Efficient Handling of Narrow Width and Streaming Data in Embedded Applications

by

Bengu Li

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2006

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Bengu Li

entitled Efficient Handling of Narrow Width and Streaming Data in Embedded Applications

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

______________________________ Date: May 25th, 2006
Rajiv Gupta

______________________________ Date: May 25th, 2006
Peter Downey

______________________________ Date: May 25th, 2006
Bongki Moon

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

______________________________ Date: May 25th, 2006
Dissertation Director: Rajiv Gupta

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: Bengu Li

Acknowledgements

My special thanks go to my adviser, Prof. Rajiv Gupta. His motivating encouragement was important when I started on this work. During the years in pursuing my degree, his guidance helped me catch the ideas and develop the solutions. He also taught me how to express an idea, especially in writing. I learnt a lot from him. I thank him for his continuous support. Without his guidance, this dissertation would never be completed.

I am also grateful for other committee members, Prof. Peter Downey and Prof. Bongki Moon, for carefully reading a draft of the dissertation and pointing out many mistakes and suggesting many improvements.

I also thank many office-mates, with whom I had great discussions on ideas and technical problems. They are Youtao Zhang, Jun Yang, Wen-Ke Chen, Xiangyu Zhang, Arvind Krishnaswamy, Sriraman Tallam, Vijayanand Nagarajan and Mohan Rajagopalan.

Finally I want to especially thank my wife Xialing for her unconditional support during these years.

Table of Contents

List of Figures **8**

List of Tables **10**

Abstract **11**

Chapter 1. Introduction **12**
- 1.1. Narrow Width Data 13
- 1.2. Streaming Data 15
- 1.3. Overview of the Dissertation 18
- 1.4. Organization 19

Chapter 2. A Study of Narrow Width Data **21**
- 2.1. Narrow Width Data in Embedded Applications 21
- 2.2. Maximum Observed Bitwidth 22
- 2.3. Dynamic Variations in Bitwidth 25
- 2.4. Related work 29
- 2.5. Summary 30

Chapter 3. An Instruction Set Extension for Narrow Width Data **32**
- 3.1. Bit Section eXtensions (BSX) 33
  - 3.1.1. Bit Section Descriptors 33
  - 3.1.2. Bit Section Addressing Modes 34
  - 3.1.3. Bit Section Instructions and their Encoding 36
  - 3.1.4. BSX Implementation 40
- 3.2. Generating BSX ARM Code 41
  - 3.2.1. Fixed Unpacking 41
  - 3.2.2. Fixed Packing 43
  - 3.2.3. Dynamic Unpacking 44
  - 3.2.4. Dynamic Packing 46
- 3.3. Experimental Evaluation 47
  - 3.3.1. Experimental Setup 47
  - 3.3.2. Results 48
- 3.4. Related Work 49
- 3.5. Summary 53

Table of Contents—*Continued*

Chapter 4. Register Allocation Exploiting Narrow Width Data **54**
4.1. The Challenges of Bitwidth Aware Register Allocation 55
4.2. Register Allocation in the GCC Compiler 59
4.3. Static Subword Register Allocation 61
4.4. Architectural Support for Speculative Subword Register Allocation . 65
4.4.1. Register File Enhancement 65
4.4.2. New Instructions 66
4.4.3. Hardware Implementation 68
4.5. Compiler Algorithm for Speculative Subword Register Allocation . . . 69
4.5.1. The Framework 69
4.5.2. Priority-Based Speculative Allocation 70
4.5.3. Profiling Pass 73
4.5.4. Speculative Reload Pass 74
4.6. Experimental Results 75
4.6.1. Benchmarks 76
4.6.2. Avoided Spill Cost 76
4.6.3. Performance 79
4.7. Related Work 81
4.8. Summary 82

Chapter 5. An Offset Assignment Method Exploiting Narrow Width Data **83**
5.1. SubWord Offset Assignment 85
5.2. Algorithms 93
5.2.1. Alg. I - Cover First, Coalesce Later 94
5.2.2. Alg. II - Coalesce First, Cover Later 98
5.2.3. Alg. III - Integrated Covering and Coalescing 102
5.2.4. Further Extensions 106
5.3. Experimental Results 108
5.4. Related Work 113
5.5. Summary 114

Chapter 6. A Study of Streaming Data **115**
6.1. Streaming Data vs. Utility Data 115
6.2. Measurement of Streaming Data 117
6.3. Summary 122

Chapter 7. Energy-efficient Caches for Network Processors . **123**
7.1. Selective Cache 125
7.2. Computation Reuse Cache 128

Table of Contents—*Continued*

7.3. Using Fetch Gating 130
7.4. Experimental Evaluation 133
7.4.1. Cache Behavior 134
7.4.2. Fetch Gating and Energy Savings 135
7.5. Related Work 137
7.5.1. Locality and Caching in Network Processors 137
7.5.2. Instruction Reuse 138
7.5.3. Clock and Fetch Gating 139
7.6. Summary 141

Chapter 8. Conclusions **143**
8.1. Contributions 143
8.2. Future Directions 146

References **148**

List of Figures

Figure 2.1. Cumulative Percentage of Register Operands with Maximum Bitwidth in Bits for Benchmark adpcm decode 22
Figure 2.2. Percentage of Register Operands with Maximum Bitwidth of 1, 2, 3, and 4 Bytes 23
Figure 2.3. Average Maximum Bitwidth of Register Operands 24
Figure 2.4. Percentage of Register Operands with Maximum Bitwidth Less than 16 24
Figure 2.5. Cumulative Number of Register Operand Instances with Dynamic Bitwidth in Bits for a Selected Register Operand in Benchmark adpcm decode 25
Figure 2.6. Percentage of Register Operand Instances with Dynamic Bitwidth within 25%, 50%, 75%, and 100% of Maximum Bitwidth 26
Figure 2.7. Average Dynamic Bitwidth of Register Operand Instances . . . 27
Figure 2.8. Percentage of Register Operand Instances with Dynamic Bitwidth Less than 16 27
Figure 2.9. Dynamic Bitwidth Averaged over Register Operands 28
Figure 2.10. Percentage of Register Operands with Average Dynamic Bitwidth Less than 16 29

Figure 3.1. Register Bit Section Descriptors 36
Figure 3.2. Dynamic Instructions Count of Selected Instructions for BSX as a Percentage of Total 37
Figure 3.3. BSX Instruction with Immediate BSD (First Variation) 37
Figure 3.4. BSX Instruction with Immediate BSD (Second Variation) . . . 38
Figure 3.5. BSX Instruction with Immediate BSD (Third Variation) 38
Figure 3.6. BSX Instruction with Register BSD 39

Figure 4.1. Bitwidth Aware Register Allocation. 56
Figure 4.2. An Example of Static Subword Register Allocation 62
Figure 4.3. The Framework for Static Subword Register Allocation. 62
Figure 4.4. The Compiler Algorithm of Static Subword Register Allocation 63
Figure 4.5. Interference Graphs in Static Subword Register Allocation . . . 64
Figure 4.6. Accessing Register R as a Source Operand 65
Figure 4.7. An Example of Speculative Subword Register Allocation 67
Figure 4.8. Hardware Implementation 68
Figure 4.9. The Framework of Speculative Subword Register Allocation. . . 69
Figure 4.10. Interference Graphs in Speculative Subword Register Allocation 71
Figure 4.11. Avoidance of Spill Cost (Cache, Speculative) 78
Figure 4.12. Performance Improvement in Cycles (Non-cache) 80

List of Figures—*Continued*

Figure 5.1. Simple Offset Assignment (SOA) 87
Figure 5.2. SubWord Offset Assignment (SWOA) 88
Figure 5.3. Positive Coalescence 91
Figure 5.4. Negative Coalescence 91
Figure 5.5. An Example of Alg. I 96
Figure 5.6. Alg. I - Cover First, Coalesce Later. 97
Figure 5.7. Motivating Example of Alg. II 99
Figure 5.8. Local subgraph: Path Cover before and after Coalescing 100
Figure 5.9. Alg. II - Coalesce First, Cover Later. 101
Figure 5.10. Right Amount of Coalescing 103
Figure 5.11. Integrated Coalescing 105
Figure 5.12. Alg. III - Integrated Covering and Coalescing 107

Figure 6.1. Streaming Data vs. Utility Data 118
Figure 6.2. Short Lifetime 118
Figure 6.3. Frequently Observed Values 121

Figure 7.1. Selective Cache 126
Figure 7.2. Timeline of Selective Cache 127
Figure 7.3. Timeline of Computation Reuse Cache 131
Figure 7.4. Network Router with Fetch Gating (F-G) logic 132

List of Tables

Table 1.1. Data Decomposition in Embedded Applications 14
Table 1.2. Streaming Nature vs. Utility Data in Embedded Applications . . 17

Table 3.1. Reduction in Dynamic Instruction Counts 50
Table 3.2. Reduction in Dynamic Cycle Counts 51
Table 3.3. Reduction in Code Size 52

Table 4.1. Benchmark Characteristics. 77
Table 4.2. Avoidance of Spill Cost (Non-cache) 77
Table 4.3. Performance Improvement in Cycles (Speculative, Cache) 80

Table 5.1. Reduction in Static Code Size 109
Table 5.2. Reduction in Stack Frame Size 111
Table 5.3. Reduction in Dynamic Instruction Count 112

Table 6.1. Application Properties 116
Table 6.2. Packet Stream Characteristics 119
Table 6.3. Average Unique Reuse Distance. 121

Table 7.1. Application Properties 134
Table 7.2. Selective Cache Hit Rate 135
Table 7.3. Computation Reuse Cache Hit Rate 135
Table 7.4. Program Behavior with Selective Cache 136
Table 7.5. Program Behavior with Computation Reuse Cache 136

Abstract

Embedded environment imposes severe constraints of system resources on embedded applications. Performance, memory footprint, and power consumption are critical factors for embedded applications. Meanwhile, the data in embedded applications demonstrate unique properties. More specifically, *narrow width data* are data representable in considerably fewer bits than in one word, which nevertheless occupy an entire register or memory word and *streaming data* are the input data processed by an application sequentially, which stay in the system for a short duration and thus exhibit little data locality. Narrow width and streaming data affect the efficiency of register, cache, and memory and must be taken into account when optimizing for performance, memory footprint, and power consumption.

This dissertation proposes methods to efficiently handle narrow width and streaming data in embedded applications. Quantitative measurements of narrow width and streaming data are performed to provide guidance for optimizations. Novel architectural features and associated compiler algorithms are developed. To efficiently handle narrow width data in registers, two register allocation schemes are proposed for the ARM processor to allocate two narrow width variables to one register. A static scheme exploits maximum bitwidth. A speculative scheme further exploits dynamic bitwidth. Both result in reduced spill cost and performance improvement. To efficiently handle narrow width data in memory, a memory layout method is proposed to coalesce multiple narrow width data in one memory location in a DSP processor, leading to fewer explicit address calculations. This method improves performance and shrinks memory footprint. To efficiently handle streaming data in network processor, two cache mechanisms are proposed to enable the reuse of data and computation. The slack created is further transformed into reduction in energy consumption through a fetch gating mechanism.

Chapter 1
Introduction

An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls [1]. Embedded systems have been widely deployed in recent years. They affect many aspects of human life (e.g., from personal media players and cellphones to mobile computing devices and network routers, etc.). As essential components of an embedded system, the embedded processor and the embedded application face severe constraints of system resources imposed by the embedded environment, such as time (e.g., soft/hard real-time requirements), hardware (e.g., small memory size, precious CPU bandwidth, no external storage), and power supply (e.g., battery, power dissipation). As a result, performance, memory footprint, and power consumption become critical factors in designing embedded processors and embedded applications.

Embedded workloads exhibit behavior significantly different from that of general purpose workloads. In particular, the properties of the data processed by embedded applications are significantly different. Typical embedded applications routinely process streams of formatted data. Studies of the data reveal the presence of *narrow width data* and *streaming data* in these applications. Narrow width data are data representable in considerably fewer bits than in one word, which nevertheless occupy an entire register or memory word. Streaming data are the input data processed by an application sequentially, which stay in the system for a short duration and thus exhibit little data locality. These properties affect the efficiency with which the components of a processor (e.g., register, cache, and memory) function. Therefore narrow width and streaming data must be taken into account when optimizing for performance, memory footprint, and power consumption for embedded systems. This

dissertation proposes methods to efficiently handle narrow width data and streaming data in embedded applications.

1.1 Narrow Width Data

Embedded applications often work on narrow width data. *Narrow width data* are data representable in considerably fewer bits than in one word, which nevertheless occupy an entire register or memory word. Narrow width data lead to inefficient utilization of system resources.

Narrow width data are common in embedded applications. This is primarily because the input/output data processed by embedded applications contain a large amount of narrow width data. For example, in network applications there are many narrow width fields in packet headers and in media applications there are many small data values in data frames. Narrow width data also come from variables or data structures used by both embedded applications and general purpose applications, such as character and boolean variables.

Narrow width data lead to inefficient utilization of system resources. First, they lead to severe underutilization of registers. When narrow width data are present in registers, many high bits in the registers are actually not needed in the computation. However, with current architecture design, a narrow width data item occupies a whole register word when the instructions are executed. Second, narrow width data result in bloated memory footprint when they reside in memory. Since memory is a precious system resource, using an entire memory word to store one narrow width data item results in an inefficient use of memory. Third, narrow width data also cause performance degradation when unpacking operations are needed to extract narrow width data from the input and additional operations are required to pack the narrow width data into the output.

Table 1.1 shows application behaviors related to narrow width data for selected

Table 1.1. Data Decomposition in Embedded Applications

Benchmark	Description
`adpcm codec`	A stream of 16-bit PCM codes is converted to 4-bit ADPCM codes and vice versa. Inside the algorithm, codes are even further decomposed into sign bit, magnitude, difference, etc.
`jpeg codec`	A stream of 24-bit RGB format of pixels is decomposed into three colors of 8-bit each. These colors are used to compute the 8-bit brightness, two types of 8-bit chrominance coefficients. Images are divided into 8x8 MCUs. Discrete Cosine Transform converts the MCUs into a matrix of 12-bit per element. After quantization and trophy encoding, the data is output in a format of header with many narrow width fields and compressed data.
`mpeg2 codec`	A stream of video frame data in a hierarchy organization consisting of video sequence, GOP, picture, slice, macro-block, block and pixel. Each level has a header with many narrow bit fields. The pixels are decomposed into 8-bit magnitude and two types of 8-bit chrominance coefficients. After inter-frame and intra-frame compression, the result is generated still in the format of header with many narrow width fields and compressed data.
`ip lookup`	A stream of IP packets with packet header and payload is decomposed into many narrow width field. The 32-bit IP destination field is further decomposed into 4-bit or 8-bit segments to lookup in the routing table.
`nat lookup`	A stream of IP packets with packet header and payload containing TCP packet is decomposed into many narrow width field. The destination IP address and TCP port are used to look up in the port mapping table for incoming traffic. The source IP address and TCP port are used to look up in the port mapping table for outgoing traffic. These fields are further decomposed into 4-bit or 8-bit segments during lookup.
`packet classification`	A stream of IP packets with packet header and payload containing TCP packet is decomposed into many narrow width field. The source and destination IP address and TCP ports and protocol field are used to classify the packets into flows based on classification table. The address and port fields are further divided into 4-bit or 8-bit segments during processing. Protocol field itself is 4-bit.

benchmarks taken from embedded benchmark suites of Mediabench [28], Commbench [50] and Intel IXP Workbench [2]. Mediabench benchmark suite consists of a set of applications which focus on media processing. Commbench benchmark suite consists of a set of applications which focus on networking and communication processing. Intel IXP Workbench is a set of network processing benchmarks on Intel IXP network processor.

In these applications, the input data units are often decomposed into many smaller narrow width data items before processing. These narrow width data may exist during the whole processing cycle. After processing, the results, which are often also narrow width data, may need to be packed back into the output data units.

In presence of narrow width data, it is a challenge to find solutions that improve the efficiency of register and memory usage. Performance that is degraded because of narrow width data is also an important issue.

1.2 Streaming Data

Embedded applications are often streaming applications which contain a main *processing loop*, in which one iteration of the loop processes one unit of data. For example, in program `adpcm codec` from `Mediabench`, one loop iteration processes one unit of Pulse Code Modulation (PCM) code. This type of application often works on streaming data. *Streaming data* are the input data processed by an application sequentially, which stay in the system for a short duration and thus exhibit little data locality.

There are two reasons for this phenomenon. Loops in embedded applications process consecutive data items in consecutive iterations. The amount of processing within one loop iteration is limited and hence little locality is observed due to lack of reuse of data. The locality across loop iterations is also often absent because different loop iterations process different data items.

Streaming data leads to degradation of overall data locality in an application. In a typical streaming application, the memory accesses to data can be classified into two categories. One is *streaming data* which are processed sequentially, such as network packets, audio and video data frames. The other is *utility data* which come from application-specific data structures required for processing streaming data, such as routing tables, etc. Within one loop iteration, utility data also show little data locality because of a limited amount of processing and reuse. Across consecutive loop iterations, utility data may demonstrate high locality since the same part of the data can be accessed repetitively (e.g., same routing table entries may be referenced separately in IP Lookup application). When the memory accesses to streaming data and utility data are mixed, overall data locality is reduced. Although within one loop iteration the number of accesses to each type of data can be small, nevertheless across consecutive loop iterations they both occupy a significant percentage of total number of memory accesses. The same fields in different streaming data units often contain the same values and thus exhibit *value locality*. This leads to redundant memory accesses to utility data and also redundant computation sequences.

Table 1.2 shows in more detail, for selected streaming applications, how streaming data and utility data are accessed. In these applications, each loop iteration processes one unit of input data. Within each loop iteration, input data are read before processing, and output data are written after processing. Different memory references to streaming data usually access different portions of the data and thus show little temporal locality within one loop iteration. At the same time, utility data are also accessed during processing.

Streaming data pose a big challenge to the efficiency of the data cache. With degraded data locality, the data cache cannot be used efficiently. Streaming data pollute the data cache. Intel IXP 1200 network processor does not include a cache in its microengine core, because with low data locality, cache cannot be efficiently used to improve the performance or the throughput [9].

TABLE 1.2. Streaming Nature vs. Utility Data in Embedded Applications

Benchmark	Description
adpcm codec	One loop iteration processes one unit of PCM code. Within each loop iteration, there are two memory references to streaming data, including one read from the input buffer and one write to the output buffer. Within each loop iteration, there are two memory references to utility data, including one read from index table and one read from step size table.
jpeg codec	One loop iteration processes one one full MCU row. Within each loop iteration, a significant part of memory references are used to read in the image data within the MCU row from input buffer and write the compressed data to output buffer. These reads and writes access different part within one row. Another significant part of memory references are used to access utility data used in DCT transformation, quantization and trophy encoding such as the DC entropy table.
mpeg2 codec	One loop iteration processes one video frame. Within each loop iteration, a significant part of memory accesses are used to read in the data in video frame from input buffer and write compressed data to output buffer. These reads and writes access different part of within the frame. Another significant part of memory references are used to access utility data used in compression such as the DCT table.
ip lookup	One loop iteration finds the nexthop information for one IP packet. Within each loop iteration, there is one SDRAM read of destination field in packet header. Within each loop iteration, there may be up to five SRAM reads of routing table.
nat lookup	One loop iteration translates the source IP address and TCP port for one IP packet. Within one loop iteration, there are two SDRAM reads of address and port fields in packet header. Within each loop iteration, there may be up to six SRAM reads of port-mapping table.
packet classification	One loop iteration identifies the flow ID for one IP packet. Within each loop iteration, there are three SDRAM reads of address, port and protocol fields in the packet header. Within each loop iteration, there may be up to thirteen SRAM reads of classification table.

1.3 Overview of the Dissertation

This dissertation tackles the problems caused by narrow width data and streaming data in embedded applications. New architectural features are introduced and associated compiler algorithms are developed to address the problems. These techniques lead to optimizations in critical factors in embedded applications such as performance, memory footprint, and power consumption.

Narrow width data are manipulated efficiently by packing multiple narrow width data items into one register or memory word. Using register allocation and memory layout methods, the problems of underutilization of registers and the underutilization of memory and performance degradation are addressed. Streaming data are handled efficiently by providing cache mechanisms which enable reuse of utility data or computation sequence. This in turn is transformed into the reduction of energy consumption. This dissertation makes the following contributions:

- *Measurement of Narrow Width and Streaming Data in Embedded Applications*
  Quantitative measurements of the presence and properties of narrow width and streaming data are performed for typical embedded applications from benchmark suites of Mediabench and Commbench. The measurements provide guidance in developing mechanisms to efficiently handle narrow width and streaming data.

- *An Instruction Set Extension for Narrow Width Data*
  An instruction set extension for the ARM processor is proposed to manipulate narrow width data directly. The cost of packing and unpacking operations can be reduced using this instruction set extension.

- *Efficient Handling of Narrow Width Data in Registers*
  Two register allocation schemes are proposed to efficiently handle narrow width data in registers in an ARM processor. The algorithms allocate two narrow

width variables to one register. A static scheme determines an assignment of narrow width variables based on maximum bitwidth information. To exploit more optimization opportunities, a speculative register allocation scheme allocates into one register two narrow width variables that can be packed together most of the time. Performance improvement is achieved through avoidance of register spills because of the reduction in register pressure.

- *Efficient Handling of Narrow Width Data in Memory*

  A memory layout scheme is proposed to coalesce multiple narrow width data in one memory location in a DSP processor. Memory footprint can therefore be shrunk. Furthermore, performance is improved. With this coalescing, explicit address calculation operations are reduced, by using a DSP specific addressing mode of *address register autoincrement/autodecrement.*

- *Energy-efficient Cache for Network Processor Exploiting Streaming Data*

  Two energy-efficient cache mechanisms are proposed to reduce energy-consumption on the Intel IXP1200 network processor by exploiting streaming data. The caches are designed to exploit the value locality of streaming data to reduce the redundancy in computation or memory accesses to utility data. This ameliorates traditional cache problem that is not effective for a network processor due to a balanced design and low temporal locality in streaming data. The caches create slack in the processing schedule, which is transformed into reduction in energy-consumption through a fetch gating mechanism.

1.4 Organization

The rest of the dissertation is organized as follows. In Chapter 2, a study of narrow width data in embedded applications is performed. In Chapter 3, an instruction set extension of the ARM processor, which directly manipulates narrow width data,

is introduced. In Chapter 4, register allocation schemes for narrow width data in ARM processors are proposed. In Chapter 5, a memory layout scheme of narrow width data for a DSP processor is presented. In Chapter 6, a study of streaming data in network processing applications is performed. In Chapter 7, energy-efficient cache mechanisms exploiting streaming data in a network processor are proposed. In Chapter 8, a summary of the contributions of this work are provided, together with discussions of future work.

Chapter 2
A Study of Narrow Width Data

In this chapter, a study of narrow width data in embedded applications is presented. Section 2.1 discusses the causes, formats, and consequences of narrow width data in embedded applications. Statistics are collected to demonstrate the prevalence of narrow width data in embedded applications and presented in Sections 2.2 and 2.3. In Section 2.4, related work is discussed. In Section 2.5, observations based upon this study are made that will guide the development of architectural and compiler optimizations presented in subsequent chapters.

2.1 Narrow Width Data in Embedded Applications

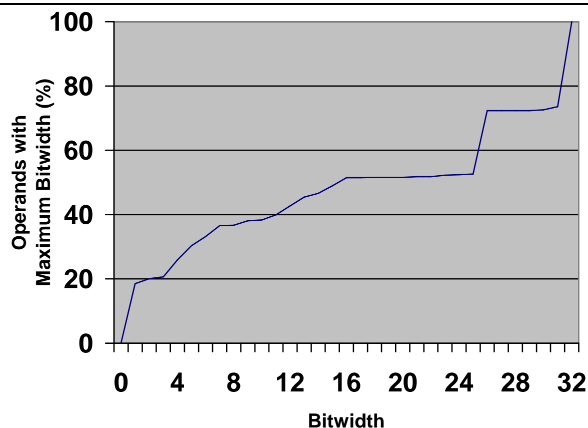
Narrow width data exist in a wide range of applications. Brooks et al. [10] observe that even in general purpose applications such as SPEC95, narrow width data widely exist. This is not surprising since many of these applications have remained the same while the general purpose systems have evolved from 16-bit to 32-bit to 64-bit. For example, no matter what the word size is, a boolean variable only needs one bit to represent. In embedded applications, the above situation is even more commonplace. As mentioned in the previous chapter, applications like media or network processing have the feature of working on large amount of narrow width data. The presence of these narrow width data together with narrow width data found in all applications comprise the source of narrow width data in embedded applications. Narrow width data are observed at different points across the memory hierarchy of an embedded system. It exists in registers which causes the inefficient usage of registers. It exists in memory which leads to a larger memory footprint than necessary. It exists in the data cache which results in poor utilization of cache. If the input/output data are in

packed form, extra operations are required to unpack/pack them. All these factors have a negative influence on the critical optimization goals in an embedded system —high performance, small memory footprint, and low power consumption.

2.2 Maximum Observed Bitwidth

To quantify the prevalence of narrow width data in embedded systems, an experimental study of typical media processing and network processing applications was performed. In the previous chapter, the existence of narrow width data has been analyzed. Here a profile-based study shows in more detail the maximum effective bitwidth of register operands in embedded applications. Maximum effective bitwidth of a register operand refers to the maximum value of effective bitwidth of any instance of the register operand at runtime. For convenience, *maximum bitwidth* is used to refer to the above. Maximum bitwidth is an approximation of static bitwidth. Typical media and networking applications are taken from Mediabench or Commbench in this study.

Figure 2.1 Cumulative Percentage of Register Operands with Maximum Bitwidth in Bits for Benchmark `adpcm decode`



The cumulative percentage of register operands with maximum bitwidth measured in bits for benchmark `adpcm decode` is shown in Figure 2.1. This benchmark has

nearly 20% of the register operands as one bit. This is because during processing many sign bits are generated and many boolean variables are used. As the bitwidth grows to 16 bits, the percentage increases to about 50%. In other words, more than half of bits in the registers are never used for almost half of the register operands. This fact tells us that the degree of narrow width data is significant. The percentage grows slightly when the bitwidth is in the range of 16 bits to 24 bits. It is followed by a big jump of about 20% at 26 bits. This is due to heap and stack references. Finally there are more than 20% of register operands using the whole register. The following figures further show that for all the benchmarks studied, the degree of narrow width data is significant.

Figure 2.2 shows the percentage of register operands with maximum bitwidth of 1, 2, 3, and 4 bytes for a set of benchmarks. All benchmarks have 30% to 40% of the register operands with maximum bitwidth of less than one byte and about 50% with maximum bitwidth of less than two bytes. It is interesting to note that for all benchmarks the percentage grows very little for maximum bitwidth of less than three bytes, but there is a big increase for maximum bitwidth of less than four bytes.

Figure 2.2 Percentage of Register Operands with Maximum Bitwidth of 1, 2, 3, and 4 Bytes

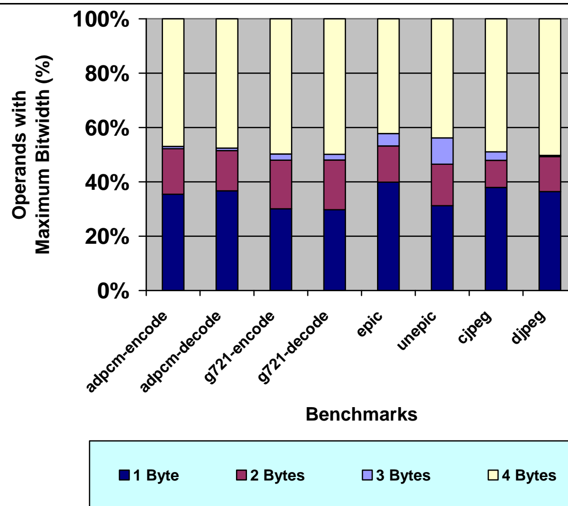


Figure 2.3 shows the average maximum bitwidth of all register operands for a set of benchmarks. For all benchmarks, the average maximum bitwidth falls in the range of 16 bits to 18 bits. Figure 2.4 emphasizes the percentage of register operands whose maximum bitwidth is less than 16 bits. For all benchmarks, this percentage is about 50%. This significant fact implies that there exists a great opportunity to pack two data with maximum bitwidth of less than 16 bits into one 32-bit register.

Figure 2.3 Average Maximum Bitwidth of Register Operands

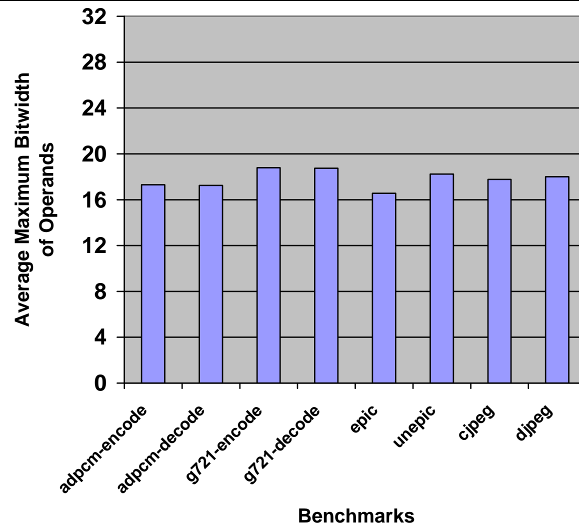
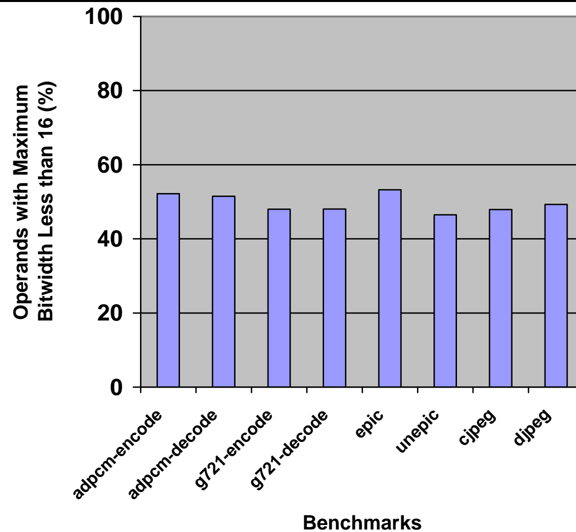


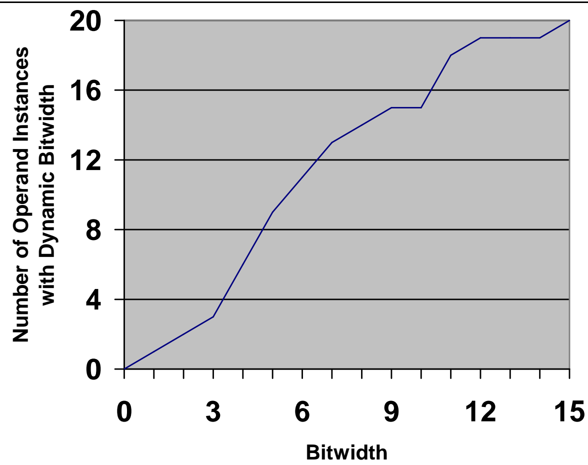
Figure 2.4 Percentage of Register Operands with Maximum Bitwidth Less than 16



2.3 Dynamic Variations in Bitwidth

Even if the required maximum bitwidth is allocated to the variables, there still exists a significant amount of underutilization of the bits at runtime. It is possible that the following case arises. The maximum bitwidth is used only once. During most of the runtime only a portion of the bitwidth is used. There are several reasons leading to this underutilization. The actual operand bitwidth may be very dependent on the input data which is unknown at compile time. Even with fixed input data, the bitwidth may vary across different loop iterations of the program execution. This dynamic information cannot be revealed by a static bitwidth study. To show what really happened to the operand bitwidth during runtime, a study of the *dynamic bitwidth* of register operands is performed. Dynamic bitwidth refers to the value of the bitwidth of one instance of the register operand at runtime.

Figure 2.5 Cumulative Number of Register Operand Instances with Dynamic Bitwidth in Bits for a Selected Register Operand in Benchmark `adpcm decode`



To appreciate the difference of dynamic bitwidth and maximum bitwidth, first a register operand is randomly selected in the benchmark `adpcm decode` and the cumulative number of register operand instances with dynamic bitwidth measured in bits for this register operand is shown in Figure 2.5. In this figure, the register operand has a maximum bitwidth of 15 bits. There are 20 instances of this register

operand in total. Among the 20 instances, there is only one instance using 15 bits but 15 instances using less than 9 bits. This implies there may exist a significant difference between the values of maximum and dynamic bitwidth.

This phenomenon exists in all the benchmarks studied. Figure 2.6 shows the percentage of register operand instances whose dynamic bitwidth is less than 25%, 50%, 75%, and 100% of the maximum bitwidth of the corresponding register operand. Take `cjpeg` for example. More than 20% of the dynamic register operand instances use less than 25% of the maximum bitwidth, 30% use less than 50% of the maximum bitwidth and almost 50% use less than 75% of the maximum bitwidth.

Figure 2.6 Percentage of Register Operand Instances with Dynamic Bitwidth within 25%, 50%, 75%, and 100% of Maximum Bitwidth

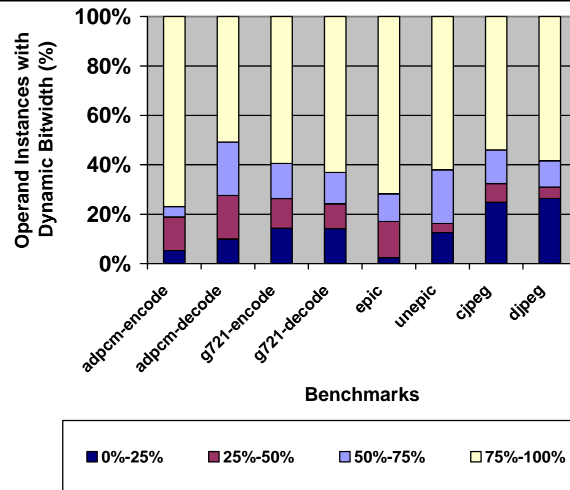


Figure 2.7 shows the average dynamic bitwidth over all register operand instances for a set of benchmarks. The average dynamic bitwidth varies from 10 bits to 14 bits. Compared to the average maximum bitwidth, which varies from 16 bits to 18 bits in Figure 2.3, the average dynamic bitwidth is significantly smaller.

Figure 2.8 shows the percentage of register operand instances whose dynamic bitwidth is less than 16 bits. Observe that from 65% to more than 80% of the register references have dynamic bitwidth of less than 16 bits. This is significantly higher than the corresponding number for maximum bitwidth (around 50% shown in Figure 2.4).

Figure 2.7 Average Dynamic Bitwidth of Register Operand Instances

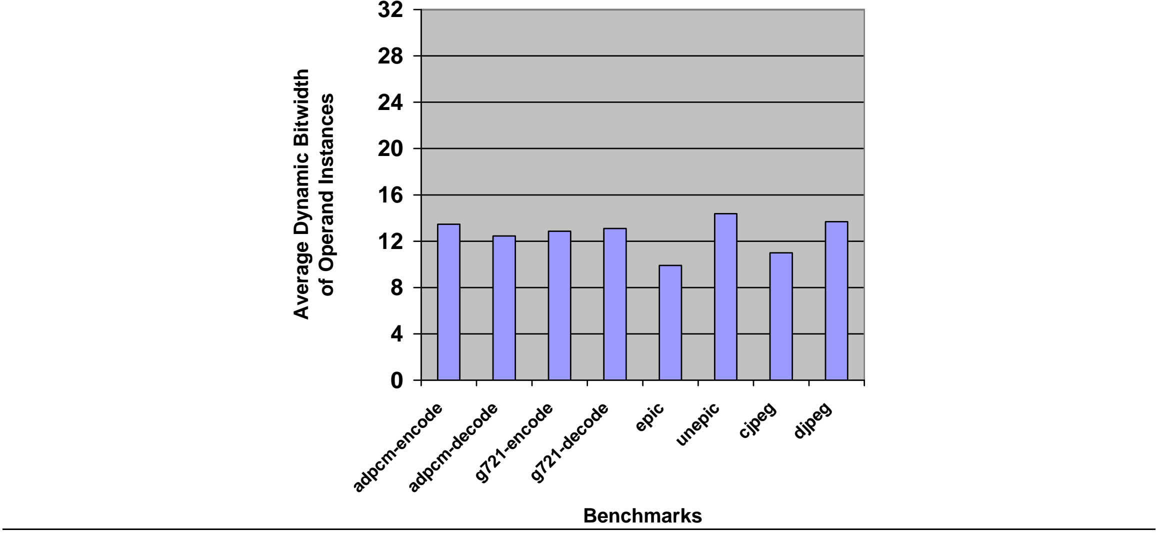
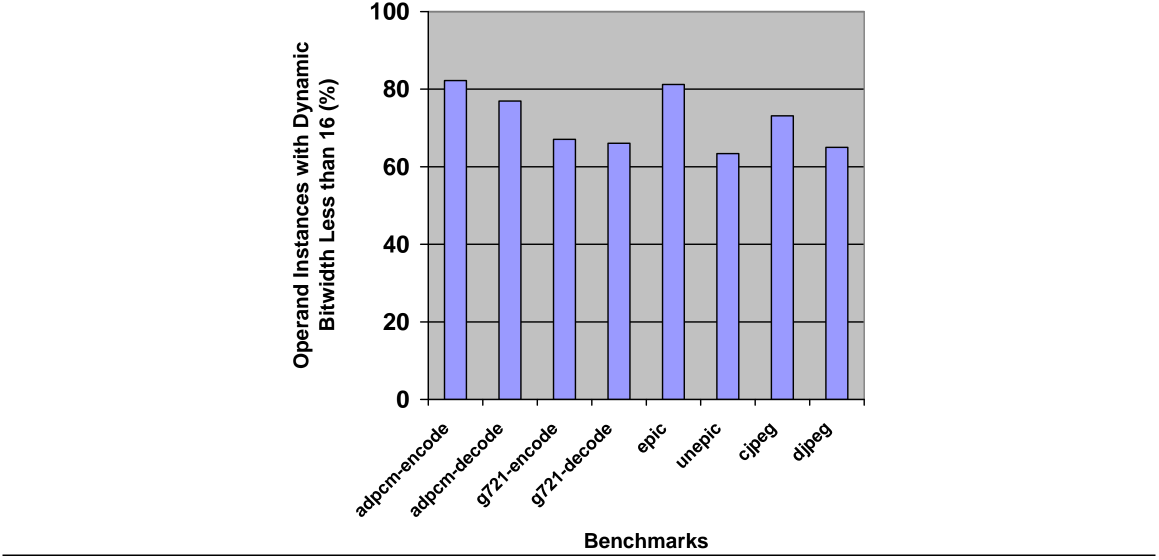


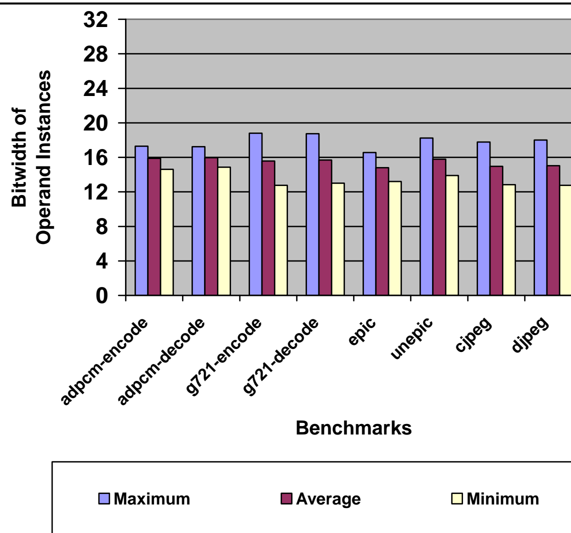
Figure 2.8 Percentage of Register Operand Instances with Dynamic Bitwidth Less than 16



The above observations show that there is a big gap between maximum bitwidth and average dynamic bitwidth. Optimization techniques that are based upon maximum bitwidth do not take advantage of this phenomenon.

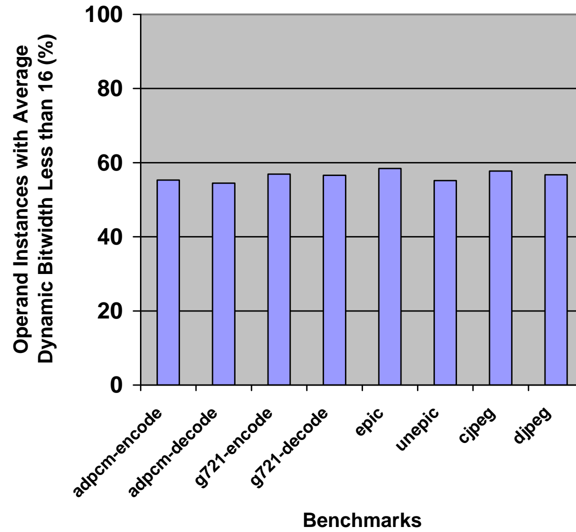
Another observation is that the streaming data processed by these applications show a higher narrow width property than utility data. Figure 2.9 shows the maximum, average, and minimum dynamic bitwidth of a register operand averaged over all register operands. Figure 2.10 shows the percentage of register operands whose average dynamic bitwidth is less than 16. The difference between these two figures and Figures 2.7 and 2.8 is that the data in the latter figures are not weighted by the number of register operand instances. Observe that Figure 2.7 and Figure 2.8 show much higher narrow width property than these two figures. This means that those register operands within the processing loops that have a large number of instances, demonstrate a higher narrow width property.

Figure 2.9 Dynamic Bitwidth Averaged over Register Operands



The data in this section show an opportunity for a compiler and/or an architecture optimization to exploit narrow width data by using register allocation based on dynamic bitwidth.

Figure 2.10 Percentage of Register Operands with Average Dynamic Bitwidth Less than 16



2.4 Related work

Several researchers have studied the presence of narrow width data in different kinds of applications. Brooks et al. [10] study the availability of narrow width instruction operands in SPEC95 in a 64-bit Alpha-like processor. Their study shows that roughly 50% of the dynamic instructions have both operands of less than or equal to 16 bits and that the percentage of instructions that have both operands of less than or equal to 32 bits is slightly higher. This demonstrates that narrow width data exist in general purpose applications. While their study focuses on general purpose applications, the study in the present chapter measures narrow width data in embedded applications. The latter shows that about 60% to 80% of register operand instances are less than or equal to 16 bits in a 32-bit processor. The comparison between the above observations concludes that there are significantly more narrow width data in embedded applications than in general purpose applications. A more recent study is provided by Pokam et al. [40]. They study narrow width data in Powerstone benchmark in a 32-bit RISC-like processor. They show that narrow width data exhibit a property of bitwidth convergence at the granularity of basic blocks. This means that many basic

blocks do not encounter operands with bitwidth more than certain value. The study in the present chapter show that narrow width data are commonplace in embedded applications and more importantly the difference between the maximum and dynamic narrow width data is significant.

Some other researchers have studied the prevalence of narrow width data through *static bitwidth analysis*. Budiu et al. [11] provide such an algorithm called BitValue. It uses forward and backward data flow analysis similar to a generalization of constant folding and dead-code detection at the bit-level. They applied this algorithm on benchmarks such as SPECINT95 and Mediabench. They found that about 31% of bytes are never used regardless of what the input data are. An even earlier bitwidth analysis algorithm is provided by Stephenson et al. [45]. They treat the bitwidth analysis problem as a value-range propagation problem and also use a bi-directional data flow analysis. They study media processing related benchmarks for silicon compilation. Their results show many of the bits are not needed. Gupta et al. [23] provide a representation of programs that makes it easy to reason about subword data entities. Tallam et al. [47] provide an algorithm which can analyze the bitwidth information for live ranges of variables at any program point by eliminating leading and trailing zero and dead bits. All these papers use compile-time analysis techniques. The study in the present chapter does not provide another static analysis technique. Instead, profiling is used to determine the maximum and dynamic bitwidth of register operand. This study demonstrates that there are significantly more dynamic narrow width data than static narrow width data.

2.5 Summary

In this section, the prevalence of narrow width data in embedded applications is quantitatively measured. This study reveals that there exists a significant amount of static and dynamic narrow width data. This study shows great promise for both

compiler and architectural techniques to optimize registers and memory utilization by exploiting narrow width data. The study yields the following important observation. It is not enough to exploit only static narrow width data. A technique that exploits dynamic narrow width data is expected to bring more benefits.

Chapter 3
An Instruction Set Extension for Narrow Width Data

In previous chapters, it was shown that narrow width data are commonplace in embedded applications. However, existing architectures have weak ability to expose bitwidth information to a compiler. Thus a compiler is not able to provide optimization techniques that efficiently handle narrow width data. The first step towards efficient exploiting narrow width data is to extend an existing architecture with the ability to manipulate narrow width data directly so that multiple narrow width data are allowed to stay in one register or memory location.

In this chapter, an extension to the ARM processor's instruction set, called *Bit Section eXtension* (BSX), is proposed. BSX allows multiple narrow width data to stay in packed form and manipulates the data directly. The cost of operations for packing/unpacking of narrow width data can be reduced. In addition, such an instruction set extension gives the compiler an opportunity to come up with an intelligent allocation of narrow width data to a register or memory location. In doing so, register pressure and memory footprint can be reduced. The ARM processor is selected for the base architecture because it is one of the most popular embedded processors and is being used by many commercial network processing architectures being built today.

As analyzed in Chapter 1 of this dissertation, in most cases the input or the output of an embedded application consist of packed data. If the input consists of packed data, the application typically unpacks it for further processing. If the output is required to be in packed form, the application explicitly packs the processing results before generating the output. The following examples are taken from benchmarks `adpcm` (audio) and `gsm` (speech). The first example is an illustration of an unpacking

operation that extracts a 4-bit entity from `inputbuffer`. The second example illustrates the packing operation of a 5-bit entity in `LARc[2]` with a 3-bit entity in `LARc[3]` into the output buffer.

Example of Unpacking Operation:
`delta = (inputbuffer>>4)&0xf;`
Example of Packing Operation:
`*c++ = ((LARc[2]&0x1f)<<3)|((LARc[3]>>2)&0x7);`

With existing architectural support, packing/unpacking of narrow width data are implemented with additional instructions that carry out shift and logical bitwise operations. These instructions cost CPU cycles and increase the application code size. BSX can manipulate narrow width data directly in packed form and reduce the packing/unpacking cost.

The rest of this chapter is organized as follows. In Section 3.1, the design of BSX is presented. Section 3.2 describes an approach to generating code that makes use of BSX instructions. Section 3.4 describes the experimental results. Related work is discussed in Section 3.5. Concluding remarks are given in Section 3.6.

3.1 Bit Section eXtensions (BSX)

3.1.1 Bit Section Descriptors

Narrow width data entity is also called *bit section*. A bit section is a sequence of consecutive bits within a word. The length of a bit section can vary from 1 bit to 32 bits. Bit sections are specified through the use of *bit section descriptors* (BSDs).

There are two options for specifying a bit section within a word. One way is to specify the starting bit position and the ending bit position within the word. Another way is to specify the starting bit position and bit section length. Either way it takes 10 bits to specify a single bit section: 5 bits for the starting position and 5 bits

for the length or ending position. The latter option is used in BSX. By analyzing the MediaBench and CommBench programs it is found that many instructions have multiple bit section operands of the same size. Therefore when one instruction has multiple bit section operands, they can share the same bit section *length* specification. The lengths of multiple bit sections used by an instruction are often the same and can be specified just once. The starting bit positions of these bit sections are often different and thus they cannot be shared.

3.1.2 Bit Section Addressing Modes

There are two different *addressing modes* through which bit section descriptors can be specified. While the position of many bit sections within the word boundary can be determined at compile time, the position of some bit sections can only be determined at runtime. Therefore two addressing modes are needed for specifying bit sections: a bit section operand can be specified as an *immediate value* encoded within the instruction; or a bit section can be specified in a *register* if it cannot be expressed as an immediate constant. The number of bit section operands that are used by various BSX instructions can vary from one to three.

Immediate Bit Section Descriptors. An immediate bit section descriptor is encoded as part of the instruction. Let us assume that `R` is a register operand of an instruction. It can be specified using 4 bits (ARM contains 16 general purpose registers). If the operand is a bit section within `R` whose position is known to be fixed, then an *immediate bit section descriptor* is associated with the register as follows:

`R[#start,#len]` refers to:
bits `[#start..#start+#len-1]` of `R`.

The constant `#start` is 5 bits long since the starting position of the bit section may vary from bit 0 to bit 31. The constant `#len` is also 5 bits long since the number of

bits in the bit section may include all the bits (0..31) of the register. Note that for a valid bit section descriptor `#start+#len-1` is never greater than 31.

Immediate bit section descriptors are used if either the instruction has only one such bit section operand or two such bit section operands. When two bit section descriptors need to be specified, the `#len` specification is the same for both descriptors and hence it can be shared by the two descriptors as follows:

`R1[#start1],R2[#start2],#len` refers to:
bits `[#start1..#start1+#len-1]` of `R1`; and
bits `[#start2..#start2+#len-1]` of `R2`.

Register Bit Section Descriptors. When both the source operands of an instruction as well as its destination operand are bit sections, three bit section descriptors need to be specified. Even though all three bit sections share the same length, it is impossible to specify all three bit sections as immediate values because not enough bits are available in an instruction. Therefore in such cases the specification of the bit section descriptors is stored in a register rather than as an immediate value in the instruction itself.

There is another reason for specifying bit section descriptors in registers. In some situations the positions and lengths of the bit sections within a register are not fixed but rather determined at runtime by the program. In this case the bit section descriptor is not an immediate value specified as part of the instruction but rather the descriptor is computed into the register which is then specified as part of the instruction. The register which specifies the bit section descriptor may specify one, two, or three bit sections in one, two, and three (possibly different) registers. Register BSDs are shown as follows:

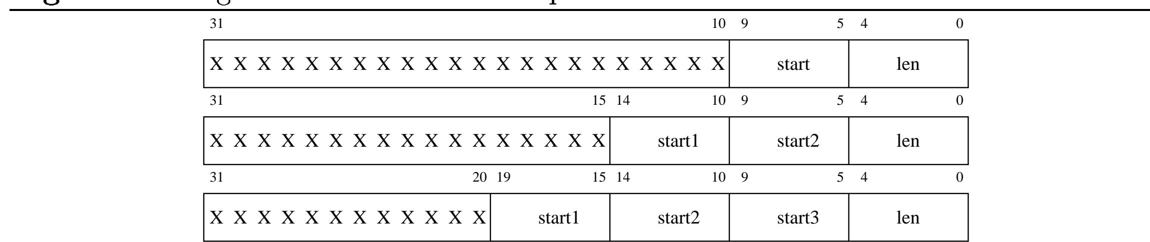
R1[R]

R1,R2[R]

R1,R2,R3[R]

where register R contains the bit section descriptors for the appropriate operand registers R1, R2, and R3. The contents of R are organized as shown in Figures 3.1.

Figure 3.1 Register Bit Section Descriptors

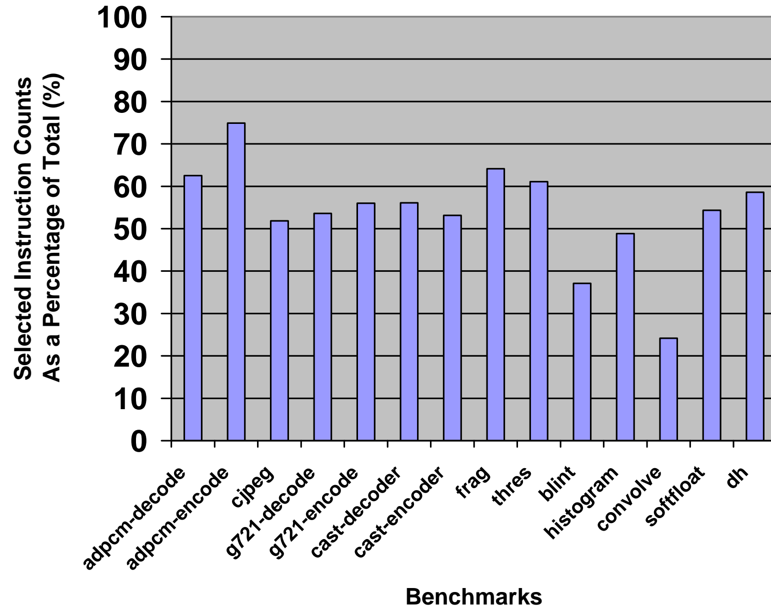


3.1.3 Bit Section Instructions and their Encoding

Next ARM instructions are described that use bit section operands. While one might think it is possible to allow any existing ARM instruction with register operands to access bit sections as operands, this cannot be allowed for all instructions as there would be too many new variations of instructions and there is not enough space in the 32-bit encoding of ARM instructions to accommodate these new instructions. Therefore a subset of instructions are selected which are most likely to be involved in bit section operations. Eight data processing instructions are selected from version 5T of the ARM instruction set which include six ALU instructions (add, sub, and, eor, orr, and rsb), compare (cmp), and move (mov) instructions. The selection of these instructions was based on a study of a number of multimedia benchmarks which determined that these instructions are the most commonly used.

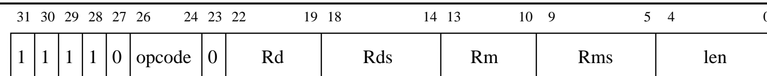
Figure 3.2 shows dynamic instruction count of selected instructions for BSX as a percentage of total dynamic instruction count. The selected instructions account for a significant percentage of total dynamic instruction count.

Figure 3.2 Dynamic Instructions Count of Selected Instructions for BSX as a Percentage of Total



Instructions with Immediate BSDs. For each of the above instructions three variations are provided when immediate bit section operands are used. In version 5T of the ARM instruction set, the encoding space with prefix 11110 is undefined. The remaining unused 27 bits of space is used to deploy the new instructions. Three bits are used to specify the operation.

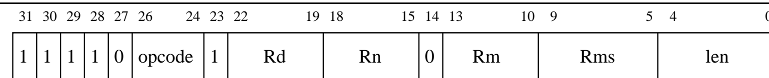
Figure 3.3 BSX Instruction with Immediate BSD (First Variation)



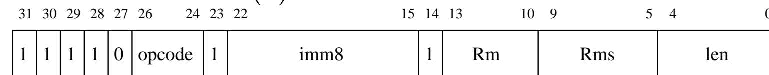
In the *first variation* (FV) of the above ALU instructions, the corresponding instructions have two bit section operands. Therefore one operand acts both as a source operand and a destination operand. The variants of `cmp` and `mov` instructions are slightly different as they require only two operands, unlike the ALU instructions which require three operands. For `cmp` the two bit section operands are both source operands and for `mov` one operand is the source and the other is the destination. We

do not allow all three operands to be bit section operands at the same time because three bit section operands will need at least 32 bits to specify. The encoding of these instructions is shown in Figure 3.1.3. The prefix 11110 in bits 31 to 27 indicates the presence of BSX instruction. Three bits that encode the eight operations are bits 24 to 26. Bit 23 is 0, which indicates this is the first variation of the instruction. The remaining bits encode the two bit section descriptors: `Rd[Rds,len]` and `Rm[Rms,len]`.

Figure 3.4 BSX Instruction with Immediate BSD (Second Variation)



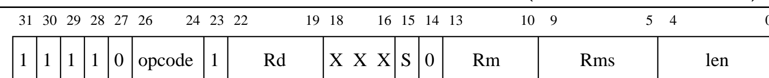
(a) ALU Instructions



(b) `cmp` and `mov` Instructions

The *second variation* (SV) of instructions has three operands. One is a destination register (not a bit section), one is a source register (not a bit section), and the third operand is a bit section operand. In this variation the operation is done as if the bit section is zero-extended. To specify this variation bit 23 must be 1 and bit 14 must be 0. The instruction format and encoding is shown in Figure 3.4. `cmp` and `mov` instructions are again slightly different as they need only two operands. Bit 15 is a flag to indicate whether the bit section is to be treated as an unsigned or signed entity. If it is 0, then it is unsigned and then zero-extended before the operation. If it is 1, the bit section is signed, and therefore the first bit in the bit section is extended before the operation.

Figure 3.5 BSX Instruction with Immediate BSD (Third Variation)

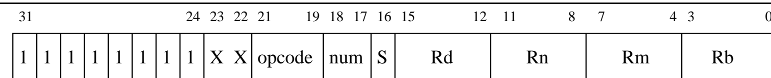


The *third variation* (TV) has one 8 bit immediate value which is one of the

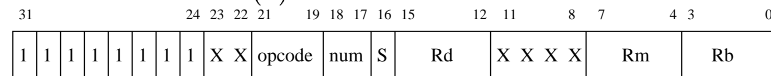
operands and one bit section descriptor which represents the second operand. The latter bit section also serves as the destination operand. To specify this variation, bit 23 must be 1 and bit 14 must be 1. The instruction format and encoding is shown in Figure 3.5.

Instructions with Register BSDs. For each of the above instructions three variations are available when register bit section operands are used. These variations differ in the number of bit section operands. Another undefined instruction space with prefix 11111111 is used to encode these instructions into version 5T of the ARM instruction set. The encoding of the instructions is shown in Figure 3.6.

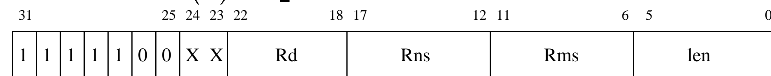
Figure 3.6 BSX Instruction with Register BSD



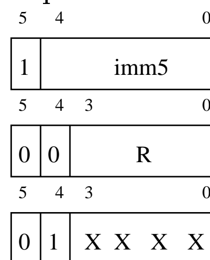
(a) ALU Instructions



(b) `cmp` and `mov` Instructions



(c) Setup BSD Instruction



(d) Setup Specifier

Bits 19 to 21 contain the opcode while bits 17 and 18 stand for the number of bit section operands in the instruction. Therefore 01, 10, and 11 correspond to presence of 1, 2, and 3 bit section operands. The `S` bit specifies whether the bit section contains unsigned or signed integer. The format and encoding of the instructions is given in

Figure 3.6 (a). Instructions `cmp` and `mov` are a little bit different, they can have at most two bit section operands. Therefore bits 17 and 18 can only be 01 or 10 and bits 8 to 11 are not used. The encoding is given in Figure 3.6 (b). The bit section descriptor in itself contains several bit sections. Therefore setup costs of a bit section descriptor in a register can be high. Therefore a new instruction with opcode `setup` is introduced to set up the bit section descriptors efficiently. This instruction can set multiple values in bit section descriptor simultaneously. The format and encoding of this instruction is given in Figures 3.6 (c) and (d). The instruction `setup Rd, Rns, Rms, len` can set up the value of `Rns`, `Rms` and `len` fields in bit section descriptor held in`Rd` simultaneously. A 6-bit setup specifier describes how a field is set up. In each setup specifier, if bit 5 is 1, then bits 0 to 4 represent an immediate value. The field is setup by copying this immediate value. If bit 5 is 0, and bit 4 is 0, then bits 0 to 3 are used to specify a register. The field is setup by copying the last five digits in the register. For Rns specifier, if bit 5 is 0 and bit 4 is 1, then Rns is not a valid bit section specifier and must be ignored. In general, since all three values (`Rns`, `Rms`, and `len`) can be in registers, it needs to read these registers to implement the instruction in one cycle. However, in practice situations where there was a need to read three registers are never encountered.

3.1.4 BSX Implementation

To implement the BSX instructions two approaches are possible. One approach involves redesign of the register file. The bit section can be directly supplied to the register file during a read or write operation and logic inside the register file ensures that only the appropriate bits of a register are read or written.

An alternative approach which does not require any modification to the register file reads or writes an entire register. During a read, the entire register is read, and then logic is provided so that the relevant bit section can be selected to generate the

bit section operand for an instruction. Similarly during a write, update only some of the bits in a register. In the cycle immediately before the write back operation occurs, the contents of the register to be partially overwritten are read. The value read is made available to the instruction during the write back stage where the relevant bit section is first updated and then written to the register file. An extra dedicated read port should be provided to perform the extra read associated with each write operation.

The advantage of the first approach is that it is more energy efficient. Even though it requires the redesign of the register file, it is also quite simple. The second approach is not as energy efficient, it requires greater number of register reads, and is also more complex to implement.

3.2 Generating BSX ARM Code

The approach to generating code with the BSX instructions is to replace a set of ARM instruction sequence patterns with BSX instructions in a compiler post pass. The optimization is aimed at packing and unpacking operations of bit sections with *compile time fixed* and *dynamically varying* positions.

3.2.1 Fixed Unpacking

An unpacking operation involves merely extracting a bit section from a register that contains packed data and placing the bit section by itself in the lower order bits of another register. The following example illustrates unpacking which extracts bit section 4..7 from `inputbuffer` and places it in lower order bits of `delta` (the higher order bits of `delta` are 0). The ARM code requires two instructions, a `shift` and an `and` instruction. However, a single BSX instruction is sufficient to perform unpacking. The `mov` instruction takes bits 4 to 7, zero-extends them, and places them in a register.

C code
`delta = (inputbuffer>>4)&0xf;`

ARM code
`mov r3, r8, asr #4` `and r12, r3, #15 ; 0xf`

BSX ARM code
`mov r12, r8[#4,#4]`

The general transformation that optimizes the unpacking operation takes the following form. In the ARM code an `and` instruction extracts bits from register `ri` and places them in register `rj`. Then the extracted bit section placed in `rj` is used possibly multiple times. In the transformed code, the `and` instruction is eliminated and each use of `rj` is replaced by a direct use of bit section in `ri`. This transformation also eliminates the temporary use of register `rj`. Therefore, for this transformation to be legal, the compiler must ensure that register `rj` is indeed temporarily used, that is, the value in register `rj` is not referenced following the code fragment.

Before Transformation

`and rj, ri, #mask(#s,#l)`
`inst1 use rj`
`...`
`instn use rj`

Precondition
the bit section in `ri` remains unchanged until `instn` and `rj` is dead after `instn`.

After Transformation

`inst1 use ri[#s,#l]`
...
`instn use ri[#s,#l]`

3.2.2 Fixed Packing

C code
`*c++ = ((LARc[2]&0x1F)<<3)|((LARc[3]>>2)&0x7);`

ARM code

`; r0 ← LARc[3]`
`;` $(LARc[3] >> 2)\&0x7$
`mov r0, r0, lsr #2`
`and r0, r0, #7`
`; r1 ← LARc[2]`
`; (LARc[2] & 0x1F) << 3`
`and r2, r1, #31`
`orr r0, r0, r2, asl #3`

BSX ARM code

`; r0 ← LARc[3]`
`; r1 ← LARc[2]`
`mov r0, r0[#2,#3]`
`mov r0[#3,#5], r1[#0,#5]`

When a bit section is extracted from a data word in ARM code, `shift`, and `and` operations must be performed. Such operations can be eliminated as a BSX instruction can be used to directly reference the bit section. This situation is illustrated by the example given above. The C code takes bits 0..4 of `LARc[2]` and concatenates them with bits 2..4 of `LARc[3]`. The first two instructions of the ARM code extract the relevant bits from `LARc[3]`, the third instruction extracts relevant bits from `LARc[2]`, and the last instructions concatenates the bits from `LARc[2]` and `LARc[3]`. The BSX ARM code only has two instructions. The first instruction extracts bits from `LARc[3]`, zero-extends them, and stores them in register r0. The second instruction moves the relevant bits of `LARc[2]` from register r1 and places them in proper position in register r0.

In general the transformation for eliminating packing operations can be characterized as follows. An instruction defines a bit section and places it into a temporary register `ri`. The need to place the bit section by itself into a temporary register `ri` arises because the bit section is possibly used multiple times. Eventually the bit

section is packed into another register `rj` using an `orr` instruction. In the optimized code, when the bit section is defined, it can be directly computed into the position it is placed by the packing operation, that is, into `rj`. All uses of the bit section can directly reference the bit section from `rj`. Therefore the need for temporary register `ri` is eliminated and the packing `orr` instruction is eliminated. For this transformation to be legal, the compiler must ensure that register `ri` is indeed temporarily used, that is, the value in `ri` is not referenced after the code fragment. The transformation is shown below.

Before Transformation

```
inst0 def ri ;bit section definition in a whole register
...
inst1 use ri ;use register
...
instn use ri ;use register
...
orr rj, rj, ri ;pack bit section
```

Precondition

the bit sections in `ri` and `rj` remain unchanged
until `orr` and `ri` is dead after `orr`.

After Transformation

```
inst0 def rj ;define and pack
...
inst1 use rj ;use bit section
...
instn use rj ; use bit section
```

3.2.3 Dynamic Unpacking

There are situations in which, while extraction of bit sections is to be carried out, the position of the bit section is determined at runtime. In the following example, a number of lower order bits, where the number equals the value of variable `size`, are extracted from `put_buffer`, zero-extended, and placed back into `put_buffer`. Since the value of `size` is not known at compile time, an immediate value cannot be used to specify the bit section descriptor. Instead the first three ARM instructions

shown below are used to dynamically construct the `mask` which is then used by the `and` instruction to extract the required value from `put_buffer`. In the optimized code the bit section descriptor is setup in register `r3` and then used by the `mov` instruction to extract the require bits and place them by themselves in `r7`.

C code
`mask = (1<<size)-1;` `put_buffer = put_buffer&mask;`
ARM code

`; r5 ← size`
`; r7 ← put_buffer`
`mov r3, #1`
`mov r3, r3, lsl r5`
`sub r3, r3, #1`
`and r7, r7, r3`

BSX ARM code

`; start = 0`
`; len = r5`
`setup r3, _, #0, r5`
`mov r7, r7[r3]`

The general form of this transformation is shown below. The ARM instructions that construct the mask are replaced by a single `setup` instruction. The `and` instruction can be replaced by a `mov` of a bit section whose descriptor can be found in the register set up by the `setup` instruction.

Before Transformation

`mov ri, #1`
`mov ri, ri, lsl rj`
`sub ri, ri, #1`
`and rd, rn, ri`

Precondition
value in `ri` should be dead after `and` instruction.
After Transformation
`setup ri, rj, rj` `mov rd, rn[ri]`

3.2.4 Dynamic Packing

C code
`o = (m&((1<<p)-1)|(n&(((1<<(16-p))-1)<<p))`

ARM code

```
; r3 ← p
mov r12, #1
rsb r2, r3, #16 ; 16-p
mov r2, r12, lsl r2 ; 1<<(16-p)
sub r2, r2, #1 ; (1<<(16-p))-1
; r1 ← n
and r1, r1, r2, lsl r3 ; n&((1<<(16-p))-1)
mov r12, r12, lsl r3 ; 1<<p
sub r12, r12, #1 ; (1<<p) - 1
; r0 ← m
and r0, r0, r12 ; m & ((1<<p)-1)
orr r0, r0, r1 ; o ← r0
```

BSX ARM code

```
; r3 ← p
setup r12, _, #0, r3 ; descriptor for m's bit section
rsb r2, r3, #16
setup r2, _, r3, r2 ; descriptor for n's bit section
; r0 ← m
mov r0, r0[r12] ; put m's relevant bits in r0
mov r0, r1[r2] ; put n's relevant bits in r0
```

Packing of bit sections together, whose sizes are not known till runtime, can cost several instructions. The C code given above extracts lower order `p` bits from `m` and higher order `16-p` bits from `n` and packs them together into `o`. The ARM code for this operation involves many instructions because first the required masks for `m` and `n` are generated. Next the relevant bits are extracted using the masks and finally they are packed together using the `orr` instruction. In contrast the BSX ARM code uses far fewer instructions. Since the value of `p` is not known at compile time, register bit section descriptors for `m` and `n` must be used.

In general the transformation for optimizing dynamic packing operations can be described as follows. Two or more bit sections, whose positions and lengths are unknown at compile time, are extracted from registers where they currently reside

and put into separate registers respectively. A mask is constructed and an `and` instruction is used to perform the extraction. Finally they are packed together into one register using `orr` instruction. In the optimized code, for each bit section, a register bit section descriptor is set up first, and then move the bit section into the final register with the bit section descriptor directly. As a result, `orr` instruction is removed. By using the `setup` instruction to simultaneously setup several fields in the bit section descriptor, the number of instructions is reduced in comparison to the instruction sequence used to create the masks in the original code. Different types of instruction sequences can be used to create a mask and thus it is not always possible to identify such sequences. The current implementation can only handle some commonly encountered sequences. The transformation is shown below.

Before Transformation

`instruction sequence to create mask1`
`and ra, rb, mask1`
`instruction sequence to create mask2`
`and rc, rd, mask2`
`orr rm, ra, rc`

After Transformation

`setup register bit section descriptor 1`
`move bit section 1 to rm using bit section descriptor 1`
`setup register bit section descriptor 2`
`move bit section 2 to rm using bit section descriptor 2`

3.3 Experimental Evaluation

3.3.1 Experimental Setup

Before the results of the experiments are presented, the experimental setup is described. The setup includes a simulator for ARM, an optimizing compiler, and a set of relevant benchmarks.

Processor Simulator. The experiment starts with a port of the cycle level simulator Simplescalar [12] to ARM available from the University of Michigan. This version

simulates the five stage pipeline described in the preceding section which is the Intel's SA-1 StrongARM pipeline [18] found, for example, in the SA-110. The I-Cache configuration for this processor are: 16Kb cache size, 32b line size, and 32-way associativity, and miss penalty of 64 cycles (a miss requires going off-chip). The timing of the model has been validated against a Rebel NetWinder Developer workstation [42] by the developers of the system at the University of Michigan.

Optimizing Compiler. The compiler used in this work is the `gcc` compiler which was built to create a version that supports generation of ARM code. Specifically the `xscale-elf-gcc` compiler version `2.9-xscale` is used. All programs are compiled at -O2 level of optimization. -O3 level is not used because at that level of optimization function inlining and loop unrolling is enabled. Clearly since the code size is an important concern for embedded systems, function inlining and loop unrolling should not be used.

The translation of ARM code into optimized BSX ARM code was carried out by an optimization post-pass. Only the frequently executed functions in each program that involve packing, unpacking and use of bit section data were translated into BSX ARM code. The remainder of the program was not modified.

Representative Benchmarks. The benchmarks used are taken from the Mediabench [28], Commbench [50], Netbench [5], and Bitwise [45] suites as they are representative of a class of applications important for the embedded domain. An image processing application `thres` is also added.

3.3.2 Results

Experimental results are presented to measure the improvements in performance and code size due to the use of BSX instructions. The reductions in both the instruction counts and cycle counts are measured for BSX ARM code in comparison to pure ARM

code. The results are provided for each of the functions that were modified as well as for the entire program. Table 3.1 shows the reduction in the dynamic instruction counts. The reductions in *instruction counts* for the modified functions vary between 4.26% and 27.27%. The net instruction count reductions for the entire programs are lower and range from 0.45% to 8.79%. This is to be expected because only a subset of functions in the programs can make significant use of the BSX instruction. Table 3.2 shows the reduction in the cycle counts. The reductions in *cycle counts* for the modified functions vary between 0.66% and 27.27%. The net cycle count reductions for the entire programs range from 0.39% to 8.67%. Table 3.3 shows the reductions in code size for the functions that were transformed to make use of BSX instructions. The code size reductions for modified functions range from 1.27% to 21.05%. However the reudctions in code size for the entire programs are small.

3.4 Related Work

A wide variety of instruction set support has been developed to support multimedia and network processing applications. Most of these extensions have to do with exploiting subword [21] and superword [27] parallelism. The instruction set extensions proposed by Yang and Lee [52] focus on permuting subword data that are packed together in registers. These techniques support manipulating narrow width data of bitwidth in bytes on a high-end processor. In contrast, BSX can address narrow width data of flexible width on a 32-bit embedded processor.

In [39], a notion of bit section referencing in a network processor is presented. Two fields, including the register number and the bit offset, are used to specify narrow width operands. However the design details are not presented. In contrast, BSX addresses more narrow width operands with immediate and register bit section descriptors. Furthermore, a detailed design of BSX on ARM processor and a method to use BSX for code optimization are presented in this chapter.

Table 3.1. Reduction in Dynamic Instruction Counts

Benchmarks	Functions & Total	Instruction Count		Savings [%]
		ARM	BSX ARM	
adpcm.decode	adpcm_decoder	6124744	5755944	6.02%
	Total	6156561	5787760	5.99%
adpcm.encode	adpcm_encoder	7097316	6654756	6.24%
	Total	7129778	6687534	6.20%
jpeg.cjpeg	emit_bits	634233	586291	7.56%
	Total	15765616	15694887	0.45%
g721.decode	fmult	47162982	43282495	8.23%
	predictor_zero	9293760	8408640	9.52%
	step_size	1468377	1320857	10.05%
	reconstruct	2628342	2480822	5.61%
	Total	258536428	253180667	2.07%
g721.encode	fmult	48750464	44367638	8.99%
	predictor_zero	9293760	8408640	9.52%
	step_size	2372877	2225357	6.22%
	reconstruct	2645593	2498073	5.58%
	Total	264021499	258163419	2.22%
cast.decoder	CAST_encrypt	41942016	37850112	9.76%
	CAST_ofb64_encrypt	26980992	25190784	6.64%
	Total	109091228	103209100	5.40%
cast.encoder	CAST_encrypt	41942016	37850112	9.76%
	CAST_ofb64_encrypt	26980992	25190784	6.64%
	Total	105378485	99496358	5.58%
frag	in_cksum	26991150	25494952	5.54%
	Total	37506531	36010318	3.99%
threshold	coalesce	3012608	2602208	13.62%
	memo	3223563	2814963	12.68%
	blocked_memo	2941542	2531826	13.93%
	Total	20959630	19730898	5.86%
bilint	main	87	79	9.20%
	Total	496	488	1.61%
histogram	main	317466	301082	5.16%
	Total	327311	310857	5.03%
convolve	main	30496	30240	0.84%
	Total	30799	30542	0.83%
softfloat	float32_signals_nan	132	96	27.27%
	addFloat32Sigs	29	23	20.70%
	subFloat32Sigs	29	23	20.70%
	float32_add	8	7	12.50%
	float32_sub	8	7	12.50%
	float32_mul	30	23	23.33%
	float32_div	30	24	20.00%
	float32_rem	28	23	17.86%
	float32_sqrt	23	20	13.04%
	float32_eq	23	17	26.09%
	float32_lt	17	11	35.29%
	Total	898	819	8.79%
dh	NN_DigitMult	153713163	141768387	7.77%
	NN_DigitDiv	19319249	18495393	4.26%
	Total	432372762	419604191	2.95%

Table 3.2. Reduction in Dynamic Cycle Counts

Benchmark	Functions & Total	Cycle Count ARM	Cycle Count BSX ARM	Savings [%]
adpcm.decode	adpcm_decoder	6424241	6202961	3.44%
	Total	6499880	6278786	3.40%
adpcm.encode	adpcm_encoder	7958088	7515456	5.56%
	Total	8035001	7592761	5.50%
jpeg.cjpeg	emit_bits	1047235	999163	4.59%
	Total	19611965	19535002	0.39%
g721.decode	fmult	63914793	60034237	6.07%
	predictor_zero	12834446	11949382	6.90%
	step_size	1564728	1269752	18.85%
	reconstruct	2601534	2454014	5.67%
	Total	347037906	341531879	1.59%
g721.encode	fmult	65798336	61415518	6.66%
	predictor_zero	12834447	11949327	6.90%
	step_size	2630082	2335106	11.22%
	reconstruct	2636030	2488439	5.60%
	Total	353610636	347605462	1.70%
cast.decoder	CAST_encrypt	46557053	40674664	12.63%
	CAST_ofb64_encrypt	32224708	30434422	5.56%
	Total	141113081	133440304	5.44%
cast.encoder	CAST_encrypt	46557174	40674817	12.63%
	CAST_ofb64_encrypt	32224703	30434428	5.56%
	Total	135572465	127900147	5.66%
frag	in_cksum	32698919	31205099	4.57%
	Total	57745393	56207197	2.66%
threshold	coalesce	4355796	3937458	9.60%
	memo	4725060	4307735	8.83%
	blocked_memo	22092904	21683166	1.85%
	Total	181425566	180186381	0.68%
bilint	main	887	808	8.91%
	Total	5957	5878	1.32%
histogram	main	481531	462532	3.95%
	Total	496650	477807	3.79%
convolve	main	40215	39949	0.66%
	Total	44945	44803	0.32%
softfloat	float32_signals_nan	132	96	27.27%
	addFloat32Sigs	324	247	23.77%
	subFloat32Sigs	675	595	11.85%
	float32_add	81	80	1.23%
	float32_sub	217	153	29.49%
	float32_mul	577	513	11.09%
	float32_div	397	380	4.28%
	float32_rem	453	314	30.68%
	float32_sqrt	383	295	22.98%
	float32_eq	242	227	6.20%
	float32_lt	305	229	24.92%
	Total	10255	9366	8.67%
dh	NN_DigitMult	236874768	224929644	5.04%
	NN_DigitDiv	26223468	25400239	3.14%
	Total	578187905	565434086	2.21%

TABLE 3.3. Reduction in Code Size

Benchmark Function	Code Size		Reduction
	ARM	BSX ARM	[%]
adpcm.decoder			
adpcm_decoder	260	248	4.62%
adpcm.encoder			
adpcm_encoder	300	284	5.33%
jpeg.cjpeg			
emit_bits	228	216	5.26%
g721.decode and g721.encode			
fmult	196	176	10.2%
predictor_zero	92	84	8.7%
step_size	76	72	5.26%
reconstruct	96	92	4.17%
cast.decoder and cast.encoder			
CAST_encrypt	1328	1200	9.64%
CAST_ofb64_encrypt	428	400	6.54%
frag			
in_cksum	108	88	18.52%
threshold			
coalesce	148	136	8.11%
memo	296	284	4.05%
blocked_memo	212	200	5.66%
bilint			
main	352	320	9.09%
histogram			
main	316	312	1.27%
convolve			
main	652	648	0.61%
softfloat			
float32_signals_nan	52	40	20.0%
addFloat32Sigs	348	324	6.90%
subFloat32Sigs	396	372	6.06%
float32_add	40	36	10.0%
float32_sub	40	36	10.0%
float32_mul	428	400	6.54%
float32_div	544	520	4.41%
float32_rem	648	628	3.09%
float32_sqrt	484	464	4.13%
float32_eq	152	120	21.05%
float32_lt	176	144	18.18%
dh			
NN_DigitMult	112	104	7.14%
NN_DigitDiv	420	404	3.81%

3.5 Summary

This chapter presents BSX instruction set extension of ARM processor that can manipulate narrow width data directly in packed form. BSX can be easily encoded into the free encoding space of the ARM instruction set. A compiler optimization post-pass can generate efficient BSX ARM by replacing a set of instruction sequence patterns. The cost of packing/unpacking operations of narrow width data is reduced. BSX achieves performance improvement and code size reduction. Furthermore, BSX exposes narrow width data to a compiler. Thus a compiler has opportunities to perform an intelligent allocation of narrow width data in registers or memory.

Chapter 4
Register Allocation Exploiting Narrow Width Data

The study in Chapter 2 has shown that embedded applications, including network and media processing applications, contain significant levels of narrow width data. In this chapter it is shown how narrow width data can be exploited to make effective use of the small number of registers provided by embedded processors. This problem is addressed in context of the ARM processor [22] which supports 16 registers.

The main objective of this chapter is to develop the architectural and compiler support through which the registers can be used more effectively in presence of narrow width data. Generally the observed narrow width data can be divided into two categories: (static) a variable is declared as a word but in reality the values assigned to the variable can never exceed 16 bits; and (dynamic) a variable is declared as a word but in practice the values assigned to it do not exceed 16 bits most of the time during a program run. As shown in the study in Chapter 2, embedded applications contain a significant amount of static narrow width data and even more dynamic narrow width data.

To exploit dynamic narrow width data in registers using a global bitwidth aware register allocation method, we need architectural supports other than ARM BSX. Although the previous chapter has shown that ARM BSX is effective in achieving performance improvement with a local peephole optimization, ARM BSX is not a suitable architectural support for a global bitwidth aware register allocation method to exploit dynamic narrow width data in registers. First, ARM BSX only contains a limited set of most frequently used instructions because not enough instruction encoding space is available. A global bitwidth aware register allocation method may

require that arbitrary instruction has the ability to address narrow with data. Second, it is difficult for ARM BSX to capture the dynamic bitwidth because ARM BSX can not automatically check the bitwidth at runtime.

Instead, we are seeking to develop a scheme with a simplified architectural support such that a compiler register allocation algorithm can effectively take the advantage of the architectural support. In particular, we allow two variables to be simultaneously assigned to the same register such that if their values can be represented using 16 bits most of the time, they can be simultaneously held in a register most of the time.

In this chapter, a speculative scheme is designed to exploit narrow width data in registers. As a comparison, a static scheme is also developed to show that the speculative scheme is efficient in handling dynamic narrow width data in registers. The proposed schemes are based on the register allocation in `gcc`.

In the rest of the chapter, first the compiler and architectural challenges of developing such a register allocation scheme are described in Section 4.1. A brief introduction to the register allocation scheme in `gcc` is given in Section 4.2. In Section 4.3 a static subword register allocation scheme is proposed. In Section 4.4 and 4.5 a speculative subword register allocation scheme is presented to exploit dynamic narrow width data. In Section 4.4 the architectural supports are described and Section 4.5 the register allocation algorithm is presented. The implementation and experimental results are provided in Section 4.6. Related work and conclusions are given in Sections 4.7 and 4.8 respectively.

4.1 The Challenges of Bitwidth Aware Register Allocation

The discussion of register allocation in this chapter targets embedded processors where small modifications can be made to effectively manipulate narrow width data. Each register `R` can contain up to two entities that are of same size, i.e., it can hold a whole word $R_{1..32}$ or two half-words $R_{1..16}$ and $R_{17..32}$.

Let us consider the existing approach to bitwidth aware register allocation [48]. Figure 4.1(a) contains a code fragment in which variables `b` and `c` are assigned to registers `R1` and `R2` respectively. Moreover there is no free register for variable `a`. Thus, spill code is generated to store new value of `a` to memory after computation and a load is introduced to bring the value from memory before `a` is used (Figure 4.1(b)). The approach in [48] determines the bitwidths of the variables using data flow analysis. Lets assume that the resulting bitwidths indicate that all three variables can be stored in 16 bits. Thus, the variables can each be assigned to use half of a register as shown in Figure 4.1(c). In the transformed code, instead of being spilled to memory, variable `a` is assigned to use half of a register $\texttt{R1}_{17..32}$ while the other half of the same register is used to hold the value of variable `b`.

Figure 4.1 Bitwidth Aware Register Allocation.

```
/* b can always be
represented using
16 bits*/


a = b + 1;
…

c= a – 10;
```

(a) Original Code

```
; b → R1
; c → R2


add R3, R1, 1
sw R3, addr_a
…
lw R4, addr_a
sub R2, R4, 10
```

(b) with Spill Code

```
; b → R1 1..16
; c → R2 1..16
; a → R2 17..32

add R2 17..32, R1 1..16, 1
…


sub R2 1..16, R2 17..32, 10
```

(c) after Bitwidth Aware RA

For the above approach to be effective, it requires that any arbitrary instruction, which takes the packed narrow width variables as its source operands, is able to address these variables. This requires at least an extra bit associated with each register name even in the best case that only two variables are packed into one register. However, in ARM processor this would require 3 extra bits in the encoding space of any arbitrary instruction which is unrealistic to implement. To solve this problem, a static register allocation scheme, called *Static Subword Register Allocation*, is proposed. This scheme is based on a simplified and implementable architectural support

and a compiler register allocation algorithm which can effectively take advantage of the support.

- *Architectural Support in Static Scheme.* The key challenge for the architecture is to provide a way to access the packed narrow width data without extra bits in the instruction format. This means that at most one narrow width data can be accessed at a time. To access the other one, special instructions should be designed to pack and unpack the other one so that it can be used. Furthermore, the architecture should indicate whether there are one or two variables in one register.

- *Compiler Support in Static Scheme.* The compiler algorithm should efficiently use the architectural support to achieve performance improvement. Obviously it is not efficient if for every reference to the other variable an extra operation is needed to extract it. However it is a good practice that packing and unpacking operations are used to replace the store and load instruction for an originally spilled variable. The cost is reduced since the result of an unpacking operation can be reused by many references. So for the compiler the challenges are picking the pair of variables to be packed and transforming the spill code with special packing and unpacking instructions. At the same time, it should achieve best performance improvement.

While the above approach looks effective, there are situations under which the static register allocation is not effective. It is possible that the compiler is unable to establish that variables `a`, `b`, and `c` occupy only half a word. In particular, there are the following three situations under which the above approach fails.

- *Program does not contain relevant information.* The static approach relies on the compiler to uncover true bitwidths of variables. However, it is possible that the program does not contain enough information to determine the true

bitwidths. For example, an input variable may be declared as a full word variable although the valid inputs do not exceed 16 bits. The compiler fails to detect that the input variable contains subword data; in addition, the bitwidths of variables that depend upon this input will also be likely overestimated.

- *Imprecision in static analysis leads to a failure in uncovering bitwidth information.* Even if the program contains information to infer the true bitwidths of variables, the imprecision in static analysis may lead us to conclude that the bitwidths exceed 16 bits. Presence of calls to libraries, complex bit operations etc. can lead static analysis to provide imprecise information.

- *Bitwidth of variables exceeds 16 bits sometimes.* The static approach misses the opportunity that some variables represent narrow width data most of the time but exceed 16 bits only in a rare case. The static approach still introduces spill code.

In summary, while the static approach can take advantage of statically known narrow width data, it cannot take advantage of dynamically observed narrow width data. A speculative approach, called *Speculative Subword Register Allocation*, is proposed to address this issue. The speculative approach speculatively packs two variables into a single register. Profiling information is used to identify pairs of variables that are highly likely to have their packed values fit into a single register together. If the values do fit into a single register at certain point at runtime, the reloading of the variables from memory at that point can be avoided; otherwise if there is a collision at certain point at runtime that the values do not fit into a single register, the reloading of the variables from memory at that point is performed. While the basic idea of the approach is clear, there are architectural and compiler challenges.

- *Architectural Support in Speculative Scheme.* The key challenge for the architecture is to design instruction set extensions so that the speculative pack-

ing/unpacking operations of two narrow width variables are exposed to the compiler. The compiler should be able to control which variable is *guaranteed* to be found in a register and which variable is *expected* to be present. If two narrow width variables are allocated to one register, the architecture should check the status of the packed variables and avoid the possible reload operations. Finally once two values are present in a register, there is a method to address them. This method should consider the limited instruction encoding space.

- *Compiler Support in Speculative Scheme.* An algorithm needs to be developed for global register allocation such that it leads to performance improvements when speculative register assignment is made. This requires that the pair of variables that are assigned to the same register are carefully chosen. The compiler must choose the variable whose value is guaranteed to be found in the register and the one whose value is expected to be found in the register. Since there is extra cost due to checking whether or not a speculatively stored value is present in a register, the compiler must use profiling information to pick the pair of variables such that it is highly likely that the values of both variables can reside simultaneously in the same register. Finally speculative register assignment should be carefully integrated into conventional global register allocation algorithm.

4.2 Register Allocation in the GCC Compiler

In this section, a brief instruction to the register allocation in gcc compiler [24] is given. The proposed register allocation schemes are based on the register allocation method in gcc compiler.

The gcc compiler performs register allocation in three passes: local register allocation pass, global register allocation pass, and reload pass. These passes operate

on the intermediate representation – register transfer language (RTL). Operands are mapped to virtual registers before register allocation. The local and global register allocation passes do not actually modify the RTL representation. Instead, the result of these passes is an assignment of physical registers to virtual registers. It is the responsibility of the reload pass to modify the RTL and to insert spill code if necessary.

The local register allocation pass allocates physical registers to virtual registers that are both generated and killed within one basic block, i.e., live ranges that are completely local to a basic block are handled in this pass. The allocation is driven by live range priorities. Register coalescing is also performed in this pass. Since local register allocation works on linear code, it is inexpensive. Local allocation reduces the amount of work that must be performed by the more expensive global allocation pass.

The global register allocation pass allocates physical registers to the remaining virtual registers. This pass may change some of the allocation decisions made during the local register allocation pass. This pass performs allocation by coloring the global interference graph. Virtual registers are considered for coloring in an order determined by weighted counts. If a physical register cannot be found for a virtual register, none is assigned and the virtual register is handled by generation of spill code in the reload pass.

The reload pass replaces the virtual register references by physical register names in the RTL according to the allocations determined by the previous two passes. Stack slots are assigned to those virtual registers that were not allocated a physical register in the preceding passes. Reload pass also generates spill code for them. Unlike Chaitin-style [14] graph-coloring allocation, which spills a symbolic register, a physical register is spilled. For each point where a virtual register must be in a physical register, it is temporarily copied into a "reload register" which is a temporarily freed physical register. Reload registers are allocated locally for every instruction that needs reloads.

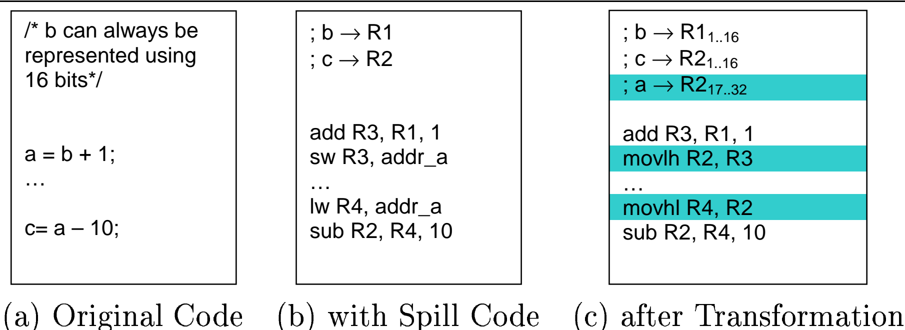
4.3 Static Subword Register Allocation

In this section, first the necessary architectural support to enable static subword register allocation is discussed and the the static subword register allocation algorithm is described.

The following enhancements are made in the architecture. To indicate whether there are two 16-bit values or one 32-bit value in the register, one status bit is attached with each register. The values in the register are addressed as follows. When the bit is clear, the references to the register act like normal register. When the bit is set, normal instruction can only access the lower half of the register. In case of a register read, the lower half is sign-extended. In case of a register write, only the lower 16 bits of the data are written into the lower half of the register. The semantics of the program is guaranteed to be correct by the compiler which only allocates variables whose static bitwidth is less than 16. When the bit is set, the upper half can only be accessed by new instructions. Two new instructions are used to pack and unpack the upper half of a register and set the status bit accordingly. For packing, the instruction moves the lower upper of the source register to the upper half of the destination register. At the same time, it sets the status bit of the destination register. For unpacking, the instruction moves the upper half of the source register, sign-extends it, puts it into the destination register and sets the status bit of the destination register to 0. The mnemonic names of the instructions are `movlh` and `movhl`.

Let us consider the example in Figure 4.2(a) where variable `a`, `b`, and `c` are narrow width. The code in Figure 4.2(b) shows the code generated by a traditional register allocation. Spill code is generated for node `a`. Figure 4.2(c) shows the code if `a`is allocated to the upper half of `R2` and new instructions are used to replace the load and store instructions in the spill code. In this way, it is obvious that the expensive cost of the spill code can be significantly reduced by packing two narrow width variables into one register.

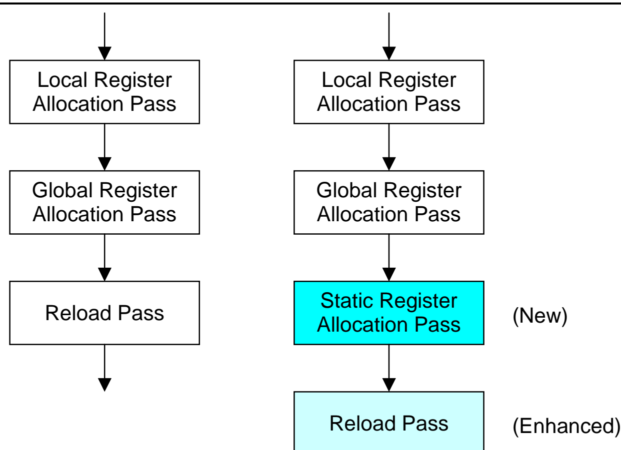
Figure 4.2 An Example of Static Subword Register Allocation



(a) Original Code (b) with Spill Code (c) after Transformation

With the above architectural support, a static subword register allocation algorithm is designed to allocate two narrow width variables to the two halves of one register. This is achieved by allocating those narrow width variables, which should have been spilled in traditional register allocation, to the upper halves of those registers that contain values no more than 16 bits.

Figure 4.3 The Framework for Static Subword Register Allocation.



The static subword register allocation algorithm is integrated into the `gcc` compiler as shown in Figure 4.3. The algorithm is implemented as a separate pass after the local and global register allocation passes. At this point, the following information is available, including whether each variable is narrow width, the assignment of variables to the physical registers and which nodes are spilled. Based on these infor-

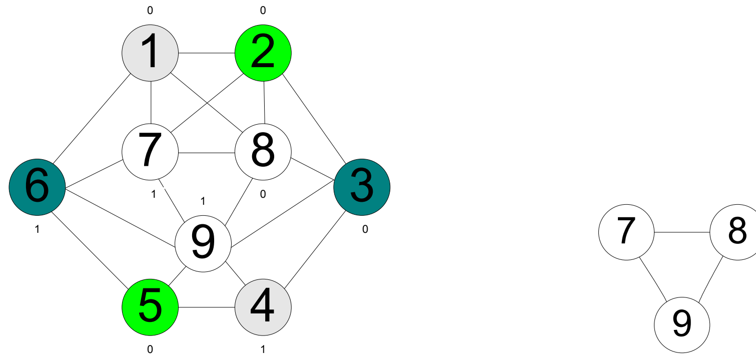
Figure 4.4: The Compiler Algorithm of Static Subword Register Allocation

```
input  : AIG
         RIG
output: Colored RIG
compute the priority of each node in RIG using the priority function;
put these nodes in an ordered list in descending order;
while the list is not empty do
    remove the head node n from the list;
    if n is narrow width variable then
        let AIGColor contain all available colors;
        for each node m of node n's AIG neighbor do
            if node m is not narrow width then
                remove the color of m from AIGColor;
            end
        end
        let RIGColor contain all available colors;
        for each node m of node n's RIG neighbor do
            remove the color of m from AIGColor;
        end
        let Color be the intersection of AIGColor and RIGColor;
        if Color is not empty then
            randomly choose one color from Color;
            update RIG with node n colored;
        end
    end
end
return RIG;
```

mation, the static subword register allocation works as a secondary pass of the global register allocation targeting at allocating narrow width variables. The result of this pass together with the results from local and global passes are used by an enhanced reload pass which generates code with part of the spill stores and load instructions replaced by the new instructions.

The static subword register allocation uses a priority-based graph coloring algorithm which is described in Figure 4.4. The algorithm is based on two interference graphs. One is *Annotated Interference Graph* (AIG) which is the original interference graph used by `gcc` annotated with the coloring decisions from the local and global

Figure 4.5 Interference Graphs in Static Subword Register Allocation



(a) Annotated Interference Graph (b) Residual Interference Graph

passes and the tags which indicate whether each variable is a narrow width variable. An example of AIG is illustrated in Figure 4.5 (a). Another is *Residual Interference Graph* (RIG). It is a subgraph of the AIG which contains only those nodes which do not get their physical register and the interference relationship between them. An example of RIG is illustrated in Figure 4.5 (b). The algorithm colors RIG based on the information from AIG. First, all nodes in RIG are sorted based on the priority information which indicates the spill cost saved by allocating one variable to the register. Next, we examine these nodes in descending order of priority and assign color to each node if it is narrow width. The color selected does not cause any interference in RIG. It also does not lead to any interference between this node and those neighboring nodes in AIG which are not narrow width. Note that if a neighboring node in AIG is narrow width, this node can choose the same color because the two nodes are allocated to different halves of the same register. The priority function is shown as follows.

$$Priority(n) = REF(n) * (COST - 1)$$

$REF(n)$ is the number of references (stores and loads) of this variable n. $COST$ is the cost in CPU cycles of a spill load or store instruction. Since an extra new instruction is used for each reference, the saving for each reference should be $COST - 1$. Thus

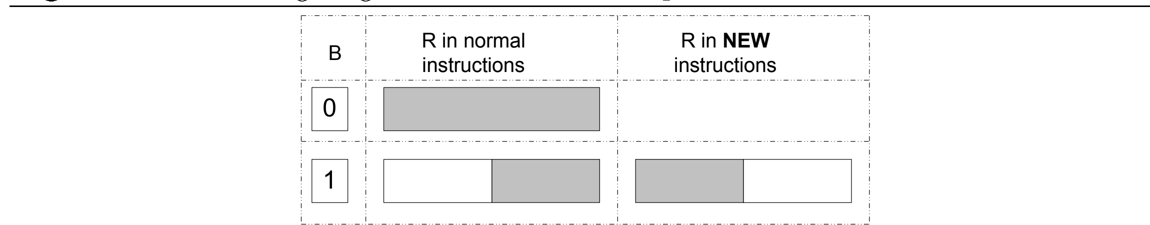
this function approximates the amount of saving achieved by allocating one variable to a register instead of memory. The priority function works as a heuristic to achieve best performance improvement.

4.4 Architectural Support for Speculative Subword Register Allocation

In this section, the architectural support to enable speculative subword register allocation is discussed. To support accessing the packed narrow width data, one bit is attached to each register, a global bit is added to the CPU status register and four new instructions are introduced. Two new instructions, `Ssw` and `Smv`, are always used one after another to fulfill a speculative packing operation. Two new instructions, `Emv` and `Sld`, are always used one after another to fulfill a speculative unpacking operation.

4.4.1 Register File Enhancement

Figure 4.6 Accessing Register `R` as a Source Operand



One extra bit `B` is attached to each register. It is used to indicate whether the register currently holds packed narrow width data. As shown in Figure 4.6, if `B` is cleared, the register holds one 32-bit value. If the bit `B` is set, the two halves of the register store two 16-bit values. The upper half of the register is used to store the speculatively saved value. If a register `Rs` is a source operand of a normal instruction or a speculative packing instruction, `Rs` is always interpreted as follows. If `B` is cleared,

the whole register is used. If `B` is set, the lower half is used and sign-extended. If a register `Rs` is a source operand of a speculative unpacking instruction `Emv`, it is interpreted as follows. If `B` is set, the upper half is used and sign-extended. If `B` is cleared, `Rs` is not used since the speculative unpacking is not successful.

4.4.2 New Instructions

To speculatively store a value in the upper half of the register and to access it later on, four new instructions are introduced. An additional bit `G` is added to the CPU status register which is set and examined by these new instructions. These instructions are described in detail below. The first two instruction are used one after another to speculatively pack a value to the upper half of a register when a spill store happens. The next two instructions are used one after another to unpack a value that was saved by a previous speculative packing operation when a spill reload happens.

1. *Speculative store:* `Ssw Rs, addr`

   This instruction stores the value of `Rs` into the memory address `addr`. In addition, it checks the value in `Rs` to see whether it can be packed. In particular, it sets the `G` bit in the status register if the upper 16 bits are the same as its 16th bit, i.e., they are sign extensions.

2. *Store move:* `Smv Rd, Rs`

   This instruction checks the global condition bit `G` and examine whether the upper half of `Rd` is available to store a packed value. If `G` is set, and the upper half of `Rd` is available, the half of `Rs` is moved into the upper half of `Rd`. The bit `B` of `Rd` is set to indicate that it contains two values.

3. *Extract move:* `Emv Rd, Rs`

   This instruction checks the `B` bit of `Rs`. If the bit is set, the upper half of `Rs` is sign-extended and moved to `Rd`. In addition, the global bit `G` is set to indicate

that the instruction successfully unpacked a speculatively stored value in `Rs`. If bit `B` of `Rs` is not set, bit `G` is cleared to indicate that the value was not found.

4. *Speculative load:* `Sld Rd, addr`

   This instruction checks bit `G`. If it is clear, the value in memory at address `addr` is loaded into `Rd`; otherwise, the instruction acts as a `nop` instruction. In the latter case, the load does not happen because the preceding `Emv` instruction must have successfully unpacked a speculatively stored value into `Rd`.

Figure 4.7 An Example of Speculative Subword Register Allocation

```
/* most of the time
b can be represented
using 16 bits*/


a = b + 1;
…

c= a – 10;
```

(a) Original Code

```
; b → R1
; c → R2


add R3, R1, 1
sw R3, addr_a
…
lw R4, addr_a
sub R2, R4, 10
```

(b) with Spill Code

```
; b → R1[1..32] or R1[1..16]
; c → R2[1..32] or R1[1..16]
; a → in memory or R2[17..32]
add R3, R1, 1
Ssw R3, addr_a
Smv R2, R3
…
Emv R4, R2
Sld R4, addr_a
sub R2, R4, 10
```

(c) after Transformation

Let us consider the example in Figure 4.7(a) and (b) where the value of `a` in register `R3` is being spilled. Assume that it is preferred to speculatively pack the value of `a` into the register which now holds the value of `c` (say register `R2`). The code in Figure 4.7(c) shows how instructions `Ssw` and `Smv` are used to speculatively pack the value into `R2` and then later instructions `Emv` and `Sld` are used to speculatively reload the value of `a` from `R2` into `R4`. If the speculative reload of `a` is successful, the `Sld` instruction turns into a `nop` instruction; otherwise the value of `a` is reloaded from memory.

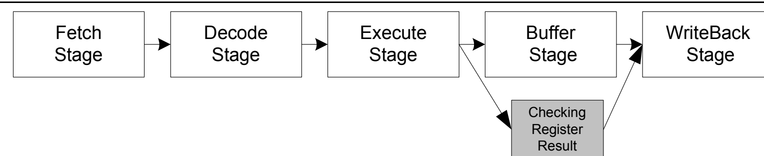
From the above description it is clear that there are cost and benefit in this approach. The cost is the two extra move instructions (`Smv` and `Emv`) that are introduced. The benefit of the technique is that it is possible to avoid a reload from

memory. If it is highly likely that the reload will be avoided, the approach brings more benefit than the cost paid for the extra move instructions. For obtaining benefits, the compiler must make good decisions on using the instructions.

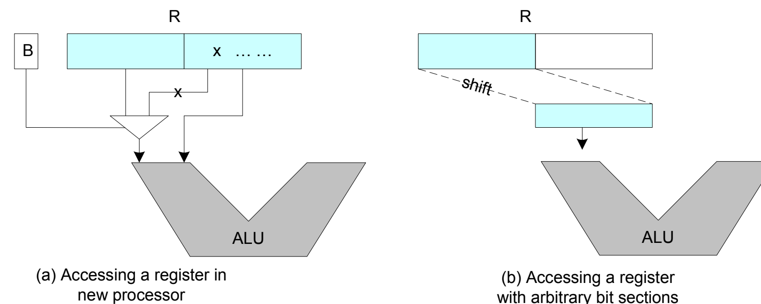
4.4.3 Hardware Implementation

Modifications must be made to the processor pipeline when the above mentioned instructions are introduced. Normal instructions must always check bit `B` of a register that it reads from or writes to. This is required so that it can interpret and update the contents of the register. For modern embedded processors such as ARM SA-110 (see Figure 4.8(a)) changes affect the second and fourth stage of the pipeline.

Figure 4.8 Hardware Implementation



(a) ARM processor pipeline



(b) Accessing registers with subword support

ALU instructions compute the result in `Execute` stage while the register is updated at `WriteBack` stage. A small component can thus be added in the intervening `Buffer` stage to ensure that the upper half is sign-extended. If not, the entire register is used to hold the result and the corresponding bit `B` of the register is cleared.

For memory access instructions that use a register as the destination, i.e., load instructions, register is ready at the end of `Buffer` stage and thus there is no time to perform validation. Simply clear its bit `B` and use all the bits in this case.

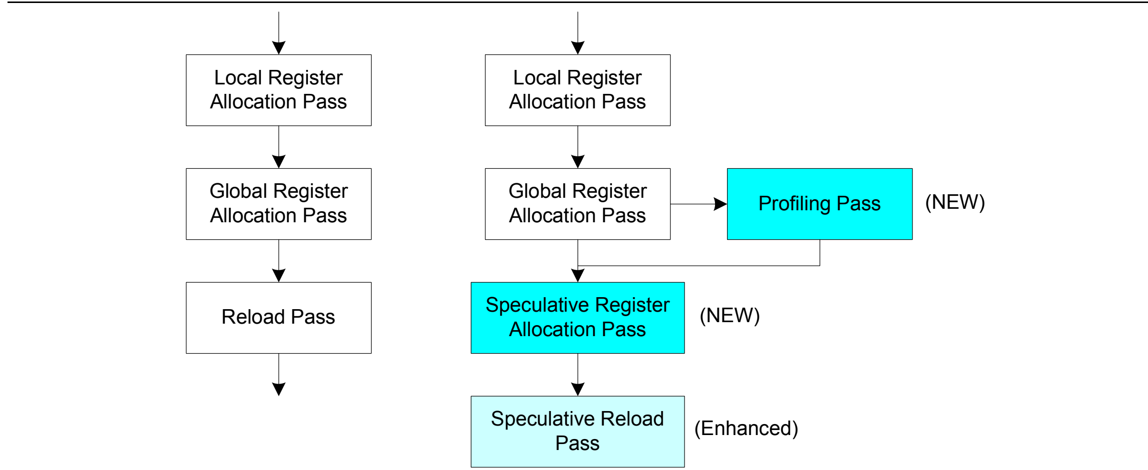
Since bit `B` of a register determines whether a 16-bit or a 32-bit value is involved in a computation, a multiplexer is needed for the upper 16 bits. Therefore in comparison to a machine without subword support, extra delay is introduced due to the multiplexer (Figure 4.8(b)). However, this delay is smaller than the delays in processors that support accesses to arbitrary bit sections such as the Infineon processor [39].

4.5 Compiler Algorithm for Speculative Subword Register Allocation

Given the hardware support designed in the preceding section, compiler support is needed to carry out speculative subword register allocation.

4.5.1 The Framework

Figure 4.9 The Framework of Speculative Subword Register Allocation.



The compiler algorithm is implemented by three passes that are integrated into `gcc` compiler. The framework of the speculative subword register allocation is shown

in Figure 4.9. The *profiling pass* collects information about how likely two variables can fit into one register and how long is the lifetime during which they coexist. Based on the above information, the new *speculative allocation pass* allocates those narrow width variables, which originally should have been spilled, to the possible upper halves of the registers. The decision is made to maximize the avoidance of spill cost for achieving high performance improvement. This pass works as a secondary global register allocation pass following the original global register allocation pass in gcc. In the *enhanced reload pass*, the compiler generates transformed code with the new instructions based on the decision made in the previous register allocation passes.

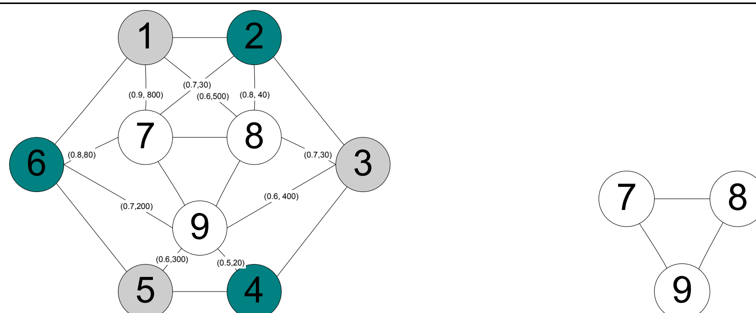
4.5.2 Priority-Based Speculative Allocation

The speculative allocation pass makes decision to allocate an originally spilled narrow width variable to the possible upper halves of the registers. This pass uses priority-based graph coloring approach to make the decision. The decision is made based upon the following profiling information: *coexisting lifetime* of variables $v1$ and $v2$ refers to the period of time during which $v1$ and $v2$ are both alive during program execution; and *coalescing probability* of variables $v1$ and $v2$ refers to the percentage of the coexisting lifetime during which $v1$ and $v2$ can be coalesced. Note that during program execution one variable may be coalesced with different variables at different program points.

The speculative allocation pass makes use of two interference graphs. One is called the *Annotated Interference Graph* (AIG) and the other is called the *Residual Interference Graph* (RIG).

Annotated Interference Graph (AIG). This graph is built from the interference graph after global allocation pass. At this point, some nodes are not colored since they are spilled. For each spilled node, the edges between this node and its colored neighbors are annotated with a 2-tuple (coalescing probability, coexisting life-

Figure 4.10 Interference Graphs in Speculative Subword Register Allocation



(a) Annotated Interference Graph (b) Residual Interference Graph

time). Figure 4.10(a) shows a simple example with nine variables (nodes) and two physical registers (colors). After global register allocation pass, nodes 1-6 are colored with two colors and nodes 7-9 are not colored. The edge labels are interpreted as follows. Label (0.9, 800) on edge (7,1) indicates that during 90% of 800 units of the time that variables 1 and 7 coexist, both of them are expected to require no more than 16 bits to represent and thus they can simultaneously reside in one register.

Residual Interference Graph (RIG). The RIG is a subgraph of AIG that consists of those originally spilled nodes and edges between them. RIG is not annotated. The RIG for the above example is shown in Figure 4.10(b). Note that two variables in RIG may be speculatively allocated to the same register if they do not interfere with each other.

The speculative allocation pass colors the nodes in RIG using the annotation information in AIG. After coloring a node in RIG, the colored node shares the same color as at least one of its colored neighbors in the AIG. While all colored nodes in AIG are colored in the local or global allocation passes, the colored nodes in RIG are colored in the speculative pass.

Next let us discuss in greater detail how nodes are chosen for coloring from RIG

and how colors are selected for them. This process uses priority based graph coloring algorithm. Our priority function is based on the expected net saving by speculatively assigning an originally spilled narrow width variable. The saving comes from the avoidance of reloads from memory; however, a cost of one cycle is incurred for each speculative spill load or store because there is an extra instruction in speculative load or store compared with traditional load or store. The saving is also a function of coalescing probabilities. The priority of node n, which is the estimated net savings by speculatively assigning node n to a register, is given by:

$$Priority(n) = READ(n) \times READCOST \times MCP(n) - REF(n)$$

where $READ(n)$ is the total number of reads of node n, $REF(n)$ is the total number of references (reads and writes) to node n, $READCOST$ is the number of cycles needed to finish a normal read from memory (i.e., it is the memory latency), and $MCP(n)$ is the maximum coalescing probability of n. Note that the above $Priority$ value can be negative for some nodes in which case they are not considered for speculative register assignment. Nodes with higher priority are considered before those with lower priority.

The maximum coalescing probability of n, i.e., $MCP(n)$, is determined by considering all available colors for n and finding which color is expected to result in most savings. The best choice for a color depends upon the following factors. The higher the coalescing probability of a pair of variables, the more beneficial it is to allocate them to the same register. The longer two variables co-exist, the more beneficial it is to allocate them to the same register. Based upon these two factors the maximum coalescing probability is computed. Moreover it should be noted that the physical register being speculatively assigned to a node n in RIG may have been allocated to several non-interfering virtual registers in earlier passes. The coalescing probabilities and lengths of coexistence of each of these virtual registers with n must be considered as long as a virtual register interferes with n. The following formula computes the

current maximum coalescing probability $MCP(n)$ for a node n in RIG:

$$MCP(n) = \max_{c \in C(n,RIG)} \frac{\sum_{n' \in Nb(n,AIG) \wedge Cl(n')=c} CLt(n,n') \times CPb(n,n')}{\sum_{n' \in Nb(n,AIG) \wedge Cl(n')=c} CLt(n,n')}$$

Where $C(n, RIG)$ is the set of currently available colors for node n in RIG (i.e., these colors have not been assigned to neighbors of n in RIG), $Nb(n, AIG)$ is the set of neighboring nodes of n in AIG (i.e., these nodes were colored during local or global allocation passes), $CLt(n, n')$ is the length of coexisting lifetime of nodes n and n', $CPb(n, n')$ is coalescing probability of nodes n and n' and $Cl(n')$ is the color assigned to node n'. The max function finds the maximum coalescing probability across all potential colors in $C(n, RIG)$.

4.5.3 Profiling Pass

The profiling pass is used to collect the above method coalescing probability and coexisting lifetime information.

The profiling pass is implemented by instrumenting the intermediate representation of the code. Profiling is performed after the global allocation pass when the objects of the optimization, those variables which do not get a register, have been identified. At this point the liveness information of the variables at each program points is available since data flow analysis is done before register allocation. The intermediate representation still contains virtual registers instead of physical register since register reloading pass has not been performed.

The relation between colored and non-colored nodes is considered. Whether two variables can be coalesced or not depends on the status of the variables (i.e., whether they fit in 16 bits or not). A variable read will not change the status of the variables and thus consecutive variable reads will share the same coalescing probability. Variable definitions can change the status of variables and thus change the coalescing

probability between two variables. Therefore variable definitions play an important role in the coexisting lifetime of two variables. In the priority function described earlier, the number of references has already been considered. Here, the length of the status history of two variables is used to approximate the coexisting lifetime.

During profiling, two 3-dimensional arrays $lifetime[i][j][k]$ and $count[i][j][k]$ are used. Here index i identifies the function, index j identifies the colored variable, and index k identifies the non-colored variable. The lifetime array records the length of overlap of live ranges of two variables while the count array records the duration over which the two variables are likely coalescable. The coalescing probability is computed by dividing latter by the former.

The instrumentation process is as follows. Assume that variable a is colored, variable b is non-colored, and their live ranges overlap. The instrumentation code is only inserted in the overlapping regions. For each definition of either a or b, coexisting lifetime is absolutely increased by one while coalesce count is conditionally increased by one. In practical benchmarks, every definition point is followed by instrumentation code for multiple variables. The instrumentation algorithm is quite straight forward. Counter initialization code is inserted in the entry point of main function and profiling information reporting code at the exit point of the main function.

4.5.4 Speculative Reload Pass

Our speculative reload pass is an enhanced version of the `gcc` reload pass. According to the decisions made in the previous passes, code is generated to access physical registers, access upper halves of physical registers and access memory. In summary, three categories of variables have to be handled in this pass.

1. For variables that are assigned to physical registers in local and global register allocation, the compiler replaces the virtual register names with physical register names in the intermediate representation.

2. For variables that remain in virtual registers after speculative allocation pass, the compiler allocates slots on the stack and generates spill code. For each definition or use point, a reload register is identified and spill code is generated by placing the store after the definition and load before the use.

3. For variables that are assigned in the speculative pass, the compiler needs to allocate slots on the stack and generate spill code for them. Instead of using ordinary load and store instructions, speculative load and store instructions are generated. At a definition point, the compiler identifies a reload register and temporarily stores the computed value into this register. A speculative store instruction `Ssw` is generated to speculative store the value back to the stack slot. It is followed by a speculative `Smv` instruction which speculatively moves the value from the temporary register into the upper half of the assigned register.

   At each use point, a reload register is identified and the value from the assigned upper half of the register is speculatively extracted using a speculative `Emv` instructions. It is followed by a speculative load instruction `Sld` which actually loads the value from the stack slot if at runtime the required value is not in the register.

4.6 Experimental Results

The static and speculative subword register allocation techniques have been implemented and evaluated. The proposed algorithms have been incorporated in the `gcc` compiler (version 2.95.02) for the ARM processor. The suggested architectural enhancement has been included in the Simplescalar simulator for ARM.

The experiments use selected benchmarks from embedded benchmark suites of Mediabench and Commbench. Two categories of configurations are used. One category is configured to have an L1 data cache with cache hit latency from 2 to 5 cycles and memory access latency of 64 cycles which means that in case of a successful

avoidance of spill reload the saving is at least 1 to 4 cycles. The other category is configured with no L1 data cache and 64-cycle memory access latency.

To measure the effectiveness of the proposed schemes the overall reduction of execution time of the program is evaluated. To reveal how the schemes lead to this performance improvement, statistics of algorithm-internal behaviors are also presented. For both static and speculative version, a measurement of the percentage of dynamic spill cost, which is really avoided at runtime, is provided. It is called *Avoidance Rate*. For static subword register allocation, this rate is calculated based on the total amount of dynamic stores and reloads in spill code. For speculative version, this rate is based on reloads only. The percentage of dynamic spill reloads, which are transformed into speculative reloads at run time, is measured. It is called *Speculation Rate*. Avoidance Rate is different from Speculation Rate in that the latter is the effort to eliminate the spill cost while the former is the actual success in doing so. In contrast, the spill cost is always avoided at runtime once it is avoided statically.

4.6.1 Benchmarks

Table 4.1 describes the benchmarks selected from the suites of Mediabench and Commbench. The second column is the number of residual virtual registers present in the intermediate code generated by the `gcc` compiler at the end of the global register allocation pass. Column 3 is the number of static reloads generated by the original `gcc` reload pass. Column 4 is the number of dynamic reloads that can be attributed to these static reloads at runtime.

4.6.2 Avoided Spill Cost

The effectiveness of the techniques is brought by the avoidance of the spill cost. Table 4.2 shows the avoidance of the spill cost in a configuration of non-cache, 64-cycle memory access latency. Column 2 shows the avoidance rate by static subword register

allocation. The result shows that a moderate amount, from about 3% to 10%, of spill cost is avoided by static register allocation. Column 3 and 4 respectively show the speculation rate and avoidance rate by speculative subword register allocation. The technique transforms most of the dynamic reloads into speculative reloads (from 86% to 99%). As a result, 33% to 95% of dynamic reloads are avoided. The comparison of the avoidance rate shows that speculative version has much higher avoidance rate than static subword register allocation. Although static version avoids both loads and stores while speculative version only avoids loads, we conclude that speculative subword register allocation is much more effective than static subword register allocation in avoiding spill cost.

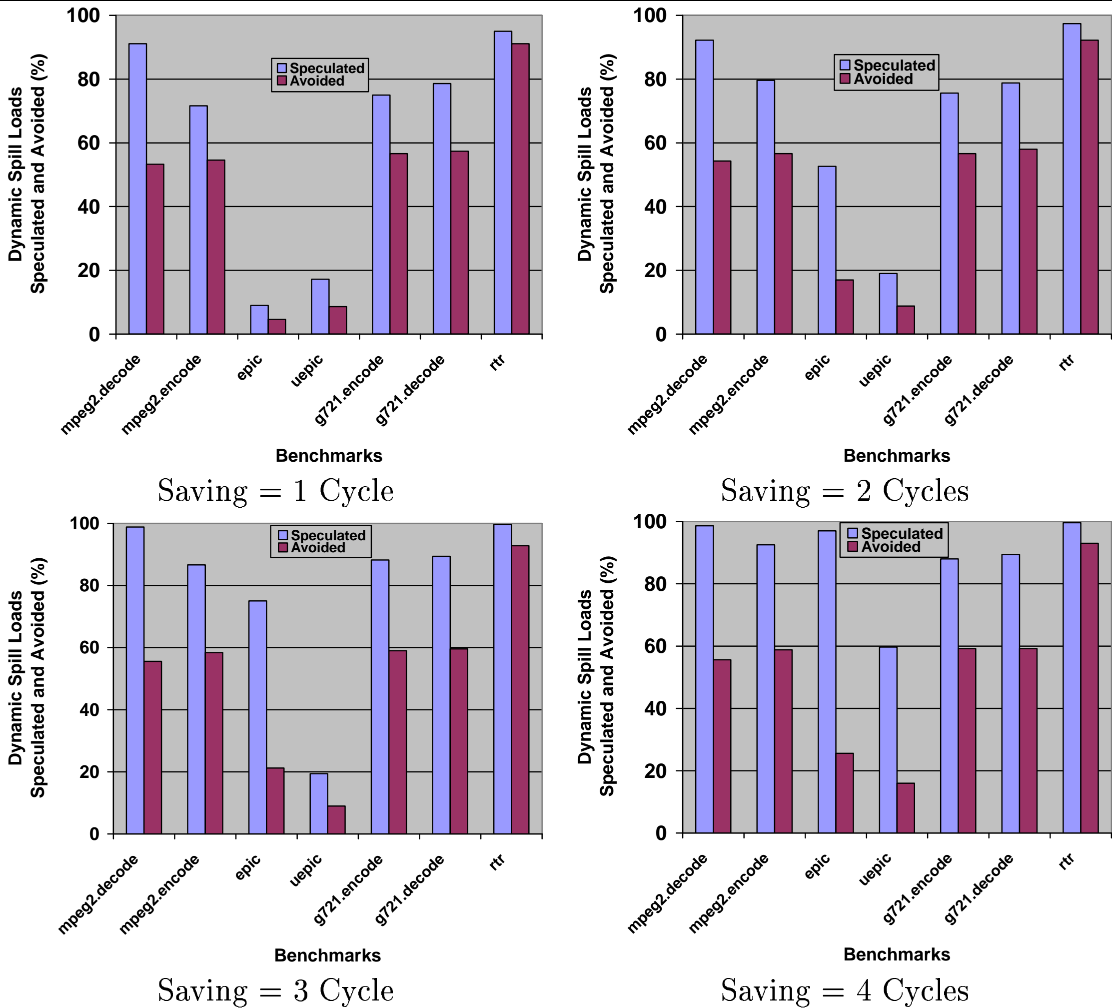
TABLE 4.1. Benchmark Characteristics.

Benchmarks	Res. Vir. Regs	Static Reloads	Dynamic Reloads
mpeg2.decode	114	483	952060
mpeg2.encode	278	1234	29738119
epic	109	527	6307287
unepic	48	225	107759
g721.encode	9	33	3543596
g721.decode	9	33	4046653
rtr	12	28	3855503

TABLE 4.2. Avoidance of Spill Cost (Non-cache)

Benchmarks	Static	Speculative	
	Avoidance Rate	Speculation Rate	Avoidance Rate
mpeg2.decode	6.27%	99.17%	58.06%
mpeg2.encode	6.45%	93.51%	61.12%
epic	3.65%	97.80%	33.56%
unepic	4.22%	86.63%	35.21%
g721.encode	7.10%	89.52%	61.07%
g721.decode	6.71%	90.33%	61.92%
rtr	10.53%	99.58%	95.26%

Figure 4.11 Avoidance of Spill Cost (Cache, Speculative)



The spill cost saved per operation differs for the two techniques. This can actually be inferred from the priority function used by them. For static technique, a change in cost may change priority value but it would not change the priority order and the range of variables which are allocated. In contrast, for a speculative technique, a change in cost does change the range of variables which are allocated. As the cost saved per operation increases, more variables have positive priority value given the same coalescing probability and thus become candidates of allocation. This is also shown by the experimental results. Figure 4.11 shows the speculation rate and

avoidance rate in a configuration with cache. The cache hit latency is from 2 to 5 cycles. This corresponds to a saving of 1 to 4 cycles. Although the saving could be more if a cache-missed reload is avoided, a conservative estimation in the cost saved by using the lower bound is used. The results show that on average, the speculation rate is 57% when the saving is one cycle and 82% when saving is four cycles. Furthermore, when the saving is one cycle, the speculative technique achieves an average avoidance rate of 41% (with upper bound of 91% and lower bound of 5%). When the saving is four cycles, avoidance rate increases to 47% (with upper bound of 93% and lower bound of 16%). Compared with the configuration without cache, a configuration with cache has lower speculation rate and avoidance rate. But the difference is not significant.

4.6.3 Performance

The final performance improvement brought by the spill cost avoided is measured. Figure 4.12 shows the percentage of reduction of execution time in cycles of the two techniques on a non-cache configuration. The results show that the static subword register allocation achieves a moderate percentage of execution time reduction from 1.51% to 3.06%. In contrast, the speculative subword register allocation achieves significant performance improvement. The speculative technique reduces the execution time by 6.35% to 14.34%. The conclusion can be reached that speculative subword register allocation is more effective to improve performance than static subword register allocation.

Again the performance improvement on a configuration with cache is measured. Table 4.3 shows the performance improvement in cycles of the speculative technique. On average, it achieves a moderate performance improvement of 4.76% when the saving is four cycles and 1.38% when the saving is one cycle. Comparing this with the results from non-cache configuration, it is found that the speculative technique

is more effective in the latter. This is due to the fact that the total spill cost of a program is much less with cache than without cache. The performance improvement of the static register allocation is measured. However, the results show negligible improvement. Thus the static technique is not as effective as speculative technique in performance improvement.

Figure 4.12 Performance Improvement in Cycles (Non-cache)

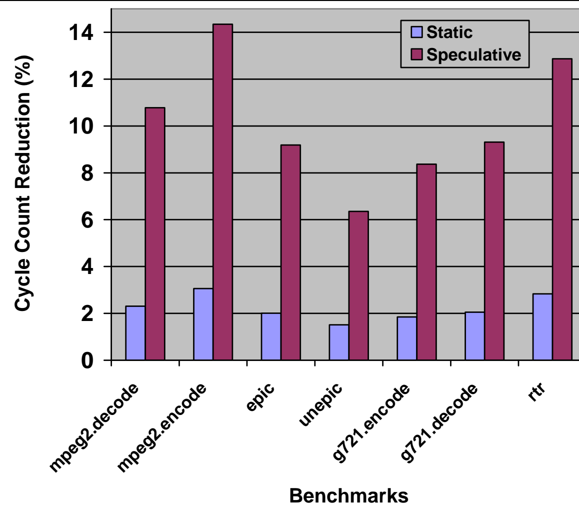


TABLE 4.3. Performance Improvement in Cycles (Speculative, Cache)

Benchmarks	Saving = 1 Cycle	Saving = 2 Cycles	Saving = 3 Cycles	Saving = 4 Cycles
mpeg2.decode	0.38%	0.65%	0.97%	1.27%
mpeg2.encode	2.53%	4.09%	5.70%	7.33%
epic	0.38%	1.31%	3.06%	5.17%
unepic	0.10%	0.21%	0.32%	0.45%
g721.encode	0.95%	1.53%	2.11%	2.71%
g721.decode	1.13%	1.82%	2.53%	3.26%
rtr	0.79%	1.20%	1.61%	2.02%

4.7 Related Work

It is well known that often applications contain significant amounts of narrow width data. The presence of narrow width data has been exploited to optimize power consumption, memory needs, and program performance. There are two types of techniques: *static* techniques rely on the compiler to identify variables that are guaranteed to be narrow width; and *dynamic* techniques are able to exploit narrow width data that is observed during program execution but cannot be necessarily identified at compile time.

Static techniques for identifying narrow width data include a number of *Bitwidth analysis*. Stephenson et al. [45] developed bitwidth analysis based upon analyzing value ranges of interesting variables. Additional bitwidth analysis include the bit section analysis proposed by Tallam and Gupta [48] and Gupta et al. [23]. A more expensive bitwise analysis was proposed by Budiu et al. [11]. The results of these analysis have been used to carry out memory coalescing for load elimination [19], storage offset assignment to reduce code size [32], silicon compilation [45], and register allocation [48].

Dynamic techniques have been developed to handle situations in which the width of the data cannot be determined to be narrow at compile time but during execution it is observed that the data values are likely to fit in half words. The presence of dynamically observed narrow width data has been exploited to develop profile guided algorithm for reducing heap allocated memory [53] and to improve data cache performance and power in [54, 51]. In contrast, in this chapter it has been shown how narrow width data can be exploited to make efficient use of registers.

Coloring-based register allocation has been studied extensively in [16, 14]. A recent work on bitwidth aware global register allocation [48] exploited statically known narrow width data by packing multiple narrow width variables into a single register. It relied upon hardware support in form of instructions that allow subsections within

registers to be addressed [31]. In contrast, this chapter presents a technique in which two variables are able to be packed in one register even if it cannot be determined at compile time that they can be stored in a single register. As a result the solution needs to be speculative in nature, i.e., while two values are expected to fit in a single register according to profile data, it is not guaranteed that they will always fit in a single register. Finally it should be pointed out that this work is orthogonal to speculative register promotion technique of [4]. While speculative register promotion allows allocation of registers to variables in presence of aliasing, this work assigns already allocated registers to additional variables when no free registers are available.

4.8 Summary

In this chapter two techniques are presented to exploit presence of narrow width data in programs to more efficiently make use of limited register resources in embedded processors. Narrow width variables otherwise spilled by a coloring allocator are statically or speculatively saved in the upper halves of the registers occupied by other narrow width variables. Register assignment is made such that two narrow width variables will reside in the same register simultaneously or expectedly simultaneously most of the time. For both techniques, a small set of new instructions are designed through which the feature of subword register assignment can be implemented without requiring significant amounts of instruction encoding space. Priority-based graph coloring register allocation algorithms are proposed to achieve maximum performance improvement. The experimental results show that the static technique avoids up to 10.53% of the spill cost (store and load) and the specultaive technique avoids from 33% to 95% of the spill cost (load). This leads to a performance improvement of up to 3.06% for static technique and 14.34% for speculative technique. The speculative technique is more effective to avoid spill cost than the static technique. Both techniques are more effective in absence of cache than in presence of cache.

Chapter 5
An Offset Assignment Method Exploiting Narrow Width Data

Chapter 2 has shown that narrow width data is common in embedded applications. An instruction set extension to ARM architecture was proposed in Chapter 3 to manipulate narrow width data directly. With such an extension, multiple distinct narrow width data can be packed into a single memory location or a register. This architectural support brings us further opportunities for developing new solutions to the memory layout problem.

The main benefit brought by the exploitation of narrow width data in memory layout problem is the shrinking of memory footprint. Memory footprint is a key factor in embedded systems. For example, in the system-on-chip design of embedded microprocessors, memory footprint is directly translated into silicon area and cost. Memory footprint can be divided into two parts. One is the data memory footprint which is occupied by the local, global, and heap data. The other is instruction memory footprint which is occupied by the instructions of the program code.

In this chapter we show that the exploitation of narrow width data can contribute to the reduction of both data and instruction memory footprint by solving the memory layout problem in a new way. First, it is obvious that packing multiple data items into a single memory location can shrink the data memory footprint. Second, a packed memory layout can help reduce the program code size in the context of solving the offset assignment problem for architectures that support autoincrement/autodecrement features associated with the indirect addressing modes. While not in the context of memory layout, the program size can be reduced by replacing packing and unpacking operations with BSX ARM instructions as shown in Chapter 3.

The autoincrement/autodecrement feature is associated with the indirect addressing modes in many embedded processors and digital signal processors (e.g., TI TMS320C25, Motorola DSP56K). This instruction set feature has received considerable attention for achieving code size reduction. Variables that are frequently accessed in sequence are placed into neighboring storage locations during storage assignment so that the autoincrement/autodecrement feature can be used and explicit address arithmetic instructions to set up the contents of the address register are avoided. The problem of finding a storage layout which maximizes the use of autoincrement/autodecrement to reduce code size using a single address register is called the *Simple Offset Assignment* (SOA) problem [8, 34].

In this chapter it is shown that the presence of subword data, as well as the support of instructions for direct manipulation of subword data, presents an opportunity for achieving effective storage offset assignment that can reduce the need for explicit address arithmetic instructions. By packing multiple subword variables into a single word, memory layout can be generated to further reduce the cost of address arithmetic in two ways. First, the need for address arithmetic instructions is reduced as variables that are packed together share the same address. Second, there are more opportunities for using autoincrement/autodecrement instructions as a variable can be adjacent to more than two variables. The *SubWord Offset Assignment* (SWOA) problem is introduced and solved using a *Path Cover with Node Coalescing* (PCwNC) formulation. Node coalescing corresponds to the packing of multiple subword variables into a single word while a path cover corresponds to the traditional placement of variables in adjacent locations. Three heuristics are presented to solve the PCwNC problem. Experimental results show that the developed techniques are effective.

The remainder of the chapter is organized as follows. Section 5.2 introduces the processor model that supports the features described above and illustrates how they can be used to reduce code size. It also formalizes the layout problem. In Section 5.3, three algorithms are developed to solve this problem. In Section 5.4 experimental

results are given to demonstrate the benefits of the approach. In Section 5.5, related work is discussed. Conclusions are given in Section 5.6.

5.1 SubWord Offset Assignment

Processor Model and Illustration. The processor model is defined and the advantages of SWOA over SOA are illustrated. Noticing that there is an opportunity to improve the storage assignment if multiple subword data can be put into a single memory location, a new processor model is proposed which integrates the autoincrement/ autodecrement feature and the ability to directly manipulate subword data.

The processor model considered is an *accumulator-based* machine where, for each instruction, one operand resides in the accumulator and the other operand resides in the memory. A specialized hardware unit, called Address Generation Unit (AGU), is used to handle memory addressing. AGU contains a set of address registers. The operand in the memory is referenced through one of the *address registers* (`AR0 AR1...`). Address register can be set up explicitly using any instruction that can write to it. Address register can also be automatically incremented or decremented by one in parallel with the memory access operation in any instruction. Explicitly setting up address register consumes processor cycles while autoincrement/autodecrement operation does not cost any extra cycles. In assembly, autoincrement of the address register `ARi` is indicated by `*(ARi)+` and autodecrement is indicated as `*(ARi)-`. In SWOA problem, only one address register will be considered. In a more general version of SWOA problem, multiple address registers can be exploited. As an example of autoincrement/autodecrement feature, consider the following instruction:

```
ADD *(AR0)+
```

This instruction adds the operand in memory pointed to by `AR0` to the accumulator and at the same time automatically increases the value in `AR0` by the size of one memory unit.

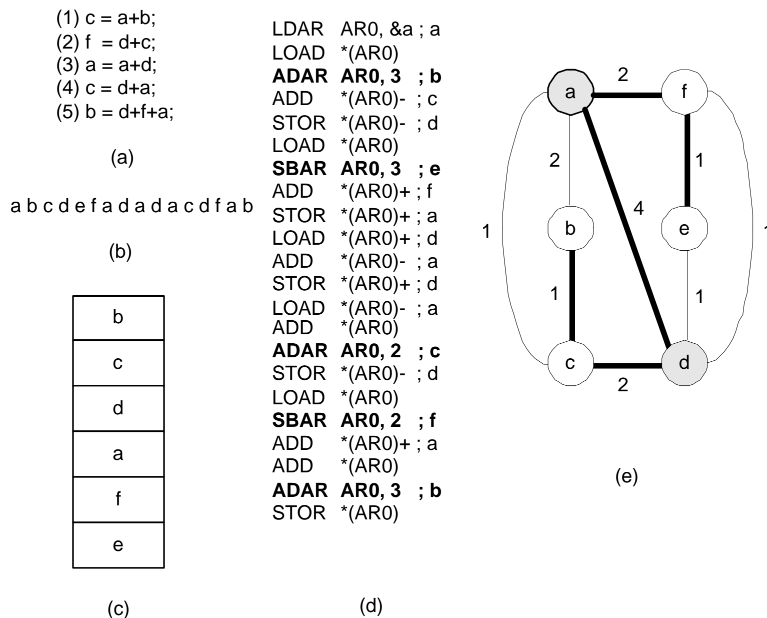
For both the accumulator and the memory operand, *bit section addressing* mode is available, i.e., the operands can be subword entities within the accumulator or the referenced memory location. A *bit section descriptor* of the form `[l,s]` is used to specify the subword entity such that `l` is the starting bit position of the subword operand and `s` is the number of bits in the subword operand. This addressing mode is quite similar to the immediate static descriptor proposed as part of a Bit Section Extension for ARM architecture in Chapter 3. The main difference is that the subword operand being read resides in the accumulator or memory while in the ARM Bit Section Extension the subword operand can come from any general purpose register. To reference subword data in memory, special logic which can read and write subword in a single memory location is added. In an instruction, either one or both of the operands can represent subword data. If only one operand is subword data, it must appropriately be extended (zero or sign) before the operation is performed. The autoincrement/autodecrement feature and subword accessing mode can be used simultaneously in an instruction. As an example, consider the instruction below:

```
ADD [16,8], *(AR0)[24,8]+
```

The above instruction performs an `ADD` over the subword operand in the memory address pointed by `AR0` starting at bit position 24 and 8 bits long and the subword operand in the accumulator starting at bit position 16 and 8 bits long. The result is stored in the accumulator and the value of `AR0` is increased to address the next word by the autoincrement operation.

To illustrate the benefits of above instructions an example is presented next. Consider the code fragment in Figure 5.1(a). Further assume that variables `a` and `d` are subword variables of length 16 bits and all other variables are 32 bits. Figure 5.1(b) shows the access sequence of the variables. Using the algorithm for solving the SOA problem proposed by Liao et al. [34], the layout shown in Figure 5.1(c) is obtained. Notice that this algorithm assigns variables a and d to separate locations. The as-

Figure 5.1 Simple Offset Assignment (SOA)

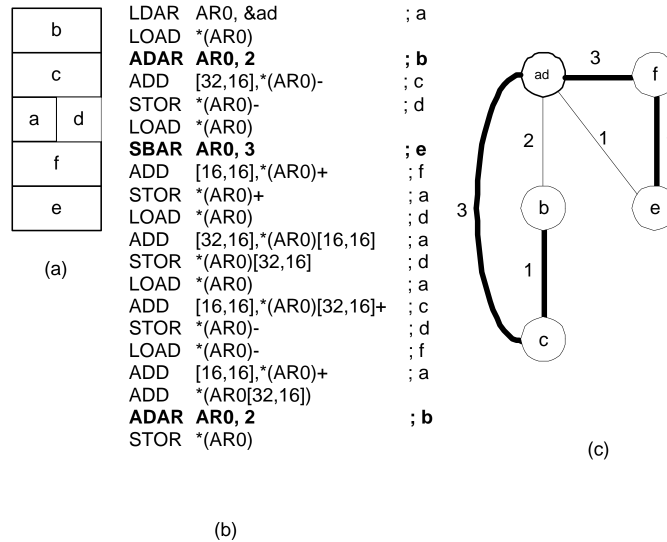


sembly code corresponding to this offset assignment is shown in Figure 5.1(d), where the instructions shown in bold type are explicit address arithmetic instructions.

Assume that the knowledge that a and d are subword variables is exploited. By packing the two variables together into a single location, the storage offset assignment will be changed as shown in Figure 5.2(a). The corresponding assembly code is shown in Figure 5.2(b) which contains fewer explicit address arithmetic instructions. The number of explicit address arithmetic instructions is reduced from 5 to 3. The packing of variables helps in two ways. First a and d can be accessed in sequence without modifying the contents of the address register as they belong to the same storage location. Second since a and d are both simultaneously made adjacent to variables c and f, it is possible to handle access sequences af and df using autoincrement and access sequences ac and dc using autodecrement.

Problem Formulation. Now let us further discuss the formulation of the layout determination problem based upon which the above layouts are derived. First let us

Figure 5.2 SubWord Offset Assignment (SWOA)



consider the formulation of SOA proposed by Liao et al. [34]. An undirected edge-weighted access graph is constructed from the access sequence where the vertices of the graph are the variables and the weights of the edges are the number of times in the static code that two variables are accessed in sequence. Figure 5.1(e) shows the access graph of this example. 16-bit variables are shown as shaded nodes while unshaded nodes represent 32 bit variables. A layout corresponds to *path cover* of the graph. The edges included in the path cover can be handled using autoincrement and autodecrement while the edges that are not covered lead to insertion of explicit address arithmetic instructions. To minimize the insertion of additional instructions, a *maximum weight path cover* should be selected. However, this problem is NP-complete and therefore a greedy heuristic was proposed by Liao et al. [34] to find the path cover. The thick edges shown in Figure 5.1(e) represent the path cover generated by the greedy algorithm. Since the sum of the weights of edges that are not covered is 5, five explicit arithmetic instructions are inserted in the generated code. The sum of the weights of edges that are not covered is also called the *cost* of the storage layout.

In order to address the SWOA problem, the above formulation is enhanced as

follows. It is allowed to apply *node coalescing* – two nodes can be coalesced into one if they together can fit into a single word. For example, the graph shown in Figure 5.2(c) is obtained by coalescing nodes for variables a and d, creating the node ad. Following coalescing, if the greedy heuristic proposed by Liao et al. [34] is applied, the path cover obtained is shown by the thick edges in Figure 5.2(c). By coalescing the edge $a - d$ is covered and the path cover further exploits the benefits of this coalescing leaving two edges uncovered with collective weight of 3. Since the cost of the layout following path covering with coalescing is 3, three explicit address arithmetic instructions are generated following SWOA.

Definition 1. (Node Coalescing) Given an access graph $G\langle V, E\rangle$, nodes $u, v \in V$ can be coalesced into uv if the following condition holds:

$$Bitwidth(u) + Bitwidth(v) \leq 32\ bits.$$

The transformed graph $G'\langle V', E'\rangle$ obtained after coalescing of u and v is obtained as follows:

V':
- $V' = V - \{u, v\} \cup \{uv\}$
- $Bitwidth(u, v) = Bitwidth(u) + Bitwidth(v)$

E':
- **for** each node $x \in V - \{u, v\}$ **do**
  - **if** $u - x, v - x \in E$ **then** $uv - x \in E'$ and
    - $weight(uv - x) = weight(u - x) + weight(v - x)$
  - **elseif** $u - x \in E$ and $v - x \notin E$ **then** $uv - x \in E'$ and
    - $weight(uv - x) = weight(u - x)$
  - **elseif** $v - x \in E$ and $u - x \notin E$ **then** $uv - x \in E'$ and
    - $weight(uv - x) = weight(v - x)$
  - **elseif** $x - y \in E$ and $\{x, y\} \cap \{u, v\} = \phi$ **then**
    - $x - y \in E'$
  - **endif**
- **endfor**

From the above discussion it is clear that it needs to solve the *Path Cover with Node Coalescing* (PCwNC) problem where coalescing operations can be applied to the

graph and a path cover needs to be found for the transformed graph such that the cost of the resulting storage layout is the minimum. While, in the example shown, only two variables could be coalesced together as each was 16 bits, in general more than two variables may be coalesced into a single location assuming that their combined bitwidths do not exceed 32 bits. While a single coalescing operation involves exactly two nodes, by iteratively performing coalescing, more than two variables can be made to share the same location.

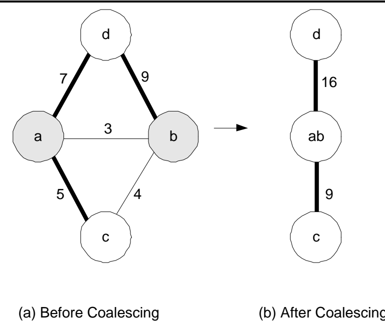
The node coalescing operation is formally defined in Definition 1. The definition specifies the precondition under which nodes u and v can be coalesced and replaced by node uv. Furthermore, edge weight $weight(x-y)$ is the number of times variables x and y are accessed one after another. To facilitate the checking of the precondition, with each node n, its bitwidth $Bitwidth(n)$ is associated. The graph update following a coalescing operation is also specified.

While the problem of finding the maximum weight path cover is already known to be NP-complete, providing the flexibility to perform node coalescing further complicates the design of a heuristic algorithm. The obvious approach that comes to mind is to develop a greedy prepass for performing node coalescing. The transformed graph can then be passed on to the greedy path covering heuristic proposed by Liao et al. [34]. However, there are intricate interactions between coalescing and path covering that make this approach not so attractive. In particular, coalescing is not always good. Since coalescing can result in nodes with higher degrees, it can adversely effect the ability to find a good path cover. In the remainder of this section the situations in which coalescing is beneficial and situations in which it is harmful are characterized. This analysis will guide the design of algorithms which will be presented in the subsequent section.

Positive and Negative Coalescence. Consider the example in Figure 5.3(a). By applying the SOA algorithm proposed by Liao et al. [34], the path cover obtained includes

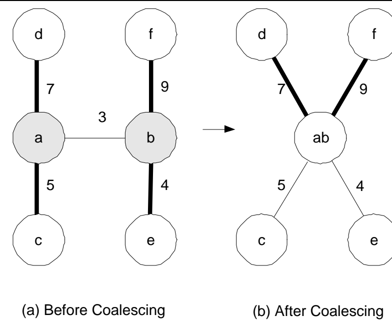
edges $b-d$, $a-d$, and $a-c$ shown as thick edges in the graph. Edges $a-b$ and $b-c$ with a collective cost of 7 are not covered. If a and b are coalesced which are both 16 bit variables shown as shaded nodes in the graph, all edges are covered in the resulting graph. Thus the impact of coalescing is positive. On the other hand if the example in Figure 5.4 is considered something completely different happens. Before coalescing, edges $b-f$, $a-d$, $a-c$, and $b-e$ are covered by the path cover and as a result only one edge, $a-b$ with cost of 3, is not covered. However, after shaded nodes of 16 bit variables a and b are coalesced, two edges, $c-ab$ and $e-ab$ with collective cost of 9, are not be covered. Thus, node coalescing has a negative impact. A quick look at these examples reveals the key difference between them. In Figure 5.3 the nodes a and b both have degree of 3 while the degree of resulting node ab is 2 which is lower. On the other hand, in Figure 5.4, the nodes a and b both have a degree of 3 while the degree of ab is 4 which is higher.

Figure 5.3 Positive Coalescence



(a) Before Coalescing (b) After Coalescing

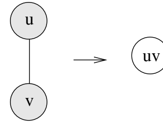
Figure 5.4 Negative Coalescence



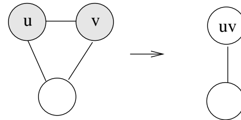
(a) Before Coalescing (b) After Coalescing

In order to characterize the possible positive or negative impact of node coalescing, it is useful to consider the impact of coalescing two nodes on the edges connected to these nodes by dividing them into the following three categories.

1. *Edge between coalesced nodes.* This edge will be removed by node coalescing and thus this is a positive effect of node coalescing. The bigger the weight of the edge between a pair of coalesced nodes, the greater is the benefit of eliminating the edge through coalescing. However, this is not the only effect of coalescing. Other effects of coalescing are described next.

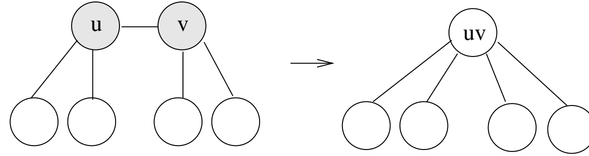


2. *Edges between coalesced nodes and their common neighbors.* By node coalescing, the two edges between nodes being coalesced and their common neighbor will become merged into one edge. The weight of the new edge is the sum of the two original edges. Since one edge each connected to the original nodes is replaced by exactly one edge connected to the coalesced node, such edges are not a contributing factor to the formation of a coalesced node with a higher degree than the degrees of the nodes before coalescing. Thus, these edges are not considered harmful.



3. *Edges between coalesced nodes and their non-common neighbors.* It is the presence of these edges that results in the creation of a coalesced node which has a degree higher than the degrees of the original nodes. Since only two of the neighbors of a node can ever be in the path cover, the amount by which the degree of a coalesced node exceeds two is the number of edges connected to the

coalesced node that will not be covered. However, if the degree of the coalesced node is not greater than two, then these edges are not harmful.



The above qualitative analysis sheds light on the factors which should influence coalescing decisions. The positive factors should be maximized and the negative factors should be avoided while doing node coalescing. In the next section, algorithms are developed to incorporate coalescing into path covering.

5.2 Algorithms

Three algorithms have been developed to solve the PCwNC problem. The discussion below outlines the motivation and differences in the three algorithms.

Alg. I - Cover First, Coalesce Later. This algorithm first uses the greedy algorithm proposed by Liao et al. [34] to find a path cover and then performs coalescing to handle as many of the uncovered edges as possible. The advantage of this approach is that coalescing will only improve the solution produced by the algorithm proposed by Liao et al. [34]. Each coalescing operation will eliminate at least one uncovered edge. The drawback of this approach is that it is conservative in that, while it completely avoids the negative impact of coalescing, it does not fully exploit the positive potential of coalescing.

Alg. II - Coalesce First, Cover Later. This algorithm first carries out all the coalescing operations, and then applies the algorithm proposed by Liao et al. [34] for finding the path cover for the transformed graph. The advantage of this approach is that when the path cover is selected, it can take advantage of the

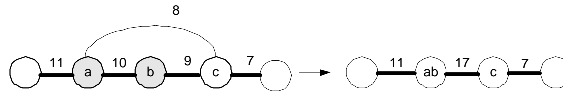
coalescing decisions made earlier. In contrast to the first algorithm, this algorithm is more aggressive in carrying out coalescing and thus it can potentially lead to coverage of more edges.

Alg. III - Integrated Covering and Coalescing. There is an inherent interaction between coalescing decisions and path covering decisions. If path cover is computed first, as is the case in Alg. I, then the knowledge of this cover is exploited in carrying out node coalescing decisions. If coalescing is performed first, as is the case in Alg. II, then the knowledge of these decisions is exploited during path covering decisions. This algorithm interleaves covering and coalescing decisions so that both types of decisions have an opportunity to influence each other. This integrated algorithm strikes a balance between the two algorithms. It is observed that Alg. I suffers when it performs too little coalescing and Alg. II suffers when it performs too much coalescing. Alg. III attempts to perform just the right amount of coalescing.

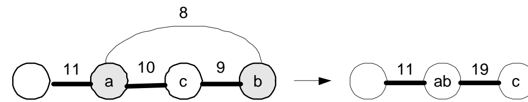
5.2.1 Alg. I - Cover First, Coalesce Later

As the name implies, this algorithm first finds a path cover using the algorithm proposed by Liao et al. [34] and then it carries out node coalescing to merge as many of the uncovered edges as possible into the path cover. This approach guarantees that coalescing only improves upon the solution provided by the initial path cover. However, in this approach, the path cover cannot be arbitrarily altered and thus coalescing decisions cannot be used to select a different and perhaps superior path cover. The key issue to address is the identification of coalescing opportunities that eliminate an uncovered edge. Four distinct situations have been identified in which coalescing is possible assuming that the nodes being coalesced satisfy the bitwidth precondition. These four cases are as follows:

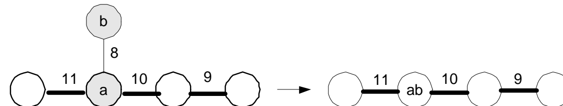
Case 1: In this case both of the nodes that are connected by an uncovered edge lie in the same covered path and there is only one node in between them. If any one of the nodes' from the uncovered edge can be coalesced with the intervening node, the uncovered edge will merge into the path cover and thus would be effectively covered. This situation is illustrated in the figure below. The shaded nodes in the figure represent nodes that can be coalesced.



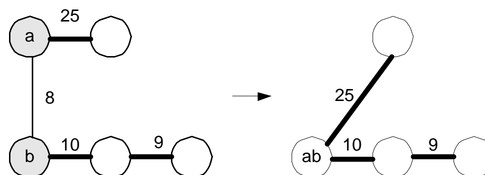
Case 2: In this case also both of the nodes that are connected by an uncovered edge belong to the same covered path and there is only one node in between them. However, in addition one of the endpoints of the uncovered edge is at the end of a path cover. In this case the two nodes connecting the uncovered edge may be coalesced together as shown in the figure below.



Case 3: In this case, for an uncovered edge, if one node is left by itself and the other node is part of a path cover, then the two nodes can be coalesced as shown below.



Case 4: In the fourth case, for an uncovered edge, if both of the nodes are the endpoints of a covered path, then the nodes can be coalesced. Actually this will connect the two covered subpaths. This situation is illustrated below.



However, it should be noted that the above situation cannot occur when the initial path cover is examined. This is because the algorithm proposed by Liao et al. [34] would have simply included this uncovered edge into the path cover. But this situation can be created after application of other coalescing transformations. In the figure below, after coalescing of nodes c and d according to case 2, an opportunity for coalescing nodes a and b according to case 4 arises.

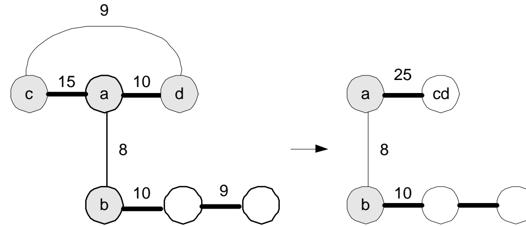
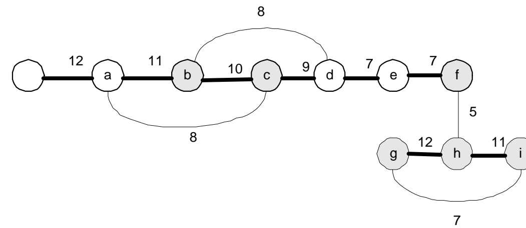
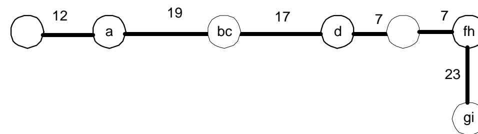


Figure 5.5 An Example of Alg. I



(a) Original graph and its cover.



(b) Coalesced graph and its cover.

An example of the application of a sequence of above coalescing transformations is shown in Figure 5.5. Figure 5.5(a) gives the original access graph and the thick edges form the path cover identified using the heuristic proposed by Liao et al. [34]. The cost of this cover is 28. In this graph the shaded nodes correspond to 16 bit variables and unshaded nodes are assumed to correspond to 32 bit variables. Using case 1, nodes a and b are coalesced. By applying case 2, nodes c and d are coalesced. By applying

Figure 5.6: Alg. I - Cover First, Coalesce Later.

```
input  : A set N of nodes with bitwidth information.
         An access sequence L for these nodes.
output: A subword offset assignment A.
G(V,E) ← AccessGraph(L,N);
NC ← ∅ ;
PC(V′,E′) ← SolveSOA(G);
F ← list of edges in E − E′ in descending order of weight;
while F ≠ ∅ do
    e(n1,n2) ← first edge in F;
    F ← F − {e};
    s ← DetectCase(e, G);
    switch s do
        case 1
            n3 ← the node between n1 and n2;
            if Bitwidth(n1)+Bitwidth(n3) ≤ 32 bits then
                G ← Coalesce((n1,n3),G);
            else
                G ← Coalesce((n2,n3),G);
            end
            update PC;
            update F;
            record this coalescing in log NC;
            break;
        case 2,3 or 4
            G ← Coalesce((n1,n2),G);
            update PC;
            update F;
            record this coalescing in log NC;
            break;
        otherwise
            break;
    end
end
A ← build SWOA from PC and NC;
return A;
```

case 4, nodes e and f are coalesced. The resulting graph is shown in Figure 5.5(b). The cost of this cover is reduced to 0 as all edges have either disappeared due to coalescing or covered by the path cover.

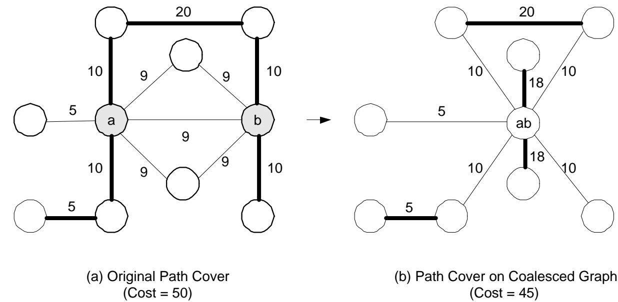
Figure 5.6 summarizes this algorithm. First, a path cover PC is computed as a normal SOA solution of the access graph. The uncovered edges are sorted in decreasing order of their weights and then processed one by one. In each step an attempt is made to handle the uncovered edge using coalescing operations defined by the four cases discussed. If coalescing is successful at any step, bookkeeping actions are performed to update the path cover and the access graph. The path cover PC and the log of coalescing actions NC together define the SWOA solution A.

5.2.2 Alg. II - Coalesce First, Cover Later

The drawback of Alg. I is that it sometimes does too little coalescing, i.e., it fails to carry out coalescing that can reduce the cost of SWOA further. Consider the access graph shown in Figure 5.7(a). As before, the shaded nodes can be coalesced together. If we consider the four cases under which Alg. I performs coalescing, none of the four cases are applicable to nodes a and b in the Figure 5.7(a). Thus, Alg. I terminates with the path cover shown by thick edges in Figure 5.7(a). On the other hand assume that the two nodes given the access graph shown in Figure 5.7(b) are coalesced. The path cover for this transformed graph is shown by the thick edges. The cost of SWOA is reduced from 50 without coalescing to 45 with coalescing.

The example in Figure 5.7 shows the limitations of the Alg. I. The path cover is found first and only node coalescings that preserve the path cover are applied. In the example shown above the node coalescing performed does not preserve the path cover of Figure 5.7(a) and thus after coalescing a fresh path cover was computed. This path cover had a lower cost. This observation suggests another approach which forms the basis of the second algorithm. Aggressive node coalescing is carried out first, and

Figure 5.7 Motivating Example of Alg. II



(a) Original Path Cover (Cost = 50)

(b) Path Cover on Coalesced Graph (Cost = 45)

after all coalescings have been performed, the path cover is computed. Thus, during coalescing, there is no path cover implied restrictions on coalescing.

While it is clear that we should use the coalesce first and cover later policy, a method must be devised for deciding when two nodes should be coalesced. It is impossible to simply exploit all opportunities for coalescing because as discussed in Section 5.1, coalescing can sometimes be harmful as it may create high degree nodes. To evaluate a particular opportunity for coalescing, a notion of *locally beneficial* coalescing is formulated as follows. Given nodes u and v which are candidates for coalescing, consider the *local subgraph* of the access sequence graph such that nodes in this subgraph are u, v, and all nodes connected by an edge to u and/or v while the edges in the subgraph are all edges emanating from nodes u and v. A cover is found for the local subgraph and then nodes u and v in the local subgraph are coalesced and a cover for this transformed subgraph is found. If the cost of the cover of the transformed local subgraph is lower than the cost of the untransformed local subgraph, then the coalescing is considered to be *locally beneficial.* If a coalescing opportunity is found to be locally beneficial, the two nodes involved are coalesced; otherwise the algorithm does not take advantage of the coalescing opportunity.

It should be noted that the covers of the transformed and untransformed local subgraph that are found above are merely used to make the coalescing decision. No covering decision is made. The above process is applied repeatedly to carry out

as much coalescing as possible. Once no more opportunities of locally beneficial coalescing exist, the path cover for the transformed graph is computed using the algorithm proposed by Liao et al. [34].

Figure 5.8 Local subgraph: Path Cover before and after Coalescing

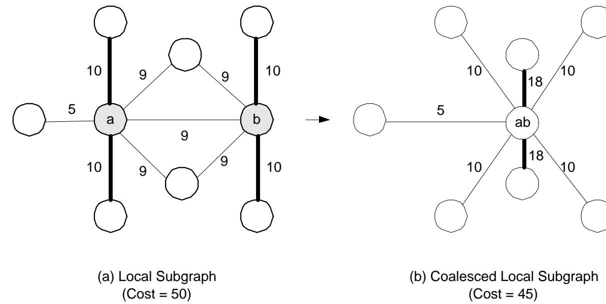


Figure 5.8(a) shows the local subgraph corresponding to nodes a and b of Figure 5.7(a). The cost of the cover for this local subgraph is 50. Figure 5.8(b) shows the resulted subgraph after coalescing nodes a and b. The cost of the cover for this subgraph is 45. Therefore, this coalescing opportunity is locally beneficial. Following coalescing, the access graph obtained is shown in Figure 5.7(b). By applying the algorithm proposed by Liao et al. [34], its cover shown in Figure 5.7(b) is computed. In the remainder of this section, the notion of *locally beneficial* coalescing is formally defined. The pseudo code for this algorithm is presented.

Definition 2. (Local Subgraph) Given coalescing candidate nodes u and v in access sequence graph $G\langle V, E\rangle$. The *local subgraph* for u and v, denoted as $LG(u,v) = \langle V', E'\rangle$, is defined as:
$V' = \{u, v\} \cup \{n :\ n \in V\ and\ n - u\ or\ n - v\ \in\ E\}$
and $E' = \{a - b :\ a - b \in E\ and\ \{a, b\} \cap \{u, v\} \neq \phi\}$.

Definition 3. (Locally Beneficial Coalescing) Given the local subgraph $LG(u, v)$, if the cost of path cover for LG(u,v) (also referred to as the *precost*), is higher than the cost of path cover for local subgraph obtained after coalescing u and v (also referred to as the *postcost*), then coalescing of u and v is said to be *locally beneficial.*

In Figure 5.9 the pseudo-code of this algorithm is given. First, all those edges that connect nodes that can be coalesced are identified. Then we go through these edges one by one and apply the *locally beneficial* test to them. In this process some edges may be collapsed through coalescing while other may be left as is. Once coalescing is over, the algorithm proposed by Liao et al. [34] is applied to find the path cover for the resulting graph giving us the SWOA solution.

Figure 5.9: Alg. II - Coalesce First, Cover Later.

```
input  : A set N of nodes with bitwidth information.
         An access sequence L for these nodes.
output: A subword offset assignment A.
G(V,E) ← AccessGraph(L,N) ;
F ← ∅ ;
NC ← ∅ ;
for all edges e(n1,n2) ∈ V do
    if Bitwidth(n1)+Bitwidth(n2) ≤ 32 bits then
        F ← F ∪ {e} ;
    end
end
sort the edges in F in descending order of weight;
while F ≠ ∅ do
    e(n1,n2) ← first edge from F ;
    G'(V',E') ← LocalSubgraph(e,G) ;
    precost ← ComputePreCost(e,G') ;
    G''(V'',E'') ← Coalesce (e,G') ;
    postcost ← ComputePostCost(G'') ;
    if precost − postcost > 0 then
        G ← Coalesce(e, G);
        record the coalescing information in the log NC;
        update F;
    else
        F ← F − {e} ;
    end
end
PC ← SolveSOA(G);
A ← build SWOA from PC and NC;
return A;
```

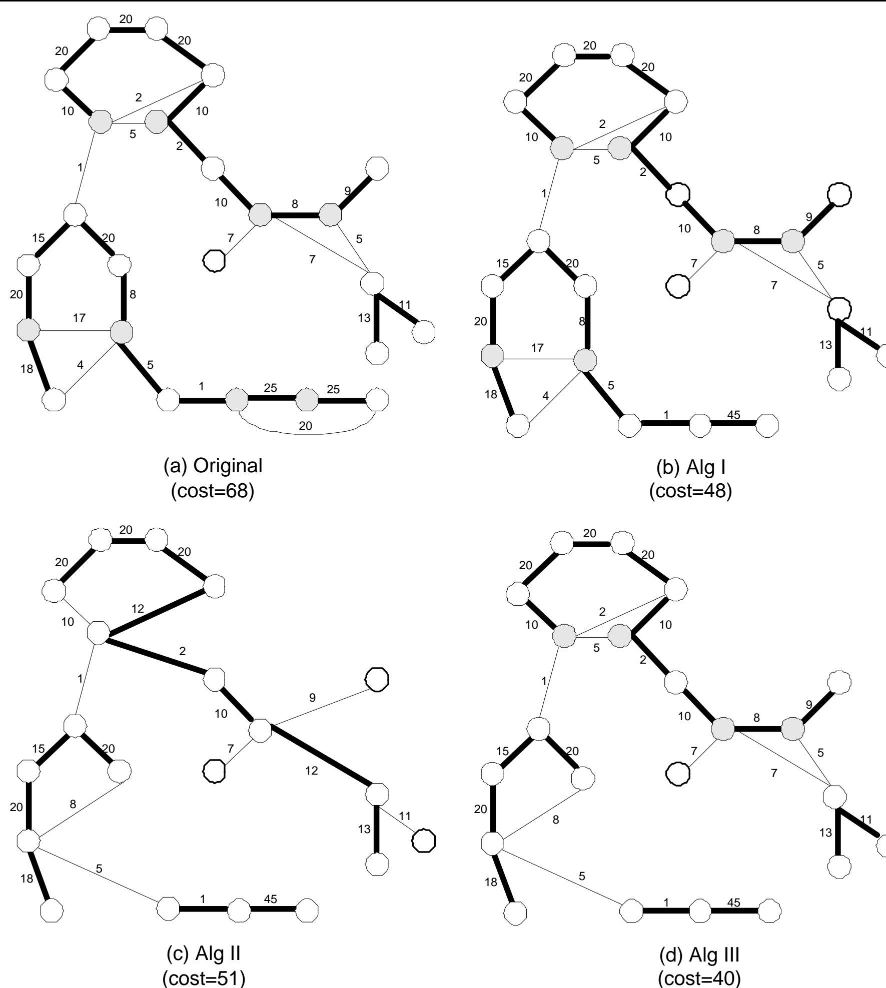
5.2.3 Alg. III - Integrated Covering and Coalescing

In the preceding sections it has been shown that while conservative coalescing performed by Alg. I can sometimes lead to inferior solutions, Alg. II performs coalescing more aggressively to achieve better performance. Next it is shown that in some situations Alg. II may perform too much coalescing leading to a worse solution than Alg. I. Consider the example shown in Figure 5.10. Solution generated by the algorithm proposed by Liao et al. [34], which does not coalesce at all, leads to the path cover with a cost of 68 as shown in Figure 5.10(a). There are four coalescing opportunities in this graph. Alg. I exploits one of these opportunities leading to the solution in Figure 5.10(b) which has a cost of 48. Alg. II exploits all four coalescing opportunities leading to the solution in Figure 5.10(c) which has a cost of 51. In other words, Alg. II performs too much coalescing in this case.

In some situations Alg. I performs better than Alg. II while in other situations Alg. II performs better than Alg. I. In fact, looking further into the above example, there is another solution that is better than the solutions produced by Alg. I and Alg. II. This solution takes advantage of two of the four coalescing opportunities as shown in Figure 5.10(d) leading to a solution with a cost of only 40. In other words, sometimes Alg. I performs too little coalescing while Alg. II performs too much coalescing. To get the best result it needs to strike a balance between the two algorithms. The real reason for the observed behaviors of Alg. I and Alg. II is as follows. There is an inherent interaction between coalescing decisions and path covering decisions. If node coalescing is performed after the path cover has been computed, as is the case in Alg. I, then the knowledge of coalescing decisions cannot influence the path covering decisions. If path covering is carried out after coalescing has been performed, as is the case in Alg. II, then the knowledge of the path covering decisions cannot influence coalescing decisions. If node coalescing and path covering decisions influence each other, an integrated algorithm is needed that interleaves

covering and coalescing decisions. Our Alg. III is one such algorithm which produces the result of Figure 5.10(d).

Figure 5.10 Right Amount of Coalescing



The integrated algorithm prepares a sorted list of edges in the decreasing order of the edge weights. Then it examines the edges one by one, where in each step, the algorithm determines whether coalescing is possible and desirable. Coalescing is possible if the combined bitwidth condition is met and coalescing will maintain the integrity of the partial cover. When coalescing is possible, it is also desirable if it satisfies the *globally beneficial* criteria described later. If the edge is not handled

through coalescing, then an attempt is made to include it into the path cover. Note that when an attempt is made to include the current edge into the partial cover, success is not guaranteed. This is essentially due to the same reason that sometimes the greedy algorithm proposed by Liao et al. [34] is not able to include an edge in the partial cover.

From the above discussion it is clear that at any point in time a partial cover exists. A coalescing decision is made based upon prior path covering decisions and a path covering decision exploits the knowledge of prior coalescing decisions. In this way it is able to tightly integrate coalescing and covering.

Definition 4. (Global Subgraph) Given a pair of coalescing candidate nodes u and v in access sequence graph $G\langle V, E\rangle$. The *global subgraph* for u and v, denoted as $GG(u, v) = \langle V', E'\rangle$, is the union of the *local subgraph* $LG(u, v)$ and the current *partial cover*.

Definition 5. (Globally Beneficial Coalescing) Given the global subgraph $GG(u, v)$, if the cost of path cover for GG(u,v) (also referred to as the *precost*), is higher than the cost of path cover for the global subgraph obtained after coalescing u and v (also referred to as the *postcost*), then coalescing of u and v is said to be *globally beneficial*. The path covers considered in above analysis must be consistent with the current partial cover.

The notion of *globally beneficial* coalescing is an extension of *locally beneficial*. Given a pair of nodes coalesceable with the current partial cover, we identify the *global subgraph* as the union of the *local subgraph* and the *partial cover* which is the cover that has already been built. The complete path cover of this subgraph is found by extending the current partial cover that is a part of it and then determining the cost of the cover and referring to the cost as the *precost*. Next the nodes are coalesced and a complete cover is found for the transformed global subgraph by extending the current partial cover that is a part of it. The cost of this cover is determined and

called the *postcost*. If the *postcost* is less than the *precost*, coalescing is performed.

Let us revisit the example of Figure 5.10. Alg. III does produce the result shown in Figure 5.10(d). There are four places in the graph where coalescing could be performed. Figure 5.11 shows the four cases. In Figure 5.11 only the relevant parts of the global subgraphs are shown both before and after coalescing and their corresponding cover costs are given. The graphs contain three types of edges: thick dark edges that are part of the current partial cover; thick dashed edges which were included to form the complete cover for the global subgraph; and thin edges that are not part of the complete cover of the global subgraph. The thin edges contribute to the precost and postcost corresponding to the two covers. In the first two cases, coalescing is found to be globally beneficial. In the third case coalescing cannot be carried out because it would require a change in the current partial cover. In the final case coalescing does not yield any benefit. Therefore the first two coalescing opportunities are exploited while the other two opportunities are ignored. The results of the Alg. III correspond to the solution shown earlier in Figure 5.10(d).

Figure 5.11 Integrated Coalescing

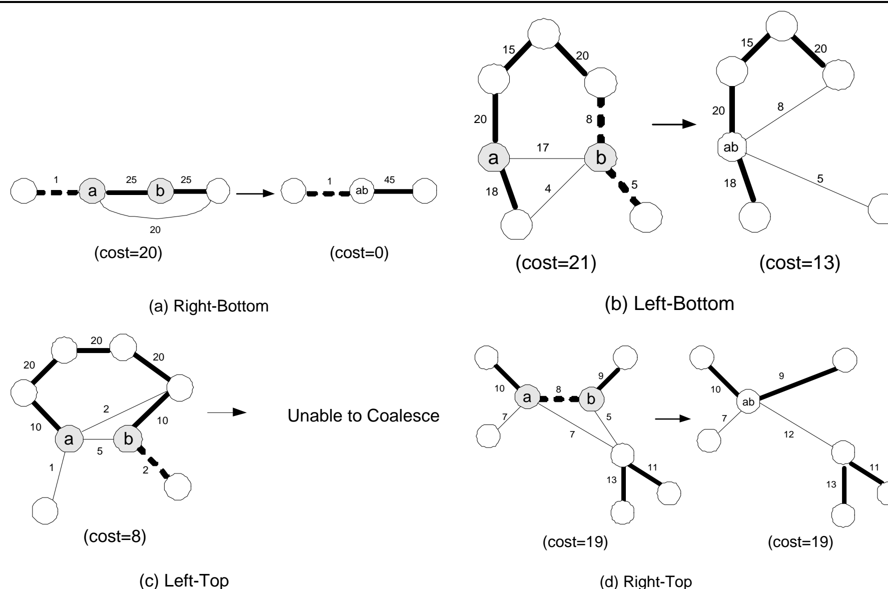


Figure 5.12 is the pseudo-code of this algorithm. First, the partial cover is set to be empty. Then run through the edges in the access graph in weight-descending order. First coalescing is attempted to handle the edge. In order for coalescing to occur, three conditions must be true: the bitwidths must not exceed 32 bits; the current partial cover should be maintained; and finally coalescing should be globally beneficial. If any one of these conditions is not satisfied, the edge cannot be coalesced. Thus, an attempt is made to incorporate it into the partial cover by checking the same conditions that were formulated by Liao et al. [34], i.e., no loop is formed in the partial cover after including the edge and degree of no node exceeds 2 after the edge is included in the partial cover. If neither coalescing nor covering succeeds, the edge is left uncovered.

5.2.4 Further Extensions

While the focus of the work in this chapter is on the simple offset assignment problem, the approach can be extended to consider the global offset assignment (GOA) problem. The SWOA can be easily used in conjunction with the variable partitioning strategy described in [34] where two variables with largest cost are selected and placed in separate partitions. However, it is observed that given the bitwidth information of each variable, a better variable partitioning strategy can be devised by taking the bitwidth information into consideration. For example, if there is a pair of nodes in the access graph that can possibly be coalesced, one should avoid placing them into separate partitions. This is because only by placing them in the same partition the possibility of coalescing is left open. In other words a partitioning strategy should also consider interactions with coalescing and covering.

Access sequence reordering optimization, either by code scheduling or algebraic transformation on expression trees, can also be combined with subword offset assignment. One must consider that different access sequences may lead to different access

Figure 5.12: Alg. III - Integrated Covering and Coalescing

```
input  : A set N of nodes with bitwidth information.
         An access sequence L for these nodes.
output: A subword offset assignment A.
G(V,E) ← AccessGraph(L,N);
NC ← ∅ ;
PC(V',E') : V' ← V, E' ← ∅;
F ← list of edges in E in descending order of weight;
while F ≠ ∅ do
    e(n1,n2) ← first edge in F;
    F ← F − {e};
    if Bitwidth(n1)+Bitwidth(n2) ≤ 32 bits then
        if n1 and n2 can be coalesced then
            G1(V1,E1) ← LocalSubgraph(e,G) ;
            G2(V2,E2) : V2 ← V', E2 ← E' ∪ E1;
            precost ← ComputePreCost(e,G2);
            G3(V3,E3) ← Coalesce (e,G2) ;
            postcost ← ComputePostCost(G3);
            if precost − postcost > 0 then
                G ← Coalesce(e, G);
                update PC; update F;
                Record this coalescing information in log NC;
            else
                if e can be covered then
                    E' ← E' ∪ {e};
                end
            end
        end
    else
        if e can be covered then
            E' ← E' ∪ {e};
        end
    end
end
A ← build SWOA from PC and NC;
return A;
```

graphs with different number of node coalescings. Intuitively, reordering should arrange variables that can be coalesced such that they are accessed in sequence so that the cost of edges corresponding to them can be removed by coalescing.

5.3 Experimental Results

The tool used for experimentation is the Motorola's gcc-based DSP56000 C compiler with source code available in the public domain. The three algorithms and the greedy algorithm proposed by Liao et al. [34] have been implemented in order to evaluate this work. The latter algorithm is used as the baseline algorithm to measure the effectiveness of the newly proposed algorithms. Our implementations focus on local variables and optimize stack frame memory accesses. To implement the algorithms the bitwidth of each variable is obtained by the bitwidth analysis techniques developed in the prior work [47]. The applications considered were taken from the Mediabench [28] (`adpcm`, `g721`, and `mpeg2`) and Bitwise Project at MIT (`softfloat`) [45].

In general, a good storage layout has two positive effects: it reduces code size and it also improves performance. Two experiments are carried out. In the first experiment higher priority is given to reducing code size and lower priority is given to improving performance while in the second experiment the priorities are reversed. In the first experiment access graphs as described in the discussion were used, i.e., the weight of an edge between two variables was the number of times two variables were observed as being accessed in sequence in the static code. This approach gives priority to reducing the static number of explicit address arithmetic instructions. In the second experiment profile information is used to determine weights as being the dynamic number of times two variables are accessed in sequence. Clearly this approach gives greater priority to reducing dynamic count of explicit address arithmetic instructions.

The reduction of the stack frame size due to variable packing is reported in the experiments. Node coalescing causes multiple subword variables to be packed into

TABLE 5.1. Reduction in Static Code Size

Benchmark / Function	SOA	Alg I		Alg II		Alg III	
Priority – Code Size / Priority – Dynamic Instruction Count	#cost	#cost	[%]	#cost	[%]	#cost	[%]
adpcm							
coder	27	27	0.0	23	14.8	21	22.2
	28	24	14.3	24	14.3	22	21.4
decoder	21	16	23.8	14	33.3	14	33.3
	23	16	30.4	14	40.9	14	40.9
g721							
encoder	7	5	28.6	5	28.6	5	28.6
	7	5	28.6	5	28.6	5	28.6
decoder	8	3	62.5	5	37.5	3	62.5
	9	5	55.6	6	33.3	5	55.6
fmult	3	1	66.7	1	66.7	1	66.7
	5	2	66.7	1	66.7	1	66.7
quantize	0	0	0.0	0	0.0	0	0.0
	0	0	0.0	0	0.0	0	0.0
update	15	8	46.7	5	66.7	5	66.7
	15	6	60.0	6	60.0	6	60.0
mpeg2							
frametotc	1	1	0.0	1	0.0	1	0.0
	1	1	0.0	1	0.0	1	0.0
formpred	10	9	10.0	9	10.0	9	10.0
	10	9	10.0	9	10.0	9	10.0
softfloat							
m32i_64	12	8	33.3	10	16.7	8	33.3
	12	8	33.3	10	16.7	8	33.3
f32_i32	14	12	14.3	11	21.4	10	28.6
	14	12	14.3	11	21.4	10	28.6
Average			26.0		26.9		32.0
			28.5		26.5		31.4

one storage location. Since the candidates for the optimization are the local variables and temporaries located in stack frame, stack frame size can be reduced. Smaller stack frame size directly leads to smaller memory footprint at runtime. This, together with static code size reduction, further reduces the memory needs of an application.

Static Code Size. Table 5.1 reports the percentage reduction in the number of the explicit address arithmetic instructions that the algorithms achieve over the algorithm proposed by Liao et al. [34]. When code size reduction is given higher priority, the algorithms achieve on average 26%, 26.9% and 32% reduction in the number of explicit address arithmetic instructions. As the results show the first two algorithms can outperform each other in different situations. However, the third algorithm significantly outperforms the other two algorithms in many cases and does at least as good as the best of the other two algorithms in other cases. Thus, we conclude that interleaving covering and coalescing decisions is the best approach. When the reduction in dynamic instruction count of explicit address arithmetic instructions is given a higher priority, it observes an average reduction of 28.5%, 26.5%, and 31.4% in the explicit address arithmetic instructions.

Stack Frame Size. Table 5.2 reports the reduction of the size of stack frame. In the experiments when code size reduction has higher priority, the algorithms achieve on average 14.5%, 22.1% and 22.7% reduction in stack frame size. In the experiments when higher priority is given to dynamic instruction count reduction, the algorithms achieve on average 16.4%, 19.5% and 21.7% reduction. The effects of this reduction can be combined with the reduction of static code size as to the memory requirement of an application. From the results, the second and third algorithms achieve more reduction in the stack frame size. This is because of the first algorithm takes a path cover first policy and reduces the number of possible node coalescing.

TABLE 5.2. Reduction in Stack Frame Size

Benchmark / Function	Priority – Code Size / Priority – Dynamic Instruction Count						
	SOA	Alg I		Alg II		Alg III	
	#size	#size	[%]	#size	[%]	#size	[%]
adpcm							
coder	12	12	0.0	10	16.7	10	16.7
	12	11	8.3	11	8.3	10	16.7
decoder	10	8	20.0	7	30.0	7	30.0
	10	8	20.0	7	30.0	7	30.0
g721							
encoder	9	7	22.2	6	33.3	5	44.4
	9	7	22.2	6	33.3	6	33.3
decoder	8	7	12.5	6	25.0	6	25.0
	8	6	25.0	7	12.5	6	25.0
fmult	6	4	33.3	3	50.0	4	33.3
	6	4	33.3	3	50.0	4	33.3
quantize	6	6	0.0	6	0.0	6	0.0
	6	6	0.0	6	0.0	6	0.0
update	14	11	21.4	9	35.7	9	35.7
	14	11	21.4	10	28.6	9	35.7
mpeg2							
frametotc	6	6	0.0	6	0.0	6	0.0
	6	6	0.0	6	0.0	6	0.0
formpred	9	8	11.1	8	11.1	8	11.1
	9	8	11.1	8	11.1	8	11.1
softfloat							
m32_64	8	6	25.0	7	12.5	6	25.0
	8	6	25.0	7	12.5	6	25.0
f32_i32	7	6	14.3	5	28.6	5	28.6
	7	6	14.3	5	28.6	5	28.6
Average			14.5		22.1		22.7
			16.4		19.5		21.7

TABLE 5.3. Reduction in Dynamic Instruction Count

Benchmark / Function	Priority – Dynamic Instruction Count / Priority – Code Size						
	SOA	Alg I		Alg II		Alg III	
	#cost	#cost	[%]	#cost	[%]	#cost	[%]
adpcm							
coder	1677709	1489644	11.2	1489644	11.2	1363256	18.7
	2169286	2169286	0.0	1874394	13.6	1710628	21.1
decoder	1702288	1702288	0.0	1185524	30.4	1185524	30.4
	1719386	1497810	12.9	1185524	31.0	1185524	31.0
g721							
encoder	1610	1150	28.6	1150	28.6	1150	28.6
	1610	1150	28.6	1150	28.6	1150	28.6
decoder	885120	442560	50.0	590080	33.3	442560	50.0
	885120	442560	50.0	442560	50.0	442560	50.0
fmult	5528	1840	66.7	1840	66.7	1840	66.7
	3688	1840	50.1	1840	50.1	1840	50.1
quantize	0	0	0.0	0	0.0	0	0.0
	0	0	0.0	0	0.0	0	0.0
update	1770	920	48.0	920	48.0	690	61.0
	2760	1840	33.3	920	66.7	920	66.7
mpeg2							
frametotc	1	1	0.0	1	0.0	1	0.0
	1	1	0.0	1	0.0	1	0.0
formpred	54534	46296	15.1	46296	15.1	46296	15.1
	54534	46296	15.1	46296	15.1	46296	15.1
softfloat							
m32_64	12	8	33.3	10	16.7	8	33.3
	12	8	33.3	10	16.7	8	33.3
f32_i32	14	12	14.3	11	21.4	10	28.6
	14	12	14.3	11	21.4	10	28.6
Average			24.3		24.7		30.2
			21.6		26.6		29.5

Dynamic Instruction Count. Table 5.3 reports the percentage reduction in the number of dynamically executed explicit address arithmetic instructions. When reducing dynamic instruction count is given a higher priority, The algorithms achieve on average 24.3%, 24.7%, and 30.2% reduction in executed explicit address arithmetic instructions over the algorithm proposed by Liao et al. [34]. As expected, the third algorithm performs the best overall. When the reduction in static code size of explicit address arithmetic instructions is given a higher priority, it observes an average reduction of 21.6%, 26.6%, and 29.5% in the dynamic instruction count of explicit address arithmetic instructions executed by the three algorithms.

The result shows that all three algorithms achieve significant reductions in static and dynamic instruction counts over the algorithm proposed by Liao et al. [34]. The three algorithms also achieve significant reductions in stack frame size. The integrated algorithm, Alg. III, gives the best performance.

5.4 Related Work

The problem of finding a storage layout which maximizes the use of autoincrement/autodecrement to reduce code size using a single address register is called the *Simple Offset Assignment* (SOA) problem. When multiple address registers are exploited, this problem is called *General Offset Assignment* (GOA) problem. This problem was first formulated as a *Path Cover* (PC) problem by Bartley [8] and Liao et al. [34, 33]. Further extensions have since been developed by many researchers. In [30], Leupers and Marwedel improved the work by Liao et al. [34] by proposing a tie-breaking heuristic for SOA and a variable partitioning strategy for GOA to further reduce storage assignment cost. Offset assignment with varying number of address registers and autoincrement by a range of values have also been studied by Sudarsanam et al. in [46] and Leupers and David in [29]. Rao and Pande [41] proposed a method to achieve better access sequence to reduce the code size by per-

forming algebraic transformations on expression trees. Similar problem is studied by Choi and Kim using an offset assignment algorithm tightly coupling with code scheduling [15]. Kandemir et al. use both access pattern modification and memory storage reordering in [25]. *Modify registers* are also considered together with offset assignment in [29, 49]. Zhuang et al. also consider coalescing non-interfering variables into the same memory location to reduce code and data size [55].

5.5 Summary

In this chapter it was demonstrated that the combination of subword referencing and indirect addressing with autoincrement and autodecrement is effective in reducing code size and stack frame size. By allowing packing of multiple subword variables into a single memory location, storage layouts are obtained which both reduce the need for address arithmetic and increase the use of autoincrement and autodecrement when address arithmetic is needed. It has been shown how variable packing can be expressed in terms of node coalescing operations on access graph. The interactions between node coalescing and path cover selection were studied and three algorithms were developed to solve these problems together. The experimental results show that these three algorithms provide significant reductions in code size over the algorithm in Liao et al. [34]. The integrated algorithm that interleaves coalescing and covering decisions performs the best.

Chapter 6
A Study of Streaming Data

In this chapter, a study of streaming data in networking applications is performed. Section 6.1 provides a classification of memory accesses into streaming data and utility data and analyzes their differing behaviors with respect to temporal reference locality. Section 6.2 shows the presence of streaming data. Section 6.3 measures the value locality in streaming data. Section 6.4 summarizes the opportunities exposed by this study.

6.1 Streaming Data vs. Utility Data

An embedded application typically consists of a processing loop such that during each iteration of a loop a new set of input data is processed. In such an application, the memory accesses to data can be mainly classified into two categories. One category is composed of memory accesses to the object data being processed, such as network packets, audio, and video data frames. We call it *streaming data* since these data come and go and exhibit streaming nature. Because of this nature, streaming data will never be reused across loop iterations. The other category is composed of memory accesses to those application-specific data structures which the application relies on to process the streaming data, such as routing table etc. We name this kind of data *utility data*. Utility data can be reused across loop iterations because the same data structures can be used to process different streaming data units. One important observation of the relationship between streaming data and utility data is that the value of streaming data often determines which part of the utility data is accessed. Table 6.1 illustrates this relationship in selected networking benchmarks for the Intel

Table 6.1. Application Properties

Application	Utility Data	Streaming Data	Description
`IP Lookup`	Routing Table	IP Dest. Addr.	Dest. address field is used to lookup next hop information in the routing table.
`NAT Protocol`	Port Mapping Table	IP Src. Addr. TCP Src Port IP Dest. Addr. TCP Dest. Port	The fields of IP address and TCP port are used to lookup translated source port for outgoing traffic in `NAT-Out` and lookup original destination and port for incoming traffic in `NAT-In`.
`Packet Classification`	Classification Table	IP Src. Addr. TCP Src. Port IP Dest. Addr. TCP Dest. Port Protocol	The 5-tuple is used to lookup ilookup the classification table and identify the flow that the packet belongs to.

IXP1200 network processor.

Streaming data references generally do not demonstrate temporal locality. First, it is often the case that there is little temporal locality within one iteration of processing. The data-plane processing in networking applications usually does not involve very complex computation. The code is often written in a way to avoid loading the same part of streaming data unit more than once during processing. Second, streaming data cannot be reused across different loop iterations of processing. This is because different loop iterations work on different streaming data units. Even in the case that the streaming data buffer is used in a modular style and the same memory addresses are used to load streaming data repetitively, different data unit is loaded each time. All together, streaming data references do not show temporal locality.

In contrast, utility data shows high temporal locality. The same part of utility

data may be accessed across different loop iterations. This is because the value in streaming data often determines which part of utility data is used. It is often the case that the same value repetitively occurs in different streaming data units as shown by the measurements performed in next section. For example, in a flow of IP traffic, many packets headers contain the same destination fields. When IP lookup is performed, the same part of the routing table is thus accessed repeatedly. This leads to temporal locality in utility data accesses. Although within one loop iteration of processing utility data references have little locality, locality occurs across loop iterations.

The low temporal locality of streaming data leads to the degradation of overall performance for memory accesses. As shown in the next section, the percentage of memory accesses to streaming data is significant in these applications. This leads to the fact that Intel IXP1200 network processor is designed with no memory cache.

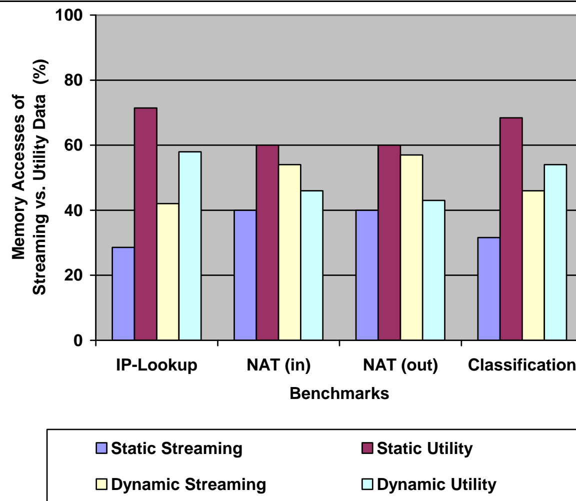
Nevertheless, the repetitive values in network packet header fields, called value locality, results in significant redundant memory loads. More importantly the entire computation within a loop iteration of processing may become redundant. In a network processor design without cache, these opportunities obviously cannot be exploited.

6.2 Measurement of Streaming Data

In this section, a quantitative measurement of the prevalence of streaming data in in selected networking application benchmarks on Intel IXP1200 network processor platform is provided.

First, Figure 6.1 shows the percentage of static and dynamic memory accesses to streaming data versus utility data. The percentage of static memory accesses to streaming data ranges from 28% to 40%. The percentage of static memory accesses to utility data ranges from 60% to 72%. The percentage of dynamic memory accesses to streaming data ranges from 42% to 57%. The percentage of dynamic memory

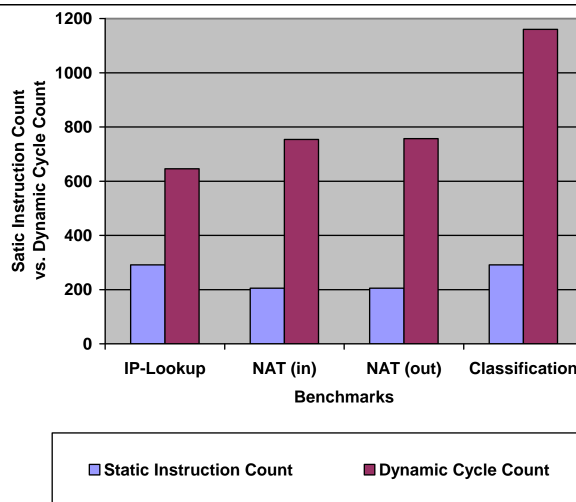
Figure 6.1 Streaming Data vs. Utility Data



accesses to utility data ranges from 43% to 58% .

It is observed that both types of memory accesses are significant. None of them is negligible. We also notice that the dynamic percentage for streaming data is higher than static one. This is probably because at runtime, the memory accesses to streaming data have to be performed to input and output object data while some of the memory accesses to utility data may or may not be performed based on the value of streaming data.

Figure 6.2 Short Lifetime



Second, it is shown that streaming data has very short lifetime. This directly leads to low temporal locality. Figure 6.2 shows the static instruction counts and the dynamic cycle counts required to process one network packet in these benchmarks. The static instruction counts range from 205 to 291. Obviously not very complex computation is required to process one unit of streaming data. The dynamic cycle counts range from 646 to 1160 cycles . Although the code contains expensive memory accesses to streaming and utility data whose latencies may be 150 cycles in Intel IXP1200 network processor, a significant part of these latencies can be hidden by the multi-core multi-thread design. The value of dynamic cycle count reveals the fact that streaming data stays in the system for a short duration.

Third, it is shown that the packet headers in network traffic show very high value locality. This gives an indication of the extent to which the network processing application perform redundant memory loads and redundant computation.

Packet Stream	Source	Num. of Packets	Num. of Distinct Destinations
1	19991129-134258-0	17045569	8920
2	20000112-111915-0	17934563	14033
3	20000117-095016-0	18433128	11746
4	20000126-205741-0	18823943	9012

TABLE 6.2. Packet Stream Characteristics

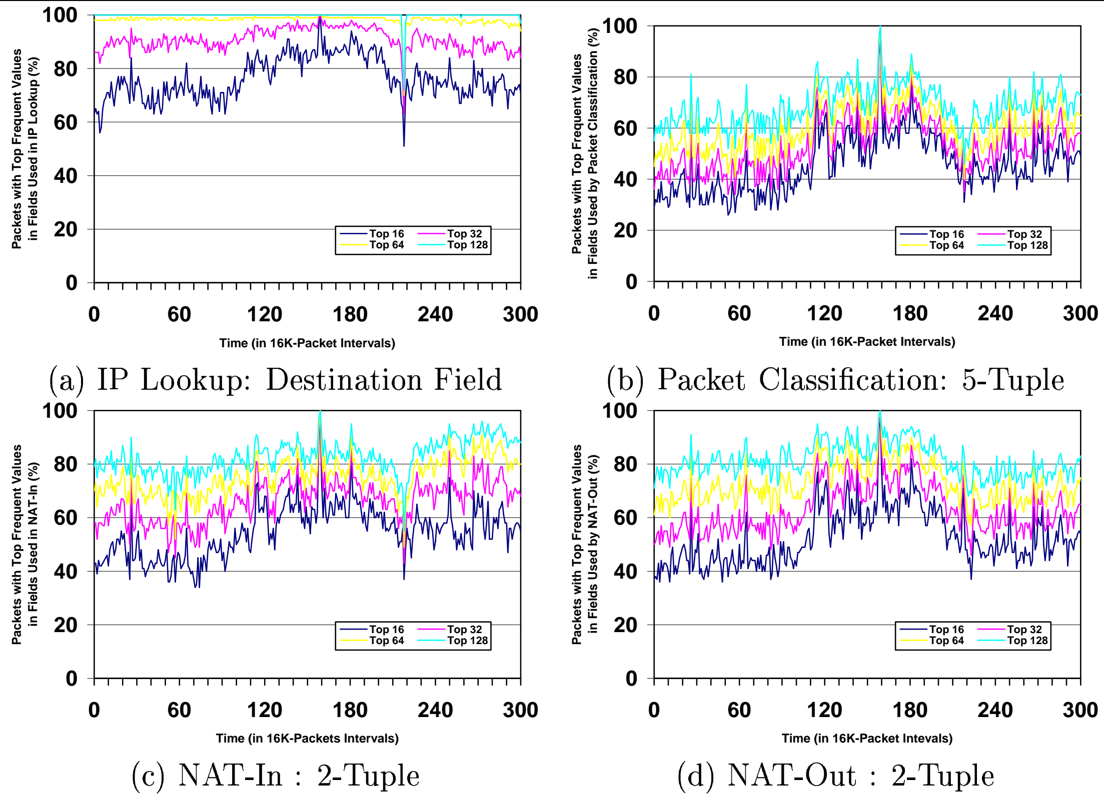
To study the value locality, traces of IP packets taken from Auckland-II Trace Archive [6] are used. Characteristics of these traces, including the total number of packets and distinct destination addresses, are given in Table 6.2. Packet traces are selected from the trace archive that had a large number of distinct destinations with respect to the number of packets in the trace.

The value locality is measured in terms of the percentage of the packets which have most frequently occurring values in the relevant n-tuple of header fields for a given application, where n is the number of relevant header fields. According to

Table 6.1, `IP Lookup` examines one field, `Packet Classification` examines five fields, and `NAT Protocol` examines four fields with two for `NAT-In` and two for `NAT-Out` respectively. To measure the value locality, the packet stream is divided into intervals of 16K consecutive packets. The top 16, 32, 64, and 128 frequent values for the above n-tuples are determined within each interval. Figure 6.3 plots the percentage of packets that have the top 16, 32, 64, and 128 frequent values for n-tuples as a function of time (in intervals) for the packet stream trace 4 in Table 6.2. Even though there are 9012 unique destination addresses, about 60% to 100% of the packets in most of the intervals have the top 128 frequent values for n-tuples This means that a significant amount of values repeatedly occur. The same experiments are performed with interval size from 1K to 1024K packets. The results are roughly the same. Experiments on other traces also show similar results. Therefore, the degree of value locality in these packet streams is quite significant.

The *unique reuse distance* between a pair of packets *P1* and *P2* with the same n-tuple value is considered next. Unique reuse distance between *P1* and *P2* is defined as the number of unique values of the n-tuple in the packets between *P1* and *P2* in the packet stream. This notion is similar to the working set size of a cache and more accurately reflects the value locality. Table 6.3 shows the average unique reuse distance for pair of packets with top 16, 32, 64 and 128 most frequently occurring n-tuple values over intervals of 16K packets for each of the packet streams. Consider the top 16 most frequent values in `IP Lookup` within packet stream `1` for example. On average these values appear repeatedly every 6.65 unique values. The statistics of unique reuse distances and frequently observed values show that a cache of a small size is enough to catch a reasonable amount of value locality.

Figure 6.3 Frequently Observed Values



(a) IP Lookup: Destination Field

(b) Packet Classification: 5-Tuple

(c) NAT-In : 2-Tuple

(d) NAT-Out : 2-Tuple

Table 6.3. Average Unique Reuse Distance.

Stream	Top 16	Top 32	Top 64	Top 128
1	6.65	8.23	9.49	10.03
2	12.43	14.57	17.08	19.97
3	5.14	6.09	7.01	7.56
4	6.34	7.28	8.11	8.49

(a) IP Lookup: 1-tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	9.12	11.69	14.56	17.02
2	16.39	20.72	26.92	31.43
3	8.41	10.36	12.53	14.71
4	9.31	11.07	12.88	14.70

(b) NAT-Out : 2-Tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	17.68	21.11	24.41	27.29
2	10.22	12.71	15.70	18.39
3	12.80	15.64	18.37	20.90
4	10.08	12.03	13.91	15.82

(b) NAT-In : 2-Tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	20.19	24.02	28.09	31.90
2	20.35	25.88	32.32	36.70
3	15.24	18. 50	21.84	24.69
4	12.57	14.78	16.84	19.20

(c) Packet Classification: 5-Tuple

6.3 Summary

In this chapter, the memory accesses of networking applications are classified into streaming data and utility data. It is observed that these two types of data have extremely different behavior in temporal locality. Streaming data demonstrates very low locality while utility data shows very high locality. The study shows that the percentage memory accesses to streaming data is significant. Thus, the overall locality is severely degraded which is one reason for the non-cache design of Intel IXP1200 network processor. Furthermore the study shows that because of the high value locality in the packet headers in network packets the memory accesses to utility data shows high temporal locality. The value locality leads to significant amount of redundant memory loads and even an entire iteration of processing computation can be redundant. These opportunities cannot be exploited by a network processor that is designed without cache. This study poses the challenge whether an efficient cache design can be used to exploit the redundant memory accesses and computations.

Chapter 7
Energy-efficient Caches for Network Processors

The study of streaming data in the previous chapter has shown that there is a significant redundancy in memory accesses and computations. This chapter presents methods for exploiting streaming data on a network processor through different designs of cache for the purpose of saving energy.

Network processors are designed to achieve high throughput. With the tremendous increase in line-speed, the amount of throughput required by network processors is increasing significantly. For example, OC-192 (10 Gig/Sec) requires a packet to be processed every 52 nanoseconds and OC-768 (40 Gig/Sec) requires a packet to be processed every 13 nanoseconds.

However, not just any cache design can be used to increase the throughput of routers. High end routers are designed to provide worst-case throughput guarantees. The line-speed is determined with respect to the maximum packet arrival rate (worst case), for which the network processor can guarantee the service of every packet with respect to its longest execution path (worst case) within the loop body. A cache design cannot change the worst case processing time. However, it may shorten the processing time for one packet. As a result, slack can be created in the processing schedule.

The proposed method in this chapter exploits this slack through fetch-gating for the data-plane algorithm while still matching the worst case throughput guarantees of the rest of the network processor. The fetch-gating brings the benefit of energy savings.

Commercial network processors (e.g., Intel IXP1200 network processor) do not

include traditional memory cache in their data-plane processor. There are two reasons behind this. First, as shown in the study in the previous chapter, the level of streaming data is significant in networking applications and the overall locality is degraded. As a result, a traditional memory cache does not bring benefits for network processor. Second, a network processor is usually targeted to have a balanced design. A *balanced design* of a network processor is one where the memory latency is already completely hidden for the desired worst case throughput [43] and each of the components of the network processor are designed for the same worst case throughput guarantees. Balanced network processors are designed with enough overlapped execution (i.e., multiple threads) that the latency to perform each memory lookup in the data-plane algorithm is completely hidden. In real world, a perfect balanced design is rare. Intel IXP1200 network processor uses four threads in each of its microengines which can can hide much of the memory latency but not all of it.

Two energy-efficient cache designs are proposed which are able to effectively create slack. One is called selective cache which is an improvement of traditional memory cache. In this scheme, the streaming data are not cached and only memory references to utility data are cached. This caching ameliorates the situation caused by the first reason mentioned above. However, the effectiveness of this scheme really depends on the extent to which the network processor is balanced. In a completely balanced network processor, there is no room for the selective cache to bring benefits. The second scheme proposed is called computation reuse cache. In this scheme, the tags are the input to the data plane algorithm query, and the data stored is the end result of the query. If there is a cache hit, one can bypass all of the computations required by the data plane algorithm query, including computation instructions and memory accesses. This scheme is effective in creating slack even in a completely balanced design.

In Section 7.1 the design and use of the selective cache is described. Section 7.2 describes the design and use of the computation reuse cache, and Section 7.3 describes

how to use it in combination with fetch gating to save energy. Section 7.4 provides performance results for the proposed approach. Section 7.5 discusses related work. Summary is given in Section 7.6.

7.1 Selective Cache

Selective cache is an improvement of traditional memory cache. In this scheme, streaming data are not cached and only memory references to utility data are cached. Previous study has shown that streaming data has little temporal locality while utility data has high temporal locality. This design is aimed at improve the locality by filtering out memory accesses to streaming data.

Figures 7.1 shows the hardware block-diagram of microengine after a cache is included. The memory reference is processed in parallel with the cache access. Once a reference is issued, it is checked whether it is a marked memory reference. If it is, the cache is accessed. If cache hit occurs, a signal is sent to the microengine controller immediately and at the same time a command is sent to memory controller to cancel the memory reference request. The data is read from the cache. If cache miss occurs, the memory access proceeds normally. Even if cache hit occurs, the context switch is not eliminated. The benefit comes from the immediate signal from cache compared with the long latency of signals from memory.

Figure 7.2 shows the execution timeline for an example on a microengine with only two active threads. In the original timeline, part of long memory access latency are hidden by context switch. There is also a part of the memory access latency that cannot be hidden by the context switch, indicated by the line segments with arrows at both ends marked as *idle time* in the figure. The cache is able to eliminate the idle times in case of cache hit. The lower half of the figure shows the new timeline of the marked code under the assumption that the first 7 marked memory accesses hit in the cache and the last marked memory access does not hit in the cache. In the timeline of this example, the idle time is successfully eliminated and thus improvement in the

Figure 7.1 Selective Cache

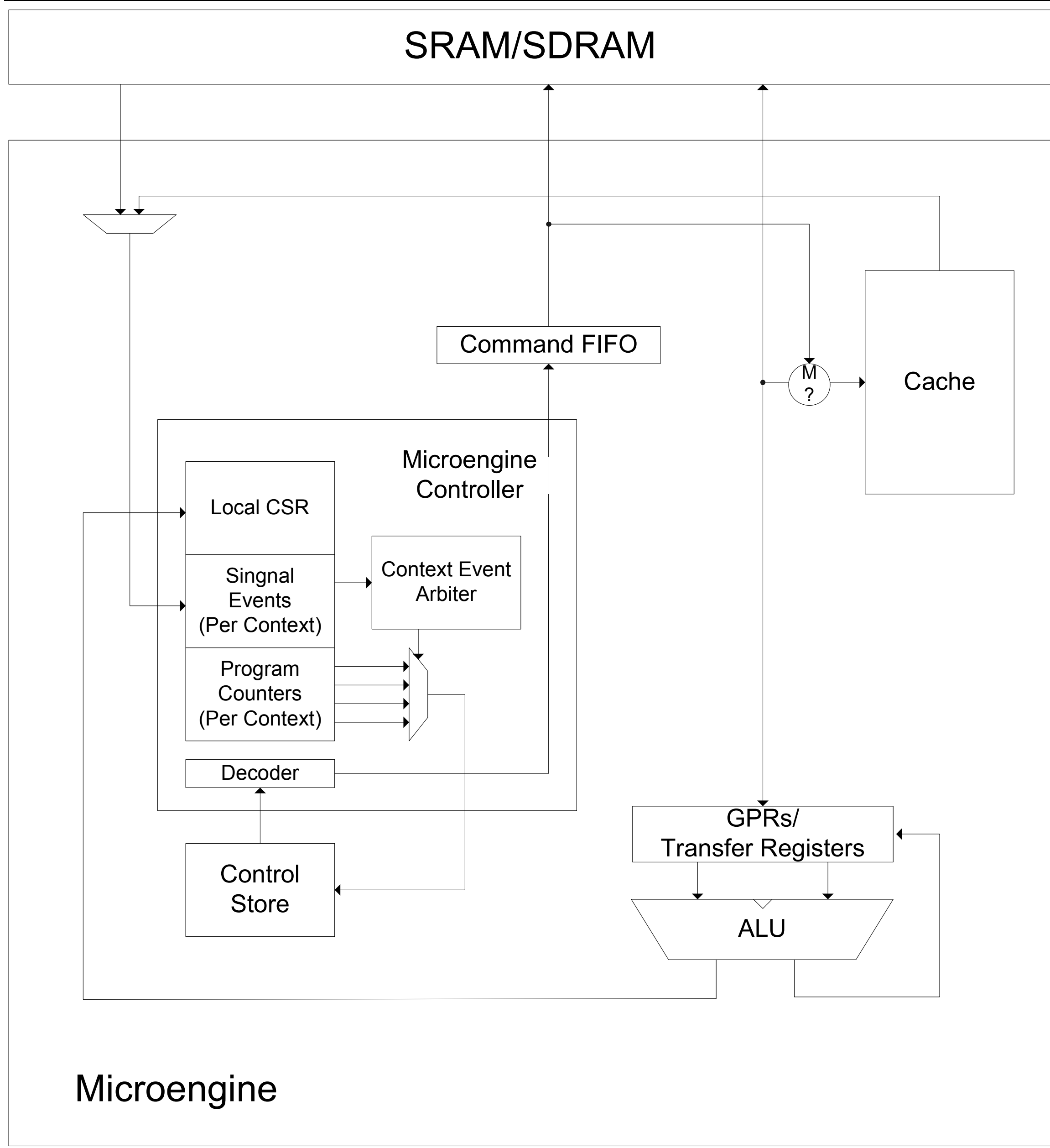
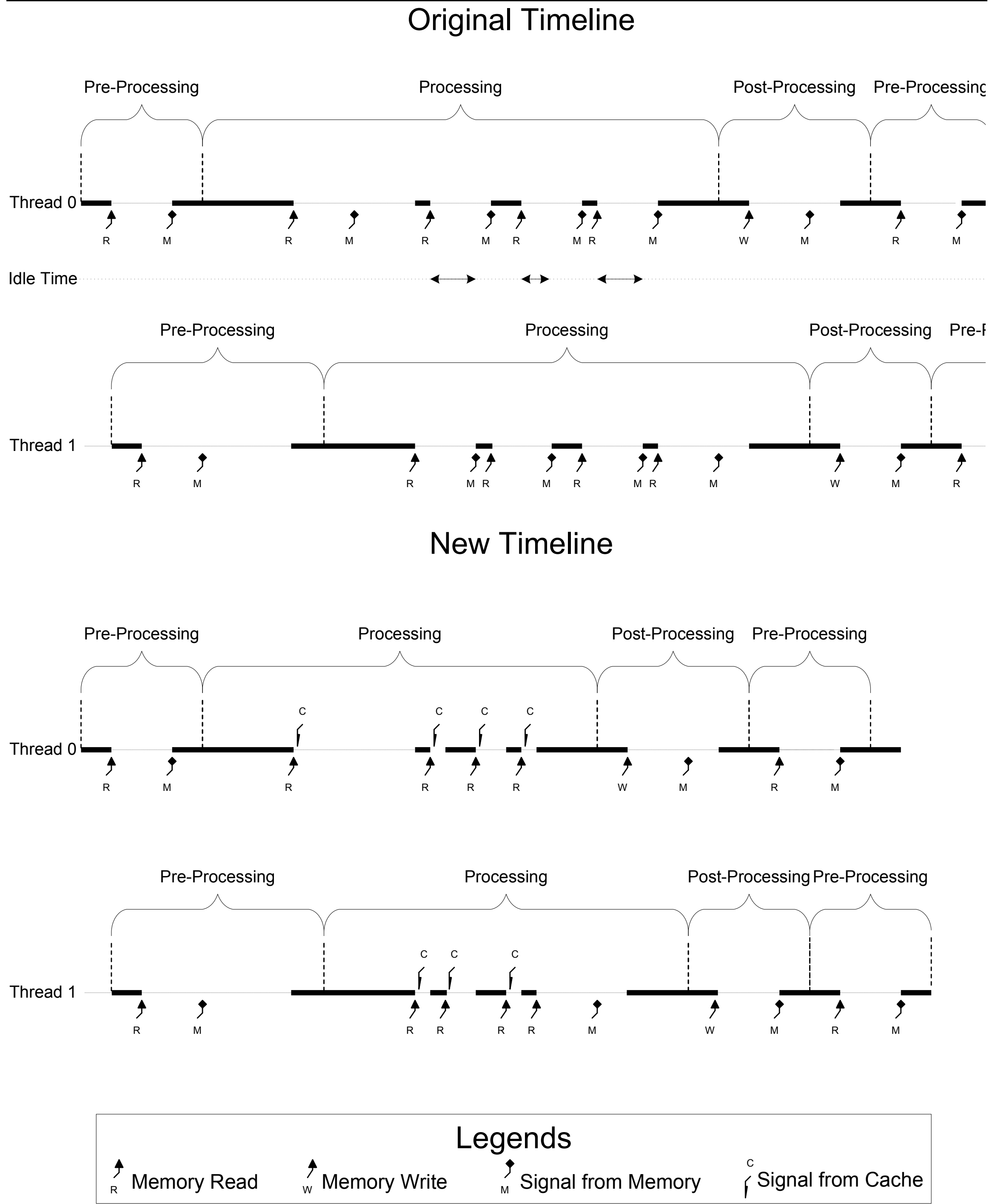


Figure 7.2 Timeline of Selective Cache



average execution time of the code is observed.

7.2 Computation Reuse Cache

In this section the computation reuse cache design is presented for network processors. The proposed cache design has two distinct features. First, the cache can be used across several applications. This goal is achieved by making the cache programmable (i.e., the composition of the tag and data parts can be changed from one application to next). Second, this cache is designed to eliminate redundant computation associated with the network processing data-plane algorithm. Because of the repeated occurrence of the same packet header fields, the data-plane algorithm often performs redundant computation. The computation reuse cache remembers previously performed computations so that later redundant data-plane algorithm queries can be avoided.

It is important to note that the caching performed corresponds to a coarse-grained computation made up of many instructions. The approach is similar in idea to the dynamic instruction reuse techniques by Sodani and Sohi [44]. Sodani and Sohi focused on identifying arbitrary dependency chains of instructions in high performance processors that perform redundant calculations. If these are discovered, then the entire computation is avoided. The above concept is used to design a programmable computation reuse cache. The level of reuse focuses on reusing complete set of function calls (i.e., the data-plane algorithm query), instead of arbitrary dependency chains as examined by Sodani and Sohi. The computation reuse cache is set up specifically for each data-plane algorithm to exploit reuse in its computations. We focus on using this approach to streamline the processing of packets in order to save energy.

For redundancy elimination, a cache line is designed to contain the input (tag) and output (data) of the computation. The inputs are the relevant fields (n-tuple) in the packet headers, working as the tags of the cache line, and the cache line data is the computation result. The cache is configurable by the application. Each data-plane

algorithm is broken into three stages – pre-processing, data-plane processing, and post-processing. A packet goes through these three stages when being processed by the data-plane algorithm. In the pre-processing stage of a packet, when portions of the packet are read in, a tag is formed from the relevant fields. Before the pre-processing stage ends, a lookup in the cache with the n-tuple for the packet is triggered. If a cache hit occurs, the hardware automatically changes the control flow to the post-processing stage for that packet's processing thread and avoids the entire execution of data-plane processing phase. During this post-processing, the computation result is copied from the cache block to registers. In case of a cache miss, the processing continues normally and when the starting point of post-processing phase is reached, the hardware updates the cache with the computation result. The cache is designed in such a way that the worst case throughput of the processing phase is not increased. This goal is achieved through a combination of software assistance and dedicated hardware.

A special register called the *jump target register* (JTR) is added in the microengine controller for each thread (hardware context). JTR remembers the starting point of the post-processing phase of the algorithm so that in case of a cache hit, the control can be directly transferred to this point. In case of cache miss, at this instruction address the computation and its result are sent to the cache for updating the cache line. This register is set during the initialization part of the data-plane algorithm.

The cache is configurable in both the input and output of the processing phase. When initializing the data-plane algorithm at boot time, the configuration of the cache is specified. Masks in the cache are set up so that appropriate header fields can be extracted from the packet for use as the index into the cache. The starting point of post-processing stage is put into JTR.

When a packet is preprocessed, the data to perform data-plane algorithm comes from the packet header. This same data is used to form the cache index and the tag. Therefore, the memory read instructions in the pre-processing phase are marked so

that as they read the relevant packet header fields, relevant values are also sent to the computation reuse cache. Multiple memory read instructions may need to be marked depending upon the number of fields in the n-tuple which act as the inputs to the data-plane algorithm. The cache is setup to receive for each thread the input n-tuple, and the arrival of the last value triggers the computation reuse cache lookup.

When performing the computation reuse cache lookup, the n-tuple is hashed into a cache set index, and then the n-tuple is compared to all of the tags in the set. Note, the tag is itself a n-tuple. If there is a hit for the n-tuple, the result data is copied into registers for the post-processing phase and control is transferred to the instruction in the JTR register. In case of a cache miss, the cache tag is updated with the n-tuple, and the cache block is updated with the result data that becomes available after the processing phase of the data-plane algorithm.

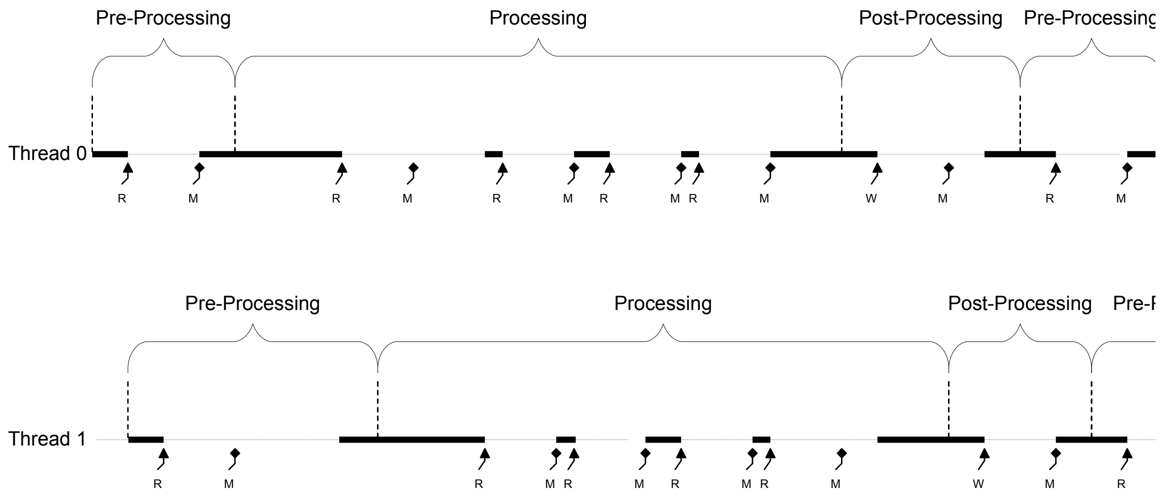
Figure 7.3 shows the timeline of the execution of an example in a microengine with only two threads active assuming computation reuse cache is hit. The processing phase of the two threads are totally avoided. The avoided code includes not only memory references but also computation instructions. This cache is effective even if the memory access latencies are hidden by thread context switch.

It is assumed that there is a separate computation reuse cache for each data-plane algorithm examined.

7.3 Using Fetch Gating

In this section, the approach for performing fetch gating while using the computation reuse cache is described. Fetch gating is a form of pipeline gating proposed by Manne et. al. [37]. Pipeline gating was proposed to stop fetching and executing instructions down wrong (branch mispredicted) paths of execution in order to save energy. The same concept is used here to stall the fetch for the data-plane algorithm when the calculations can be reused due to the computation reuse cache hits. This is possible since the overall network processor is balanced, and if the data-plane algorithm

Figure 7.3 Timeline of Computation Reuse Cache



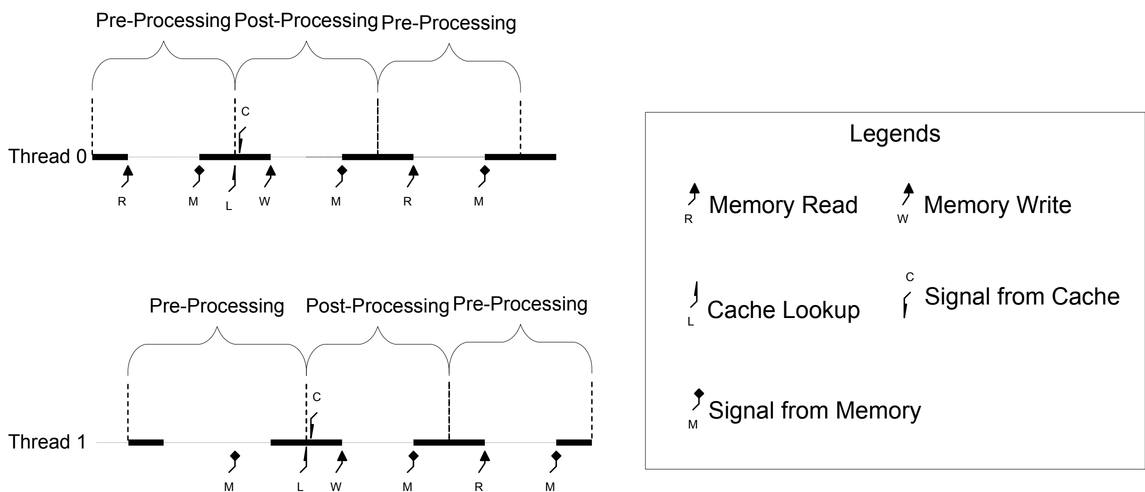
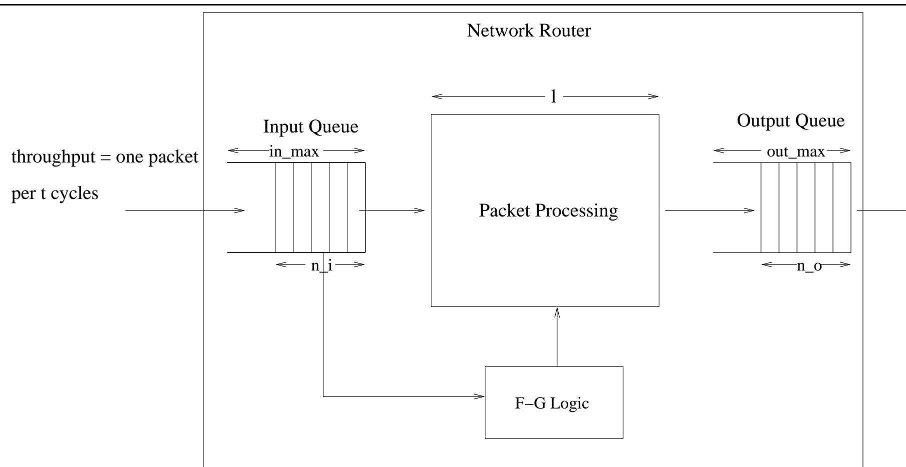


Figure 7.4 Network Router with Fetch Gating (F-G) logic



can reuse and jump ahead in its computations this creates slack in the data-plane algorithm's schedule. It can therefore gate execution while the rest of the network processor continues to process the packets for the overall designed throughput.

Figure 7.4 gives a simplified high level view of a network processor using the computation reuse cache for a data-plane algorithm. In this design it is assumed that the data-plane algorithm has two queues connecting it to the other parts of the processor so that it can be scaled independently of other stages. The input queue to the data plane algorithm initiates the fetch gating logic. Each time there is a change in the queue length, the hardware decides whether to perform fetch gating or keep the processor in normal state. The fetch gating algorithm currently uses two levels. One corresponds to the standard operation when no fetch gating is performed, and the other corresponds to the fetch gated power saving mode. During the latter mode, no fetching or execution will be performed by the fetch gated data-plane algorithm microengine.

The underlying principle of the approach is to perform fetch gating based upon the occupancy of the input queue shown in Figure 7.4. In a balanced network processor design, the occupancy of the input queue should be low. If this is the case and a reasonable number of computation reuse cache hits is available, energy can be

saved by applying fetch gating to the microengine. This can continue up to a point. Once the input queue becomes occupied enough, fetch gating is turned off in order to provide worst-case throughput guarantees and to prevent packets from being dropped.

In Figure 7.4, it is assumed that the number of packets in the input queue is n_i. The fetch gating algorithm checks whether the input queue size, n_i, is smaller than an worst case throughput input queue size threshold. If the input queue to the data-plane algorithm microengine has enough empty slots to guarantee worst case throughput for its implementation, then the microengine is allowed to be in fetch-gated state, else the algorithm performs a normal fetch.

7.4 Experimental Evaluation

The Nepsim [35] simulator is used in the experiments. Nepsim is a cycle-accurate simulator of the Intel IXP1200 network processor. Nepsim is modified to represent a balanced network processor with higher throughput and extended it with the cache designs and the fetch gating algorithm. For the results a 30 cycle on-chip SRAM latency is modeled to store the data structure for the data-plane algorithm being examined.

In the evaluation, four benchmarks are used which are modified from Intel's Workbench suite or self-developed and migrated to run on the Nepsim simulator. They are IP-Lookup, Packet Classification, and NAT Protocol (in and out). The properties of the application are summarized in Table 7.1. These applications have the code size of 200 to 300 instructions, shown in the first column. It also shows the worst-case packet processing time (latency) that was observed across the four traces in the second column. For IP-Lookup this is 646 cycles and for packet classification it is 1160 cycles. This processing time is considered as the time period between when a packet enters into the processing stage and when it leaves the processing stage (i.e., not counting the time that the data-plane algorithm is spinning and waiting for the packet to arrive). It also does not include the cycles spent on receiving and transmit-

Table 7.1. Application Properties

Applications	Code Size	Proc. Time (Worst)	Proc. Time (Average)	#SRAM Refs
ip-lookup	291	646	435	5
classification	254	1160	1010	13
nat-in	205	754	603	6
nat-out	205	757	597	6

ting the packet. It also shows the average-case packet processing time in the third column – the average is computed over the four traces. The last column shows the number of SRAM references needed in the worst case for the algorithm.

7.4.1 Cache Behavior

Table 7.2 and Table 7.3 show the hit rate of selective cache and computation reuse cache. In both schemes, a 64 entry direct mapped configuration is used. The four packet streams used in the study in previous chapter are used as input. For selective cache, the hit rate varies between 81% and 97%. For computation reuse cache, the hit rate varies between 52% and 88%. The reasonably high hit rate shows in both schemes that locality is being effectively exploited. Selective cache shows an even higher hit rate than computation reuse cache. This is because selective cache works at a finer granularity and therefore affects higher locality. The following example reveals the reason behind this behavior. Suppose there are two different values in the same field in two different packet headers and the two values share certain length of same prefix. In computation reuse cache, they are considered as two totally different tags. Thus nothing can be used between them. However in selective cache, they may cause repetitive memory references because they share the prefix.

TABLE 7.2. Selective Cache Hit Rate

Applications	Packet Stream Trace			
	1	2	3	4
ip-lookup	95.77%	92.22%	96.53%	97.10%
classification	82.12%	81.85%	85.06%	86.42%
nat-in	90.04%	89.74%	92.89%	93.50%
nat-out	91.28%	90.24%	94.01%	94.24%

TABLE 7.3. Computation Reuse Cache Hit Rate

Applications	Packet Stream Trace			
	1	2	3	4
ip-lookup	68.43%	70.18%	88.89%	85.79%
classification	60.03%	84.97%	77.42%	77.42%
nat-in	74.70%	67.87%	84.45%	82.57%
nat-out	52.52%	82.78%	71.68%	71.68%

7.4.2 Fetch Gating and Energy Savings

Table 7.4 and Table 7.5 show the effect of cache hits on the data-plane algorithm for the selective cache and the computation reuse cache respectively. These results were obtained by running trace 1 from Table 6.2 through each of the algorithms. In both tables, the first column, ME Energy, gives the energy used by the microengine on which the data-plane algorithm is being run as a percentage of total energy of the network processor without cache. The second and third columns show the average packet processing latency (cycles) in the absence of cache and the percentage reduction in the time when the cache is used. For the selective cache, the reduction varies from 3.62% for IP-Lookup to 8.23% for classification. For the computation reuse cache, the reduction varies from 19% for IP-Lookup and 47% for classification. From the results we conclude that the computation reuse cache successfully creates more slack than the selective cache despite that the selective cache has higher cache hit

rate than the computation reuse cache. This is in accordance with previous analysis. The effect of the selective cache depends on the degree of the balance of the network processor. Since much of the memory latency has been hidden by overlapping the execution of threads, there is less room for the selective cache to produce improvement. In contrast, in the computation reuse cache, not only memory latency can be hidden but also the computation instruction sequence which has much coarser granularity can be hidden. This means that once there is a hit, the computation reuse cache can save much more time than a hit in selective cache. This is why computation reuse cache achieves better time reduction even with lower cache hit rate.

TABLE 7.4. Program Behavior with Selective Cache

Applications	ME % Total Energy	Time No Cache	Time Reduction	Cycles Gated	ME Energy Reduction
ip-lookup	33.47%	435	3.62%	2.18%	1.15%
classification	29.66%	1010	8.23%	5.84%	3.94%
nat-in	27.71%	603	6.02%	4.75%	2.45%
nat-out	29.38%	597	6.31%	4.96%	2.54%

TABLE 7.5. Program Behavior with Computation Reuse Cache

Applications	ME % Total Energy	Time No Cache	Time Reduction	Cycles Gated	ME Energy Reduction
ip-lookup	33.47%	435	18.62%	22.45%	16.61%
classification	29.66%	1010	47.23%	30.78%	18.89%
nat-in	27.71%	603	51.58%	45.82%	37.69%
nat-out	29.38%	597	41.37%	42.64%	28.43%

The last two columns in Table 7.4 and Table 7.5 show the results of applying the fetch gating algorithm in the prior section. The energy is recorded using the models provided in the Nepsim [35] simulator augmented to take into consideration the cache and the network processor changes. The fourth column shows the percentage of cycles the algorithm was fetch gated. The last column shows the percentage of energy

savings when using the fetch gating when compared to the energy used for just that microengine shown in column one. For these results, only one data-plane algorithm is examined at a time, and the computation reuse cache is 64-entry and directed mapped. For the selective cache, the results show that for IP-Lookup 2.18% of the cycles is fetch gated for the data-plane algorithm microengine and an energy savings of nearly 1.15% is achieved. This results in an overall network processor energy savings of 0.4%. Across all applications, the computation reuse cache allows 2.18% to 5.84% of packet processing to be performed in fetch gated mode which produces 1.15% to 3.94% in energy savings for the microengines. For the computation reuse cache, the results show that for IP-Lookup 22% of the cycles are fetch gated for the data-plane algorithm microengine and achieve an energy savings of nearly 17%. This results in an overall network processor energy savings of 6%. Across all applications, the computation reuse cache allows from 22% to 46% of packet processing to be performed in fetch gated mode which produces 17% to 38% in energy savings for the microengines. Note that slack is generated by reducing the amount of computation needed for a packet when there is a hit in the computation reuse cache, and then energy savings comes from using fetch gating to exploit this slack. To summarize the experimental results, it can be concluded that computation reuse cache is much more effective than selective cache in saving energy.

7.5 Related Work

A summary of related work is given on temporal locality and caching in network processors, reuse caching, and fetch gating.

7.5.1 Locality and Caching in Network Processors

Memik et al. [38] examine the use of a traditional cache for a set of networking applications on a StrongARM 110 processor. They found that most of the cache

misses came from a small number of instructions. To exploit this observation, they use a filter for the data cache to remove the memory accesses with low locality. Li et al. [3] investigate a range of memory architectures that can be used for a wide range of packet classification caches. They study the impact of factors like cache associativity, cache size, replacement policy and the complexity of hash functions on the overall system performance. Both of these studies use a traditional cache, which cannot be used to increase the throughput of data plane algorithms for a balanced network processor. This is why the focus is on saving energy by using a computation reuse cache to create slack in the data-plane algorithm's schedule.

Chiueh et al. [17] use a combined hardware/software approach to construct a cache for high performance IP routing in general-purpose CPU based routers. The destination host address is mapped to a virtual address space and used to lookup a destination route in the Host Address Cache (HAC). A part of the normal L1 data cache is reserved for use as the HAC. This approach can be classified as being similar to the computation reuse cache in that a hit in the cache skips the full IP lookup. In case of a lookup miss, a 3-level routing table lookup algorithm is consulted for the final routing decision. The focus of the above work is on using the HAC to provide throughput for their network processor design, and not for energy savings. The contribution of the work in this chapter is to define the general notion of using a programmable computation reuse cache for data plane algorithms and to use it to save energy in a balanced network processor while still providing worst-case throughput guarantees.

7.5.2 Instruction Reuse

Sodini and Sohi [44] observe that many instructions or groups of instructions have the same input and generate the same output when dynamically executed. They exploit this phenomenon by buffering the computation result of the previous execution of

instruction dependency chains. They use the same result for future dynamic instances of the same dependency chains if the input to these chains are the same. In this way, the execution of many groups of instructions can be avoided and the early outcome can allow dependent instructions to proceed sooner. They use a hardware mechanism called the Reuse Buffer (RB) to store the previous computation results. The program counter is used as an index to search the RB for cached chains. Our computation reuse cache is motivated by the reuse buffer, but it differs in that it is programmable and used to cache computations at much larger levels (methods) than just dependency chains.

Ding and Li [20] present a pure compiler memoization technique to exploit value locality. They detect code segments that are executed repeatedly, which generate a small number of different values. The code segment is replaced by a table recording the previous computation results for later lookup if the same values are seen. Performance improvement and energy consumption reduction are achieved. Their work is related to the work in this chapter because a computation reuse mechanism is also used to reduce energy consumption. Their approach is a purely software technique. In contrast, a programmable hardware technique specific for data-plane algorithms is provided in order to save energy while providing worst-case throughput guarantees.

7.5.3 Clock and Fetch Gating

Luo et al. [36] use a clock gating technique to reduce power consumption in multi-core network processors. They observe that when the incoming traffic rate is low, most processing elements on the network processor are nearly idle and yet still consume dynamic power. When the number of idle threads increase, they start to gate off the clock network of a processing unit. When the pressure from the incoming buffer rises, they stop clock gating. Since the activation takes time, they need extra buffer space to avoid packet loss. They also developed strategies to terminate and reschedule

threads during activation and deactivation. This work is related to the techniques in this chapter because both aim at gating some part of the network processor to reduce energy. Both use the queue occupancy information in gating decisions. Their approach is complementary to the technique in this chapter. Their focus is on applying fetch gating when the traffic rate is low, whereas the work in this chapter focus on applying fetch gating when value locality can be found and exploited, and this includes when the traffic rate is high.

Manne et al. [37] observe that due to branch mispredictions wrong path instructions cause a large amount of unnecessary work in wide-issue super-scalar processors. They develop a hardware mechanism called pipeline gating to control rampant speculation in the pipeline. They use a confidence estimator to assess the quality of each branch prediction. In case of low confidence, they gate the pipeline by stalling instruction fetch. Baniasadi and Moshovos [7] extend this approach to throttle the execution of instruction flow in wide-issue super-scalar processors to achieve energy reduction. They use instruction flow information such as rate of instructions passing through stages to determine whether to stall stages. When the rate is sufficiently high and there is enough instruction level parallelism, they may stall fetch because introducing extra instructions would not significantly improve performance.

Karkhanis et al. [26] also propose a mechanism called Just-In-Time instruction delivery to save energy. They observe that performance-driven design philosophy causes useful instructions to be fetched earlier than needed and stall in the pipeline for many cycles or they wait in the issue queue. Also when a branch misprediction occurs, all those early-issued instruction along mispredicted branch are flushed. This wastes energy. Their suggested mechanism monitors and dynamically adjusts the maximum number of in-flight instructions in the processor according to processor performance. When a maximum number is reached, the instruction fetching is gated.

Buyuktosunoglu et al. [13] collect issue queue statistics to resize the issue queue dynamically to improve issue queue energy and performance on a super-scalar pro-

cessor. The statistics are derived from counters that keep track of the active state of each queue entry on a cycle-by-cycle basis. They divide the issue queue into separate chunks and may turn off/on certain block based on statistics.

The above prior works are related to the work in this chapter since they all gate/throttle the execution of instruction flow to achieve the goal of energy reduction. The approach in the chapter is built upon these techniques to gate fetch for the data-plane microengines.

7.6 Summary

High end network processors are built with a balanced processor design, where using a traditional cache for the data-plane algorithm will not increase throughput. These processors are built such that there is sufficient threading (packet parallelism) to hide each data-plane algorithm SRAM lookup latency, so a traditional cache cannot increase throughput.

In this chapter two schemes are presented for the sake of energy. One scheme is selective cache which is an improvement of traditional cache by filtering out memory accesses to streaming data and thus improve locality and hit rate. Selective cache is effective in hiding those memory latencies which are not totally hidden on an incompletely balanced network processor. Selective cache achieves small energy savings. Another scheme is based upon the computation reuse cache, where a hit hides the full data-plane algorithm processing of the packet, not just one SRAM lookup as in a traditional cache. This is accomplished by having a cache block contain the input as the tag and output as the data of the data-plane algorithm computation. Therefore a complete query performed by the data-plane algorithm takes one cache access if there is a hit. Slack is therefore generated by reducing the number of instructions executed when there is a hit. This reduction in number of instructions allows us to exploit the slack to save energy through fetch gating for the data-plane algorithm microengine while still matching the worst case throughput guarantees of the rest of the processor.

Overall, the computation reuse cache allowed 22% to 46% of the execution time to be performed in fetch gated mode with 17% to 38% reduction in data-plane algorithm energy across the different algorithms examined. Computation reuse cache is much more effective than selective cache in saving energy.

Chapter 8
Conclusions

This dissertation makes contributions in efficient handling of narrow width and streaming data in embedded applications. Typical embedded applications process streams of formatted data units and demonstrates unique data characteristics. Narrow width data and streaming data are the two important data characteristics observed in embedded applications. Originating from fields within the formatted data units, *narrow width data* are data representable in considerably fewer bits than in one word. They nevertheless occupy an entire register or memory word. Processing of narrow width data is often preceded/followed by unpacking/packing operations. The common presence of narrow width data in embedded applications poses challenges such as low register utilization, wasted data memory, and performance degradation. Coming from the streams of data units, streaming data often are not reusable and show short lifetime and lack temporal locality. This leads to low cache efficiency. My dissertation tackles these problems to achieve optimizations in critical factors in embedded applications, including memory footprint, power consumption, and performance. Section 8.1 summarizes the contributions made in this dissertation. Section 8.2 discusses future directions.

8.1 Contributions

To efficiently handle narrow width and streaming data, new architectural features are introduced and associated compiler algorithms are developed. Narrow width data are manipulated efficiently by packing multiple narrow width data items into one register or memory word. Using register allocation or memory layout methods, the problem of underutilization of registers, the underutilization of memory, and performance

degradation are addressed. Streaming data are handled efficiently by providing cache mechanisms which enable reuse of utility data or computation sequence across the processing iterations. This creates slack in the processing schedule, which in turn are transformed into the reduction of energy consumption. Specifically, this dissertation makes the following contributions.

Measurement of Narrow Width and Streaming Data in Embedded Applications. Quantitative measurement of the presence and properties of narrow width and streaming data are performed respectively in typical embedded applications from Mediabench and Commbench. The result shows that there is significant amount of narrow width data in embedded applications. More than half of the bits in the register file are not used. Nearly 80% of the register operand instances have less than 16 valid bits. The study reveals that there exists significantly more dynamic narrow width data than static narrow width data. The study of streaming data shows that streaming data have very short lifetime and the memory accesses to them occupies a significant percentage. Contrary to utility data, streaming data have low temporal locality. The result also shows there is a great amount of value locality in streaming data which might cause redundant computation and memory accesses to utility data. Both studies provide hints on how to efficiently handle them.

An Instruction Set Extension for Narrow Width Data. An instruction set extension for ARM processor is proposed to manipulate narrow width data directly with bit section addressing mode. With this extension multiple narrow width data can reside in one register. The cost of packing and unpacking operations can be reduced via a peephole style optimization using this instruction set extension.

Efficient Handling of Narrow Width Data in Register. Two register allocation schemes are proposed to handle narrow width data in register efficiently. The algorithms allocate two variables, whose bitwidth is less than 16-bit, to one 32-bit register. Register pressure and spill cost are reduced. The static register allocation scheme determines an assignment of such variables based on the maximum bitwidth

information. In a further step, a speculative register allocation scheme is proposed to exploit the fact that there are more narrow width data dynamically than statically. Two variables, whose bitwidths at runtime are most probably less than 16-bit, are allocated to one 32-register. In case of conflicts, architectural enhancement is able to guarantee the correct execution semantics. The results show up to 10% of the spill cost can be avoided by static scheme and up to 95% by the speculative scheme. This leads to a performance improvement of up to 3% for static scheme and up to 14% for the speculative scheme.

Efficient Handling of Narrow Width Data in Memory. A scheme is proposed to coalesce multiple narrow width data in one memory location in DSP processor. As a result, memory footprint can be shrunk. More importantly, this scheme brings significant benefit in performance. With coalescing, explicit address calculation operations can be reduced by even more extensively using DSP specific addressing mode of *address register autoincrement/autodecrement*. Three algorithms are proposed to solve the offset assignment problem in presence of narrow width data. The first algorithm takes cover-first coalesce-later policy. The second one uses coalesce-first cover-later policy. The third one interleaves the covering and coalescing operations at a finer grain and is able to maximize the benefits.

Energy-efficient Cache for Network Processor Exploiting Streaming Data. Two energy-efficient cache mechanisms are proposed to reduce energy-consumption on Intel IXP1200 network processor by exploiting streaming data. Cache is designed to exploit the value locality of streaming data to reduce the redundancy in computation or memory accesses to utility data to ameliorate the situation that traditional cache is not effective for network processor due to a balanced design and low temporal locality in streaming data. The slack created by such a scheme is transformed into reduction in energy-consumption through a fetch-gating mechanism. A selective cache is proposed to only cache memory accesses to utility data by filtering out streaming data. By doing so, cache hit rate can be maintained at a high level and memory access

redundancy is reduced. However, the effectiveness of this scheme is still limited due to a balanced design. A computation reuse cache is designed to store the computation result together with relevant packet header fields. Redundant computation sequence can be avoided. The computation reuse is effective even in a balanced design of a network processor.

8.2 Future Directions

The research in this dissertation can be expanded in two directions. First, the idea and methodology used in this dissertation can be applied in other problems. Specifically, it is possible to exploit narrow width data in high-end systems. Second, more techniques can be developed to exploit narrow width and streaming data in embedded applications. For example, it is possible to exploit narrow width data through register rematerialization and to develop a good compiler algorithm to utilize selective cache in presence of streaming data. This section describes these ideas.

Exploiting Narrow Width Data in High-End Systems. As high-end systems evolve into even wider word width such as Itanium processors with 64-bit width, there are new opportunities to exploit narrow width data. For example, no matter what word width is used, only one bit is actually needed to represent a boolean variable. As word width becomes wider, it is possible that data which is originally not narrow width becomes narrow width. A quantitative measurement is required to reveal the presence of narrow width data in high-end systems. Furthermore, wider word width means more instruction encoding space. This allows even more thorough implementation of Bit Section eXtension. With stronger architectural support, register allocation algorithm can be developed to achieve better improvement.

Exploiting Narrow Width Data by Register Rematerialization. Narrow width data can be exploited by register rematerialization. Register rematerialization recomputes the value of a variable instead of spilling it to memory if the reconstruction of the

value is possible and cheaper. If multiple narrow width data are in a packed form and reside in a register, multiple values can be rematerialized with low cost. The data processed by embedded applications naturally contains packed form of narrow width data. It is a good opportunity to apply register rematerialization. A good compiler algorithm can be developed to decide the portion of the register file to hold packed narrow width data based on the estimation of cost and savings.

Compiler Algorithm for Selective Cache. In this research selective cache is not as effective as computation reuse cache. It could be very effective on a processor which is not hardware multi-threaded. If this is true, a good compiler algorithm can be developed to decide whether a specific instance of memory access should be cached or not. In this way, cache efficiency can be greatly improved. Decisions can be made either based on static analysis or profiling information.

References

[1] Embedded system. In *Wikipedia, http://en.wikipedia.orgi/wiki/Embedded system.*

[2] Mathew Adiletta, Mark Rosenbluth, Debrea Bernstein, Gilber Wolrich, and Hugh Wilkinson. The next generation of intel ixp network processors. In *Intel Technology Journal*, volume 6, pages 6–18, 2002.

[3] Kang Li adn Francis Chang, Damein Berger, and Wu chang Feng. Architectures for packet classification caching. In *IEEE International Conference on Networks*, Sydney, Australia, 2003.

[4] Jin Lin adn Tong Chen adn Wei-Chung Hsu and Pen-Cheung Yew. Speculative register promotion using advanced load address table (alat). In *International Symposium on Code Generation and Optimization*, San Francisco, CA, 2003.

[5] Gokhan Memik andi William H. Mangione-Smith and Wendong Hu. Netbench: A benchmarking suite for network processors. In *IEEE International Conference on Computer-Aided Design*, San Jose, CA, 2001.

[6] Wand Research Group at University of Auckland. Auckland-ii trace archive. In *NLANR, http://pma.nlanr.net/Traces/long/auck2.html.*

[7] Amirali Baniasadi and Andreas Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *International Symposium on Low Power Electronics and Design*, pages 16–21, Huntington Beach, CA, 2001.

[8] David H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. In *Software Practice and Experience*, volume 22, pages 101–110, 1992.

[9] Peter Brink, Manohar Casterlino, David Meng, Chetan Rawal, and Hari Tadepalli. Network processing performance metrics for ia- and ixp-based systems. In *Intel Technology Journal*, volume 7, 2003.

[10] David Brooks and Margaret Martonosi. Value-based clock gating and operation packing: Dynamic strategies for improving processor power and performance. In *ACM Transactions on Computer Systems*, volume 18, pages 89–126, New York, NY, USA, 2000. ACM Press.

[11] Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. Bitvalue inference: Detecting and exploiting narrow width computations. In *The 6th European Conference on Parallel Computing*, 2000.

[12] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 13–25, 1997.

[13] Alper Buyuktosunoglu, Stanley Schuster, David Brooks, Pradip Bose, Peter Cook, and David Albonesi. An adaptive issue queue for reduced power at high performance. In *International Workshop on Power-Aware Computer Systems*, Cambridge, MA.

[14] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. In *Computer Languages*, volume 6, pages 47–57, 1981.

[15] Yoonseo Choi and Taewhan Kim. Address assignment combined with scheduling in dsp code generation. In *ACM/IEEE Design Automation Conference*, pages 225–330, New Orleans, LA, 2002.

[16] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. In *ACM Transactions of Programming Languages and Systems*, volume 12, pages 501–536, 1990.

[17] Tzi cker Chiueh and Prashant Pradhan. High performance ip routing table lookup using cpu caching. In *IEEE INFOCOM*, pages 1421–1428, New York, NY, 1999.

[18] Intel Corporation. Sa-110 microprocessor technical reference manual. In *Intel, ftp://download.intel.com/design/strong/applnots/ 27819401.pdf.*

[19] Jack Davidson and Sanjay Jinturkar. Memory access coalescing : A technique for eliminating redundant memory accesses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 186–195, Olando, FL, 1994.

[20] Yonghua Ding and Zhiyuan Li. A compiler scheme for reusing intermediate computation results. In *International Symposium on Code Generation and Optimziation*, Palo Alto, CA, 2004.

[21] Jose Fridman. Data alignment for sub-word parallelism in dsp. In *IEEE Workshop on Signal Processing Systems*, pages 251–260, 1999.

[22] Steve Furber. Arm system architecture. Addison Wesley Longman, 1996.

[23] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. A representation for bit section based analysis and optimization. In *International Conference on Compiler Construction*, pages 62–77, Grenoble, France, 2002. Springer Verlag.

[24] C.Evan Foster III and Harold C. Grossman. An empirical investigation of the haifa register allocation in the gnu c compiler. In *IEEE Southeast Conference*, pages 776–779, 1992.

[25] Mahmut Kandemir, Mary J. Irwin, Guilin Chen, and J. Ramanujam. Address register assignment for reducing code size. In *International Conference on Compiler Construction*, pages 273–289, Warsaw , Poland, 2003. LNCS 2622, Springer Verlag.

[26] Tejas Karkhanis, James E. Smith, and Pradip Bose. Saving energy with just in time instruction delivery. In *International Symposium on Low Power Electronics and Design*, Montery, CA, 2002.

[27] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver B.C., Canada, 2000.

[28] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture*, Research Triangle Park, NC, 1997.

[29] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In *International Symposium on Systems Synthesis*, pages 3–8, Hsinchu, Taiwan, China, 1998.

[30] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in dsp code generation. In *International Conference on Computer-Aided Design*, pages 109–112, San Jose, CA, 1996.

[31] Bengu Li and Rajiv Gupta. Bit section instruction set extension of arm for embedded applications. In *International Conference on Compilers, Architecture, and Synthesis of Embedded Systems*, Grenoble, France, 2002.

[32] Bengu Li and Rajiv Gupta. Simple offset assignment in presence of subword data. In *International Conference on Compilers, Architecture, and Synthesis of Embedded Systems*, San Jose, CA, 2003.

[33] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tijiang, and Albert Wang. Storage assignment to decrease code size. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 186–195, La Jolla, CA, 1995.

[34] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tijiang, and Albert Wang. Storage assignment to decrease code size. In *ACM Transactions on Programming Languages and Systems*, volume 18, pages 235–253, 1996.

[35] Yan Luo, Jun Yang, Laxmi Bhuya, and Li Zhao. Nepsim: A network processor simulator with power evaluation framework. In *IEEE Micro, Special Issue on Network Processors for Future High-End Systems and Applications*, 2004.

[36] Yan Luo, Jia Yu, Jun Yang, and Laxmi Bhuyan. Low power network processor design using clock gating. In *ACM/IEEE Design Automation Conference*, pages 712–715, Anaheim, CA, 2005.

[37] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, pages 132–141, 1998.

[38] Gokhan Memik and William H. Mangione-Smith. Improving power efficiency of multi-core network processors through data filtering. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Grenoble, France, 2002.

[39] Xiaoning Nie, Lajos Gazsi, and Frank Engeland Gerhad Fettweis. A new network processor architecture for high speed communications. In *IEEE Workshop on Signal Processing Systems*, pages 548–557, 1999.

[40] Giles Pokam, Olivier Rochecouste, Andre Seznec, and Francois Bodin. Speculative software management of datapath-width for energy optimization. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[41] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 128–138, Atlanta, GA, 1999.

[42] rebel.com. Netwinder family. In *Rebel, http://www.rebel.com/netwinder.*

[43] Timothy Sherwood, George Varghese, and Brad Calder. A pipelined memory architecture for high throughput network processors. In *International Symposium on Computer Architecture*, pages 299–299, San Diego, CA, 2003.

[44] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *IEEE/ACM ISCA*, pages 194–205, Denver, CO, 1997.

[45] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver B.C., Canada, 2000.

[46] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. Analysis and evaluation of address arithmetic capabilities in custom dsp architectures. In *ACM/IEEE Design Automation Conference*, pages 287–292, Anaheim, CA, 1997.

[47] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. pages 108–120. Dept. of Computer Science, University of Arizona, 2002. Technical Report.

[48] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–96, New Orleans, LA, 2003.

[49] Sathishukumar Udayanarayanan and Chaitali Chakrabarti. Address code generation for digital signal processors. In *ACM/IEEE Design Automation Conference*, pages 353–358, Las Vegas, NV, 2001.

[50] Tilman Wolf and Mark Franklin. Commbench - a telecommunications benchmark for network processors. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, 2000.

[51] Jun Yang and Rajiv Gupta. Energy efficient frequent value data cache design. In *IEEE/ACM International Symposium on Microarchitecture*, pages 197–207, Istanbul, Turkey, 2002.

[52] Xiao Yang and Ruby B. Lee. Fast subword permutation instructions using omega and flip network stages,. In *International Conference on Computer Design*, pages 15–22, Austin, Texas, 2000.

[53] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction*, pages 14–28, Grenoble, France, 2002.

[54] Youtao Zhang and Rajiv Gupta. Enabling partial cache line prefetching through data compression. In *International Conference on Parallel Processing*, Kaohsiung, Taiwan, China, 2003.

[55] Xiaotong Zhuang, Choksheak Lau, and Santosh Pande. Storage assignment optimizations through variable coalescence for embedded processors. In *ACM SIGPLAN Conference on Languages, Compiler, and Tools for Embedded Systems*, pages 220–231, San Diego, CA, 2003.