation preserves the correctness of the original code, at the cost of an additional move instruction.

We insert our new hardware into the fourth pipeline stage. At that point, the ALU output forms the memory address for any memory operation. This allows us to use the actual value of a memory address instead of an alias for it.

Figure 34 formally illustrates the finite state machine for our allocation hardware. We check if the current memory address is already stored in the table; if so, we replace the memory operation with a register move instruction. If the current memory address is not in the table, we allocate one of the remaining dead registers and create a map between the dead register and the address. In the remainder of this section, we explain in detail how this process is done in hardware.

The hardware uses pieces of the register-file to create a register lookup table–Figure 35. We make GPR 127 a special-register which contains the root entries of the table. Entry 1

Type Tag (3 bits)	Address Field (21 bits)	Register Map (5 bits)	Next Entry (3 bits)
0x1	0x0F000	0x0F	0x3

A Book–keeping Entry

Data (32 bits)
0x00000001

A Data–Entry

Entry 1 (5 bits)	E1 Valid (1 bit)	Entry 2 (5 bits)	E2 Valid (1 bit)	... ...

The Root Entry (Register 127)

Figure 35: The register lookup table components

can hold a book-keeping entry from GPR0-31, entry 2 holds a register between GPR32-63, etc. The state field indicates whether the book-keeping entry is valid.

Each book-keeping entry contains the type of memory accessed along with the address. The type tag also doubles as a valid bit, with type 0 corresponding to the invalid state. The memory is then bypassed by accessing the register in the Register Map entry. In the case of Figure 35, GPR16+offset contains the contents of memory address 0x0F000. The value 16 is relative to the table entry. The Next Entry field contains a link to another bookkeeping entry that has a position -3/+4 offset from the current register. If the value of this field is 0, then the next-entry is null.

Figure 36 illustrates an example of a register file which doubles as the components of our data structure. Register 127 has a valid entry 1 which points to Register 16. Register 16's type entry shows that there is valid SRAM data entry in register 4. It also has a link to the next bookkeeping entry located in Register 19. Register 19's next-entry field is 0, so it is final book-keeping entry in this area of the register file.

If the current address is not in the lookup table, the hardware allocates an unused register. It determines that a register is a candidate for allocation if it appears in the intersection of the dead-rest bit-vectors for all four threads at the most recent context switch. For example, if registers 0-2 are dead in threads 0-2 and registers 0,2 are dead in

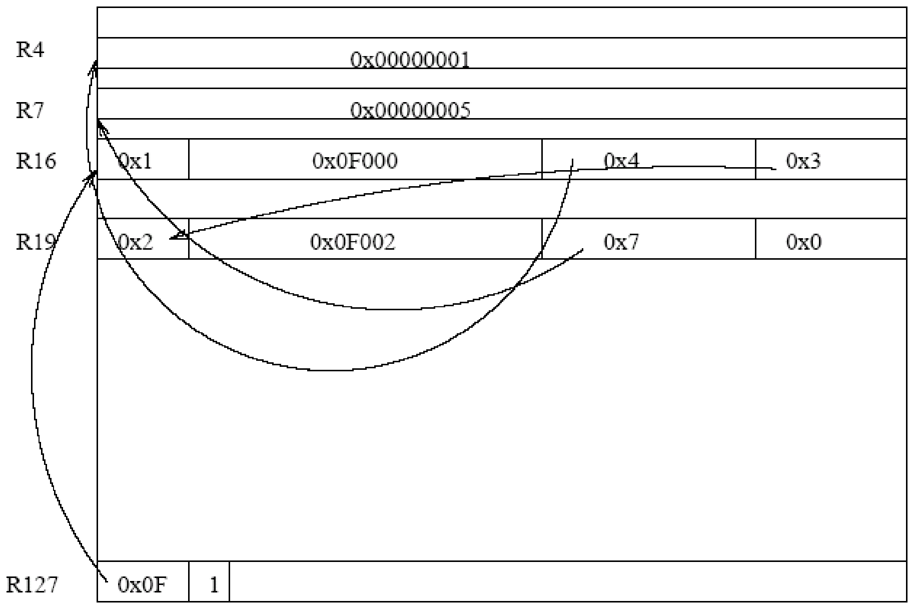


Figure 36: An example register file

thread 3, the hardware computes

```
1110...0 & 1110...0 & 1110...0 & 1010...0 = 1010...0
```

so registers 0 and 2 are candidates for allocation. It arbitrarily selects the first bit which is set, so in this case the hardware will allocate register 0 to the next memory address.

We model the latency of our algorithm as 6 cycles. This is a conservative estimate of the time we expect it would require in an actual processor. The running time of each bit operation is 1 cycle, for a total of 2 cycles. We estimate the time for a table lookup or insertion to be 2 cycles on average, for a total of 4 cycles. Thus, the total running time is 6 cycles.

5.3.3 Other Considerations

5.3.3.1 Transfer Registers

The processor uses transfer registers [34] when transmitting to external devices such as SRAM. Transfer registers are not general-purpose and can only be exclusively read or written. A transfer register must appear as the source of a store operation instruction, but it cannot be used as the corresponding source register in the new move instruction. For example, if we wished to map address 128 to register A1, transforming

```
sram_write  $0, 128, ..., CTX_SWAP
```

into

```
alu_b       A1,  $0
```

The value of $0 in the resulting instruction is the value of the read transfer register $0, not the value of the write transfer register.

For our address-register mapping scheme to work, instructions that use transfer registers must replace those registers with general purpose registers. This can mean that 2 general purpose registers are required for each memory-address transfer-register pair, but in practice one transfer register is used for a large number of memory addresses.

Statically, we need to account for the case where a transfer register is the source of a memory operation such as `t_fifo_wr`, a write to the transfer FIFO, that is not rewritten by our address-register mapper. We do this by looking for the next store instruction following a transfer register definition and the previous load instruction following a transfer register use. If the load or store instruction is not a possible remap target, we note that in the option field of the corresponding instruction that uses the transfer register. If the transfer register is the source of both types (via a branch), then we split the instruction into a mappable and non-mappable version. For example, the following program

```
    immed       $0  1
    sram_write  $0  B0  0  1
    immed       $1  5
    t_fifo_wr   $1  B1  B2 1
    t_fifo_rd   $2  B2  0  1
    alu_add     A1  A3  $2
    immed       $1  7
    br=ctx      0   L0
```

```
     t_fifo_wr   $1  B3  0  1
 L0: sram_write  $1  B0  0  1
```

is transformed into

```
     immed       $0  1
     sram_write  $0  B0  0  1
     immed       $1  5          NO_MAP
     t_fifo_wr   $1  B1  B2 1
     t_fifo_rd   $2  B2  0  1
     alu_add     A1  A3  $2     NO_MAP
     immed       $1  7
     immed       $1  7          NO_MAP
     br=ctx      0   L0
     t_fifo_wr   $1  B3  0
 L0: sram_write  $1  B0  0
```

This example shows all possible cases. The transfer register instructions associated with the FIFO operations are annotated with the NO_MAP option. The transfer register instruction that is the source of both a t_fifo_wr instruction and a sram_write instruction is split into two transfer register instructions, one of which has the NO_MAP option.

5.3.3.2 Unpacking Memory Instructions

A memory instruction can load or store a range of words at once. In that case the transfer register specified in the destination slot only represents the start of the range of transfer registers used in the operation. We need to unpack the memory operation, so that each transfer register is exposed in the instruction stream.

For example,

```
    sram_write  $0  B0  0  4
```

is transformed into

```
sram_write  $0  B0  0  1
sram_write  $1  B0  1  1
sram_write  $2  B0  2  1
sram_write  $3  B0  3  1
```

5.4 Experimental Results

We use the NePSim (*Network Processor Simulator*) open-source IXP1200 simulator [50, 74] to implement the hardware modifications. Benchmarks are selected from the Netbench [51] benchmark suite. We are only able to run a subset of the benchmark suite that has been ported to NePSim [50].

Table 12 to Table 14 shows the results of running our algorithm on the benchmark. Each benchmark executes forever, so we run the first 8000000 instructions on each benchmark before halting. The benchmarks are run on all 6 PUs (4 intermediate PUs) and on 3 PUs (1 intermediate PU).

Table 12 shows the SRAM dynamic load+store counts before and after optimization. There are two factors which limit the number of load+stores that can be removed: 1) Most importantly, our optimization requires many registers, while the number of available dead registers is limited. 2) Some of the load+store activity facilitates inter-engine communication; these cannot be removed. The store numbers are universally better than the load numbers because any store can be immediately allocated to a register, while a load requires that a corresponding store is already allocated to a register.

Table 13 shows the reduction in idle cycles. The idle cycle reduction varies wildly across benchmarks, but there exists a correlation between the dynamic load+store count and the idle cycle reduction. Also, the relative number of idle cycles removed decreases when number of PUs increases. This is probably due to the increase in cross-PU communication.

Table 14 shows the increase the packet throughput for each benchmark. Intuitively a decrease in idle cycles should cause an increase in throughput performance. The throughput however is a complicated function of many parameters not just idle cycles. We examined

Table 12: Comparison for dynamic Load+Store Counts

Benchmark	*# SRAM Loads Pre-Opt*	*# SRAM Loads Post-Opt*	*% Decrease*
ipfdwr (1 PU)	339643	301484	11%
ipfdwr (4 PU)	1284455	1134751	12%
md4 (1 PU)	674500	571901	15%
md4 (4 PU)	1739127	1471492	15%
nat (1 PU)	491110	364786	25%
nat (4 PU)	732668	568279	23%
url (1 PU)	860114	785074	9%
url (4 PU)	2612944	2407754	8%
Benchmark	*# SRAM Stores Pre-Opt*	*# SRAM Stores Post-Opt*	*% Decrease*
ipfdwr (1 PU)	91515	74127	19%
ipfdwr (4 PU)	351549	283618	19%
md4 (1 PU)	320169	256253	20%
md4 (4 PU)	894918	187932	21%
nat (1 PU)	207839	149644	28%
nat (4 PU)	346453	261299	25%
url (1 PU)	132049	94208	12%
url (4 PU)	420058	365450	13%

Table 13: Comparison for idle Cycles

Benchmark	*Idle Cycle Count Pre-Opt*	*Idle Cycle Count Post-Opt*	*% Decrease*
ipfdwr (1 PU)	139	61	221%
ipfdwr (4 PU)	88217	88210	0%
md4 (1 PU)	629387	469737	25.4%
md4 (4 PU)	9574894	9470987	1.1%
nat (1 PU)	101284	26676	73.7%
nat (4 PU)	106866	38325	64.1%
url (1 PU)	1006534	860238	14.5%
url (4 PU)	7728240	7079193	8.4%

Table 14: Comparison for throughput

Benchmark	*#Pkts Pre-Opt*	*#Pkts Post-Opt*	*% Increase*
ipfdwr (1 PU)	18297	18701	2.21%
ipfdwr (4 PU)	70302	75490	7.38%
md4 (1 PU)	4210	4485	6.53%
nat (1 PU)	28645	32755	14.35%
nat (4 PU)	44898	50101	11.59%
url (1 PU)	2174	2245	3.27%
url (4 PU)	7139	7387	3.47%

Table 15: Occurrences of memory accesses by type

Benchmark	*Removed Spills*	*Cross-Communication*	*Out of Space*
ipfdwr (1 PU)	55547	22248	353363
ipfdwr (4 PU)	217635	31599	1386770
md4 (1 PU)	166515	52399	775755
md4 (4 PU)	974621	181459	1477965
nat (1 PU)	184519	33410	481020
nat (4 PU)	249543	81628	747950
url (1 PU)	112881	47425	831857
url (4 PU)	259798	204846	3292800

the benchmarks and the throughput is mainly a function of the design and inter PE communication which is not handled by our framework. In some cases, the idle cycles form a part of the critical path and our framework optimized it away significantly. For example, there is a 14% increase in the speed of the nat benchmark, which is promising.

Table 15 shows the different reasons that the algorithm is unable to remove all spills. Cross-communication indicates a memory access that is inherently unremovable with the current IXP hardware. It is a memory access the programmer uses for communication with another microengine rather than for storage purposes. The more important reason that the hardware cannot remove a spill is due to size restrictions. The program only has a limited amount of dead registers, furthermore the CAM table can only hold 8 different addresses simultaneously.

CHAPTER VI

COMPILER OPTIMIZATION FOR MANAGING RUNTIME CONSTRAINTS

In this chapter, we will look at a completely different problem for the IXP network processor. Instead of managing register allocation, some runtime constraints could be effectively realized by the compiler. Due to the lack of OS, such functionalities are highly desirable on the network processor. We demonstrate that they can be achieved solely through compiler analysis and optimizations.

6.1 Motivation

Most network applications have real-time constraints because packets have real-time requirements such as maximal delay, deadline, sustained rate, etc. Typically network processors are programmed by putting different packet processing tasks on different micro-engines – the output of one task feeding the other. The tasks run asynchronously and are designed so that output of one is guaranteed to be available at a certain interval. This translates into real time constraints on tasks (in other words, a task must complete by a certain deadline). On the other hand, to maximize parallelism, each task is implemented through multiple threads on a given micro-engine. Also, as discussed earlier, due to increase in the size of available code store, one could now program the NPs in heterogeneous manner. Under a certain design objective, it may be desirable to put multiple heterogeneous threads on one NP to eliminate inter-thread communication that could otherwise be quite heavy in terms of latency and delays. In this work, we assume that such a design decision has been taken by the application developer analyzing higher level application characteristics. At lower (implementation) level, the big question is then how to make the design work with respect to the real time constraints? The key question is how to map the real-time constraints of tasks to run-time constraints of individual threads comprising the tasks and how to schedule the threads to meet the task requirements. The real time scheduling research has fortunately addressed this problem and it is possible to select a particular scheduling scheme to give the required guarantee [46, 68]. By choosing appropriate scheduling scheme that arbitrates among threads, one can develop requirements for runtime constraints that must be obeyed by the threads. These are the scheduling constraints which are then conveyed to our compiler by the developer which then attempts to implement them. Although the real time literature addresses the issues of how to develop constraints from real time requirements, it leaves the issues of their efficient implementation open. The focus of this work is on developing compiler strategies that implement such run time constraints efficiently (without performance degradation) in the absence of any OS or hardware support.

As we know, programs executing on a general purpose processor require runtime support both from the hardware and the operating system. Scheduling of multiple programs sharing a CPU must be orchestrated by the OS and the hardware using certain sharing policies. Real time applications require a real-time aware OS kernel to meet their specified deadlines.

Modern computers have been built with increasingly sophisticated hardware and OS to ascertain that an application's runtime constraints are satisfied. However, in case of network processors, both OS and hardware supports are normally absent. This is due to multiple reasons. First, the OS overhead is unbearable in most cases. For example, on general purpose processors, the operating system can take actions during the context switch. Normally, context switches happen infrequently, therefore pose very small overhead to the program execution. However, on NPs a context switch might occur after a small amount of processing has taken place (e.g. as observed on Intel's IXP processor, context switches happen every 10-20 cycles to hide long latency operations) and such overhead could become a huge bottleneck for an application, even if a small amount of OS intervention is imposed during context switches. Thus, OS solutions to such processors are not really feasible. On the other hand, network processors are geared towards very high-speed lines. As the speed of the underlying network keeps increasing (10 giga-bits per second etc), resorting to the hardware to provide runtime supports would be simply unacceptable due to large silicon real estate needs and mechanisms that might unduly slow the critical path and the clock.

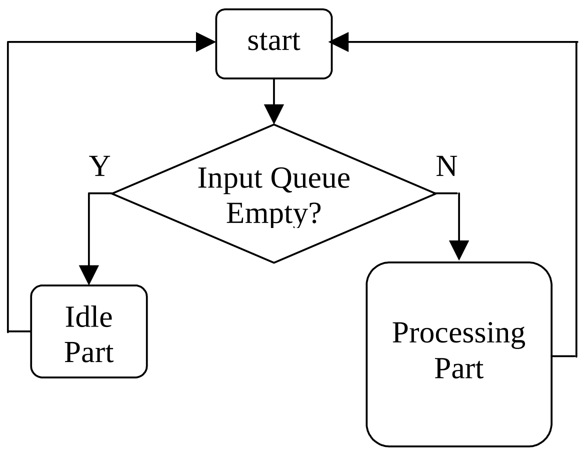


Figure 37: Thread execution model

In short, to provide comparable performance to their counterparts, i.e. the traditional ASIC routers, runtime support from both hardware and the operating system are largely sacrificed. For example, IXP 1200 almost gives a raw machine to the running applications. The CPU is scheduled in a round-robin fashion. Threads are not pre-emptable unless they give up the CPU by executing the instruction which causes a context switch. Also, no hardware interrupts and OS are currently available on the packet processing engine due to reasons discussed above. As a result, special approaches must be devised for network processors to satisfy applications' runtime constraints without involving hardware or OS.

6.2 Background

As mentioned in the previous chapters, there are several PUs on the network processor. These PUs can be assumed to work in a pipeline fashion. Each PU gets packets from its input queues, processes it and then writes to its output queues, or the input queues of the PU in the next pipeline stage. With pipeline processing, typically some PUs are in charge of getting packets from the input ports; some handle packet processing and some are for output ports.

With multiple threads on each PU, the CPU is better utilized since when one thread is stalled, the CPU can context switch to another thread. As will be addressed later, context switching is lightweight in that only PC is saved, therefore threads normally have non-overlapping regions of the register file and the memory for their private use (therefore

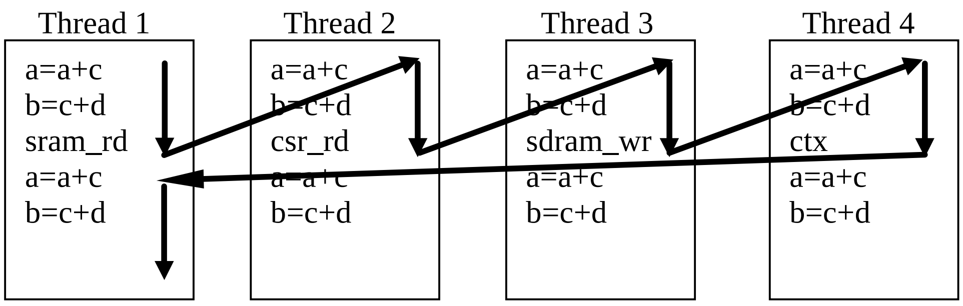


Figure 38: Thread execution on the IXP

there is no need to save them during context switches), although each thread can access all registers and memory locations. A small amount of data such as the routing table, might be shared which is typically held in the main memory.

We perform our optimizations on a per thread basis. The thread execution model is shown in Figure 37. Code on each thread must be running in an infinite loop servicing packets. If there are packets in the input queue, the thread enters the *processing part*, otherwise it is in the *idle part*. After idling for a short period, it checks the input queue again. Since there is no interrupt supported, each thread must poll the input queue frequently.

The thread code is stored on-chip, so there is no latency for instruction fetching. Multi-threading and context switching are extremely light-weight. At any moment, the CPU executes code from one of the threads if at least one thread is not stalled. No cycles are wasted during a context switch; the hardware only saves the PC of the current thread and then switches to a new thread. As mentioned before, thread state can be stored in their private registers or in memory, thus there is no need to save/restore anything except the PC during context switches.

Most instructions can be completed in 1 cycle. Long latency instructions must perform context switch to hide the latency. Here, we define *ctx-trigger instructions* as instructions that can cause context switch. There are many types of ctx-trigger instructions, including long latency instructions such as memory accesses, hash operations, etc. are included as ctx-trigger instructions. The ctx-trigger instructions in the original code before our optimization are called *native ctx-trigger instructions.*

Since there is no interrupt present which can stop a thread's execution and no thread

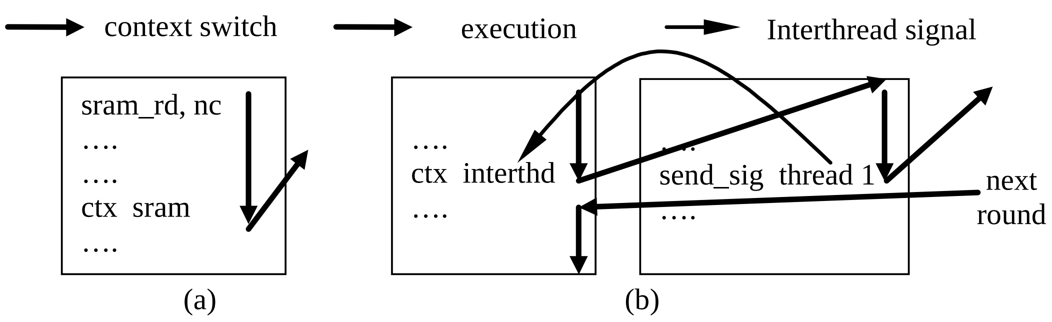


Figure 39: Use ctx instruction to wait on signals

can preempt other threads, a thread will always take the CPU if no ctx-trigger instruction is encountered. Therefore, manually written code could easily lead to imbalanced sharing of the CPU [1]. To help with that, IXP has a particular kind of ctx-trigger instruction called the *ctx instruction.* The basic functionality of the ctx instruction is to voluntarily surrender the CPU to other threads without doing any long latency operations (another use is to wait on signals as discussed later). After executing a ctx instruction, the thread gives up CPU to other thread (in a round robin manner) and it becomes ready immediately. In other words, it can resume execution as soon as the CPU schedules it again (after finishing one round of round-robin). The programmer can intentionally insert some ctx instructions if he perceives that one thread might take too much CPU time.

When the thread executes a ctx-trigger instruction, the CPU searches threads sequentially from the current thread, e.g. 1,2,3,4,1,2,3,4 ... until a non-stalled thread is found and schedules it for execution. If all threads are stalled, the CPU is idle. We show an example in Figure 38 to illustrate how multi-threaded execution might occur. Assume the CPU starts execution from the first instruction in thread 1. It executes instructions until the sram_rd instruction is encountered. sram_rd issues read command to the SRAM memory, which causes the thread to give up CPU and stall until the completion of the read operation. The hardware then looks for the next thread to execute. If the next thread is not stalled at this time, it starts execution. Thus, threads 2 and 3 execute their respective instructions as shown. After thread 4 executes ctx, it gives up the CPU and thread 1 is scheduled.

[1]As stated in the IXP manual [34]: "It is the programmer's responsibility to write his/her code so that this does not happen."

Notice that, it is not always true that all threads can be executed in each round. When thread 1 is scheduled again, it is possible that the read operation has not completed yet, therefore thread 1 may still be stalled. If so, the hardware will schedule thread 2 instead. In this work, we make sure that a thread is not stalled as follows. The compiler controls the time distance between two ctx-trigger instructions for each thread such that it meets the following criterion. If the sum of the execution time of all other threads in each round is greater than the time to complete any long latency instruction, the stalled thread will be ready before it is scheduled again in the next round. However, if the stall latency is too long, one could start the corresponding operation early (without a context switch), after doing some useful work, then perform context switch. For example, in Figure 39.a, after issuing a read command to SRAM (the option "nc" means "no context switch"), the thread continues execution, until the "ctx sram" instruction, which does a context switch and waits on the signal from SRAM. In this way, the thread's stall time can be reduced through compiler optimizations.

Notice that, the original code typically lacks proper control of context switches, easily leading to *unnecessary stalls*, i.e. situations in which none of the threads are ready. We observe that roughly 20% CPU time is wasted due to such stalls. Our optimization also aims to reduce such stalls leading to performance improvement.

It is also possible to perform interthread synchronization using special signals sent from one thread to another. In Figure 39.b, "ctx interthd" causes the thread to give up CPU and get stalled. When the interthread signal is received from the second thread, the stalled thread becomes ready and starts execution when the round robin control reaches it.

6.3 Runtime Constraints

We study a number of well-known constraints involving CPU scheduling, real-time scheduling and packet scheduling that can be used to fine tune task pipelining. Real-time and packet scheduling have long been studied in research area, although NOT widely implemented in ASIC dominated era. With the growing NP market, they are highly likely to be installed on NPs. For example, packet scheduling can be used to provide differential

Table 16: Runtime constraints and approaches

Category	**Constraint**	**Approach**
CPU Scheduling	(Weighted) Round Robin—(W)RR	FCS
	Priority Sharing—PS	FCS
Real-time Scheduling	Rate Monotonic—RM	FCS
	Earliest Deadline First—EDF	DCS
Packet Scheduling	Priority Class—PC	FCS
	First Come First Serve—FCFS	DCS
	(Weighted)Fair Queueing—(W)FQ	FCS

services, i.e. some packets like critical video frames are processed more urgently.

We will show that these constraints can be satisfied by the two approaches proposed. One is called *Fixed Context Switch (FCS)* and the other is *Dynamic Context Switch (DCS)*. To provide an overall picture, we list the constraints being considered and the approaches accordingly in Table 16.

As listed in Table 16, with either FCS or DCS, 7 types of constraints can be tackled. For CPU scheduling, weighted round robin (WRR) involves assigning a fixed weight to each running thread. Each thread gets a share of the CPU time that is proportional to its weight. The second one, called priority sharing (PS), involves assigning each thread a priority. High priority threads can preempt low priority threads. For real-time scheduling, two classic schemes are included, i.e. Rate Monotonic (RM) and Earliest Deadline First (EDF) [46]. RM is for periodic threads, where short period threads can preempt long period threads. EDF works for threads with different deadlines; the thread with earliest deadline is served first. The last three constraints deal with packet scheduling. All threads receive and process packets from their input queues. With the Priority class (PC) constraint, each thread serves incoming packets using a certain priority class. Packet classes with higher priority should be served earlier. First Come First Serve (FCFS) simply serves packets according to their arrival times. Finally, Weighted Fair Queueing (WFQ) [25] is similar to the WRR for CPU scheduling. The original definition of WFQ assumes packets are prioritized but could not be split; however, ideally packets should be split into very small pieces and served in a WRR fashion to achieve weighted fair sharing of the bandwidth. So, the algorithm attempts to send complete packets but approximates the ideal case as closely as possible. Normally the bandwidth is not the bottleneck due to the long processing time inside the network processor. Also packet transmission is treated as a part of the processing that moves packets in pieces from the internal SDRAM to outside transmission hardware. Thus, the WFQ is the same as sharing the processing power in a WRR fashion. In other words, WFQ is equivalent to WRR sharing of the CPU on each processing unit.

Both FCS and DCS insert ctx instructions at particular points of the program to control context switches of threads. FCS inserted ctx instructions do not wait on inter-thread signals and can resume execution in the next round. Conversely, a DCS inserted ctx instruction must wait on the inter-thread signal and the corresponding thread is stalled until the signal is received. Thus, with a control thread, DCS can manage the threads' runtime behavior more adaptively. DCS is more flexible and powerful, however, it increases execution overhead.

6.4 Fixed Context Switch (FCS)

Fixed Context Switch (FCS) problem is defined as:

The compiler inserts ctx and other padding instructions (will be defined shortly) in the threads such that the runtime constraints can be met. "Fixed" means that those ctx instructions do not wait on signals, the thread will resume execution in the next round. The optimization target is to reduce the number of useless padding instructions.

Next, we give examples of how compiler inserted ctx instructions can change the runtime behavior of threads.

6.4.1 Examples

We begin with an example that illustrates how to implement WRR CPU sharing with ctx insertions.

In Figure 40.a, we show two threads sharing one CPU. For simplicity, we assume both threads contain only sequential code, which are all non ctx-trigger instructions. Also, as mentioned in the previous section, each non ctx-trigger instruction takes 1 cycle. Thread 1

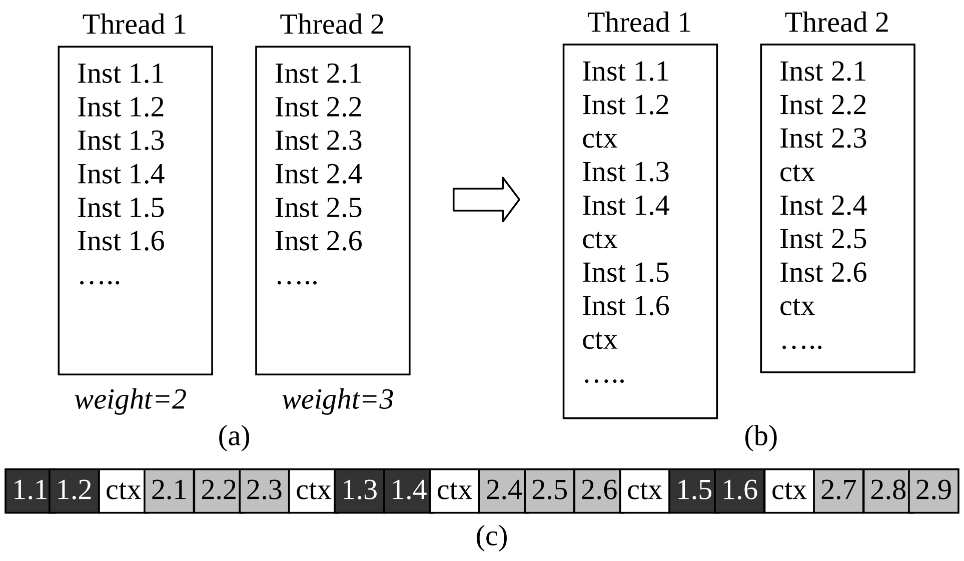


Figure 40: Example 1-Weighted Round Robin

has weight 2 and thread 2 has weight 3, i.e. the CPU time taken by thread 1 vs. thread 2 should be at the ratio 2:3. In Figure 40.b, ctx instructions are inserted. For thread 1, we insert one ctx after every two instructions. For thread 2, we insert one ctx after every three instructions. The runtime execution trace is shown in Figure 40.c. Since the hardware only schedules threads in a round robin fashion, every time a thread executes ctx, the CPU is switched to the other thread. After each thread is executed 3 times, we observe that thread 1 takes 6 units of CPU time (black boxes), and thread 2 takes 9 units of CPU time (grey boxes). Although the underlying hardware has no intention to allocate CPU time differently for the two threads, compiler inserted ctx has achieved WRR CPU sharing between the two threads. Compared with the implementation in a traditional computer, which includes hardware generated time interrupts and interrupt handlers inside the operating system, the compiler solution here is much simpler and lightweight. Notice that, actually we can insert fewer ctx instructions, e.g. insert one ctx after every 10 instructions in thread 1 and insert one ctx after every 15 instructions in thread 2 as long as the same ratio is kept.

Next, we give another example showing how priority sharing can be approximated among three threads with FCS. In Figure 41.a, thread 1 has the highest priority, thread 2 has middle priority, and thread 3 is lowest. Each solid line represents the sequential non ctx-trigger instructions for one thread. Solid dots are ctx instructions. From the figure, we can see there

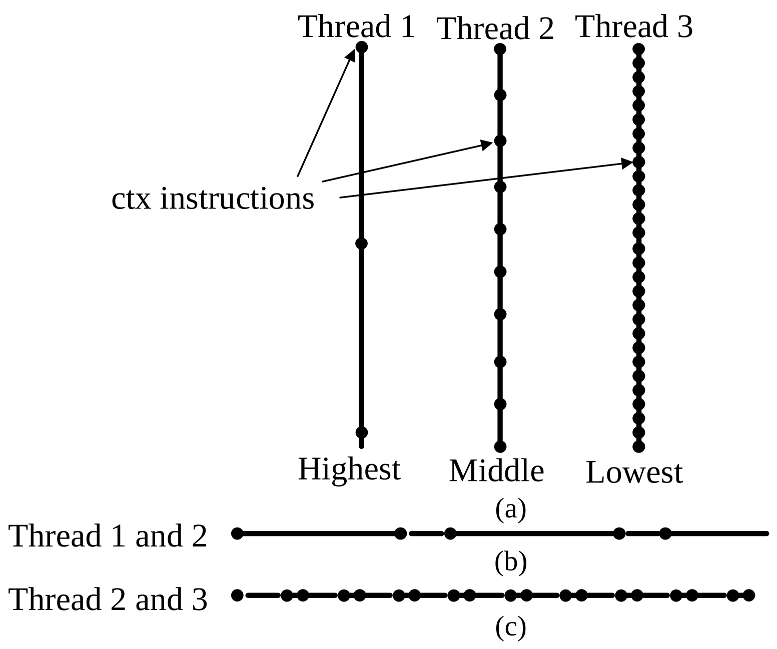


Figure 41: Example 2-Priority Sharing

are more ctx instructions in thread 1 than in thread 2, and there are more ctx instructions in thread 2 than in thread 3. Therefore, thread 2 is more frequently interrupted than thread 1 when they are both enabled; and thread 3 is more frequently interrupted when either thread 1 or thread 2 (or both) is enabled. Figure 41.b shows the runtime trace if thread 1 and 2 are enabled together (solid line is thread 1 and dotted line is thread 2). Thread 2 is forced to progress slowly. On the other hand, when thread 2 is enabled along with thread 3 (Figure 41.c, dotted line is thread 2 and solid line is thread 3), thread 2 takes most of the CPU time. Actually, in real programs, threads are always enabled, however they could be in either the idle part or the processing part-Figure 37. We put many ctx instructions in the idle part, so the thread is frequently swapped out and takes almost zero CPU time (as if it is disabled). Notice that FCS can only approximate PS, because lower priority thread cannot be completely preempted. With DCS, PS can be fully supported but with higher overhead.

So far, we have shown that insertion of ctx instructions can have an important impact on the runtime behavior of threads. If ctx-trigger instructions are located at fixed distances, we can know for sure how long a thread will execute between two context switches, which

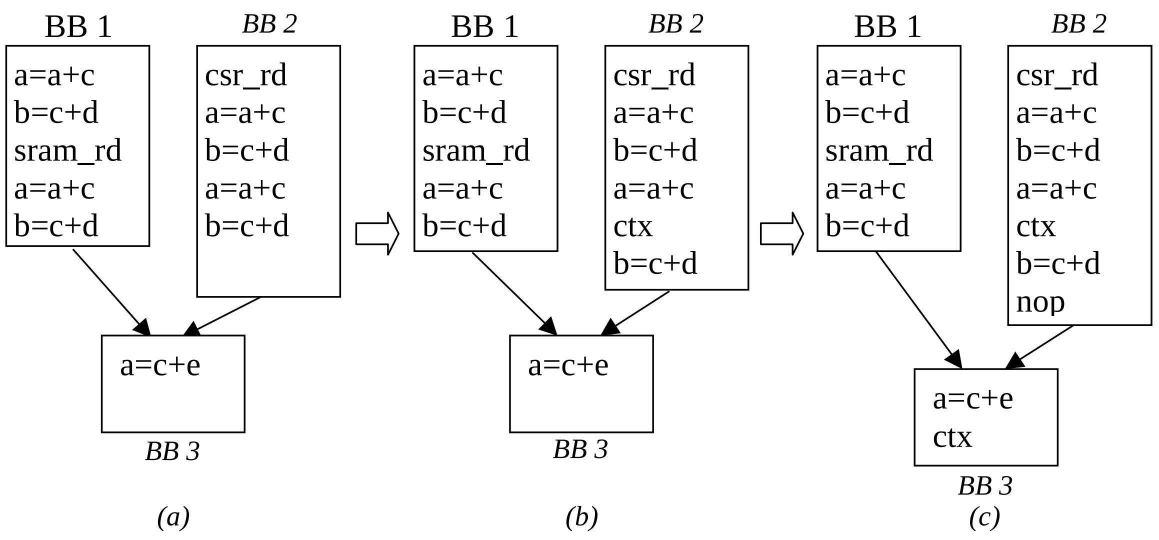


Figure 42: Example 3-control context switch distance with padding instructions

serves as a starting point for us to control a thread's runtime behavior. Both examples 1 and 2 utilize such property to enforce the runtime constraints. However, fixed distance insertion is not free when branches exist.

In Figure 42.a, we show a code segment with 3 basic blocks (BBs). In BB 1, sram_rd is a native ctx-trigger instruction followed by 2 non ctx-trigger instructions. In BB 2, csr_rd is a native ctx-trigger instruction followed by 5 non ctx-trigger instructions. If we want this thread to context switch after every 3 instructions (cycles), a ctx instructions can be inserted into BB 2 as in Figure 42.b. However, along the two paths to BB 3, we cannot insert a ctx instruction after the first instruction in BB 3 since it is only 2 instructions away from the inserted ctx instruction in BB 2 (although there are 3 instructions from this insertion to the sram_rd instruction in BB 1). The only way to achieve context switch after every 3 instructions is to insert a nop in BB 2, then insert a ctx instruction after the first instruction in BB 3. In the resulting code (Figure 42.c), (along any path) the number of instructions between any two consecutive ctx-trigger instructions is 3. It may be noted that in general the *padding instructions* need not be just nops; they could be instructions from optional computation that could be scattered in a given thread. Modern packet processing often involves optionally performing some computation, which is not a part of necessary functionality but is rather a desired functionality that could be supported if "room exists".

Such computation could involve statistics gathering for traffic analysis etc.

6.4.2 FCS Formulation

Definition 21. *Context Switch Distance (CSD)*
The number of non ctx-trigger instructions between two consecutive ctx-trigger instructions is called CSD.

For our first example in Figure 40.b, CSD=2 for thread 1 and CSD=3 for thread 2. CSD is path dependent. In Figure 42.b, if we insert a ctx instruction as the second instruction in BB3, it will have CSD=3 to the sram_rd instruction in BB1 and CSD=2 to the ctx instruction in BB 2.

Definition 22. *Padding Instruction*
Padding instructions are instructions solely used to increase context switch distance, such as the nops.

Example: the nop in Figure 42.c.

Definition 23. *CSD-k Form and CSD-k Transformation*
If by inserting only ctx instructions, a program can have CSD=k everywhere, then it is in CSD-k form. Here, k is a constant value for that program. The process of converting a program into CSD-k form by inserting padding instructions is called CSD-k transformation.

Thread 1 in Figure 40.b is in CSD-2 form and thread 2 in Figure 40.b is in CSD-3 form. The program in Figure 42.a is not in CSD-3 form, but after adding the nop as in Figure 42.c, it is in CSD-3 form.

Definition 24. *Minimal CSD-k Transformation*
If the number of inserted padding instructions is minimal after CSD-k transformation, the transformation is called a minimal CSD-k transformation.

The two threads in Figure 40.b are minimal CSD-2 and CSD-3 transformed, because no extra padding instructions are inserted.

Notice that we do not consider the number of inserted ctx instructions as a measure of effectiveness of the transformation because our main objective is to minimize the degradation in the execution speed for one thread under the given value of k. The number of inserted ctx instructions largely depends on the value of k (if we want a CSD-4 form, roughly 1/5 of the instructions are ctx instructions) and the choice of k's value is based on the runtime constraint we want to fulfill. For example, for PS, a high priority thread must be given a big k, while a low priority thread must be given a small k. We will show that even with OS scheduling, this overhead is not avoidable. We call such overhead the *Scheduling Overhead*, which is not controlled by our algorithm. However padding instructions come as an artifact of our transformation (in order to transform a flow graph into CSD-k form) and therefore should be eliminated/minimized during CSD-k transformations. It is important to note that padding instructions need not be just nops that do useless work but could comprise optional computation such as statistics gathering for traffic analysis.

6.4.3 CSD-k Form Verification

The first question we need to answer is whether a program is in CSD-k form. Hereby, we propose the *Modulo Marking Algorithm* and the *Modulo Checking Rule*. The modulo marking algorithm attempts to assign each program point a *Modulo Number*, and then the modulo checking rule will check if the modulo number is unique for each program point. First, we give some definitions.

Definition 25. *Program Points*

The intermediate points between instructions.

Definition 26. *Modulo Number*

During modulo marking, each program point is marked with a number from 0 to k-1. If there is a path from a native ctx-trigger instruction or the entry point of the code to a program point P with n intermediate instructions along the path, then the modulo number of P is mod_num(P)=n mod k. If the modulo number is not determined yet, we initialize it as mod_num(P)= $\perp$.

Definition 27. *Start Point*

The program point just before the first instruction in a BB. mod_num(start(B)) denotes the modulo number of B's start point.

Definition 28. *End Point*

The program point just after the last instruction in a BB. mod_num(end(B)) denotes the modulo number of B's end point.

Our basic idea is: ctx-trigger instructions should only exist at the program points with modulo numbers equal to 0. For native ctx-trigger instructions, the two program points before and after them are initially given modulo number 0. Also, the start point of the entry BB is assumed to have modulo number 0 (this is needed to guarantee that each point will get a modulo number eventually). It is easy to prove the following lemma and corollaries, which lay the basis for the modulo marking algorithm.

Lemma 5. *If there is a path from program point P to Q, and mod_num(P)=m, where $m \neq \perp$, then mod_num (Q)= (m + number of instructions on the path) mod k.*

Corollary 1. *If there is a path from a ctx-trigger instruction to instruction Q, then mod_num (Q)=(number of instructions on the path) mod k.*

Corollary 2. *If there is a path between two ctx-trigger instructions, then the path length is a multiple of k.*

As shown in Figure 43, modulo marking is a data flow algorithm to get modulo numbers for the start and end points of all BBs. During propagation, for each BB, its start point should get the modulo number from the end point of one of its predecessors, if that end point has a modulo number. (we use the redefined max() operation for this purpose). During propagation, for each BB, the modulo number of its start point is computed from its predecessors using the lattice shown in Figure 43. There are two cases when propagating inside each BB. For a basic block B, if there is no native ctx-trigger instruction in it, we can use Lemma 5 to get out(B)=(in(B)+Size(B)) mod k. Otherwise, we use Corollary 1 to get out(B) from the position of the last native-trigger instruction. Once the modulo numbers of

Input: CFG G(V,E).

Lattice: $\perp, \top$, 0, 1,…k-1

For any v, max(v, $\perp$)= v, v+$\perp$ =$\perp$, v-$\perp$ =$\perp$, v mod$\perp$=$\perp$,

For any v_1, v_2, $v_1 \vee v_2 = max(v_1, v_2)$,

Initialization:

1. in(entry BB)=0.
2. two program points near each native ctx-trigger instruction are give modulo number 0.
3. other program points have modulo number $\perp$.

Propagation:

$$in(B)= \bigvee_{P \in pred(B)} out(P) \quad \text{if B<> entry BB and in(B)=}\perp$$

$$out(B)= \begin{cases} \text{(in(B)+Size(B)) mod k---if no ctx-trigger inst. in B} \\ \text{(number of inst. from the last ctx-trigger inst to the end of B) mod k---if there is ctx-trigger inst. in B} \end{cases}$$

Output Values:

mod_num(start(B))=in(B)

mod_num(end(B))=out(B)

Figure 43: Modulo Marking algorithm

the start and end points of all BBs are decided by the algorithm in Figure 43, the modulo numbers for program points inside each BB can be inferred with Lemma 5.

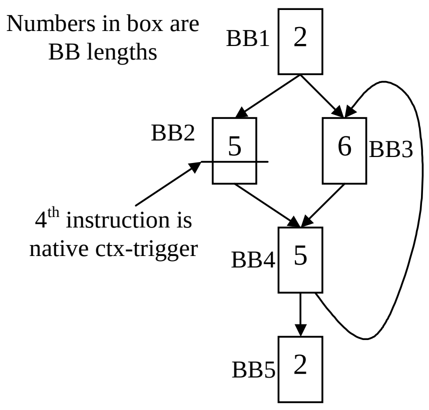
After modulo marking, we can verify whether the program is in CSD-k form by the following *Modulo Checking Rules.*

- Rule 1: Inside a basic block B, for any native ctx-trigger instruction N, (mod_num(start(B)) +(number of inst before N in B) mod k=0.
- Rule 2: For any basic block B, if P and Q are two predecessors of B, then mod_num(end(P)) =mod_num(end(Q)).

If either Rule 1 or Rule 2 fails, the code is not in CSD-k form. Actually, during modulo marking, rule 2 can be used to check if the end points of all predecessors of a BB have the same modulo number.

Theorem 2. *The modulo checking rules are correct*

Proof: If rule 1 fails, there will be a path from a ctx-trigger instruction to the start of B


Numbers in box are BB lengths
BB1
2
BB2
5
6
BB3
4th instruction is native ctx-trigger
BB4
5
BB5
2

BB#	1in	1out	2in	2out	3in	3out	4in	4out	5in	5out
Init	0									
Iter 1	0	2		1						
Iter 2	0	2	2	1	2	2	1	0		
Iter 3	0	2	2	1	2	2	1	0	0	2

Figure 44: Example for CSD-3 verification

and then to N, but its length is not a multiple of k, which contradicts with Corollary 2.

If rule 2 fails, there must be a program point P, such that two modulo numbers are possible. Suppose two modulo numbers m1 and m2 are possible for P. We can find a nearby program point with two possible modulo numbers: 0 and (m2-m1) mod k. In other words, there are two paths from native ctx-trigger instruction(s) to this instruction such that one path length is a multiple of k but the other is not. So padding instructions must be added to resolve this conflict.

On the other hand, if both rules hold, the modulo number of each program point is unique. We can insert ctx instructions to all program points with modulo number 0 (except those with native ctx-trigger instructions). It is easy to verify CSD=k for all consecutive ctx-trigger instructions.

Figure 44 gives an example which shows how to verify if a program is in CSD-3 form. The number in each BB represents the size of the BB. The fourth instruction in BB 2 is a native ctx-trigger instruction. The entry point (start point of BB 1) always has modulo number 0 (a native ctx-trigger instruction is assumed there). In iteration 1, the end point of BB 2 gets modulo number from the ctx-trigger instruction in that BB. In the meantime, the end point of BB 1 is assigned a modulo number from its start point. In iteration 2 and 3, all BBs get modulo numbers. Next, we apply the Modulo Checking Rules. In BB 2, mod_num(start(BB 2))=2 and there are 3 instructions before the fourth native ctx-trigger instruction, however (2+3) mod 3 $\neq$ 0. This breaks rule 1. It is easy to see that there are 5 instructions between the entry point of the code and this ctx-trigger instruction along the path, which violates Corollary 2. Next, for BB 3, the two predecessors are BB1 and BB4. However, mod_num(end(BB 1)=2, and mod_num(end(BB 4))=0. This breaks rule 2. We can identify two paths: one from the ctx-trigger instruction at the beginning of the code going through BB 1 to the start point of BB 3 with length 2; the other goes from the ctx-trigger instruction in BB 2, traversing through BB 4 till the start point of BB 3 with length 6. The second path suggests that there should be a ctx instruction at the beginning of BB 3, however we must add more padding instructions to the first path to make its length a multiple of 3.

6.4.4 Modulo Graph

Next, we present another representation for the control flow graph, which facilitates CSD-k form verification and minimal CSD-k transformation. We first introduce the concept of a *Connection Point.* Notice that if the CFG is in CSD-k, according to modulo checking rule 2, the modulo number of a BB's start point must equal to the modulo number(s) of the end point(s) of its predecessor(s). Therefore, we can assume a single point for these start and end points. A connection point is defined as follows:

Definition 29. *Connection Point*
A set of start and end points. It is the transitive closure of start and end points through the predecessor/successor relationship on the CFG. If the code is in CSD-k form, the modulo number of a connection point is the modulo number(s) of its element(s).

For example, in Figure 44, there are four connection points, i.e. end (BB1), start (BB2), start (BB3), end(BB4), start(BB5), start(BB4), end(BB2), end(BB3), start(BB1) and end(BB5). In addition, it is easy to observe that:

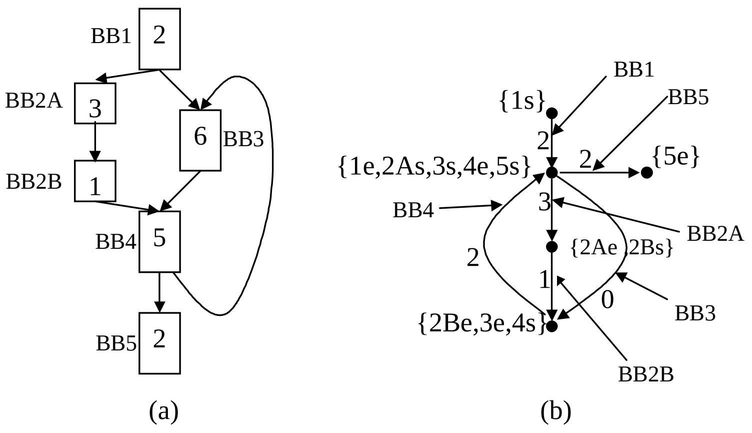


Figure 45: Modulo graph for the example

1. A start or end point must belong to one and only one connection point. In other words, any two connection points cannot share elements.

2. If the code is not in CSD-k form, the modulo number of a connection point may not be determined due to the inconsistency of its elements' modulo number.

3. Each BB links one connection point to another connection point.

Next, we define modulo graph.

Definition 30. *Modulo Graph*

A weighted directed graph, on which nodes are the connection points. Each BB becomes a directed edge connecting one connection point (which contains its start point) to another connection point (which contains its end point). The weight of an edge is: (size of the BB) mod k. Each ctx-trigger instruction splits its BB and becomes a connection point with modulo number 0.

In Figure 45, we show how to build a modulo graph for the example in Figure 44. In Figure 10.a, BB 2 is split at the native ctx-trigger instruction and becomes BB2A and BB2B. In Figure 45.b, the modulo graph is constructed. The notation in the figure needs explanation. Here, 1s means start(BB1) and 5e means end(BB5), etc.

Next, we give the definitions of ground point.

Definition 31. *Ground Point*

A ground point is a connection point that contains the start point of the entry BB or a ctx-trigger instruction after that splits a BB. Its modulo number is 0.

In Figure 45.b, {1s} and {2Ae,2Bs} are ground points.

Definition 32. *Super Path, Positive Edge, Negative Edge*

A super path is a path on the modulo graph regardless of the direction of the edges. If an edge follows the direction of the super path, we call it a Positive Edge, otherwise, it is a Negative Edge. Length of a Super Path is defined as ($\sum$(length of all positive edges)-$\sum$(length of all negative edges)) mod k

For example, on Figure 45.b, there is a super path BB1 $\rightarrow$ BB4 $\rightarrow$ BB2B, BB1 is a positive edge, BB4 and BB2B are negative edges. Its length is (2-2-1) mod k.

Definition 33. *Super Loop*

A closed super path. Its length is the length of that super path.

Lemma 6. *If a program is in CSD-k form, after modulo marking, for any super path from P to Q, its length is (mod_num(Q)-mod_num(P)) mod k*

Proof: One observation is that if a program is in CSD-k form, according to rule 2, all program points belonging to the same connection point should have the same modulo number. Path from P to Q can be separated into many edges $P \rightarrow C_1 \rightarrow C_2 L Q$, based on Lemma 5, (mod_num(P)+ $\sum$(length of all positive edges)- $\sum$(length of all negative edges)) mod k=mod_num(Q). Therefore, length of the super path = (mod_num(Q)-mod_num(P)) mod k.

Theorem 3. *The Modulo Graph is in CSD-k form iff the length of any super path between two ground points is zero and the length of any super loop is zero.*

Proof: If the Modulo Graph is in CSD-k form, based on Lemma 2, any super path between two ground points is zero because mod_num(Q)=mod_num(P)=0. For any super loop, we can pick an arbitrary point P on the loop, and regard it as a path from P to P. Since mod_num(P)-mode_num(P)=0, its length is 0.

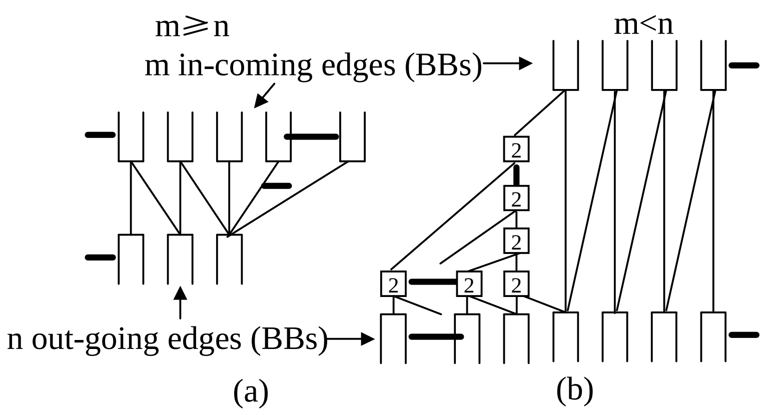


Figure 46: Illustration for the NP-complete proof

On the other hand, if any super path between two ground points has length 0 and any super loop has length 0, we first apply the modulo marking algorithm on the graph, then check if the two rules are satisfied.

Rule 1: Assume N is the first native ctx-trigger instruction in the BB. Now it becomes a ground point. Suppose start(B) belongs to the connection point CP, according to the modulo marking algorithm, there is a super path from a ground point G to CP, therefore mod_num(CP)=length of super path G to CP. We extend this super path to N, therefore the length of the extend path is 0 (between two ground points). Thus, rule 1 is correct. If N is not the first ctx-trigger instruction. It must has a path with length equal to a multiple of k to the first ctx-trigger instruction, thus, rule 1 is still valid.

Rule 2: We can find a ground point GP and GQ, so that there is a path from GP to P with length mod_num(end(P)), and there is path from GQ to Q with length mod_num(end(Q)). Since, now both end(P) and end(Q) belong to the same connection point, we can combine the two paths into a path from GP to GQ, which should have length 0. Thus, mod_num(end(P))= mod_num (end(Q)).

Theorem 4. *The Minimal CSD-k problem is NP complete.*

Proof: We reduce the *Maximal Bipartite Subgraph* problem [57] to CSD-2. Proof for higher value k is omitted due to the space limitation. Maximal bipartite subgraph problem is to delete minimal number of edges from an undirected graph to make it bipartite, i.e.

no odd length cycle in the graph. For each instance of a graph, we build a modulo graph accordingly. All nodes and edges are mapped to the modulo graph and all edge weights equal 1 (we will show details later). No native ctx-trigger instruction is in the graph (ctx-trigger instruction at the entry point can be omitted, since we can always flip the modulo numbers, i.e. $0 \rightarrow 1$ and $1 \rightarrow 0$, of all program points to make the entry point has modulo number 0). According to the constraints in Figure 47, for each edge, we either add 1 instruction to make it length 2 or leave it unchanged. The objective is to make minimal number of edges into length 2 (since 2 mod 2=0, making the edge length 2 is to remove the edge from length counting). After the CSD-2 transformation problem is minimally solved, all super loop lengths should equal 0, i.e. all cycle length on the original graph should be an even number. Therefore, solving the minimal CSD-2 transformation problem is equivalent to solving the maximal bipartite subgraph problem.

To map the graph to a modulo graph, all nodes are ordered first. Edges (BBs) always flow from lower number nodes to higher number nodes. Assume for each node (i.e. a connection point), there are m in-coming edges and n out-going edges. We show the mapping so that each BB has at most two successors as required on IXP. If m≥n, we show the connection point in Figure 46.a below. If $m < n$ (Figure 46.b), we need some additional BBs. All additional BBs have size 2 and it is easy to show that no padding instruction should be added to these BBs in the optimal solution.

6.4.5 Solving Minimal CSD-k Transformation

Modulo Graph gives an elegant way to model the problem mathematically. By defining the number of padding instructions in each BB as a variable and their total number as our objective function that should be minimized, we can set up a number of constraints and solve this problem mathematically. In Figure 47, we assume that there are n connection points and p (p≤n) of them are not ground points. The modulo number of the connection points are defined as variables $CP_1, CP_2 \ldots CP_n$. The number of edges (BBs) is m; thus for each edge (BB), we define the number of padding instructions as variables $PI_1, PI_2, \ldots PI_m$. The objective is to decide CPs and PIs such that the total number of padding instructions

Variables:

1. Connection Points: CP_1, $CP_2 \ldots CP_n$
   Among them CP_1, $CP_2 \ldots CP_p$ are not ground points
   $CP_{p+1}, CP_{p+2} \ldots CP_n$ are ground points
2. Number of Padding Instructions for each edge (BB):
   PI_1, $PI_2 \ldots PI_m$

Constants:
Edge Weights: E_1 $E_2 \ldots E_m$

Objective:
Min ($PI_1 + PI_2 + \ldots + PI_m$)

Constraints:
(1) All variables ϵ [0, k-1]
(2) For edge E_i from CP_j to CP_k,
($CP_j + E_i + PI_i$) mod k=CP_k (Lemma 1)
(3) $CP_{p+1} = CP_{p+2} \ldots = CP_n = 0$

Figure 47: Mathematical modeling

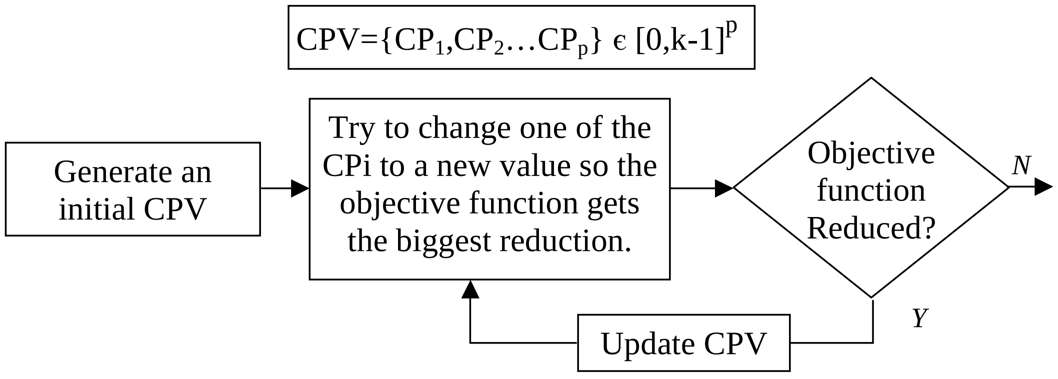
is minimized.

Obviously, all variables should be in the range [0, k-1]. For an edge (BB) from connection point CP_j to CP_k, it must satisfy $(CP_j + E_i + PI_i)$ mod k=CP_k (Lemma 5).

This modeling gives us a way to solve the problem using techniques like integer linear programming (ILP)(modulo operations can be converted in ILP). However, due to its NP-completeness, resorting to an ILP solver would take a long time [2]. Next, we give a greedy heuristic solution, which finds an approximate solution quickly.

From Figure 47, we can observe that the solution should determine CP_1 to CP_p. We can group these values into a p-dimensional vector called CPV. After the p values (or the CPV) are determined, using constraint (2) in Figure 47, all PIs and the value of the objective function are decided as well. Therefore, we have a function from $CPV \in [0, k-1]^p$to an integer. The problem is to find the minimal value of this function.

Since the greedy heuristic may end up with a local minimum depending on the initial value, our heuristic starts with a number of (we set it to 1000) initial values for CPV. These initial values are randomly generated. From each initial CPV, it attempts to reduce the objective function in a greedy manner (Figure 48). Each time, one of the CP_i is changed



CPV={CP1,CP2…CPp} ϵ [0,k-1]^p
Generate an initial CPV
Try to change one of the CPi to a new value so the objective function gets the biggest reduction.
Objective function Reduced?
N
Y
Update CPV

Figure 48: Heuristic algorithm

to a new value so the objective function gets the biggest reduction. The complexity of the algorithm can be calculated as O((max #iteration) × (#new values tried in each iteration) × (time to calculate the objective function))= $O(mk \times kp \times m) = O(m^2k^2p)$.

6.4.6 Relaxed Minimal CSD-k Transformation

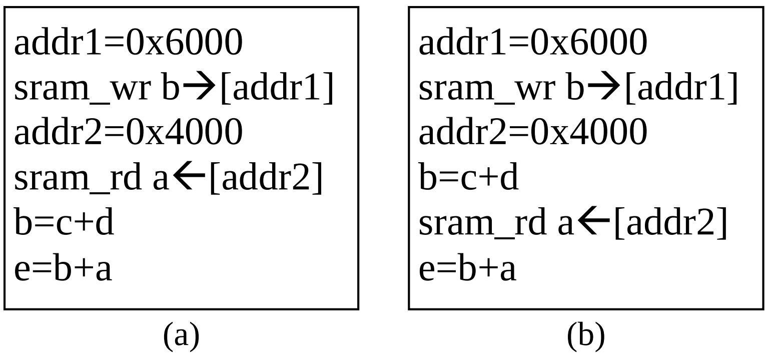
Enforcing CSD strictly could be expensive. In real world applications, a rough approximation of the context switch distance is typically enough. To relax the CSD-k transformation, we can slightly change the formation in Figure 47. The second constraint is relaxed with a D on the right hand side and becomes:

$$(CP_j + E_i + PI_i) \bmod k = CP_k \pm \Delta$$

In this way, we allow the CSD to be slightly changed along each path. In our experiments, we found that even a small Δ could help reduce the padding overhead a lot. Accordingly, the heuristic algorithm in Figure 48 must also take this into consideration by factoring in the change in the second constraint.

6.4.7 Overhead Reduction via Code Transformations

The above approaches attempt to insert padding instructions without other code transformations. In this section, we discuss several compiler techniques that can reduce the number of padding instructions. All these optimizations are applied before the minimal CSD-k transformation.



```
addr1=0x6000
sram_wr b→[addr1]
addr2=0x4000
sram_rd a←[addr2]
b=c+d
e=b+a
```

(a)

```
addr1=0x6000
sram_wr b→[addr1]
addr2=0x4000
b=c+d
sram_rd a←[addr2]
e=b+a
```

(b)

Figure 49: Code motion to increase distance

6.4.7.1 Code Motion for longer Context Switch Distance

When native ctx-trigger instructions are located too close to each other, the compiler can move these instructions to increase the distance. Code motion is restricted by dependencies among instructions. The example in Figure 49.a shows a code segment, in which the two ctx-trigger instructions have distance 1. If we want it to be in CSD-2 form without extra padding instructions, we can simply move the sram_rd instruction down by one slot (Figure 49.b). Our approach first finds all ctx-trigger instruction pairs with distance shorter than k, then attempts to move them one by one. It might happen that moving one instruction can affect several pairs. We heuristically require that the sum of the distances for all pairs should be increased by each code motion.

6.4.7.2 Distance Compensation

Due to the existence of native ctx-trigger instructions, sometimes we must insert a few padding instructions to increase the distance between these instructions. For example, in Figure 50.a, the two native ctx-trigger instructions sram_rd and csr_rd are at CSD=2. If we want to convert the code into CSD-3 form, a padding instruction must be added between them (Figure 50.b). However, since the code is in the same BB, it will be executed with the same frequency, we can keep the CSD between these two native ctx-trigger instructions, but give a longer CSD later to compensate. Overall, the CPU time given to the thread is not changed. As shown in Figure 50.c, the ctx instruction is added 4 instructions away from the csr_rd. One limitation is that distance compensation may shorten the round trip

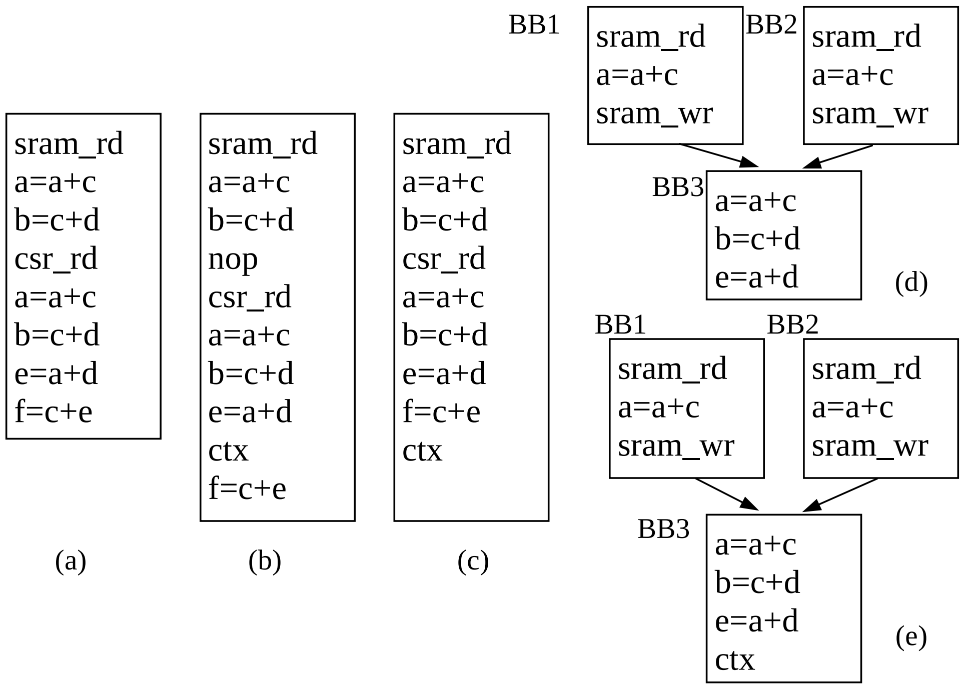


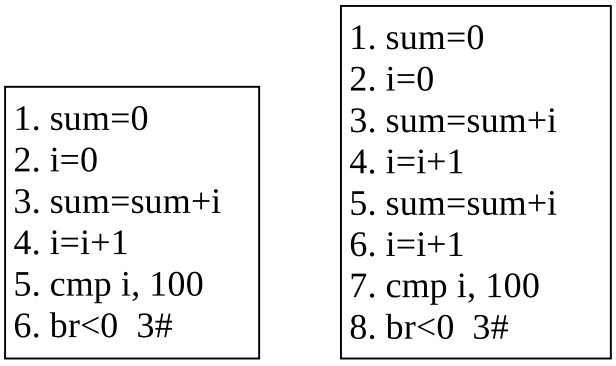
Figure 50: Distance compensation

time. We should make sure that time for each round trip is still bigger than the stall time of each thread, such that threads can be ready when scheduled again.

Moreover, we can apply the same technique across basic block boundaries. In Figure 50.d, the CSDs are all 1 in BB1 and BB2. To make the code CSD-2, we need at least two padding instructions. However, in Figure 50.e, we compensate it by lengthening the CSD in BB3, as the sum of the execution frequency of BB1 and BB2 equals the execution frequency of BB3.

6.4.7.3 Loop Unrolling

For loops that are too short or their lengths are not a multiple of k, loop unrolling can help reduce the number of padding instructions. In Figure 51.a, the loop contains only 4 instructions, i.e. instruction 3 to 6 (for clarity, branch delay slots are ignored). If we want it to be in CSD-8 form, 4 nops should be added. However, in Figure 51.b, we can unroll the loop so no padding instruction is needed. Actually, we replace padding instructions with the unrolled loop body. Although the code size is still increased, the runtime slowdown due to useless padding instructions can be avoided.



```
1. sum=0
2. i=0
3. sum=sum+i
4. i=i+1
5. cmp i, 100
6. br<0  3#
```

(a)

```
1. sum=0
2. i=0
3. sum=sum+i
4. i=i+1
5. sum=sum+i
6. i=i+1
7. cmp i, 100
8. br<0  3#
```

(b)

Figure 51: Loop unrolling

6.4.8 Apply to Runtime Constraints

We now describe briefly how to apply FCS to the other runtime constraints in Table 16. WRR and PS were illustrated in Section 6.4.1. For RM, among all threads in their processing part, the one with shortest period occupies the CPU, so idle parts are given a very short CSD. Processing parts are given a longer CSD for shorter period threads. PC is similar to RM, except that threads with higher priority classes are given a longer CSD. Finally, as explained in Section 6.3, WFQ is similar to WRR.

6.4.9 Other Considerations

Finally, it might be noted that the choice of k really depends upon the application needs. The value of k should be picked depending on the runtime constraint an application needs to fulfill (for example, in PS, the thread with lowest priority must have a very small k). When there are multiple options, the programmer can make the decision based on the scheduling overhead (due to ctx insertions) and the padding instruction overhead given by our algorithm. As mentioned in Section 6.4.2, scheduling overhead largely depends on k's value, therefore choosing the biggest CSD allowed can minimize the scheduling overhead, i.e. the dynamic number of ctx instructions executed. However, a big CSD could introduce more padding instructions. Thus, there is an inherent tradeoff involved between the number of ctx instructions and padding instructions in the range of allowed k values. The programmer can use our algorithm to find the overhead for different options, then decide which one to

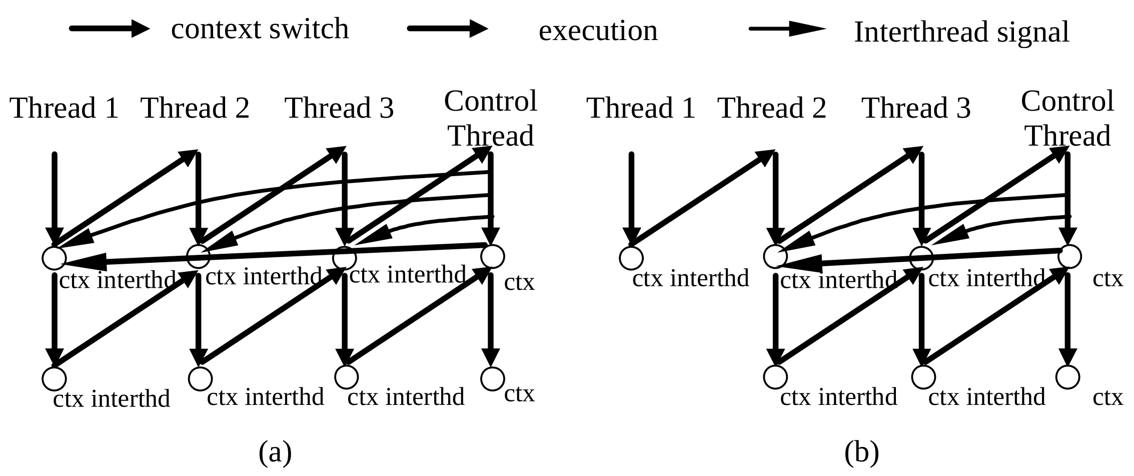


Figure 52: Dynamic Context Switch

choose.

6.5 Dynamic Context Switch (DCS)

To control the context switch more adaptively, DCS makes use of the inter-thread signal to control the length of time a thread should be delayed. Instead of ctx, "ctx interthd" is issued. This will cause the thread to be stalled until the inter-thread signal is received (which works like a wake-up signal). Moreover, a *Control Thread* is needed to send inter-thread signals to control their execution.

In Figure 52.a, we designate the fourth thread as the control thread. All threads are CSD-k transformed. The control thread sends out inter-thread signals according to the runtime policy. In Figure 52.a, when control thread is running, it sends out signals to other 3 threads. Therefore, in the next round, the other 3 threads can proceed to the next "ctx interthd" instruction. In Figure 52.b, we show how one thread is stalled by DCS. If the control thread only sends out signals to thread 2 and thread 3, thread 1 will be stalled in next round. The control thread knows which thread to stall by reading their registers. As mentioned earlier, the register file is accessible to all threads, but each thread only uses part of the register file. The control thread can look at particular registers belonging to other threads to decide which thread(s) to stall.

For example, DCS can implement the two runtime constraints in Table 16-FCFS and

Table 17: Benchmark properties

Benchmarks	Code size	Cycle/ iteration	#native ctx-trigger
Drr	108	207037	9
Fir2dim	447	159149	13
Frag	271	28620	14
Kmp	123	148059	15
Lzw	126	43163	13
Md5	913	3983292	49
Wraps_recv	875	2048.37	35
Wraps_send	921	1264.87	37

EDF. For FCFS, each thread should have a register indicating the timestamp of the first packet in its input queue (or no packet, the thread is in idle part). Once the control thread detects the thread with earliest packet to service, it blocks all other threads from executing in their processing part until that thread finishes. For EDF, the deadlines are stored in threads' registers. Only the thread with earliest deadline can be executed in its processing part.

The overhead of DCS includes: 1) one thread is taken as control thread and the control thread occupies part of the CPU time; 2) some registers are used to convey information to the control thread. 3) inter-thread signals cannot be used for other purposes.

6.6 Evaluation

We evaluate the algorithms with the Intel-provided IXP1200 Developer Workbench 2.01, including a cycle-accurate simulator for IXP. We experiment with 8 benchmark programs collected from Commbench [72], Netbench [51], one program from DSPstone (fir2dim) and a packet scheduling algorithm [75]. We apply our algorithms on generated assembly code to automatically insert padding and ctx instructions. We also observe that unnecessary stalls are reduced.

Table 17 shows some properties of the benchmark programs. The code size is the number of instructions before our optimization. The cycle count is the average number of cycles for

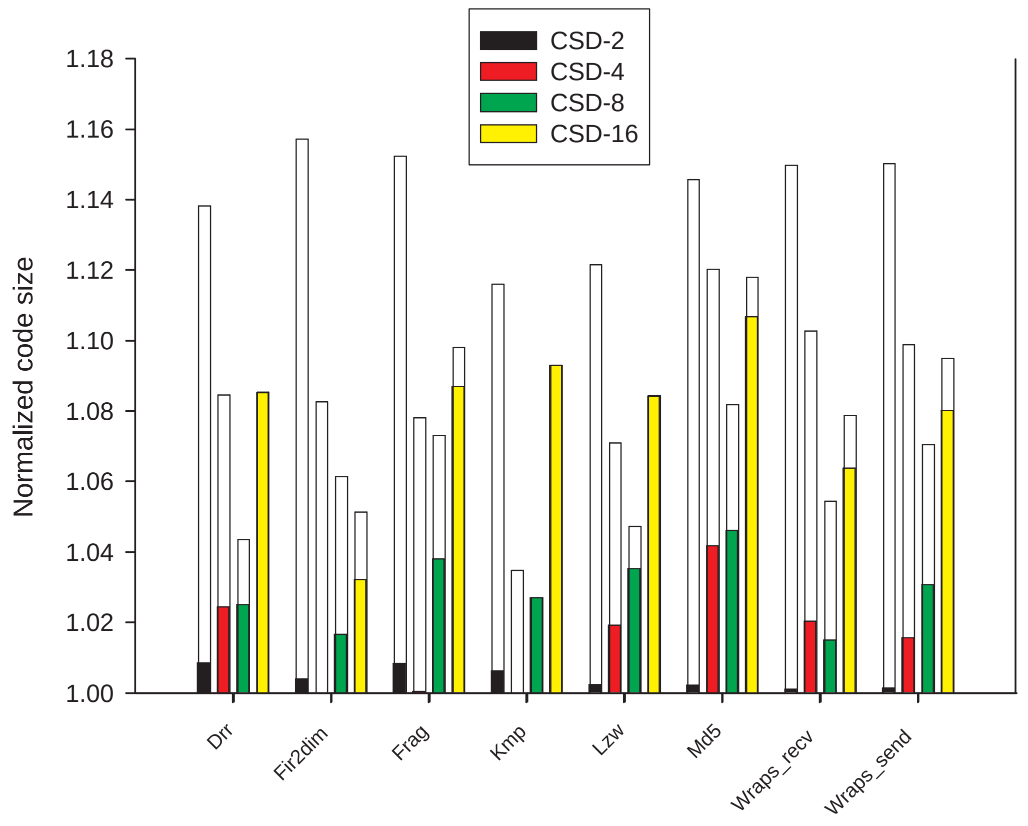


Figure 53: Code size growth under different CSDs

each iteration in the processing part. The last column gives number of native ctx-trigger instructions in the code. Since most of these instructions are for data input/output, we observe that it takes higher percentage in small benchmarks.

In all cases, we use relaxed minimal CSD-k transformation and set Δ to 1. Figure 53 gives code growth due to padding and ctx instructions after applying our CSD-k transformations. Data is normalized to the original code size. The bottom part of each bar represents code growth due to padding instructions, while the upper part represents code growth due to ctx instructions. Total code sizes are shown as absolute numbers at the top of each bar. In general, with larger CSD, more padding instructions are added. On average, the code growth due to padding instructions is 0.4%(CSD-2), 1.3%(CSD-4), 2.9%(CSD-8), 7.9%(CSD-16). Notice that, large benchmarks induce fewer padding instructions (except md5, which has too many native ctx-trigger instructions). It is probably due to their bigger

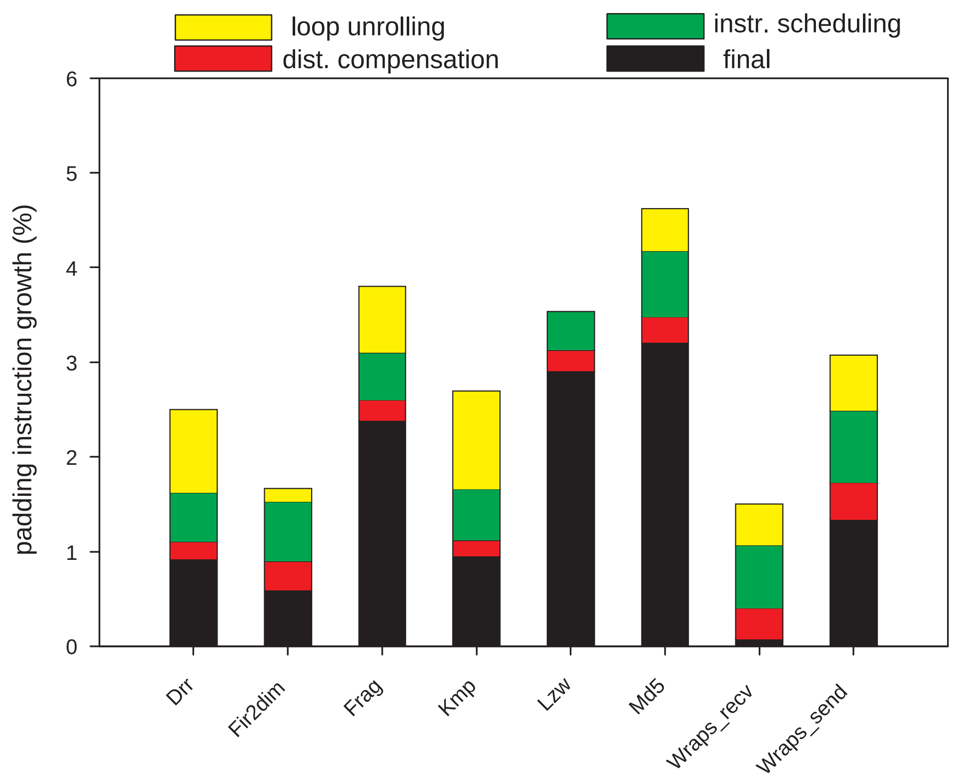


Figure 54: Padding instruction reduction through code transformation (CSD-8)

BBs as we have observed in these benchmarks. The number of additional ctx instructions is in reverse proportion to the CSD. Also, if the code contains more native ctx-trigger instructions, fewer extra ctx instructions are needed. On average, the code growth for ctx instructions is 13.7%(CSD-2), 7.0%(CSD-4), 2.8%(CSD-8), 0.8% (CSD-16). Finally, the average numbers for total code growth are 14.1%(CSD-2), 8.4%(CSD-4),5.7%(CSD-8), 8.7%(CSD-16).

Figure 54 demonstrates the effectiveness of the 3 code transformations in Section 6.4.7. All numbers are normalized to the original code size. Only code growth due to padding instructions is considered. Loop unrolling replaces padding instructions with useful loop body, which reduces padding instructions by 0.5%. The effectiveness of loop unrolling depends on how many small loops are encountered. With instruction scheduling, code growth is cut by 0.6%, while distance compensation contributes about 0.3%. We also observe that distance compensation is more useful when the CSD is small, because less

Table 18: Runtime performance

Constraint	Benchmark	Orig. Stall Cycle (%)	CSD	CPU time (%)	Padding (%)
WRR	Frag	22.5	8	61.23	2.83
	Lzw		4	31.01	1.2
PS (FCS)	Wraps_recv (high)	24.3	16	89.56	7.41
	Wraps_recv (low)		2	7.15	0.1
PS (DCS)	Wraps_recv (high)	24.3	8	70.67	2.5
	Wraps_recv (low)		8	0	0

distance needs to be compensated.

Table 18 illustrates the runtime performance for 3 scenarios. In the first scenario, we run two different applications on threads. For the other two, the same application is run on two threads with different runtime constraints. third column shows the cycles spent on unnecessary stalls in the original code. They are quite significant.

The first scenario schedules two programs-Frag and Lzw-on two threads with WRR. For WRR, we can choose a large CSD for both threads to reduce scheduling overhead. After balancing both overheads, we found that CSD-8 for Frag and CSD-4 for Lzw are the best. Simulation shows Frag takes 61.23% of the CPU time, while Lzw takes 31.01% of the CPU time. The ratio is very close to the ideal ratio of 2:1. The imprecision might come from stalls due to long latency ctx-trigger instructions, which are not fully covered when the total cycles of each round are less than the latency of such instructions. This is likely to happen since there are only two threads running together. The cycles spent for unavoidable scheduling overhead are roughly 8%. The last column shows the percentage of cycles spent on padding instructions, which is less than 4% – a very small amount. As a matter of fact, the original code of the two threads wastes over 20% CPU due to improper sharing of CPU and long latency instructions. In other words, with precise timing control of the threads, CPU cycle wastage is negligible.

The second scenario tests priority sharing (PS) for two threads running the same benchmark. With FCS, the first thread (high priority) should be CSD-16 transformed and the second one (low priority) should be CSD-2 transformed. During runtime, the first thread takes most of the CPU time (89.56%), while the second gets 7.15%. Therefore, it approximates PS closely. Cycles wasted for padding instructions take over 3.7% due to the large CSD value given to the first thread. The scheduling overhead is about 4%. A precise implementation should rely on DCS as shown in the third scenario. Both threads are CSD-8 transformed, so we get a small overhead due to ctx and padding instructions (as shown in Figure 53). With DCS, the second thread is completely stalled, which means it truthfully enforces PS. Compared with FCS, the overhead of DCS is higher due to the control thread (about 30%), but the padding instruction overhead is only 2.5% for the first thread, because we do not need to enforce a long CSD in the code as long as the control thread can take control of its execution. Notice that, for FCS we actually save the CPU cycles compared with the original code with over 20% CPU time loss due to unnecessary stalls for long latency instructions. DCS has higher overhead but enforces the constraints with high fidelity.

Finally, the running time of our algorithm is only a few seconds even for the biggest benchmark. Since the algorithm is in polynomial time, it should scale very well for larger benchmarks.

CHAPTER VII

RELATED WORK

The research problems we have encountered for network processors are very special, because network processors are designed for meeting the new requirements of network applications that have not been targeted by any prior processors.

7.1 Intra-thread Register Allocation

Although our main focus of intra-thread register allocation is on optimizations for a special network processor architecture with dual-register file, partitioned register file has long been adopted by commercial DSP products like Texas Instruments' VLIW chips. The IXP network processor's register file differs from theirs in that 1) only one function unit is processing the instructions; 2) the parallel access to the register file is restricted to the two source operands of the ALU instruction. A recent architecture paper [22] studies multi-banked architecture and shows its performance advantage.

[39] talks about register allocation for VLIW machines. Although their problem is different from ours, the proposed component graph method gives us valuable enlightenments.

[58] studies the register allocation problem for a dual-bank register file. Register access requires both register number and a bank specifier, which is decided by a control register. The paper proposes to use one register bank called primary bank for global register allocation and the other called secondary bank for local register allocation. Since their architecture does not require source operands to be in different banks, the approaches used are different from us. Also, the separated global and local register allocation may cause imbalanced register pressure as discussed in this work.

7.2 Inter-thread Register Allocation

The multi-threaded architecture of the IXP model differs from traditional general purpose multi-threaded processors or SMT processors in that the number of threads and the code for each thread is known beforehand (at compile time). Further, the architecture exposes context switch to users (actually the programmer should handle everything except the save/restore of PC for each thread). This makes register sharing across threads difficult, since registers are not saved during context switches. On the other hand, exposed architecture features allow the compiler to undertake inter-thread register allocation.

This problem is also different from the traditional concepts of caller-save and callee-save registers, since registers cannot be saved to the memory during context switches due to the high cost of memory operations. The only chance to use a register that is also used (shared) by other threads is to guarantee the register is dead during the context switch. [30] studies live range analyses for context switches at the procedure boundary on Alpha machines. Optimizations are conducted to minimize the number of registers that should be saved during context switches–in [30] it is equivalent to reduce the number of callee-save registers. In contrast, context switches taking place on the network processor are more frequent (reaching basic block level) thus require analyses at finer granularities; it is not profitable to save/restore but allow small deadness at context switch points for a fine granularity allocation.

[6] talks about the inter-task register allocation problem for embedded systems with static OS and predetermined tasks. The goal of the paper is to minimize the number of registers that should be saved during context switches. However, the assumption that tasks have fixed priority and saving the variables to memory during context switch is not applicable in our model. Finally, their assumption that tasks can be preempted at any given program points except for critical sections is not true for our network processor model either. Therefore, the techniques proposed in their work are not applicable to our problem.

A recent publication [29] studies register allocation problem for single thread on the IXP network processor. The compiler is dedicated to the particular processor with consideration to many architectural details that involve irregularities but they do not focus on an important IXP feature–multi-threading. The goal of our paper is to study a generalized model for multi-threaded register allocation, so it can be extended to other network processors with similar designs.

Register renaming is an old concept in superscalar processors[54, 65]. There are two key differences between the renaming mechanism presented here and the renaming unit in a superscalar processor. The hardware presented here attempts to rename memory addresses to registers, rather than renaming virtual registers to physical registers. Also, the goal here is to reduce memory activity, while traditionally it is to remove data dependencies among instructions, increasing the amount of parallelism.

We focus on the balancing of registers among different threads and the allocation of shared registers to meet the overall register demands across threads. We show that the problem is quite involved and provide a systematic solution to balance register requirements across threads by determining the number of registers that can be put into shared registers to reduce the overall register requirements.

7.3 Managing Runtime Constraints

Most previous compiler work for real-time or packet processing are at the language level. Realtime programming languages like Ada can provide language support for helping specify and implement realtime programs. However, they require realtime constraints to be written explicitly by the programmer or indirectly through a type system. However, no implementation details with regard to the particular platform are considered. In this work, we are interested in a very efficient solution for a specialized network processor. In addition, the runtime constraints under study are not limited to realtime constraints.

PLAN [33] is a language that can be used to program mobile agents that evaluate on remote hosts across the network. It also provides language features such that the computation could be carried out safely and all programs will terminate eventually. In contrast, our work on the network processor is at a much lower level, which implements some of the functionalities of the OS and packet scheduler. Besides, the compiler analysis and optimization proposed in this work do not require any language changes, therefore is completely transparent to the programmers.

Real-time operating systems (RTOS) like eCos, RTLinux can manage runtime constraints. However, they are typically too expensive to be implemented on network processors. Even a small amount of OS intervention is imposed, the overhead could be unbearable for fast packet processing. Thus, OS solutions to such processors are not really feasible.

Although implementing runtime constraints through compiler has limitations. For example, only some functionalities are implementable with FCS, whereas DCS is much more expensive than FCS. Also, padding overhead and scheduling overhead have to be considered. It's probably the only feasible way for the IXP network processor due to the lack of OS and full-fledged hardware support.

7.4 Others

The ShangriLa compiler [16] developed by Intel helps the programming on the IXP network processor by defining a C-like high-level language, together with a number of compiler optimizations to speedup the execution. The infrastructure relies on a runtime system. It also proposes stack optimization and profile directed soft cache. We are currently not sure about how register allocation is conducted in ShangriLa, but it is believed to share some similarities with our register allocation pass (either intra-thread or inter-thread register allocation). Satisfying runtime constraints through pure compiler techniques hasn't been considered in the ShangriLa compiler.

J. Dai et. el. [23] proposed a technique to automatically partition a sequential packet processing program into parallel subtasks which can be naturally pipelined and mapped to the network processor. The transformation ensures that packet processing tasks are balanced among pipeline stages. Also, minimizing data transmission between pipeline stages is part of the objective. A compiler phase is designed for the IXP network processor, which achieves significant speedup for commonly used NPF IP forwarding benchmarks. Although code and data partitioning hasn't been seriously studied in this work. We are also interested in this topic and plan to work on it in the future.

Much research has been conducted from other aspects such as application level implementations, architectural design optimizations etc. Since the focus of this work is on compiler optimizations, we will briefly mention these related work. A recent workshop on network processors held with HPCA collects many interesting papers in this area.

For example, [62] describes the design and implementation of the Dynamic Window-Constrained Scheduling (DWCS) [70, 69, 68] algorithm to schedule packets on network processors. The DWCS algorithm characterizes multimedia streams with diverse Quality of Service (QoS) requirements. Earlier implementations of DWCS on Linux and Solaris machines use a heap-based implementation, which requires O(n) time to find the next packet and send it out, and which frequently moves heap elements. For speed improvements and conservation of memory bandwidth, our design uses a Hierarchically Indexed Linear Queue (HILQ). The HILQ substantially reduces the number of memory accesses by scattering packets sparsely into the queue. Experimental results demonstrate improved scalability compared to a heap-based implementation in supporting thousands of streams with strict real-time constraints, while causing no loss in accuracy compared to the standard DWCS algorithm.

Various of hardware improvements or special hardware units are explored in literature as well. For example, [61] talks about how to design pipelined memory architecture to improve the throughput of network processors. [44] and [32] proposed microarchitectural design for routers. Architectural innovations for IP look-up are studied in [5, 4, 40]. [60] evaluates flexible network processor interfaces. Some other special hardware units for packet buffers [27], regular expression [9], priority queues [80], packet classification [14] are also evaluated. For special applications running on network processors, [73] explores how to accelerate protein motif finding and [21] looked at mixed real-time workloads.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1 Conclusion

We have looked at several important and also interesting problems regarding compiler optimizations for the IXP network processor. Largely, they can be categorized into two types. The first big category is the register allocation problem for both intra-thread and inter-thread register allocations. The second problem is how to manage runtime constraints across multiple threads for the IXP.

The study on intra-thread register allocation [76] discusses three different approaches for performing register allocation and bank assignment. Bank assignment can be performed before register allocation, after register allocation or combined with register allocation. We propose a structure called register conflict graph (RCG) to capture the dual-bank constraints. To further improve the effectiveness of the algorithm, we also propose some enabling transformations. Our results show that the phase ordering of first doing register allocation and then assigning banks can reduce the number of spills with affordable cost in terms of additional instructions.

Allocating registers across threads [77] could further increase the number of effective registers by sharing some of the registers across threads. The values that are not live across context switch program points are held in shared registers. Maximizing shared registers in turn reduces spills and context switches making it safer to keep more ranges in shared registers. We approach this problem from zero-spill accounting only for mandatory load/stores. The results show that we are able to minimize register requirements in SRA setting and are able to improve the cycle counts substantially in the ARA setting for large benchmarks executing on different threads. This means that it is viable to develop multi-threaded large applications on IXP effectively with a good compiler support. The solution is able to speedup the performance of critical threads by meeting their demands through maximal sharing of registers.

The dynamic register allocation approach [18] presented earlier attempts to go beyond the best statically available allocation techniques, by combining static analysis with dynamic allocation. By dynamically mapping memory addresses onto registers, it can reduce the total number of dynamic memory operations. In turn, reducing the total number of memory operations reduces the idle cycle count, which is the goal of all optimizations for systems with real-time constraints. The results show that this approach is able to reduce idle cycle counts in all benchmarks and achieve an unweighted average decrease of 51% in idle cycles with a 8 cycle latency. These results also show that idle counts can be reduced even further if hardware supporting next neighbor registers is available. The hardware overhead introduced by our method is insignificant and is off the critical path. It demonstrates that it is viable to use smart dynamic allocation techniques over existing static algorithms.

To manage runtime constraints, our work [79] provides an effective framework for realizing these constraints purely via compiler-controlled context switches across threads. Without our compiler optimizations, these constraints can only be enforced by the programmer, which is time-consuming and error-prone (or even impossible in some cases), and it is very difficult to achieve with a reasonably small overhead. Although our approach introduces extra no-ops, we have shown that they do not pose a big performance drag, the code growth is also small. Also, we put the guarantee for runtime constraints at the highest priority (as actually intended by the applications); the paper gives a way to achieve these otherwise impossible goals on the particular network processor. Note that most applications currently running on network processors haven't seriously considered runtime constraints. However they are surely on the horizon for academic and industrial researchers as network processors provide the right platform to achieve what they have envisioned long time ago. Our experimental results show the runtime constraints are properly fulfilled. The runtime overhead is low compared with the original code, where arbitrary context switches can cause unnecessary stalls and cycle wastage.

Although our current research focuses on a particular network processor, we believe the techniques that have been proposed in this thesis could potentially have bigger impact.

Solving dual-bank register allocation shares similarity with dual-bank memory access optimization studied in [81]. The design of new generations of the IXP network processor kept all the features targeted by this work. For IXP 2400/2800 series network processors, more threads are available on each processor, therefore the cost of applying dynamic context switch is relatively lower. Meanwhile, with multi-core design being widely adopted, sharing critical resources (like the register file) would continue to be an important topic for future researches. We believe the same architecture could be used for processors under extreme performance demands. As long as a processor's thread context switch mechanism is similar to IXP, the same technique can be applied to achieve better register sharing and to manage runtime constraints with compiler techniques.

8.2 Future Work

There are several other topics related to optimizations on the IXP network processors that might be interesting to explore in the future. It is clear that compiler optimization should target automatic translation from the language semantics to the machine code which could achieve better performance at runtime by exploiting application properties and hardware resources. We intend to look at the following problems in our future work.

8.2.1 Heterogeneous Memory

The IXP network processor provides heterogeneous memory components with different capacities and latencies. For example, the size of the scratchpad and local memory is only several KB, but can be accessed very fast. SRAM is slower than scratchpad but much larger as well. DRAM is very big (up to 2G on IXP 2800) but takes long access time (300 cycles). Besides, additional features exist for scratchpad and SRAM. For example, each bit of the scratchpad can be operated individually; SRAM memory cell can be locked/unlocked. It becomes an interesting topic to investigate how to map the program data to different parts of the memory. During such mapping, we need to consider the properties of the data as well as the properties of the memory components. In other words, for data we should consider how frequently they will be used and how much latency they can tolerate. It has been studied in compiler and architecture research that some of the memory accesses are critical, while others could allow a few cycles of slack [26, 56]. If such information can be distilled through compiler analyses, it will help allocate data to the memory components properly. Local memory is fast but precious, therefore should be used for the maximal profit. Possible candidates could be compiler controlled memory [19] for storing spill variables, extended register file [82] etc. In addition, since each processor core is quite simple, we could think about how to use compilers to help improve the performance of a single processor via the given memory hierarchy. In this manner, there will be no overhead being imposed on the hardware. One possibility would be software prefetching. Notice that no cache is currently available on packet processors, partly because there is little locality that can be exploited among independent packets. However as we put more code on the processor, there are other data which might have good locality, such as the state information if the packet flow is stateful.

8.2.2 Workload Assignment

A packet in the network processor typically undergoes several processors in a pipelined fashion. During each stage, a particular processor works on the packet and performs its task. The reason packet processing typically involves multiple processors is due to the limited code store size on each processor. Naturally, processors are assigned with different tasks. For example, some are for receiving, some are for sending, etc. However, currently it is always users' responsibility to assign tasks to multiple processors. Consequently, as we have observed, the task assignment by users normally leads to load imbalance on processors, since it is very difficult to estimate the computation needs of each task. We believe task assignment can be improved if more information is obtained for each task, either through profiling or compiler analyses. Secondly, a program cannot be arbitrarily split and assigned to different processors. The communication cost must be considered when we split the program and perform task assignments.

8.2.3 Data Communication Through Next Neighbor Registers

Next neighbor register is a new feature available on IXP 2400/2800. Each processor has 128 next neighbor registers and these registers can be used in two modes. When they are used in next neighbor mode, writing to these registers actually causes data to be moved to the register file of the next processor (numerically), i.e. the next neighboring processor can get these data by reading from their next neighbor registers. Alternatively, if next neighbor mode is turned off, these registers are used just as general purpose registers. We believe the main purpose of next neighbor register is to conveniently transfer data to the next processor. Since packet processing on network processors is typically in a pipelined fashion involving a sequence of processors, transferring data through the next neighbor registers is the fastest way. It would be interesting to study how next neighbor registers should be most efficiently used. The small number of next neighbor registers, i.e. 128 limits the amount of data transfer, therefore it is important to identify which data should be passed through the next neighbor registers and which data should be passed via other ways, such as the SRAM and DRAM. Moreover, next neighbor registers allow some of the registers to be shared between two processors. It is possible for the compiler to make use of these registers for other purposes.

8.2.4 Profile and Application Driven Optimizations

Up till now, all our compiler optimizations rely entirely on static compiler analyses. We have not attempted to improve the optimizer through profile and application specific information. Although static compiler analyses are more widely applicable, profile information could help a lot especially for network applications. Notice that programs running on the network processors are relatively stable, i.e. it is not easy to reload new applications at runtime and code typically run for a long time once installed, therefore in some sense, profile driven optimizations are more effective on network processors than on general purpose processors. Similarly, compiler optimizations designed for particular applications, i.e. application-specific optimizations can improve the performance of each application individually. For network processors, both profile and application driven optimizations deserve more research efforts to harness their advantages over static compiler analyses. Also, we need to investigate issues like: what kind of profile information should be collected, what kind of application specific properties should be looked into, how such knowledge (hint) is conveyed along with the code and what kind of runtime supports are necessary to utilize them.

8.2.5 Optimization Phases

Currently, our optimizations are mostly at the assembly level, i.e. we first translate benchmarks into assembly with one-to-one mapping to instructions, then the compiler optimizes the code. This requires the benchmarks to be rewritten or the generated code being recovered to a stage without register allocation. It is possible that our algorithm can do a source level transformation before the Intel Micro-C compiler. For example, the Micro-C compiler requires explicit specification of where each data unit resides (in SRAM, DRAM, etc.). By parsing a generic C program, our pass could attempt to automatically give such specifications. The source code is then transformed to the format the Micro-C compiler is able to compile.

REFERENCES

[1] "The great telecoms crash, the economist print edition," July 2002.

[2] Appel, A. W. and George, L., "Optimal spilling for cisc machines with few registers," in *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, (New York, NY, USA), pp. 243–253, ACM Press, 2001.

[3] A.V.Aho, R.Sethi, J., "Compilers principles, techniques and tools," *Addison-Wesley, Reading*, September 1986.

[4] Baboescu, F., Tullsen, D. M., Rosu, G., and Singh, S., "A tree based router search engine architecture with single port memories," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 123–133, IEEE Computer Society, 2005.

[5] Baer, J.-L., Low, D., Crowley, P., and Sidhwaney, N., "Memory hierarchy design for a multiprocessor look-up engine," in *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), p. 206, IEEE Computer Society, 2003.

[6] Barthelmann, V., "Inter-task register-allocation for static operating systems," in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, (New York, NY, USA), pp. 149–154, ACM Press, 2002.

[7] Bergner, P., Dahl, P., Engebretsen, D., and O'Keefe, M. T., "Spill code minimization via interference region spilling," in *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 287–295, 1997.

[8] Briggs, P., Cooper, K. D., and Torczon, L., "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, 1994.

[9] Brodie, B. C., Taylor, D. E., and Cytron, R. K., "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 191–202, IEEE Computer Society, 2006.

[10] Burke, J., McDonald, J., and Austin, T., "Architectural support for fast symmetric-key cryptography," in *International conference on Architectural support for programming languages and operating systems*, November 2000.

[11] Callahan, D. and Koblenz, B., "Register allocation via hierarchical graph coloring," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[12] Chaitin, G. J., “Register allocation & spilling via graph coloring,” in *SIGPLAN ’82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, (New York, NY, USA), pp. 98–101, ACM Press, 1982.

[13] Chaitin, G., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., and Markstein, P., “Register allocation via coloring,” pp. 47–57, January 1981.

[14] Chang, F., chang Feng, W., chi Feng, W., and Li, K., “Efficient packet classification with digest caches,” in *Proceedings of the HPCA-10 Workshop on Network Processors and Applications*, (Madrid, Spain), February 2004.

[15] Chang, Y.-S., Oh, H.-S., Yi, J.-H., Lee, J.-H., Lee, S.-W., Chun, J.-B., and Kyung, C.-M., “Gep2c02: A network processor for real-time content switching,” in *Proceedings of the HPCA-10 Workshop on Network Processors and Applications*, (Madrid, Spain), February 2004.

[16] Chen, M. K., Li, X.-F., Lian, R., Lin, J. H., Liu, L., Liu, T., and Ju, R., “Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[17] Chow, F. C. and Hennessy, J. L., “The priority-based coloring approach to register allocation,” October 1990.

[18] Collins, R., Alegre, F., Zhuang, X., and Pande, S., “Compiler assisted dynamic management of registers for network processors,” in *IPDPS ’06: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IEEE Computer Society, 2006.

[19] Cooper, K. D. and Harvey, T. J., “Compiler-controlled memory,” in *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 2–11, ACM Press, 1998.

[20] Corporation, S. R., “Network processors: A dose of reality, study number: Ac102-03,” October 2003.

[21] Crowley, P., “Supporting mixed real-time workloads in multithreaded processors with segmented instruction caches,” in *Proceedings of the HPCA-10 Workshop on Network Processors and Applications*, (Madrid, Spain), pp. 1–13, February 2004.

[22] Cruz, J. L., Gonzlez, A., Valero, M., and Topham, N. P., “Multiple-banked register file architectures,” in *ISCA ’00: Proceedings of the 27th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 316–325, ACM Press, 2000.

[23] Dai, J., Huang, B., Li, L., and Harrison, L., “Automatically partitioning packet processing applications for pipelined architectures,” in *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), ACM Press, 2005.

[24] Daveau, J.-M., Thery, T., Lepley, T., and Santana, M., “A retargetable register allocation framework for embedded processors,” in *LCTES ’04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, (New York, NY, USA), pp. 202–210, ACM Press, 2004.

[25] Demers, A., Keshav, S., and Shenker, S., “Analysis and simulation of a fair queueing algorithm,” in *ACM SIGCOMM*, September 1989.

[26] Fields, B., Bodk, R., and Hill, M., “Slack: Maximizing performance under technological constraints,” in *the 29th International Symposium on Computer Architecture*, 2002.

[27] Garcia, J., Corbal, J., Cerda, L., and Valero, M., “Design and implementation of high-performance memory systems for future packet buffers,” in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 373, IEEE Computer Society, 2003.

[28] George, L. and Appel, A. W., “Iterated register coalescing,” *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 300–324, 1996.

[29] George, L. and Blume, M., “Taming the ixp network processor,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[30] Grunwald, D. and Neves, R., “Whole-program optimization for time and space efficient threads,” in *Architectural Support for Programming Languages and Operating Systems*, pp. 50–59, 1996.

[31] Gupta, R., Soffa, M. L., and Ombres, D., “Efficient register allocation via coloring using clique separators,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 370–386, 1994.

[32] Hasan, J., Cadambi, S., Jakkula, V., and Chakradhar, S., “Chisel: A storage-efficient, collision-free hash-based network processing architecture,” in *ISCA ’06: Proceedings of the 33rd International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 203–215, IEEE Computer Society, 2006.

[33] Hicks, M. W., Kakkar, P., Moore, J. T., Gunter, C. A., and Nettles, S., “PLAN: A packet language for active networks,” in *International Conference on Functional Programming*, pp. 86–93, 1998.

[34] Intel Corporation, *Intel IXP1200 Processor Family-Hardware Reference Manual*, December 2001.

[35] Intel Corporation, *Intel IXP1200 Processor Family-Software Reference Manual*, August 2004.

[36] Intel Corporation, *Intel IXP2800 Processor Family-Hardware Reference Manual*, August 2004.

[37] Intel Corporation, *Intel IXP2800 Processor Family-Software Reference Manual*, August 2004.

[38] J.A.Bondy and S.C.Locke, "Largest bipartite subgraphs in triangle-free graphs with maximum degree three," *Journal of Graph Theory*, vol. 10, pp. 477–504, 1986.

[39] Jang, S., Carr, S., Sweany, P., and D.Kuras, "A code generation framework for vliw architectures with partitioned register files," in *the Third International Conference on Massively Parallel Computing Systems*, April 1998.

[40] Kaxiras, S. and Keramidas, G., "Ipstash: a power-efficient memory architecture for ip-lookup," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 361, IEEE Computer Society, 2003.

[41] Khosravi, H., Yavatkar, R., Bakshi, S., Deval, M., Muralidhar, R., and Ahmed, S., "Using network processors to improve scalability and availability of routing/signaling protocols," in *Proceedings of the HPCA-10 Workshop on Network Processors and Applications*, (Madrid, Spain), February 2004.

[42] Kim, H., *Region-based register allocation for epic architectures.* PhD thesis, 2000. Adviser-Krishna Palem.

[43] Kim, J., S.Jung, and Y.Park, "Experience with a retargetable compiler for a commercial network processor," in *In Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, October 2002.

[44] Kim, J., Dally, W. J., Towles, B., and Gupta, A. K., "Microarchitecture of a high-radix router," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 420–431, IEEE Computer Society, 2005.

[45] Lepak, K. M. and Lipasti, M. H., "Silent stores for free," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 22–31, 2000.

[46] Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, vol. 20, pp. 40–61, 1973.

[47] Liu, J., Kong, T., and Chow, F., "Effective compilation support for variable instruction set architecture," in *In Proceedings of International Conf. on Parallel Architectures and Compilation Techniques*, September 2002.

[48] Lueh, G.-Y., Gross, T., and Adl-Tabatabai, A.-R., "Global register allocation based on graph fusion," in *Languages and Compilers for Parallel Computing*, pp. 246–265, 1996.

[49] Lueh, G.-Y., Gross, T., and Adl-Tabatabai, A.-R., "Fusion-based register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 3, pp. 431–470, 2000.

[50] Luo, Y., Yang, J., Bhuyan, L. N., and Zhao, L., "Nepsim: A network processor simulator with a power evaluation framework," *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications*, vol. 24, no. 5, pp. 34–444, 2004.

[51] Memik, G., Mangione-Smith, W., and Hu., W., "Netbench: A benchmarking suite for network processors," in *ICCAD*, November 2001.

[52] Miller, M., Cech, D., and Darnell, S., "Essential network search engine features for high speed npu-based packet searching," in *Proceedings of the HPCA-10 Workshop on Network Processors and Applications*, (Madrid, Spain), February 2004.

[53] Minkenberg, C., Luijten, R., Denzel, W., and Gusat, M., "Current issues in packet switch design," in *HotNets-I*, (Princeton, NJ), 2002.

[54] Monreal, T., Gonzlez, A., Valero, M., Gonzlez, J., and Vinals, V., "Delaying physical register allocation through virtual-physical registers," in *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 186–192, IEEE Computer Society, 1999.

[55] Muchnick, S. S., *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997.

[56] Muthler, G. A., Crowe, D., Patel, S. J., and Lumetta, S. S., "Instruction fetch deferral using static slack," in *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 51–61, IEEE Computer Society Press, 2002.

[57] Papadimitriou, C. H. and Yannakakis, M., "Optimization, approximation and complexity classes," *Journal of Computer and System Sciences*, vol. 43, pp. 425–440, 1991.

[58] Park, J., Lee, J., and Moon, S., "Register allocation for banked register file," in *LCTES*, June 2001.

[59] Poljak, S. and Tuza, Z., "Bipartite subgraphs of triangle-free graphs," *SIAM J. Discrete Math*, vol. 7, pp. 307–313, 1994.

[60] Sauer, C., Gries, M., Gomez, J. I., and Keutzer, K., "Towards a flexible network processor interface for rapidio, hypertransport, and pci-express," in *Proceedings of the HPCA-10 Workshop on Network Processors and Applications*, (Madrid, Spain), February 2004.

[61] Sherwood, T., Varghese, G., and Calder, B., "A pipelined memory architecture for high throughput network processors," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 288–299, ACM Press, 2003.

[62] Shi, W., Zhuang, X., Paul, I., and Schwan, K., "Efficient implementation of packet scheduling algorithm on high-speed programmable network processors," in *MMNS*, 2002.

[63] Spalink, T., Karlin, S., Peterson, L., and Gottlieb, Y., "Building a robust software-based router using network processors," in *PLDI '05: Proceedings of ACM Symposium on Operating Systems Principles*, 2001.

[64] T.H.Cormen, C.E.Leiserson, and R.L.Rivest *MIT Press*, 1989.

[65] Tomasulo, R. M., "An efficient algorithm for exploiting multiple arithmetic units," pp. 13–21, 1995.

[66] Wagner, J. and Leupers, R., "C compiler design for an industrial network processor," in *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2001.

[67] Welfeld, F. J., "Network processing in content inspection applications," in *International Symposium on Systems Synthesis*, September 2001.

[68] West, R. and C.Poellabauer, "Analysis of a window-constrained scheduler for real-time and best-effort packet streams," in *RTSS*, 2000.

[69] West, R. and Schwan, K., "Dynamic window-constrained scheduling for multimedia applications," in *ICMCS, Vol. 2*, pp. 87–91, 1999.

[70] West, R., Schwan, K., and Poellabauer, C., "Scalable scheduling support for loss and delay constrained media streams," in *IEEE Real Time Technology and Applications Symposium*, 1999.

[71] Wheeler, B. and Gwennap, L., "A guide to network processors, fifth edition," October 2003.

[72] Wolf, T. and Franklin, M., "Commbench - a telecommunication benchmark for network processors," in *ISPASS*, 2000.

[73] Wun, B., Buhler, J., and Crowley, P., "Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, (Washington, DC, USA), pp. 173–184, IEEE Computer Society, 2005.

[74] Yan Luo, Jun Yang, L. B. and Zhao, L., "Nepsim: A network processor simulator with power evaluation framework," *IEEE MICRO, special issue on network processors for future high-end systems and applications*, September 2004.

[75] Zhuang, X. and Liu, J., "Wraps scheduling and its efficient implementation on network processors," in *HiPC*, 2002.

[76] Zhuang, X. and Pande, S., "Resolving register bank conflicts for a network processor," in *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), p. 269, IEEE Computer Society, 2003.

[77] Zhuang, X. and Pande, S., "Balancing register allocation across threads for a multithreaded network processor," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, (New York, NY, USA), pp. 289–300, ACM Press, 2004.

[78] Zhuang, X. and Pande, S., "Differential register allocation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), pp. 168–179, ACM Press, 2005.

[79] Zhuang, X. and Pande, S., "Effective thread management on network processors with compiler analysis," in *ACM SIGPLAN Conference on Languages, Compiler, and Tools for Embedded Systems*, 2006.

[80] Zhuang, X. and Pande, S., "A scalable priority queue architecture for high speed network processing," in *Proceedings of 25th IEEE INFOCOM*, April 2006.

[81] Zhuang, X., Pande, S., and Jr., J. S. G., "A framework for parallelizing load/stores on embedded processors," in *PACT '02: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, 2002.

[82] Zhuang, X., Zhang, T., and Pande, S., "Hardware-managed register allocation for embedded processors," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, (New York, NY, USA), pp. 192–201, ACM Press, 2004.

INDEX

VITA

Xiaotong Zhuang is a doctoral student in the Computer Science department of Georgia Institute of Technology. Xiaotong also holds a BE degree in EE from Shanghai Jiaotong University, and two MS degrees in CS from Shanghai Jiaotong University and Georgia Institute of Technology respectively. His main areas of interest are compiler optimizations, computer architecture, security and parallel and distributed system. During his PhD study, he has conducted research in many emerging areas like embedded systems (including network processors), secure processor etc. Xiaotong is a recipient of the Outstanding Graduate Research Assistant Award in College of Computing for the academic year 2005-06.