ON EXTRACTING COARSE-GRAINED FUNCTION PARALLELISM FROM C PROGRAMS

BY

CHIEN-WEI LI

B.S., National Taiwan University, 1990
M.S., National Taiwan University, 1992

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

To my family, my teachers, my friends, and people who helped me.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Wen-mei Hwu for giving me this opportunity to learn how to solve important problems. He teaches me to see the big picture, as well as to pay attention to the details. I appreciate his patience in correcting my errors in speaking, writing, and thinking. I feel grateful for his generosity in financial support. He has done everything a good teacher could do, however, I am not capable enough to fully carry out his vision. Efficiently mapping complex applications onto parallel machines is a fascinating problem to me. Hope that I can work on this for the rest of my life, based on what I've learned from him.

I would like to thank Professors David Padua, Vikram Adve and Mark Hesagawa-Johnson for their courtesy of being my thesis committee members. Their experiences, comments, and critics broaden my knowledge and make me understand the problem more deeply. I would like to thank Professors Nick Carter, Matt Frank, and Steve Lumetta for their feedback on my work.

Although they may not know me, I would still like to thank the professors who taught those interesting and helpful courses that I took at UIUC. Especially, I would like to thank Professor Benjamin Wah, visiting Professor Yao-Jen Chang, and late Professor Michael Faiman for their personal instruction and assistance when I first came to America. I would like to thank my M.S. thesis advisor Professor Jie-Yong Juang and my other teachers in Taiwan, from K to 18, who really make my life at UIUC much easier.

I would like to thank my officemates Hong-Seok Kim and Dan Burke. I learned a lot from Hong-Seok about pointer analysis and program analysis. Many ideas in my research are inspired during our discussion. I also obtained a lot of hardware knowledge from Dan. I would like to thank my colleagues, Ben-chung Cheng, Hong-Seok Kim, and Erik Nystrom for their pointer analysis work; Robert Kidd, Hong-Seok Kim, Tahir Mobashir, Erik Nystrom, James Player, Shane Ryoo, John Sias, and Ian Steiner for their Pcode enhancement work. Especially, Bob and John made a lot of effort in system administration and in perfecting the IMPACT compiler. I also appreciate the help of other IMPACT colleagues, Ron Barnes, Kevin Cernekee, Marie Conte, Hillery Hunter, Geoff Kent, Matt Merten, Chris Rodriguez, Andy Schuh, Chris Shannon, Sain Ueng, and Le-chun Wu. Especially, Le-chun and Ben-chung have been helping me since we met in Taiwan.

I would like to thank the staffs of the IMPACT group, Sabrina Hwu, Marie-Pierre Lassiva-Moulin and Xiaolin Liu, and the staffs of Coordinated Science Laboratory and Computer Science department. Especially, Marie-Pierre helps me a lot for the deposit of this dissertation. I really appreciate it.

I would like to thank the friends I made at Rockwell, Conexant and Mindspeed, especially my mentor Dr. Kumar Ganapathy, for their help and sharing experience.

This research is funded by the Semiconductor Research Corporation and by the Gigascale Systems Research Center.

Not to have an Acknowledgement longer than the other chapters, I'll just stop here. Finally, I would like to thank my other friends and my family for taking care of me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The progress of IT (Information Technology) industry is driven by the simultaneous advance of semiconductor manufacturing technology, hardware, application, and design methodology. More advanced manufacturing technology enables more powerful hardware, which in turn enables more advanced application. On the other hand, more advanced application motivates more powerful hardware, which in turn motivates more advanced manufacturing technology. Although less visible, design methodology plays a crucial role in meshing technology to hardware, and hardware to application, so that the whole IT industry is not out of gear.

To put the rest of this dissertation in perspective, this chapter will examine the trends on manufacturing technology, hardware, and application, and point out, among the many challenges faced by the current design methodology, which problem domain this dissertation is trying to make some small step contributions. Chapter 2 will review previous works to understand how the problems are approached by other researchers in different ways, and to identify the specific problem that this work will focus on. Chapter 3 will present the problem statement to set the goal of this work, and outline the steps to achieve the goal by decomposing the problem into sub-problems. Later chapters of this dissertation will discuss each of these sub-problems and the proposed solutions in

detail. Finally, this dissertation will conclude with the results and insights obtained from prototyping the proposed solutions, and propose some future works.

1.1 Technology Trend

The rapid growth of the semiconductor industry is fundamentally driven by a trend observed by Gordon Moore in 1965, that is transistor density doubles every 18 months [1]. In this rate, a single chip will have a billion transistors on it in the near future, enough for the integration of a whole system [2]. However, to utilize this enormous amount of transistors, we need to solve many problems. Below is an incomplete list of the problems.

- The NRE (Non-Recurrent Engineering) cost is soaring. For example, the cost of mask set has risen from several hundred thousand dollars for 0.18-micron process to over 1 million dollars for 90-nm process, and 3 million dollars for 65-nm process [3] [4] [5]. Moreover, mask cost is only a fraction of the total NRE cost. The design and verification costs are also sky-rocketing as chip design is becoming more and more complex.

- Because of the shrinking of feature size, transistors can switch very fast, and are thus no longer the performance bottleneck. However, the RC delay of long wire does not scale down proportionally [6]. Signals can no longer propagate along long wires in one clock cycle [7]. One implication of this wire delay problem is that, because of clock skew, it is getting harder and harder to synchronize the whole chip at high clock frequency [8]. Even if technically possible, increasing clock frequency will no

longer be a feasible approach to achieve high performance, because of prohibitive power dissipation.

- Power dissipation has been a recurring problem since the early days of semiconductor industry. Integrating more transistors on a single chip will increase the power density, because more transistors switching simultaneously will cause more dynamic power dissipation. Moreover, in the deep sub-micron era, leakage power is no longer a second order effect. In the future, leakage power will even contribute more to total chip power dissipation than dynamic power [9] [10].

- Related to the power dissipation problem, energy efficiency is becoming a top design consideration for extending the operating period of small portable information appliances operating on batteries, and for reducing the utility cost of large data warehouses consisting of thousands of servers [9].

- Yet another everlasting problem is the memory bottleneck. While the density of DRAM quadruples in three years, even faster than the increase of logic density, the speed of memory cannot catch up the speed of logic. Putting more memory on chip does not necessarily solve the memory bottleneck problem, due to the wire problem and the limitation on the number of memory access ports.

The semiconductor industry will not stall building more powerful hardware because of these problems. Instead, people are developing innovative hardware architectures to more efficiently use the coming billion transistors [11].

1.2 Hardware Trend

The state-of-the-art hardware systems are composed of ASICs (Application Specific Integrated Circuits) and/or programmable devices like digital signal processors and microprocessors. The goal of hardware design is to achieve a balance among performance, cost, and flexibility for the target applications. Technology trend profoundly affects how people build hardware systems to maintain this balance.

For example, traditional standard cell based ASIC design is being challenged as a cost-effective approach to achieve low power and high performance, because of soaring NRE cost, high design risk and constantly changing industry standards. For applications which microprocessors and digital signal processors still cannot meet the performance, power, and area requirements, people are seeking alternatives like structured ASIC, FPGAs (Field Programmable Gate Arrays), and reconfigurable architectures, to replace standard cell based ASICs. These alternatives promise lower cost and/or more flexibility, without sacrificing too much performance [12] [13].

The technology trend is also challenging the conventional wisdom in microprocessor design. Because the centralized organization of current high-performance microprocessors does not scale well with the advance of semiconductor manufacturing technology, researchers are proposing alternative architectures like the M.I.T. RAW processor [14], the Stanford Stream processor [15] and Merrimac machine [16], and the U.T. Austin TRIPS processor [17], to address the issues faced by future billion transistor microprocessors [18].

For commercial microprocessors, the design objective now is not performance, but performance *per Watt*. Instead of increasing clock frequency, which will incur too much power dissipation, both Intel and AMD are shipping dual-core microprocessors and will resort to multi-core architectures to achieve high performance in the future [19] [20] [21]. The Cell processor developed by Sony, Toshiba and IBM also adopts multi-core architecture, consisting of one PowerPC Processing Unit and 8 Synergetic Processing Units for SIMD processing [22] [23].

Although microprocessors have been making significant progress in performance and will be more power efficient in the future, I believe general-purposed architecture alone is not the most efficient hardware platform. Future system on chip will consist of multiple general-purposed cores and application specific accelerators in order to power efficiently and cost effectively meet the requirements of emerging applications.

1.3 Application Trend

In the past, the growth of semiconductor industry is driven by PCs (Personal Computers) and desktop applications. As the analog world is gradually digitized, and more and more richer and richer digital contents are delivered through the Internet, (portable) telecommunication, multimedia, and gaming applications are replacing PC desk-top applications as the new driving applications.

These applications present much higher design challenges than traditional PC desktop applications because 1) they require much higher computing power for complicated

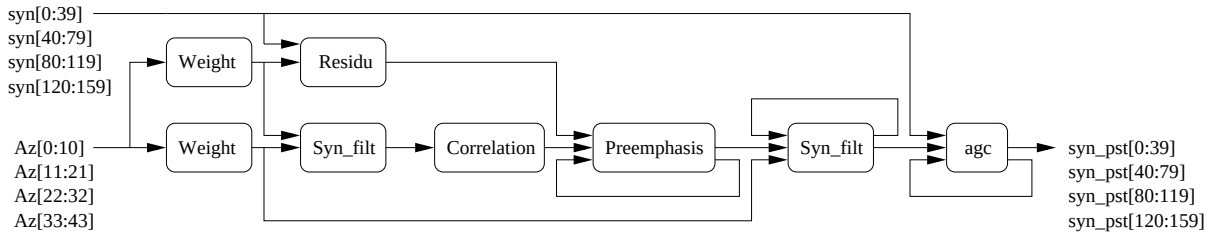


Figure 1.1 The block diagram and data-flow of the post-filter of G.724 decoder

algorithms to, for example, analyze and synthesize audio and video streams; 2) they impose much tighter design constraints on form factor, cost, power dissipation and energy efficiency.

These applications usually consist of DSP (Digital Signal Processing) kernels, with inputs and outputs of sequences of frames. Usually an input frame is further divided into sub-frames or blocks, which are then individually processed by the DSP kernels. So potentially there is abundant parallelism in processing these sub-frames or blocks.

As a simple but concrete illustrating example, Figure 1.1 shows the components and data-flow of the post-filter used in the G.724 decoder [24]. The 160-bit input speech frame `syn[0..159]` is divided into four 40-bit sub-frames to be individually processed by the post-filter.

Parallelism also exists in each computation kernel. The **Weight** block is basically a vector multiplication, scaling its input signals by different weights. The **Residu** block is a FIR (Finite Impulse Response) filter and the **Syn_filt** and **Preemphasis** blocks are IIR (Infinite Impulse Response) filters. The **Correlation** block computes two auto-correlations. The **agc** block for automatic gain control is a little more complicated, but the basic computations are still vector multiplication and accumulation.

To meet the, often conflicting, design requirements, it is necessary to exploit all the possible inherent parallelism in these applications.

1.4 Exploiting Parallelism

For the post-filter shown in Figure 1.1, potentially we can at least exploit the following parallelism.

- Frame level data parallelism. If there is no data dependence between the processing of consecutive frames, we could potentially duplicate the hardware to post-filter different frames in parallel.

- Sub-frame level data parallelism. If there is no data dependence between the processing of consecutive sub-frames, we could potentially duplicate the hardware to process each sub-frames in parallel.

- Sub-frame level function parallelism. Instead of duplicating hardware, we could pass the sub-frames through the DSP kernels in a pipelining or data-flow fashion to exploit the coarse-grained function parallelism among these kernels.

- Signal level data parallelism. For digital signal processing kernels, we could use techniques like Intel MMX/SSE [25] [26] to exploit fine-grained data parallelism.

- Instruction/operation level parallelism. We could implement these kernels using state-of-the-art high-performance digital signal processors or microprocessors, which exploit instruction level parallelism to speed up the execution. We could also

design ASIC to directly map the operations of these kernels to parallel arithmetic units.

In spite of its abundant parallelism, the post-filter contributes only about 50% of the total G.724 decoder execution time. According to Amdahl's law [27], the performance of the G.724 decoder cannot be significantly improved without speeding up the other 50% of its computation, which may exhibit different characteristics from the post-filter and thus require different approaches to improving performance.

It is no surprise that people build today's telecommunication and media applications using an array of hardware components, from ASIC and DSP (Digital Signal Processor) to micro-controller and microprocessor, exploiting coarse-grained and fine-grained, data and function parallelism to balance performance and cost.

Partitioning complex software into concurrent tasks, exploiting different forms of parallelism, mapping these tasks onto complex hardware and searching for a balance point between performance and cost is a daunting task. However, the current design practice mainly relies on designer's experience and instinct. With shorter time-to-market and product lifetime, the development of future applications needs more efficient and systematic design methodology.

1.5 Improving Design Productivity

The exponential increase of transistor density is followed by the exponential increase of hardware and software complexities. However, we cannot exponentially improve our

productivity using the same design methodology. To boost productivity, we shift to higher level design abstraction to hide complexity. In the past, software design moved from assembly language programming to high-level language programming; hardware design moved from gate-level design to RTL (Register Transfer Level) design. However, abstraction alone cannot achieve paradigm shift. We need the tools that can translate designs from higher level representation to lower level representation without sacrificing too much design quality. The success of the first high-level programming language Fortran is because of the accompanying good Fortran complier; the success of Verilog/VHDL is because of good RTL synthesis tools.

In summary, Figure 1.2 depicts the big picture of the problem domain that this dissertation is trying to make some small contributions. The problem is two-fold.

- What is the programming model for capturing complex emerging applications in a compact representation? To improve design productivity, the programming model must be simple. To cover wide range of applications, the programming model must be versatile.

- What are the compiler techniques to extract parallelism out of the compact representation, and to map concurrent tasks onto complex multi-core hardware? The complex hardware will consist of multiple general-purposed microprocessors, ASICs and even FPGAs.

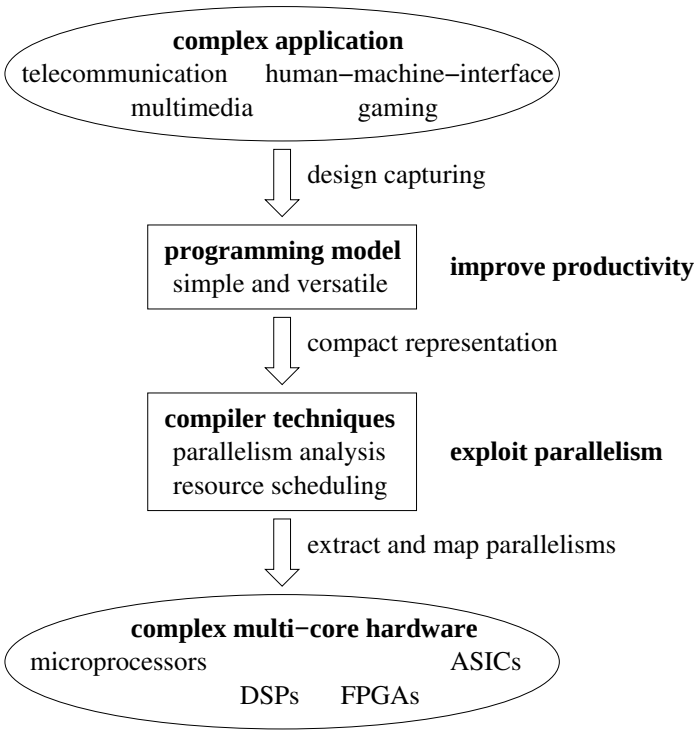


Figure 1.2 The challenge of the design methodology community.

I realize that this is not a new research topic. Many researchers have made great contributions before. The next chapter will scan the previous works, trying to find an empty slot in the book shelf for this dissertation.

CHAPTER 2

Previous Work

Exploiting parallelism and boosting productivity are the recurring challenges of the IT industry, especially when the advance of technology accumulates enough momentum to make a hardware architecture leap, or to surpass the existing design methodology. This chapter will review the previous works on optimizing compilers, high-level synthesis, and concurrent languages. Although they take different approaches, or target different hardware platforms, all these three areas concern how to exploit parallelism and boost productivity.

2.1 Optimizing Compilers

Compiler optimization is an active and exciting research area. Researchers have been innovating new techniques to efficiently implement new programming language constructs and to effectively utilize new architecture features. We can roughly divide optimizing compilers into two categories, vectorizing/parallelizing compilers targeting supercomputers [28] [29] and optimizing compilers targeting super-scalar, VLIW (Very Long Instruction Word) or EPIC (Explicitly Parallel Instruction Computing) architectures [30] [31]. Great progresses have been made in these two areas.

2.1.1 Vectorizing Compilers

Early vectorizing compiler researches [32] [33] [34] [35] [36], most notably the Parafrase project at the University of Illinois [37] and the Parallel Fortran Converter project at Rice University [36], not only formalized fundamental notions like data dependence, dependence distance, dependence direction, and dependence level, but also pioneered dependence test techniques for automatically identifying the inherent parallelism in sequential programs. Furthermore, to enable more vectorization and to better utilize the underlying hardware features, these ground-breaking works also invented program restructuring techniques [38] [39] [40] [28], for example, loop interchanging [41] [42], loop skewing [43], scalar renaming [44], array renaming [36], strip-mining, and vector register allocation [45].

Although these early vectorizing compiler works focused on exploiting fine-grained data parallelism to speed up scientific computations on vector or SIMD (Single Instruction Multiple Data) machines, they also laid the foundation for the parallelizing compilers targeting MIMD (Multiple Instruction Multiple Data) machines, and more recently for the vectorizing compilers targeting instruction sets like the Intel MMX and SSE [26] for speeding up multimedia applications on microprocessors.

2.1.2 Parallelizing Compilers

Because MIMD machines usually have high inter-processor communication cost, parallelizing compilers targeting MIMD machines must look beyond the inner-most loop to seek more coarse-grained parallelism in the outer loops [46] [47] [48] [49] [50] [51] [52].

To maximize parallelism and to increase locality, researchers have developed many program analysis and transformation techniques, for example, loop distribution [28], loop fusion [46] [53], loop tiling [54], unimodular transformation [55], array data-flow analysis [56] [57] [58] [59], and array privatization [60] [61] [60] [62] [63].

Because parallelizing compilers need to examine larger program regions for parallelism, many analyses need to cross the procedure boundaries to get more accurate analysis results. Because full program in-lining is too costly, researchers have developed many inter-procedural analysis techniques [64] [65] [66] [67] [68] [69] [70] [71].

Most of the parallelizing compiler works are based on the SPMD (Single Program Multiple Data) model to exploit coarse-grained data parallelism. This is suitable for scientific applications with data set much larger than the number of processors. However, researchers found that SPMD alone may not be the best way to parallelizing applications like many digital signal processing applications which have many kernels with small working set. For this kind of applications, it is better to exploit function (or task) parallelism in addition to data parallelism [72] [73].

In data parallelism, different processors (or functional units) execute the *same* program (or function) on different data at the same time. In function parallelism, different processors (or functional units) execute *different* programs (or functions) on different data at the same time. Researchers have developed techniques for task scheduling and resource allocation given the dependence or data-flow among the tasks [74] [75] [76].

Unlike parallelizing compilers targeting MIMD machines, which must exploit *coarse-grained* data and/or function parallelism in order to avoid excessive costly inter-processor

communication, optimizing compilers targeting high performance microprocessors exploit instruction level parallelism, which can be classified as *fine-grained* function parallelism.

2.1.3 Superscalar/VLIW/EPIC Compilers

High performance microprocessors are capable of executing multiple instructions at the same time. People have made micro-architecture and compiler innovations to increase the number of instructions available for parallel execution.

For example, Tomasulo's algorithm [77], which is widely used in superscalar microprocessors, eliminates false dependencies among instructions by register renaming [78] [79]; branch prediction [80] [81] [82] [83], trace cache [84] [85], predication [86] [87] [88], speculation [89] [90] [91], and memory disambiguation [92] enable more parallel instruction execution by eliminating the synchronization barriers caused by spurious control dependencies and memory dependencies.

Often, the micro-architecture features for exploiting ILP (Instruction Level Parallelism) rely on compiler supports to achieve better utilization. For example, to expose and schedule more instructions for parallel execution, people have developed trace scheduling [93], superblock formation [94], software pipelining, modulo variable expansion and modulo scheduling [95] [96] [97]; to enable more effective predication, people have developed hyperblock formation and predication analysis [98] [99]; to support speculation, people have developed sentinel scheduling [100]; to obtain more accurate compile-time memory disambiguation, people have been improving the accuracy and efficiency of dependence tests [101] and pointer analysis [102] [103].

It is due to the micro-architecture and compiler innovations combined, and always being manufactured using the cutting-edge process technology, that microprocessors can make such impressive progress in performance and cost. However, general-purposed architecture still cannot meet the performance and cost requirements of many applications. Many products still rely on special hardware to achieve the required performance under strict cost and power constraints.

2.2 High-level Synthesis

Hardware designers have long been exploiting parallelism to improve the performance and efficiency of their products. However, designing hardware at circuit level or gate level is tedious and difficult. Designers must determine circuit topology, size transistors, optimize logic, synchronize signals with respect to clocks and perform circuit or logic simulations for functional verification and for timing analysis. As circuits become larger and larger, it is very time consuming to capture and verify the whole design at such low level.

To improve design productivity, people developed hardware description languages like Verilog and VHDL as well as RTL synthesis tools. The hardware description languages essentially abstract hardware as a hierarchy of concurrent processes following an event-driven execution model. Instead of drawing schematics, designers can now capture their designs using hardware description languages just like writing software programs, or more precisely concurrent programs. The RTL synthesis tool will then take the high-

level descriptions along with user specified design constraints, perform all the low level design activities, and finally generate a netlist ready for the place-and-route tool [104]. This enables designers to focus on RTL and architecture level design exploration and verification.

However, RTL designers still need to take care of details like circuit reset, clock synchronization and dividing critical timing path into several pipeline stages, as well as explicitly expressing fine-grained and/or coarse-grained, function and/or data, parallelism as a hierarchy of concurrent processes with bit-level or word-level interprocess communication signals. In other words, the designers still need to describe the design *structurally*, not *behaviorally*. As ASICs are getting more complex, RTL design is also becoming too time-consuming. We are again facing the productivity crisis.

Researchers are advocating moving to even higher design abstraction and high-level synthesis [105] [106] [107]. Starting from an abstraction like data flow graph [108], which describes the dependences between fine-grained or coarse-grained tasks, people have done extensive researches on how to optimize the mapping of concurrent tasks onto hardware building blocks.

There are already commercial tools that can take C programs and generate the corresponding RTL implementation [109] [110] [111] [112]. Although the users of these tools can describe their design behaviorally, in order to obtain better synthesis results, they still need to explicitly express parallelism, especially coarse-grained function parallelism, as well as the inter-process communication mechanism using compiler directives or concurrent language constructs.

2.3 Concurrent Programming Languages

In addition to the techniques that extract parallelism from sequential programs, vectorizing and parallelizing compiler researchers also developed compiler directives and language constructs to let programmers explicitly express parallelism. For example, Fortran-D [113] and High Performance Fortran (HPF) [114] extend the Fortran language with vector operations and data partitioning directives for explicit data parallelism on top of a shared memory model; the MPI standard [115] is proposed as a portable library for explicit inter-process communication under the message-passing paradigm. While it is natural to target shared-memory programs on shared-memory multiprocessors, and message-passing programs on distributed memory multicomputers, the memory model of a concurrent programming language is not tightly coupled to the memory organization of the underlying machines. It is up to the compiler and the run-time system to bridge the semantic gap.

In addition to vector, SIMD, shared-memory MIMD, and distributed-memory MIMD machines, researchers also experimented data-flow supercomputers [116] [117] [118] to exploit massive parallelism. In parallel with the development of data-flow machines, researchers also designed data-flow languages [119] [120] for explicitly expressing fine-grained function parallelism. Different from a program written in imperative languages, a program written in data-flow language is side-effect free and each of its variables has only single assignment.

Interestingly, researchers also developed compiler analyses and transformations that can translate an imperative program to a form with some data-flow properties. For example, there exists efficient algorithms to translate an imperative program into the SSA (Single Static Assignment) form [121], gated SSA form [122], or dependence web [123]. Researchers [124] even argued that it is not necessary to design data-flow languages for data-flow machines, because imperative programs can obtain the same performance on data-flow machines using advanced compiler techniques, and the compilers for both types of languages have similar complexities. Also, the von Neumann programming model of imperative languages could be more intuitive and result in more compact programs than the data-flow programming model for some applications, especially for applications with a lot of partial state changes in complex data structures.

Because of these and other reasons, in spite of their many creative concepts, data-flow languages did not become mainstream [1]. The dominating programming languages today are still imperative languages. Instead of for expressing massive parallelism in general applications, recent data-flow language researches are more for software engineering purpose [120] and for specific application domains.

For example, to model DSP applications, researchers have developed formal representations like synchronous data flow [125] and data-flow process networks [126]. In these models, a task or a process, which could be an imperative program, represents a DSP kernel which is repeatedly applied to its input signals. Also explicitly expressed in these

[1] Neither did general purposed data-flow machines. Instead, it is the restricted data-flow model [78] that prevails in commercially successful high-performance microprocessors.

models are the signal flow among these tasks and the signal generating and consuming rates of each kernel. The motivation for these formalisms is to enable automatic synthesis and optimization of real systems from the models [127] [128] [129] [130] [131].

With similar motivation, and language semantics, researchers also developed streaming languages like StreamIt [132] and Brook [133] to ease the programming for machines like the MIT RAW machine [14], the Stanford Merrimac [16] or even graphics processors [134]. The fundamental concepts of these streaming languages are *stream* consisting of possibly infinite number of independent data, and *kernel* (or *filter*) operating on streams. Thus, a streaming program explicitly expresses the function parallelism among the execution of kernels, as well as the data parallelism among the processing of stream elements.

The previous works on exploiting parallelism and on boosting productivity are really tightly correlated, and we can unify them in a single picture, as shown in Figure 2.1. Figure 2.1 can be divided into two halves. The top half is extracting parallelism from the applications by compilers, or expressing parallelism in the applications by software programmers or hardware designers. The bottom half is mapping concurrent tasks onto hardwares exploiting various types of parallelism. Each edge in Figure 2.1 corresponds to the enormous amount of knowledge and techniques obtained in decades of compiler, high-level synthesis and programming language researches. The next chapter will discuss where my work will make a dent in this big picture, considering both the learned lessons and the projected trends.

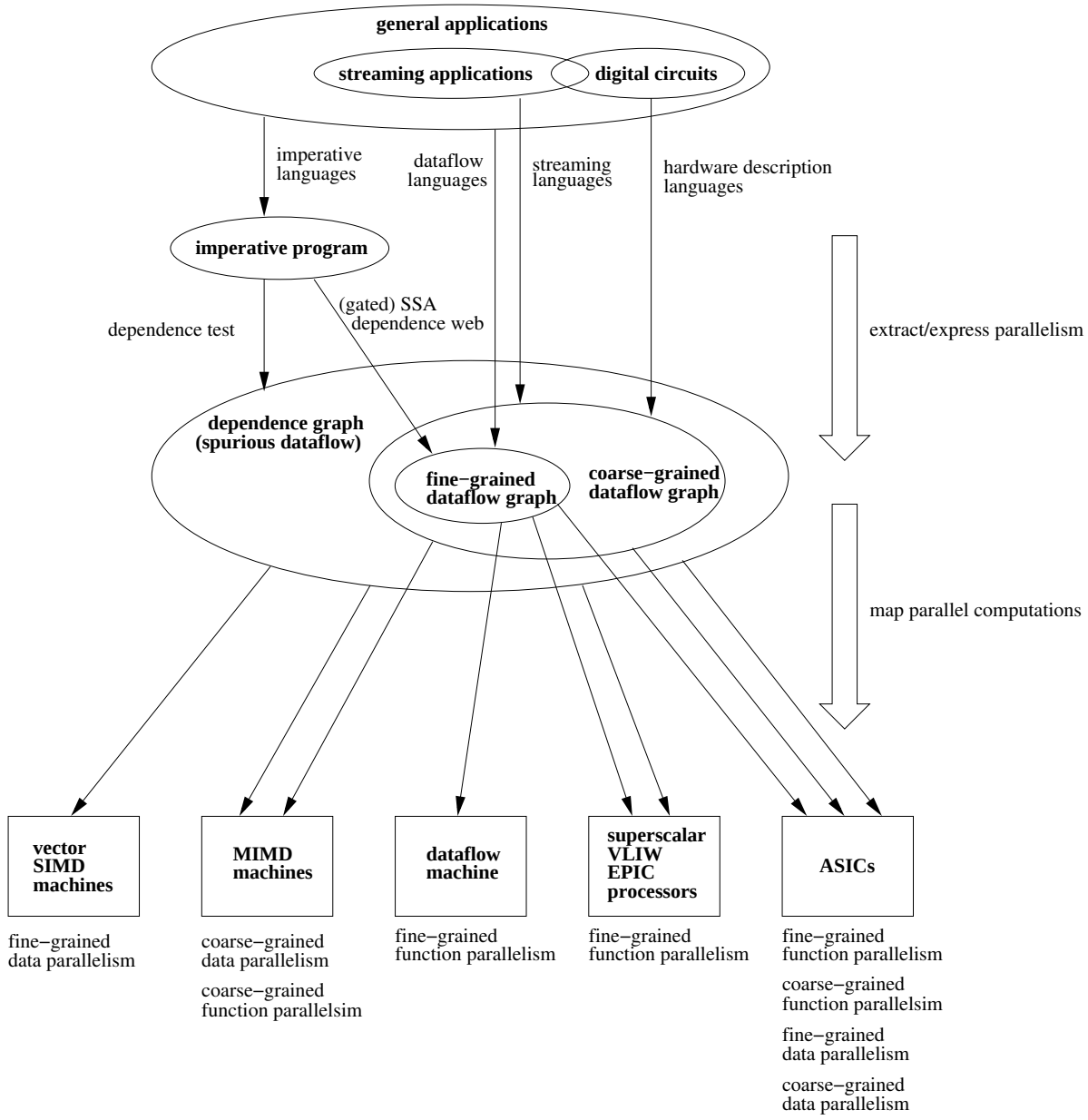


Figure 2.1 A unified view of exploiting parallelism and boosting productivity

CHAPTER 3

Thesis Overview

This chapter will serve two purposes. First, it will establish the problem statement of my PhD research based on the reflection on the technology, hardware and application trends discussed in Chapter 1 and the previous works on exploiting parallelism discussed in Chapter 2. Second, it will discuss what sub-problems we need to solve and give an overview of the remaining chapters of this dissertation.

3.1 Problem Statement

Figure 3.1 relates the previous works with the perceived multi-core architectures. Many of the works people have done for partitioning and distributing computations onto MIMD machines can be readily used for exploiting coarse-grained data and function parallelism for the multi-core architecture. For efficiently utilizing superscalar/VLIW/EPIC cores and SIMD/vector execution units, researchers have already developed a lot of techniques, and are keeping pushing the envelope. Very likely, the coming multi-core architecture will also include ASICs or coprocessors to efficiently accelerate applications [135]. The CAD community have been innovating more powerful tools to facilitate the development of these accelerators.

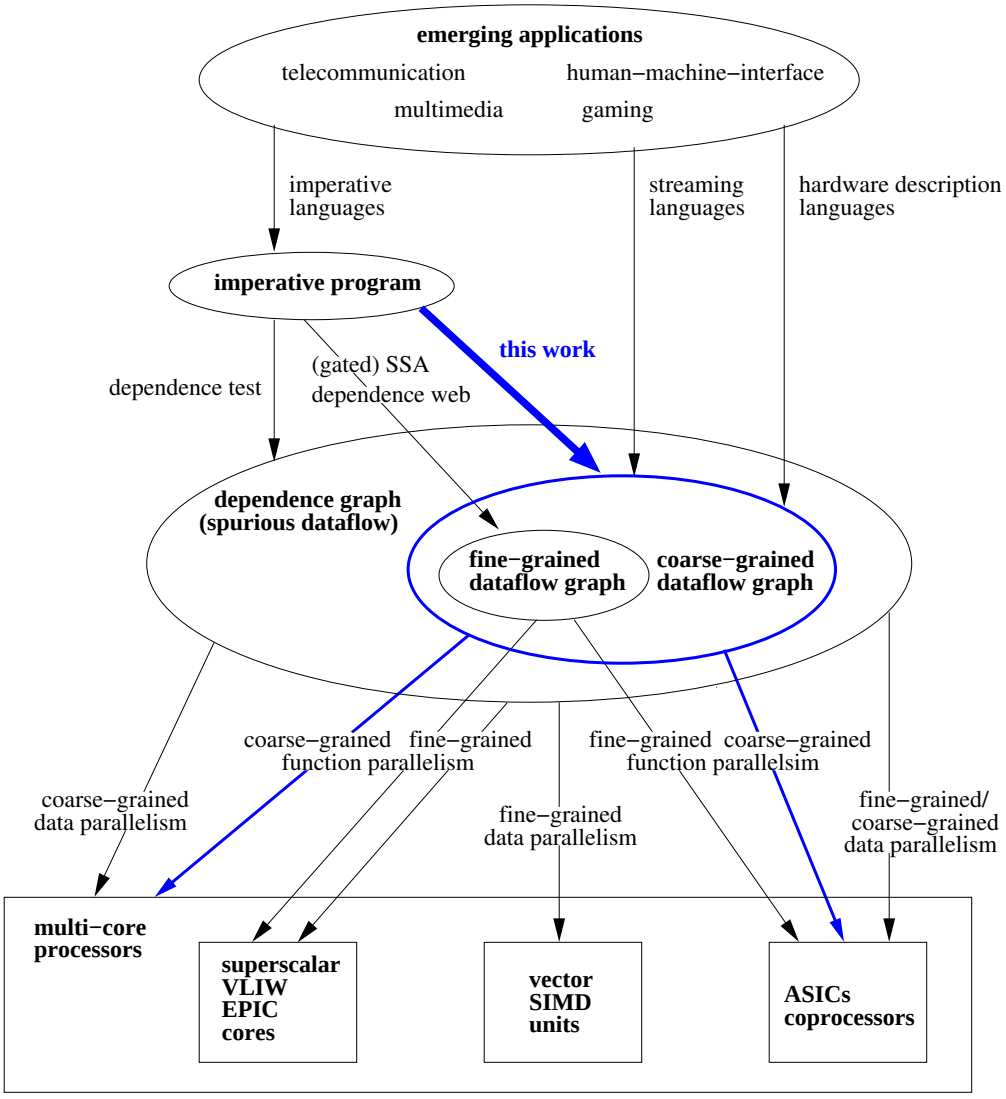


Figure 3.1 The position of this work with in mapping applications onto multi-core architectures

The works of mapping concurrent tasks onto multi-core architectures are all based on an abstraction, the dependence graph, which describes the partial order between the execution of computational tasks. Researchers have been pushing the accuracy of dependence test. There already exists exact data dependence test, the Omega test [101], which is very efficient for common cases. Because of the way they are constructed,

dependence graphs may contain many false dependences. While false dependences may not affect the effective accuracy of the dependence graph for compiler applications like vectorization, the removal of false dependences can improve the effectiveness of many other compiler optimizations [136].

Dependence graph without false dependences can be called data-flow graph, because it contains only the true data dependences, or data-flows, between computational tasks. A data-flow graph can be fine-grained, with each node corresponding to basic operations like addition, or it can be coarse-grained, with each node corresponding to more complicated computations like filters. Because they expose the maximum available parallelism, data-flow graphs are instrumental in high-level synthesis and in mapping tasks onto array of processors, and also the "programs" for the data-flow computation model.

It is indisputable that the data-flow model is ideal for building hardware, because of its localized memory access, neighboring communication, and maximum parallelism. Indeed it has been the model for designing high performance ASICs like DSP circuits [137]. For the perceived multi-core architecture, the data-flow model will also play an important role not only in building the accelerators, but also in core-to-core, core-to-accelerator, and accelerator-to-accelerator communications through the on-chip interconnection network.

However, as discussed in Chapter 2, there are two schools of thoughts about how to construct the data-flow graph. The first school of thought is to let the programmers write programs using data-flow or streaming languages. The second school of thought is to let the programmers write programs using conventional imperative languages, and

the compilers translate the imperative programs into data-flow graphs. This thesis work follows the second school of thought for the following reasons.

- There have already existed a huge code base written in imperative languages. As time goes by, more and more important imperative programs will be developed. These imperative programs will still need to run efficiently on future multi-core processors.

- The von Neumann programming model of the imperative languages is widely applicable. Many complicated applications have been written in imperative languages based on the von Neumann programming model. On the other hand, the data-flow or streaming languages are still in the stage of proving concepts. If we could develop a program analysis system to extract data-flow from imperative programs, the need for developing new data-flow or streaming languages, as well as the associated tool chains, will be questionable.

Because there are already efficient algorithms to convert imperative programs to fine-grained data-flow graphs [138] [123] [121] [139], and because exploiting coarse-grained parallelism will become more and more important for future multi-core processors, this thesis work will focus on extracting coarse-grained data-flows from imperative programs to facilitate the exploitation of coarse-grained function parallelism in multi-core processors, as indicated in Figure 3.1. More specifically, this thesis work will target programs written in the C language, partly because of the popularity of C and partly because of the

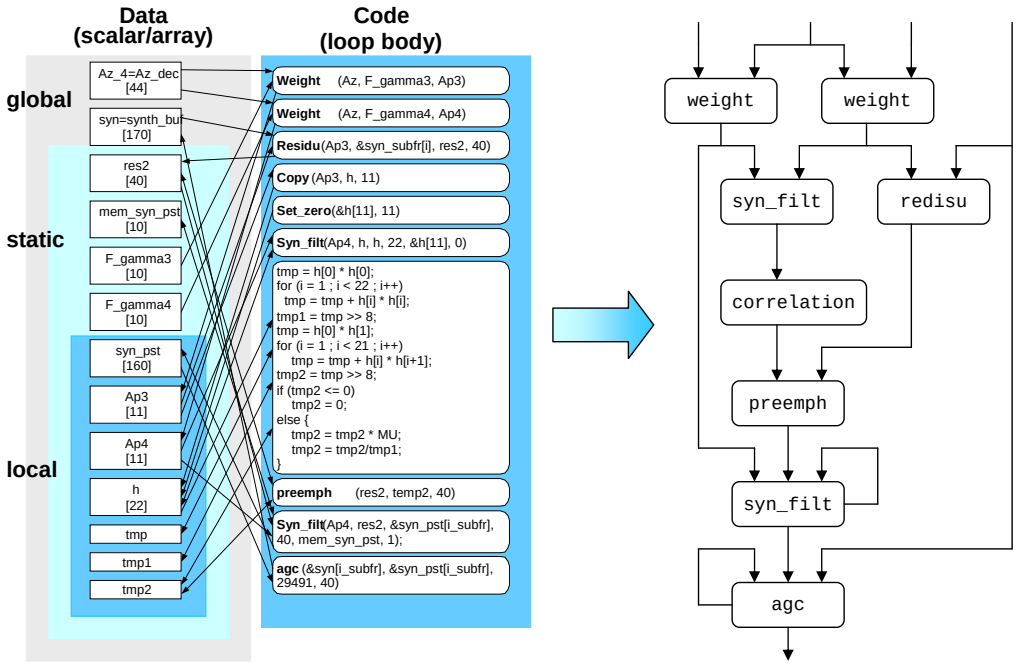


Figure 3.2 Illustration of the problem statement using the post-filter of G.724 decoder

compiler infrastructure used for prototyping and experiments. However, the techniques developed in this work could also be applied to other imperative languages.

To specifically illustrate the problem that this work is to solve, Figure 3.2(a) shows the original C code and the corresponding memory accesses of the G.724 post-filter example presented in Figure 1.1. Note that the original program accesses both memory objects statically allocated in the global memory and memory objects dynamically allocated in the stack and the heap memory. These memory objects are often shared by different functions. The challenge is to sort out the memory data-flow as shown in Figure 3.2(b) from the complicated memory accesses as shown in Figure 3.2(a).

In summary, the problem statement of this research is as follows.

Problem statement: Building a program analysis system to extract coarse-grained data-flow from C programs for exploiting coarse-grained function parallelism.

This concludes the *philosophy* part, and start the *engineering* part, of this PhD dissertation.

3.2 Fine-grained Analogy

To obtain some insights on how to extract coarse-grained data-flow from imperative programs, this section will use Figure 3.3 to review how fine-grained data-flow is extracted from imperative programs to exploit fine-grained function parallelism.

By pairwise comparison of variable reads and variable writes in the code segment of Figure 3.3(a), we can construct the dependence graph shown in Figure 3.3(b). Each dependence is annotated with the corresponding dependence distance. Note that the dependence distance is an interval, not necessarily a single integer number [101]. For clarity, only the lower bound of the dependence distance is shown in Figure 3.3(b).

These dependences prevent the parallel execution of instructions in the same iteration and/or in different iterations. However, many of the dependences in Figure 3.3(b) are false dependences caused by writing to the same variable `a`. If each dynamic instruction writes to a different memory location, we can eliminate all the false dependences and obtain the maximum parallelism which is only constrained by the true dependences and hardware resources, as shown in Figure 3.3(c) [1].

[1] Here we assume there are 1 adder, 1 multiplier, and 1 divider.

Superscalar processors achieve this by performing the architecture register to physical register renaming on the fly [79]. Renaming can also be done using compile-time techniques like SSA [121], which can easily convert the loop body in Figure 3.3(a) to the data-flow graph in Figure 3.3(d). To obtain better instruction scheduling results, software pipelining [95] or modulo scheduling [96] also perform register renaming using techniques like modulo variable expansion [95] to allocate different registers to instructions in different iterations.

The key to exploiting fine-grained function parallelism is really to extract the data-flow between instructions by eliminating false dependences through renaming. Essentially there are three issues in extracting data-flow for function parallelism.

- Defining *function*. For fine-grained function parallelism, a function is an instruction or an operation.

- Identifying the memory storages accessed by each function. For instructions operating on registers, the accessed memory storages can be identified using the specified register numbers for the source and the destination operands.

- Identifying the producer and consumer relation between functions. Superscalar processors use hardware structures like RAT (Register Alias Table) to establish the producer and consumer relation between instructions at run-time. Compilers identify the producer and consumer relation by performing data-flow analysis or by SSA construction.

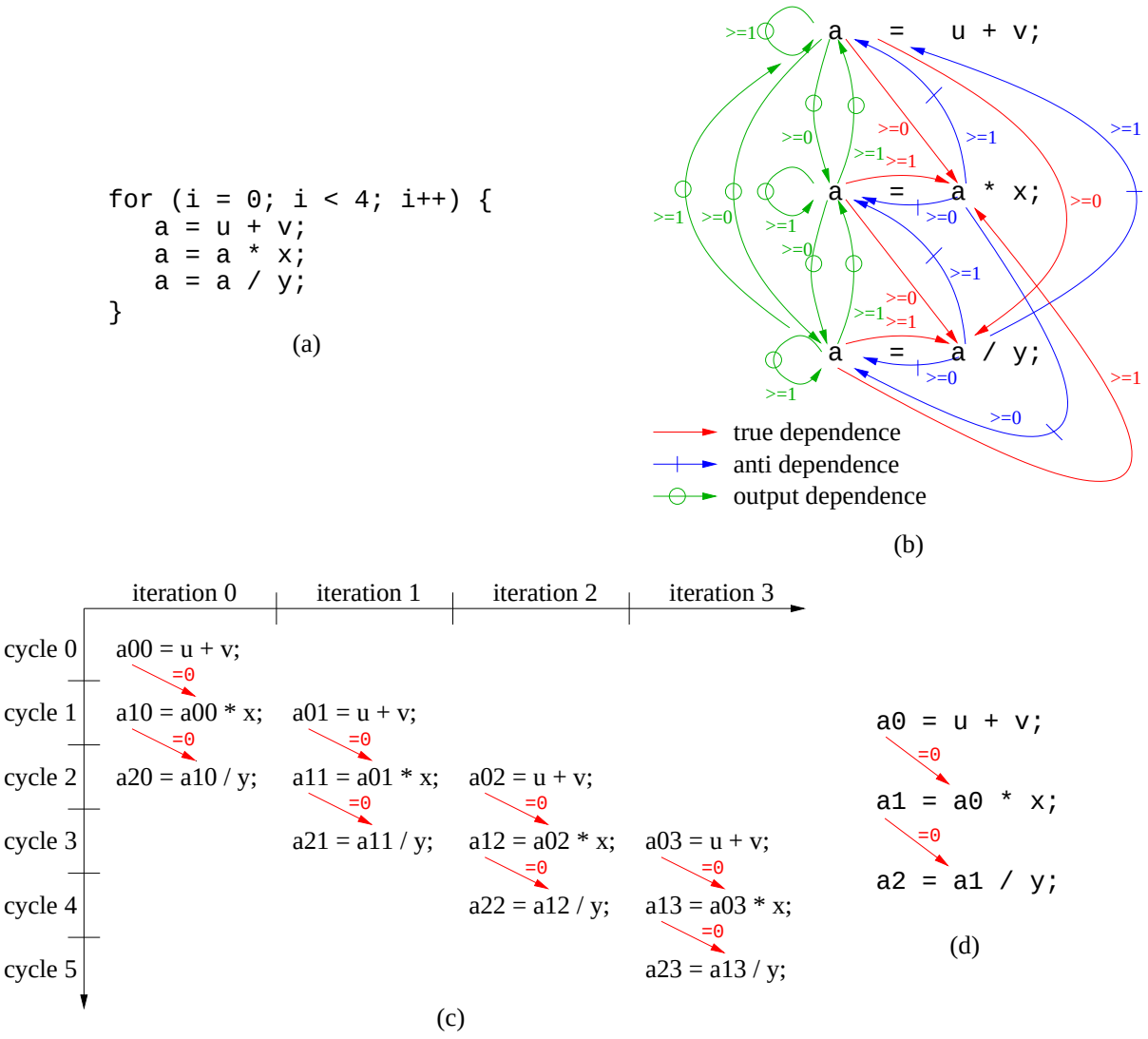


Figure 3.3 Example illustrating extracting fine-grained data-flow

The next section will address these three issues in the context of extracting coarse-grained data-flow to exploit coarse-grained function parallelism.

```
 1:    int A[40];
 2:
 3:    foo0 (...)
 4:    {
 5:        for (i0 = 0 ; i0 <= 3 ; i0++) {
 6:            foo1 (A, ...);
 7:            foo2 (... A, ...);
 8:            foo3 (... A);
 9:        }
10:    }
11:    foo1 (short y[], ...)
12:    {
13:        for (i1 = 0 ; i1 <= 39 ; i1++)
14:            y[i1] = ...
15:    }
16:    foo2 (..., short s[], ...)
17:    {
18:        for (i2 = 0 ; i2 >= 0 ; i2--)
19:            s[i2] = s[i2] ...
20:    }
21:    foo3 (... short x[])
22:    {
23:        for (i3 = 0 ; i3 <= 39 ; i3++)
24:            ... = x[i3] ...
25:    }
```

Figure 3.4 Example coarse-grained functions of subroutine calls

3.3 Coarse-grained Issues

This section will examine the issues in extracting coarse-grained data-flow from imperative programs to exploit coarse-grained function parallelism. The discussion will follow the three issues summarized in the previous section. As explained in the following sections, extracting coarse-grained data-flow is much more difficult than extracting fine-grained data-flow.

3.3.1 Defining Coarse-Grained Function

To exploit coarse-grained function parallelism, we must first define what is *coarse-grained function*, then we can discuss how to execute *coarse-grained* functions in parallel.

```
 1:    int A[40];
 2:
 3:    foo0a (...)
 4:    {
 5:        /*
 6:         * loop0a:
 7:         */
 8:        for (i0 = 0 ; i0 <= 3 ; i0++) {
 9:            /*
10:             * loop1a: write A[0..39]
11:             */
12:            for (i1 = 0 ; i1 <= 39 ; i1++)
13:                A[i1] = ...
14:            /*
15:             * loop2a: read A[39..0], write A[39..0]
16:             */
17:            for (i2 = 39 ; i2 >= 0 ; i2--)
18:                A[i2] = A[i2] ...
19:            /*
20:             * loop3a: read A[0..39]
21:             */
22:            for (i3 = 0 ; i3 <= 39 ; i3++)
23:                ... = A[i3] ...
24:    }
```

Figure 3.5 Example coarse-grained functions of loops

For the program segment in Figure 3.4, it is natural to consider the subroutine calls to `foo1`, `foo2` and `foo3` as *coarse-grained functions*. For the program segment in Figure 3.5, we may consider each inner loop as a *coarse-grained function*. Coarse-grained function is really not as well defined as fine-grained function. While subroutine calls and loops are natural candidates for program regions, there could be other ways to divide a program into regions, or coarse-grained functions.

Ideally we would like each program region, or coarse-grained function, is side-effect free and accesses most of its data in local memories. We would also like to partition a program in such a way that communication between program regions is localized in the memories only accessed by the two communicating program regions. Ideally we would like to partition a program into program regions in so that we could generalize the fine-

grained data-flow execution model to a coarse-grained data-flow execution model, and maximize the available coarse-grained function parallelism.

But this rarely happens in imperative programs which often use global variables for the communication between many program regions. To convert imperative programs into coarse-grained data-flow programs, a more practical approach is to sort out the producer and consumer relation between program regions and then convert global memory accesses to local memory accesses, as discussed in the next section.

3.3.2 Identifying Producer and Consumer Relation

Consider the example in Figure 3.5, which has three inner loops as coarse-grained functions, all accessing the same array `A`. As shown in Figure 3.6(a), we can speed up the execution of `foo0a` using three hardware accelerators for `loop1a`, `loop2a`, and `loop3a`, with a memory block for the communication between these three accelerators, just as the software implementation in the original program. This may speed up the execution of individual innder loop, but there is not too much overlap between the execution of accelerators as illustrated in Figure 3.6(a). Note that `loop1a` at outer loop iteration i can not start writing to `A[0]` until `loop3a` at outer loop iteration $i-1$ finishes reading the value of `A[0]` generated by `loop2a` at outer loop iteration $i-1$.

Similar to the example in Figure 3.3, the problem here is that both `loop1a` and `loop2a` write to the same array `A`. If we can use different buffers for `loop1a` and `loop2a` at different outer loop iterations, like renaming variables in Figure 3.3(c), we can increase

the overlapping between the execution of `loop1a`, `loop2a`, and `loop3a` at different outer loop iterations, as illustrated in Figure 3.6(b).

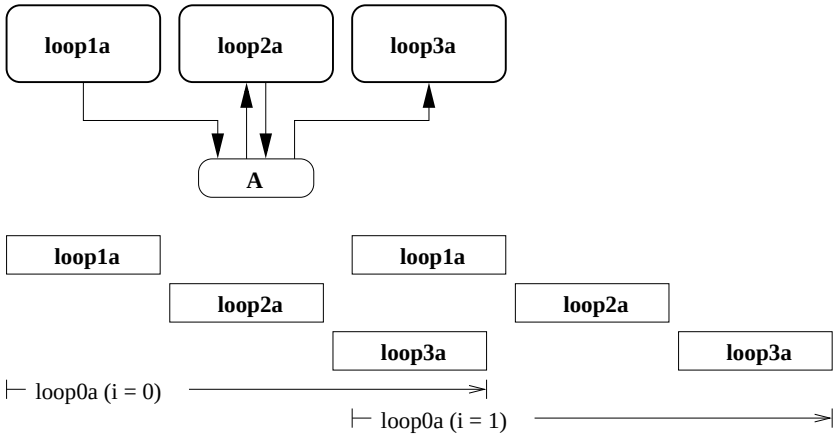
Basically, we can uncover more inherent coarse-grained function parallelism by separating the memory data-flow between `loop1a` and `loop2a` from the memory data-flow between `loop2a` and `loop3a`. However, this is possible only if we can prove the following.

- *All* the array `A` elements consumed by `loop2a` are produced by `loop1a` at the same outer loop iteration.
- *All* the array `A` elements consumed by `loop3a` are produced by `loop2a` at the same outer loop iteration.

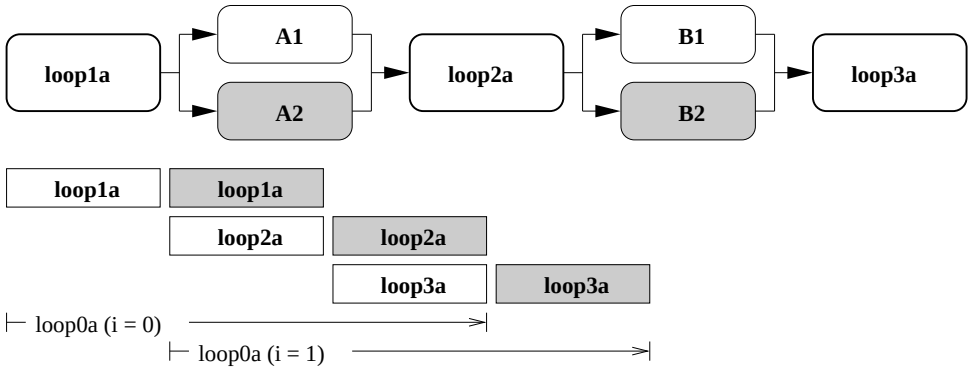
The proof for this simple example is trivial. Note that the `loop1a` produces the set of array `A` elements $\{\texttt{A[i]} | 0 \leq \texttt{i} \leq 39\}$, which is also the set of array `A` elements consumed by `loop2a`. Similarly, the same set of array `A` elements are produced and consumed by `loop2a` and `loop3a` respectively. However, in general it is not easy to identify the producer and consumer relation between coarse-grained functions, because determining the exact memory locations accessed in a coarse-grained program region is not as easy as in the fine-grained case.

3.3.3 Summarizing Coarse-grained Memory Accesses

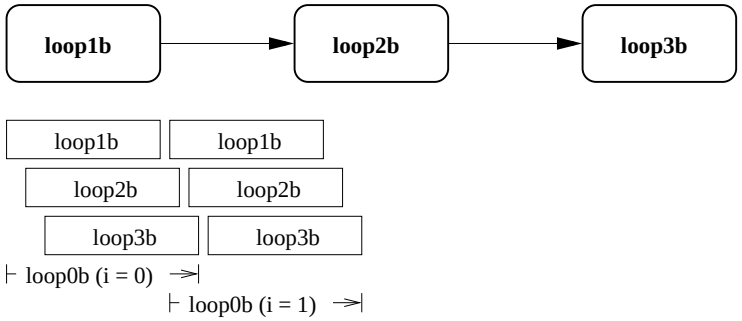
Summarizing the accessed memory locations by a coarse-grained function is more difficult than summarizing the accessed memory locations by a fine-grained function. For the fine-grained case, the memory consists of registers (or scalar variables). The



(a)



(b)



(c)

Figure 3.6 Illustration of the producer-consumer relations between coarse-grained functions

```
 1:    int A[40];
 2:
 3:    foo0b (...)
 4:    {
 5:        /*
 6:         * loop0b:
 7:         */
 8:        for (i0 = 0 ; i0 <= 3 ; i0++) {
 9:            /*
10:             * loop 1b: write A[0..39]
11:             */
12:            for (i1 = 0 ; i1 <= 39 ; i1++)
13:                A[i1] = ...
14:            /*
15:             * loop 2b: read A[0..39], write A[0..39]
16:             */
17:            for (i2 = 0 ; i2 <= 39 ; i2--)
18:                A[i2] = A[i2] ...
19:            /*
20:             * loop 3b: read A[0..39]
21:             */
22:            for (i3 = 0 ; i3 <= 39 ; i3++)
23:                s = A[i3] ...
24:    }
```

Figure 3.7 Producer and consumer program regions with the same memory access patterns

source and destination operands of an instruction unambiguously specify which registers are accessed. The set of accessed registers can be easily represented using a bit vector, with each bit corresponding to a register.

When performing data-flow analysis to identify the producer and consumer relation between instructions, we need to check whether two instructions may access the same registers. This checking can be easily done by applying bit-level operations on bit vectors.

On the other hand, a coarse-grained program region can access not only scalar variables, but also arrays and aggregates like structures or unions in the C programs. Using pointer to access dynamically allocated memories only makes the situation worse. In general, we cannot use bit vectors to represent the set of memory locations accessed by a coarse-grained function. Instead, we need to use complicated data structures to repre-

sent the accessed array elements, aggregates and dynamically allocated memories. If the array accesses are irregular, or we cannot figure out exactly which dynamically allocated memories are accessed, at best we can only have an inaccurate but conservatively correct representation.

This inevitably complicates the identification of producer and consumer relation between coarse-grained functions. When performing data-flow analysis, instead of applying bit-level operations on bit vectors, we need to apply complicated procedures on complicated data structures to check whether two coarse-grained functions may access some common memory locations.

There is another difference between accessing an array and accessing a scalar. The access order of array elements could be very useful information, as discussed in next section.

3.3.3.1 Memory Access Order

Consider the program segment in Figure 3.7, which is essentially the same as the example in Figure 3.5 except that `loop2a` in Figure 3.5 accesses array `A` from element 39 to element 0, while `loop2b` in Figure 3.7 accesses array `A` from element 0 to element 39. Because of this reversal of the array accessing order, `loop1b` and `loop2b` in Figure 3.7 not only have a producer and consumer relationship but also have the same *access pattern* of array `A`. Similarly, `loop2b` and `loop3b` also have the same *access pattern* of array `A`. Because of this, the array elements produced by `loop1b` can be immediately consumed by `loop2b`, and the array elements produced by `loop2b` can be immediately consumed

by `loop3b`, without the need to buffer the whole array `A`. This data streaming not only eliminates the buffering overhead but also increases the overlap between the execution of producer and consumer, as illustrated in Figure 3.6(c).

Note that, in Figure 3.4 and Figure 3.5, we use more memory for the communication between producer and consumer pairs to increase the available function parallelism. However, if the communication between producer and consumer can be in streaming fashion, like the one shown in Figure 3.6(c), we can increase the available coarse-grained function parallelism without using additional memory [2].

Strictly speaking, the data path in Figure 3.6(c) may not be correct, because the output of `loop2b` may be consumed by program regions outside `loop0b`. On the other hand, we are certain that the output of `loop1b` is only consumed by `loop2b`, because the writes of `loop1b` are "killed" by the writes of `loop2b`, and thus will not get exposed outside `loop0b`. Therefore, when we summarize the memory accesses of a program region, we only need to record the exposed memory accesses. The next section will elaborate on this.

3.3.3.2 Summarizing Exposed Accesses

Consider the program segment in Figure 3.8, which is different from Figure 3.7 in that `loop2c` reads and writes both array `A` and array `B`. However, knowing that `loop2c` reads array `B` from element 0 to element 39 will not help find more producers for `loop2c`, because *all* the array `B` elements are produced from *within* the loop body of `loop2c`.

[2] In this case, we even use less memory.

```
 1:    int A[40];
 2:    int B[40];
 3:
 4:    foo0c (...)
 5:    {
 6:        /*
 7:         * loop0c:
 8:         */
 9:        for (i0 = 0 ; i0 <= 3 ; i0++) {
10:            /*
11:             * loop 1c: write A[0..39]
12:             */
13:            for (i1 = 0 ; i1 <= 39 ; i1++)
14:                A[i1] = ...
15:            /*
16:             * loop 2c: read A[0..39], write A[0..39], write B[0..39]
17:             *          read B[0..39] (not exposed)
18:             */
19:            for (i2 = 0 ; i2 <= 39 ; i2--) {
20:                B[i2] = A[i2] ...
21:                A[i2] = ... B[i2] ...
22:            }
23:            /*
24:             * loop 3c: read A[0..39]
25:             */
26:            for (i3 = 0 ; i3 <= 39 ; i3++)
27:                s = A[i3] ...
28:    }
```

Figure 3.8 Example illustrating summarization of exposed accesses.

In general, to find the producers and consumers of a program region, we only need to know its *exposed* memory accesses. The exposed memory reads of a program region are the memory reads that are not "covered" by any memory write *executed earlier* within the same program region. The exposed memory writes of a program region are the memory writes that are not "killed" by any memory write *executed later* within the program region.

An exposed read should have some producer *outside* its program region, unless it is an access of some implicitly initialized memory like look-up table. Otherwise the programmer may forget to initialize some memory. On the other hand, an exposed write may or may not have consumers *outside* its program region.

In general, to exactly summarize the *truly* exposed memory accesses of a coarse-grained function is difficult, partly because of the reason discussed at the beginning of Section 3.3.3, and partly because of the difficulty in calculating the addresses of accessed memories.

3.3.3.3 Symbolic Scalar Variable Evaluation

The target language of this research work is the C language. C programs use pointers to reference memory extensively, which causes difficulty in summarizing the exposed memory accesses of program regions.

Take the program segment of `foo2d` in Figure 3.9 as example, which is simplified from the original source code of the pre-emphasis filter of G.724 decoder [24]. To determine the exposed memory reads of `loop2d`, we need to know the memory locations accessed by the pointer dereferences `*p` and `*q` in the loop body. Inter-procedural pointer analysis [140] [141] [142] [103] could tell us that both pointers `p` and `q` point to the memory object array `A`. However this information is not accurate enough for us to deduce that the reads by `*p` and `*q` at line 20 get their data from outside `loop2d`, not from the write `*p` at line 20. To figure out this, we must know that the assignment statement at line 20 is equivalent to the assignment statement in the comment at line 21. Then we can use dependence test to confirm that there is no true data dependence between the write of `*p` and the reads of `*p` and `*q` at line 20.

```
 1:    short A[40];
 2:
 3:    foo0d ()
 4:    {
 5:        ...
 6:        foo2d (A, tmp2, 40);
 7:        ...
 8:    }
 9:
10:    foo2d (short *s, short n, int L)
11:    {
12:        short *p, *q, temp, i;
13:
14:        p = s + L - 1;
15:        /* p = A + 39 */
16:        q = p - 1;
17:        /* q = A + 38 */
18:        temp = *p; /* A[39] */
19:    loop2d: for (i = 0 ; i <= L - 2 ; i++) {
20:            *p = *p - *q-- * n;
21:            /* A[39-i] = A[39-i] - A[38-i] * n; */
22:            p--;
23:        }
24:        /* p = A + 0 */
25:        *p = *p - n * mem_pre;
26:        /* A[0] = A[0] - n * mem_pre; */
27:        mem_pre = temp;
28:    }
```

Figure 3.9 Example illustrating symbolic scalar variable evaluation.

We have discussed the sub-problems we need to solve to extract coarse-grained data-flow from C programs. The next section will outline the proposed program analysis system to solve these problems and the organization of the rest of this dissertation.

3.4 Thesis Organization

The proposed memory data-flow analysis system to extract coarse-grained data-flow from C programs for the exploitation of function parallelism is sketched in Figure 3.10, which shows the components of this program analysis system, as well as the information flow between them. The current implementation first does function in-lining to embed

all C source code into the main function. Flow-insensitive pointer analysis [103] is then performed to obtain the set of objects that each pointer *may* points to. Next, control flow graph is constructed to facilitate the symbolic evaluation of scalar variables, as well as to obtain more accurate pointer information by taking control flow into consideration. The shaded components in Figure 3.10 constitute the main work of this research. The rest of this dissertation will present more detailed discussion on symbolic scalar variable evaluation (Chapter 4), program region construction (Chapter 5), exposed memory accesses summarization (Chapter 6), and producer-consumer relation analysis (Chapter 7). Chapter 8 will discuss the prototyping of the memory data-flow analysis system and the experiment results. Chapter 9 will conclude this dissertation with the obtained insights and some possible future research directions.

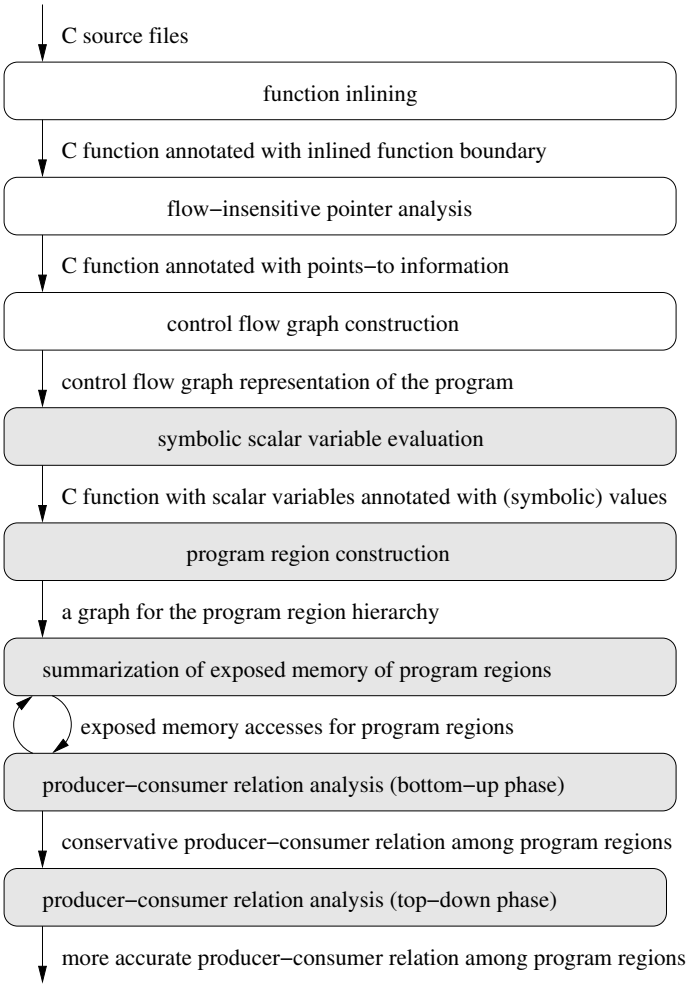


Figure 3.10 Components of the proposed memory data-flow analysis system

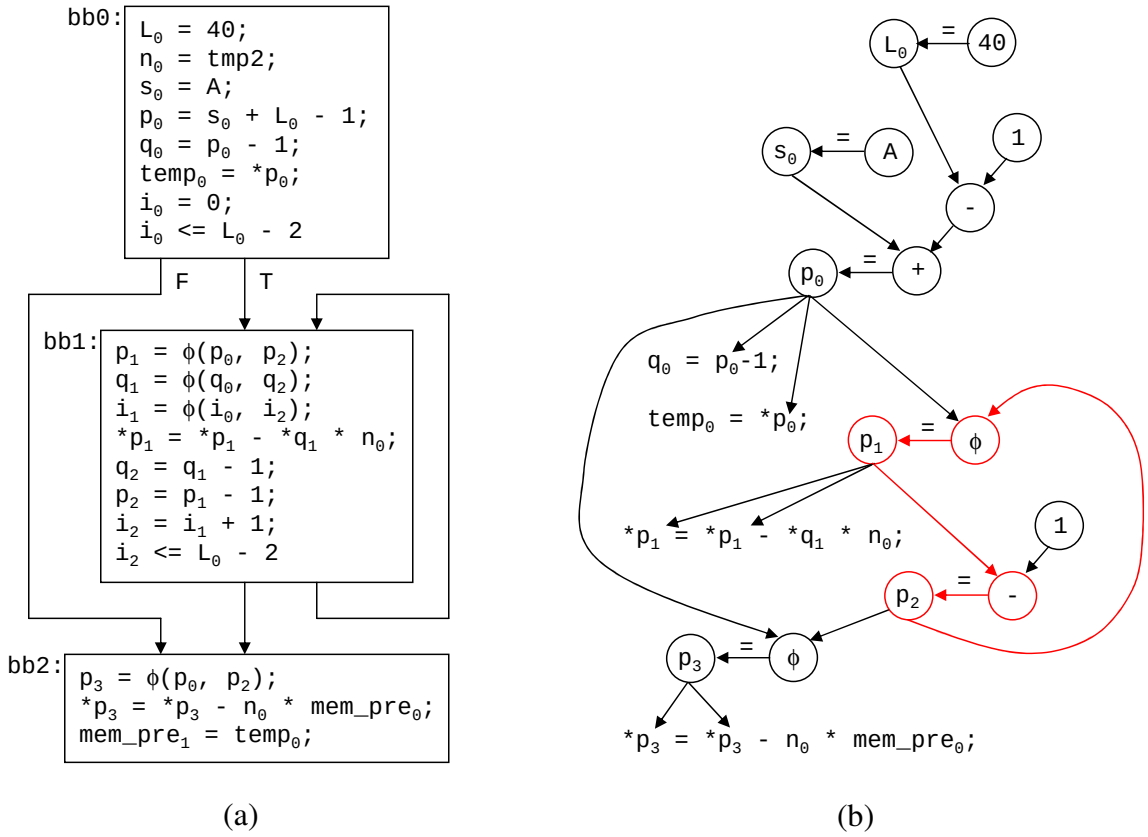
CHAPTER 4

Symbolic Scalar Variable Evaluation

This chapter will explain how the proposed memory data-flow program analysis system evaluates the symbolic value of each scalar variable, based on SSA form [121] and induction variable detection [143]. The limitation of these algorithms is that they can not go beyond the procedure boundary. To work around this limitation, procedures are in-lined first, as indicated in Figure 3.10.

4.1 SSA-based Symbolic Evaluation

Use the program segment in Figure 3.9 as example. After in-lining, we can covert the function `foo0d` into the SSA form shown in Figure 4.1(a). Note that the variables in Figure 4.1(a) are annotated with different subscripts so that the value of each variable is generated by a single assignment statement. For straight-line code, the single assignment property can be easily obtained by renaming variables. However, in an arbitrary control flow graph, different values of the same variable can reach the same program point via different paths in the control flow graph. To preserve the single assignment property, ϕ-functions are inserted at adequate confluence points in the control flow graph to represent all the possible reaching values using one dummy variable. For example, in Figure 4.1(a),



(a) (b)

Figure 4.1 Example SSA form nd value flow graph

both values of the variable p, p_0 and p_2, can reach the beginning of basic block bb1. Therefore, a ϕ-function is inserted at the beginning of basic block bb1 to represent the two possible reaching values p_0 and p_2 using the dummy variable p_1.

Basically, SSA form is a sparse representation of the value flow between variables. By back-tracking the SSA link, we can do backward substitution to find the symbolic value of a variable. For example, in Figure 4.1(b), we can find the symbolic value of p_0

as follows.

$$\begin{aligned} \mathtt{p}_0 &= (\mathtt{s}_0 + \mathtt{L}_0) - 1 \\ &= (\mathtt{s}_0 + 40) - 1 \quad \text{, given } \mathtt{L}_0 = 40 \\ &= (\mathtt{A} + 40) - \mathtt{1} \quad \text{, given } \mathtt{s}_0 = \mathtt{A} \\ &= \mathtt{A} + 39 \end{aligned}$$

The problem with back-tracking the value flow through SSA links is that there may be cycles in the value flow graph, as the one highlighted in Figure 4.1(b) by red edges. Cycles in value flow graph are caused by reading and writing the same variables within loops. These variables are called induction variables. Induction variables must be handled carefully, otherwise, back-tracking the value flow graph may get trapped in infinite loop.

4.2 Induction Variable Detection

For the detection of induction variables, we use the method invented in [144]. First, we identifies the SCCs (Strongly Connected Components) [145] in the value flow graph. Each SCC is corresponding to an induction variable. The nodes in SCC could be scalar variables, arithmetic operators, and ϕ-functions. If the combination of the operators and ϕ-functions in a SCC matches some predefined patterns, we can determine the symbolic value of each node in the SCC.

Take the SCC, marked with red edges, in Figure 4.1(b) as example. It has two operators: 1) a ϕ-function at the loop header with an operand $\mathtt{p}_0$ from outside the loop and another operand $\mathtt{p}_2$ from within the loop; 2) a "$-$" operator with the second operand

being a constant 1. For this type of SCC, each node in the SCC will be an induction expression with symbolic value of the form $d + (-1)\mathtt{h}$. Here h is called *fundamental induction variable* [144], which takes the values 0, 1, 2, 3, $\cdots$. The coefficient of the fundamental induction variable is -1, which means the value of each induction expression in this SCC will decrement by 1 every iteration. Each induction expression in this SCC will have a different offset d, depending on its position in the SCC. Below are the symbolic values of the induction expressions $\mathtt{p}_1$ and $\mathtt{p}_2$.

$$\begin{aligned} \mathtt{p}_1 &= \mathtt{p}_0 + (-1)\mathtt{h} \qquad \text{, where } \mathtt{h} = 0, 1, 2, 3, \cdots \\ \mathtt{p}_2 &= \mathtt{p}_1 - 1 \\ &= \mathtt{p}_0 + (-1)\mathtt{h} - 1 \end{aligned}$$

We can further substitute the value of $\mathtt{p}_0$ into the symbolic values of $\mathtt{p}_1$ and $\mathtt{p}_2$ as follows.

$$\begin{aligned} \mathtt{p}_1 &= \mathtt{A} + 39 + (-1)\mathtt{h} \quad \text{, given } \mathtt{p}_0 = \mathtt{A} + 39 \\ \mathtt{p}_2 &= \mathtt{A} + 38 + (-1)\mathtt{h} \end{aligned}$$

The technique presented in [144] can identify higher-order induction variables which can be represented as polynomials of the fundamental induction variable. For the current implementation, we only represent symbolic values as affine expressions of the form $d+c\cdot\mathtt{h}$, where h is the fundamental induction variable, c is an integer constant, and d can be either an integer constant or a scalar variable. Back-tracking will proceed in the value flow graph until any non-affine term is encountered. For example, in Figure 4.2, back-

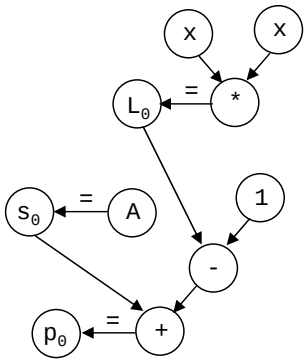


Figure 4.2 Example illustrating non-affine expressions

tracking will stop before $\mathtt{L}_0 = \mathtt{x} * \mathtt{x}$. The symbolic value of $\mathtt{p}_0$ will be represented as $\mathtt{A} + \mathtt{L}_0 - \mathtt{1}$, without further expanding $\mathtt{L}_0$ into $\mathtt{x} * \mathtt{x}$.

4.3 SSA Extension

Symbolic evaluation only based on SSA form has its limitation. For example, in Figure 4.1, the SSA form only tell us that the value of $\mathtt{p}_3$ can be either $\mathtt{p}_0$ or $\mathtt{p}_2$. Note that the value of $\mathtt{p}_3$ could be $\mathtt{p}_0$ only if the branch at the end of the basic block `bb0` is not taken. However, the branch at the end of the basic block `bb0` is always taken, because its branch condition $\mathtt{i}_0 <= \mathtt{L}_0 - \mathtt{2}$ is always true. (The value of $\mathtt{i}_0$ is 0, and the value of $\mathtt{L}_0$ is 40.) Therefore, the value of $\mathtt{p}_3$ is actually equal to $\mathtt{p}_2$.

Furthermore, $\mathtt{p}_2$ is an induction variable, and thus can take more than one values. $\mathtt{p}_3$ should take the *last* value of $\mathtt{p}_2$ when the loop terminates, because $\mathtt{p}_3$ is outside the loop, while $\mathtt{p}_2$ is inside the loop. However, we can not figure out this using the SSA representation. The fundamental problem of SSA form is that it retains only data flow

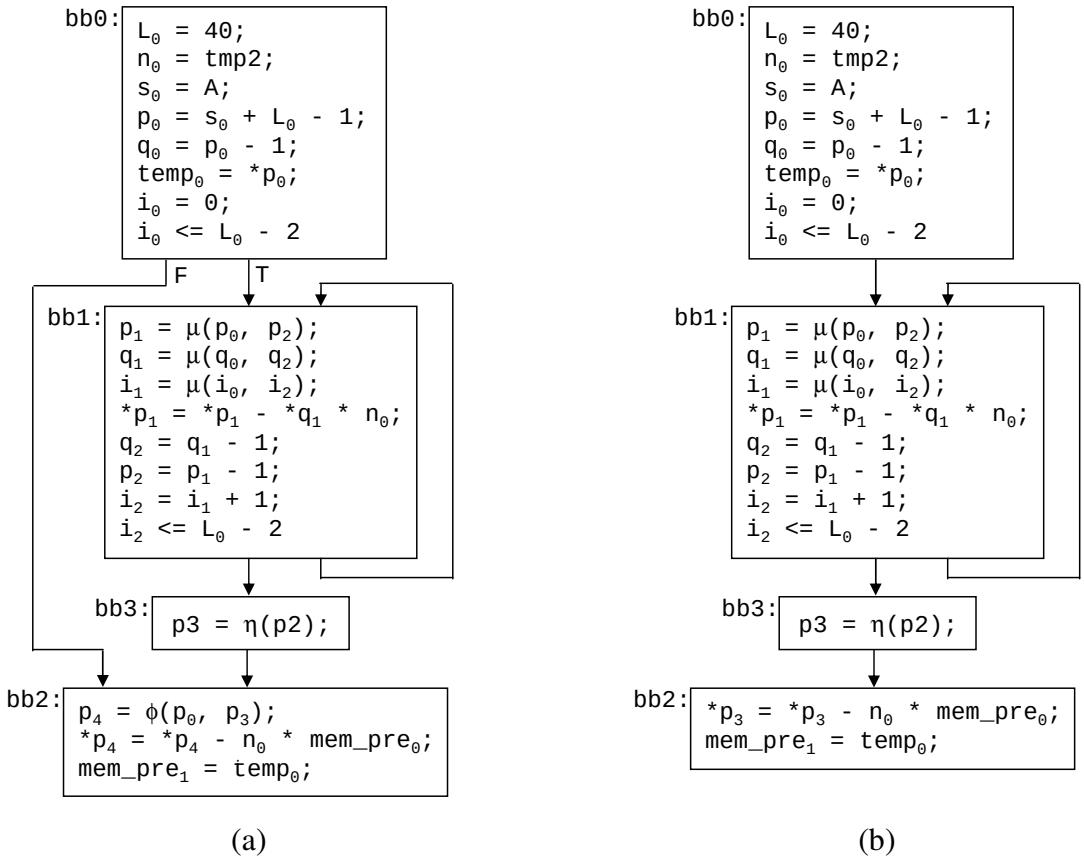


Figure 4.3 Example gated SSA form and pruned control flow graph

information but no control flow information. The ϕ-function contains no information to determine which of the reaching values it should take. To remedy this problem, people has extended SSA form to gated SSA form [123].

In gated SSA form, ϕ-function is augmented with predicate for the selection of possible reaching values. Special ϕ-functions called μ-functions and η-functions are placed at loop entry and loop exits. A μ-function has two operands. The output of a μ-function will take the value of the first operand for the first loop iteration, and the value of the second operand for the remaining loop iterations. The value of a η function is the value of the

corresponding variable when control reaches the corresponding loop exit. Figure 4.3(a) shows the corresponding gated SSA form of Figure 4.1(a). For clarity, it does not show the predicates in ϕ-functions, μ-functions and η-functions. Note that a dummy basic block is inserted at the loop exit to facilitate the insertion of η-functions.

The implementation and interpretation of gated SSA form is complicated. For the symbolic evaluation of scalar variables in the prototype memory data-flow analysis system, we implemented a simplified version of gated SSA form. To ease the job of identifying induction variables and calculating their loop-exit values, we extended the SSA form with μ-functions and η-functions, but without having predicates in ϕ-functions, μ-functions and η-functions. Without resorting to predicate evaluation, we can still prune the control flow graph by checking whether some branch conditions are always true or always false.

For the control flow graph in Figure 4.3(a), the `false`-branch at the end of basic block `bb0` is never taken, so we can prune this edge and obtain the simplified control flow graph in Figure 4.3(b). After pruning the `false`-branch edge at the exit of basic block `bb0`, we can also prune the the ϕ-function at the beginning of basic block `bb2`, because now the control can reach basic block `bb2` only through bacic block `bb3`. This can be accomplished by re-constructing the SSA form using the pruned control flow graph [1].

Prunning control flow graph and SSA enables us to have more accurate symbolic scalar variable evaluation. For example, we can easily conclude that, in Figure 4.3(b),

[1]It will be interesting to implement an algorithm to incrementally modify the SSA form from an incremetally modified control flow graph, but this is beyond the scope of this work.

variable p at the basic block bb2 has the symbolic value of $\mathtt{p}_3$. We cannot easily reach this conclusion in Figure 4.3(a) without full-blown implementation of gated SSA.

Given Figure 4.3(b), the value of $\mathtt{p}_3$ can be derived as follows.

$$\begin{aligned}
\mathtt{p}_3 &= \eta(\mathtt{p}_2) \\
&= \eta(\mathtt{p}_1 - 1) \\
&= \eta((\mathtt{A} + 39 - \mathtt{h}) - 1) \\
&= \eta(\mathtt{A} + 38 - \mathtt{h}) \\
&= \eta(\mathtt{A}) + \eta(38) - \eta(\mathtt{h}) \\
&= \mathtt{A} + 38 - 38 \\
&= \mathtt{A}
\end{aligned}$$

Note that h is the fundamental induction variable, which starts from 0. Its last value $\eta(\mathtt{h})$ is the loop trip count minus 1, that is 38. The loop trip count in this example can be calculated by checking the loop exit condition, $\mathtt{i}_2$ > $\mathtt{L}_0$ - 2. Note that the value of $\mathtt{i}_2$ is $1 + \mathtt{h}$ and the value of $\mathtt{L}_0$ is 40, and thus the value of the loop exit condition is h > 37. So, when the loop terminates, the value of h would be 38. In general, it is not so straightforward to calculate the trip counts for arbitrary loops, which is beyond the scope of this work.

CHAPTER 5

Program Region Hierarchy

This chapter will discuss how the proposed memory data-flow analysis system partitions a program into program regions as coarse-grained functions. It will also discuss the limitation of this program partitioning and how to handle library functions which have no source code available.

5.1 Program Region Hierarchy

For the current implementation, we define a coarse-grained function to be one of the following 4 program regions.

- in-lined function;
- loop with single loop entry, the so called natural loop [146];
- loop body;
- memory read;
- memory write.

By partitioning a program segment into these regions, we can impose a program region hierarchy upon the program segment. For example, Figure 5.1 is the program region

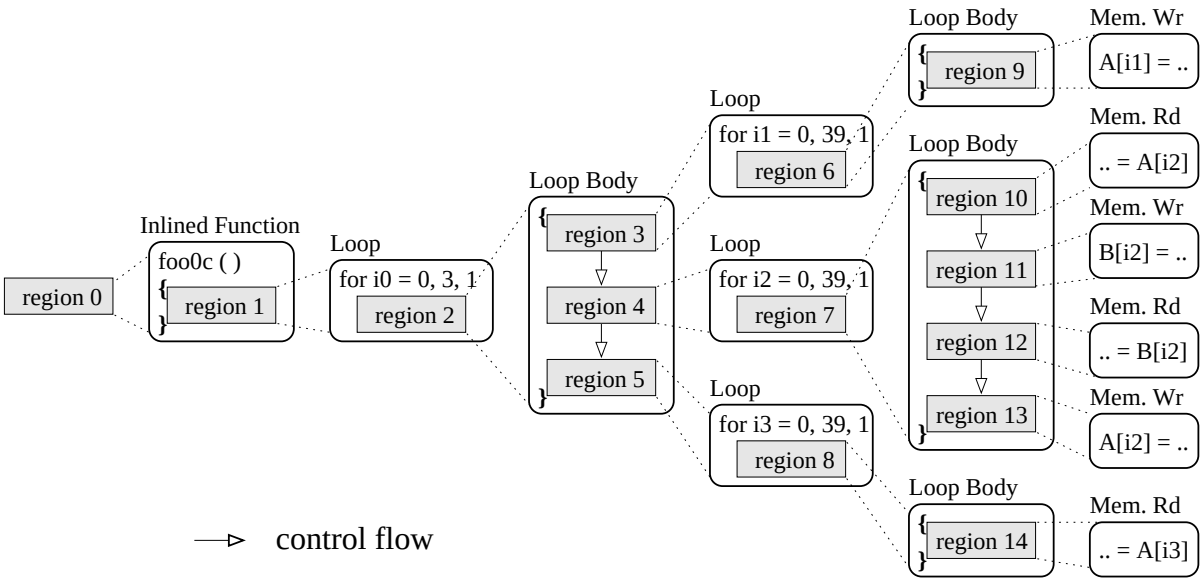


Figure 5.1 Example program region hierarchy

hierarchy of the in-lined function `foo0c` in Figure 3.8. Memory reads and memory writes are the fundamental regions which are always at the bottom of program region hierarchy, like the regions 9 to 14 in Figure 5.1. Although it is hard to call a single memory read or memory write coarse-grained, treating memory read and memory write as fundamental program regions will simplify the implementation of the the memory data-flow analysis and the discussion of later chapters.

Except the fundamental regions, all other program regions consist of sub-regions. A loop region, like the regions 1, 3, 4, and 5 in Figure 5.1, has only one sub-region, its loop body. A loop body region, like the regions 2, 6, 7 and 8 in Figure 5.1, or an in-lined function region, like the region 0 in Figure 5.1, may have more than one sub-regions. The sub-regions of a program region are represented as a directed graph called the sub-region

graph. The nodes in a sub-region graph are corresponding to the sub-regions. The edges in a sub-region graph are corresponding to the control flow between the sub-regions.

Accordingly, we use two major recursive data structures to implement the program region hierarchy, `_region` and `_subregion_graph`. Each `_region` has a reference to a `_subregion_graph`. Each node in a `_subregion_graph` has a reference to the `_region` data structure of the corresponding sub-region. We basically build the `_region` and `_subregion_graph` data structures from bottom up. To build the `_region` data structure for a program region, we first build the `_region` data structures for its sub-regions, then build a `_subregion_graph` with its nodes pointing to the `_region` data structures of the sub-regions.

Program regions are identified in the control flow graph. A loop region and the corresponding loop body region can be found using the algorithm for finding natural loops [146]. The entry basic block and exit basic block of an in-lined function are marked for the identification of the in-lined function. The marking of in-lined function entry block and exit block is done during the construction of control flow graph, with the help of in-liner generated compiler pragmas.

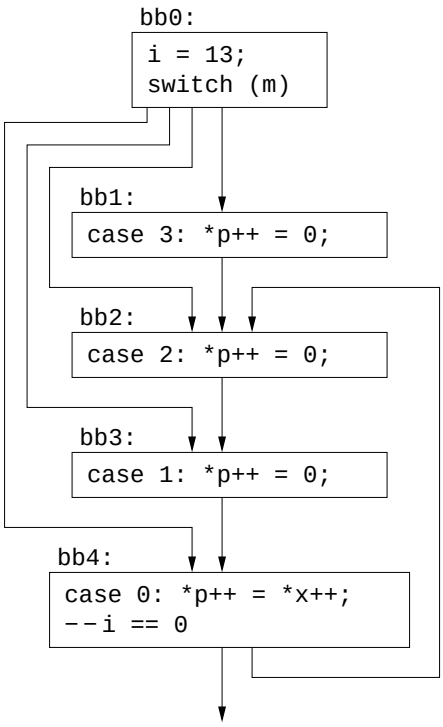
By our definition and implementation of program regions, the sub-region graph of a program region is actually a directed *acyclic* graph, since a loop region has only one sub-region and a loop body region contains no back-edges. This simplifies the implementation of the memory data-flow analysis, which will be discussed in the following chapters. However, our definition and implementation of program regions do have some limitations as discussed in the next section.

```
i = 13;
switch (m) {
   case 3: *p++ = 0;
   case 2: do {
               *p++ = 0;
   case 1:    *p++ = 0;
   case 0:    *p++ = *x++;
            } while (--i);
}
```

(a)

```
i = 12;
switch (m) {
   case 3: *p++ = 0;
   case 2: *p++ = 0;
   case 1: *p++ = 0;
   case 0: *p++ = *x++;
}
do {
   *p++ = 0;
   *p++ = 0;
   *p++ = *x++;
} while (--i);
```

(c)



(b)

Figure 5.2 Work-around of improper loop

5.2 Limitations

The definition and implementation of program regions in this work have the following limitations.

- It cannot handle improper loops, which have more than one loop entry. Figure 5.2(a) shows a program segment from one of the MediaBench programs. Note that the `do-while` loop has multiple entry points. We can not identify an improper loop using the algorithm for finding natural loops, which is based on the detection of back-edges. A back-edge is an edge in the control flow graph so that the destination basic block of the edge dominates the source basic block of the edge.

As shown in Figure 5.2(b), the "loop-back" edge from basic block `bb4` to basic block `bb2` in the control flow graph is not a back-edge, because basic block `bb2` does not dominate basic block `bb4`. Therefore we will not group basic blocks `bb2`, `bb3` and `bb4` as a loop body region. Instead, we will group all the basic blocks in Figure 5.2(b) as one program region, which will have a sub-region graph with the same structure as the control flow graph shown in Figure 5.2(b). Note that the control flow graph in Figure 5.2 is not an *acyclic* graph. This violates our assumption of sub-region graph and breaks the memory data-flow analysis.

The current remedy to this problem is to hand modify an improper loop to a natural loop by peeling out the first iteration, as shown in Figure 5.2(c). We expect improper loops will occur very rarely in common programs, as we only find one case in all the benchmark programs we tried.

- It cannot handle indirect function calls. This is really the limitation of our in-lining based approach. In the current implementation, in-lining takes place before the pointer analysis, as shown in Figure 3.10. So, the in-liner does not know the possible values of function pointers, and thus does not in-line functions at the call-site of indirect function calls.

  The current remedy to this problem is to hand replace the call-site of indirect function call with several call-sites of direct function calls based on the pointer analysis results. This is illustrated in Figure 5.3.

```
foo()
{
   sort_data(quick_sort);
   sort_data(bubble_sort);
}
sort_data(void (*sort)())
{
   (*sort)();
}
```

(a)

```
foo()
{
   sort_data(quick_sort);
   sort_data(bubble_sort);
}
sort_data(void (*sort)())
{
   if (sort == quick_sort)
       quick_sort();
   else if (sort == bubble_sort)
       bubble_sort();
}
```

(b)

Figure 5.3 Work-around of indirect function call

- It cannot handle recursive functions. This is the limitation of any in-lining based program analysis. We may convert tail-recursions to loops, but, to handle recursions in general, we must resort to inter-procedural memory data-flow analysis, which is left as future work. For the telecommunication and media benchmark programs used in this work, we only find one recursion case for implementing the intrinsic functions of `left_shift` and `right_shift`. We hand modified the program to break this recursion, as illustrated in Figure 5.4.

- It cannot handle functions with variable number of arguments, for example, `printf`. For the current in-lining based implementation, we manually replace the `printf` at different call-sites with different variants of the `printf` function according to the number and the data types of the actual parameters. For each new version of `printf`, a template function is created to model its memory access patterns.

Not just for `printf`, we also use template to model the memory access behaviors of other library functions.

```
shift_left(int shiftcnt)
{
    if (shiftcnt < 0)
       shift_right(-shiftcnt);
    /* do shift left */
}
shift_right(int shiftcnt)
{
    if (shiftcnt < 0)
       shift_left (-shiftcnt);
    /* do shift left */
}
foo ()
{
    shift_left(shiftcnt);
    shift_right(shiftcnt);
}
```

(a)

```
shift_left(int shiftcnt)
{
    /* do shift left */
}
shift_right(int shiftcnt)
{
    /* do shift left */
}
foo ()
{
    if (shiftcnt > 0)
        shift_left(shiftcnt);
    else
        shift_right(-shiftcnt);

    if (shiftcnt > 0)
        shift_right(shiftcnt);
    else
        shift_left(-shiftcnt);
}
```

(b)

Figure 5.4 Work-around of recursive function call

```
 1: size_t fread (void *ptr, size_t size, size_t nitems, FILE *file)
 2: {
 3:     int i, j;
 4:
 5:     *file = *file;
 6:     for (i = 0 ; i < nitems ; i++) {
 7:         for (j = 0 ; j < size ; j++)
 8:             ((char *) ptr) [i * size + j] = 0;
 9:         if (i)
10:             break;
11:     }
12:     return i;
13: }
```

Figure 5.5 A template describing the memory access behavior of `fread`

5.3 Handling Library Functions

The proposed program analysis system is trying to do whole program memory data-flow analysis. No matter it is in-lining based or inter-procedural, whole program analysis cannot proceed if the source code of some function is not available. However, it is very common in a program to call library functions which have no source code available. This

work uses template function to model the memory access behavior of library functions, similar to the approach used in [141] for whole program pointer analysis. For example, Figure 5.5 shows the template function for the library function `fread`. The outer loop of the template in Figure 5.5, lines 6-11, models writing the items to the buffer pointed by the formal parameter `ptr`. The trip count of the outer loop, the maximum number of items to read, is given by the formal parameter `nitems`. Lines 9-10, Figure 5.5, models that the outer loop can exit early and read fewer data items. The inner loop (lines 7-8, Figure 5.5) models writing each byte of the read item to the buffer. The formal parameter `size` gives the size of each item in bytes.

In addition to modeling the memory access behavior of software library functions, hardware IP (Intellectual Property) providers can also provide the templates that model the memory access behavior of their IPs [1]. Template is really a way to enable whole *system* memory data-flow analysis in order to optimize the communication between software and/or hardware components.

5.4 Related Work

This work has the same program regions as those used in [62] and [147]. The goal of [62] and [147] is to exploit coarse-grained data parallelism in outer loops, while the goal of this work is to exploit coarse-grained function parallelism among the program regions. It is not clear how [62] and [147] handled improper loops, indirect function

[1] A hehavioral C model also works, but from the memory data-flow analysis point of view, a template only modeling the memory access behavior is accurate enough and requires less analysis time.

calls, recursion and library functions. Some of these issues may not matter in their case, because their target language is Fortran.

The templates used in [141] only model the accessed memory objects. This is enough for the purpose of whole program pointer analysis. For whole program memory data-flow analysis, we may obtain more accurate analysis results by using templates which have more detailed modeling of the memory access patterns.

CHAPTER 6

Exposed Memory Access Summarization

For each program region, exposed memory access analysis tries to find its exposed memory reads that consume the data generated by other program regions, and the exposed memory writes that produce the data for other program regions. For each exposed read and exposed write, we use a data structure called Memory Access Descriptor (MAD), as explained below, to describe its memory access pattern. The exposed reads, and the exposed writes, of a program region is a set of MADs which have mutually exclusive memory accesses.

6.1 Memory Access Descriptor

For our current implementation, the MAD data structure for describing memory access pattern can be represented as a 6-tuple, $\langle size, alias, base, offset, displace, type \rangle$.

- *size*: This is the size of each access in terms of bytes.
- *alias*: This is the alias set given by the flow-insensitive pointer analysis [102] [103], which gives us the most conservative information about the memory objects which *may* be accessed.

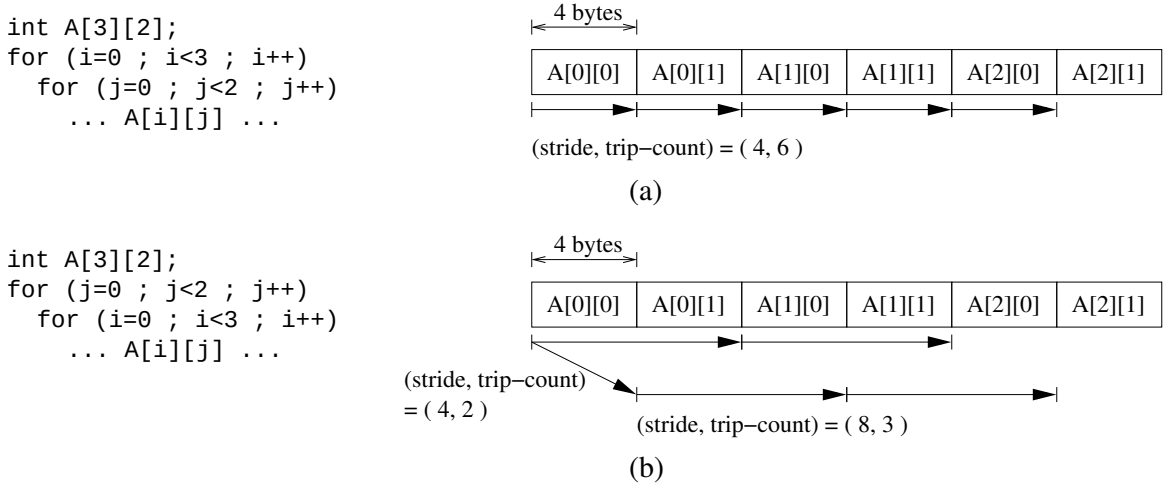


Figure 6.1 Example illustrating the *displace* field of the MAD data structure

- *base*: This is the base address of the accessed memory locations. A base can be static or dynamic. A static base is like the array `A` in the memory reference `A[i]`. A dynamic base is like the pointer `p` in the memory reference `*p`.

- *offset*: This is *byte* offset from the *base* address. For example, for the memory accessed by `*(p+7)`, the *base* is `p`, and the *offset* is 28, assuming `p` is a pointer to 4-byte integers.

- *displace*: This is a list of (stride in bytes, trip-count) pairs. For example, accessing all the elements of a 3-by-2 integer array `A` in row major order will have *base* `A`, *offset* 0, and *displace* $[(4, 6)]$, as shown in Figure 6.1(a). On the other hand, accessing all the elements of `A` in column major order will have *displace* $[(8, 3)(4, 2)]$, as shown in Figure 6.1(b).

```
if (...) {
   /* access A[0] to a[9] */
   for (i = 0 ; i <= 9 ; i++)
      ... A[i] ...
} else {
   /* access a[9] down to a[0] */
   for (i = 9 ; i >= 0 ; i--)
      ... A[i] ...
}
```

(a)

```
if (...) {
   /* access A[0] to a[9] */
   for (i = 0 ; i <= 9 ; i++)
      ... A[i] ...
} else {
   /* access a[1] to a[10] */
   for (i = 1 ; i <= 10 ; i++)
      ... A[i] ...
}
```

(b)

```
int A[10];
int B[20];
if (...) {
   p = A;
} else {
   p = B;
}
while (...) {
   ... p[x] ...
}
```

(c)

```
int A[10];
int B[20];
if (...) {
   p = A;
} else {
   p = B;
}
for (i = 0 ; i < 10 ; i++) {
   ... p[x] ...
}
```

(d)

```
struct {
   int A[10];
   int B[20];
} s;
for (i = 0 ; i < 20 ; i++) {
   ... s.B[i]...
}
```

(e)

```
struct {
  int a;
  int b;
} A[10];
for (i = 0 ; i < 10 ; i++) {
   ... A[i].b ...
}
```

(f)

Figure 6.2 Examples for illustrating different MAD structures

- *type*: Different *type* values represent different accuracy levels of the memory access description. Below are the 4 possible values of *type*, from the most accurate to the least accurate.

  - *Seq*: This type of MAD is the most accurate memory access description. A *Seq*-type MAD describes not only the accessed memory locations but also the access order. For example, we can figure out the exact accessed memory locations and the access order for the `for`-loops in Figure 6.1. Using the 6-tuple notation, $\langle size, alias, base, offset, displace, type \rangle$, the MADs describing the ac-

cesses of array `A` by the `for`-loops in Figure 6.1 are $\langle 4, \{\texttt{A}\}, \texttt{A}, 0, [(4,6)], Seq\rangle$ and $\langle 4, \{\texttt{A}\}, \texttt{A}, 0, [(8,3)(4,2)], Seq\rangle$ respectively.

– *Must*: A *Must*-type MAD describes only the accessed memory locations, but not the access order. For example, in Figure 6.2(a), one of the *for*-loops accesses array `A` from element `A[0]` to element `A[9]`, and the other from `A[9]` down to `A[0]`. We are certain that the code segment in Figure 6.2(a) accesses the set of array elements $\{\texttt{A}[i]|0 \leq i \leq 9\}$, but we cannot determine the access order at compile time. Therefore, we describe these memory accesses using a *Must*-type MAD, $\langle 4, \{\texttt{A}\}, \texttt{A}, , [(4,10)], Must\rangle$.

– *May*: While a *Must*-type MAD describes the *exact* set of accessed memory locations, a *May*-type MAD describes only an upper bound of the possibly accessed memory locations. Some memory locations in the set described by a *May*-type MAD may not be accessed. For example, the code segment in Figure 6.2(b) will access the set of array elements $\{\texttt{A}[i]|0 \leq i \leq 9\}$ if the branch condition is true, or $\{\texttt{A}[i]|1 \leq i \leq 10\}$ if the branch condition is false. Therefore, the *May*-type MAD for Figure 6.2(b) is $\langle 4, \{\texttt{A}\}, \texttt{A}, , [(4,11)], May\rangle$. Note that, at run time, it will access either `A[0]` or `A[10]`, but not both.

– *Doomed*: If we cannot even determine an upper bound of the accessed memory locations, we can only conservatively use a *Doomed*-type MAD to describe the possibly accessed memory *objects* given by the pointer analysis. For example, we cannot determine the memory locations accessed by the code segment

in Figure 6.2(c). We only know it may access the memory objects, array `A` or array `B`, but we are not certain which array elements of which array the `while`-loop will access. The corresponding *Doomed*-type MAD is thus $\langle 4, \{\texttt{A},\texttt{B}\}, \perp, \perp, \perp, \mathit{Doomed}\rangle$.

Figures 6.2(d) to (f) illustrate the MAD structures for other memory access cases. Like Figure 6.2(c), we cannot determine, at compile time, whether the `for`-loop in Figure 6.2(d) will access array `A` or array `B`. However, instead of giving up too early and using a *Doomed*-type MAD, we can still describe the memory accesses of the `for`-loop in Figure 6.2(d) using a *Seq*-type MAD, $\langle 4, \{\texttt{A},\texttt{B}\}, \texttt{p}, , [(4, 10)], \mathit{Seq}\rangle$. Note that the *alias* set of the MAD is {`A`, `B`}, and the *base* address of the MAD is `p`.

For the loop in Figure 6.2(e), which accesses the array `B` in the structure `s`, we can use $\langle 4, \{\texttt{s.B}\}, \texttt{s}, 40, [(4, 10)], \mathit{Seq}\rangle$ to describe its memory accesses, which has the starting address of the structure `s` as the *base*, and the offset of array `B` from the starting address of structure `s`, 40, as the *offset*.

For the loop in Figure 6.2(f), which accesses an array of structures, we can use $\langle 4, \{\texttt{A.b}\}, \texttt{A}, 4, [(8, 10)], \mathit{Seq}\rangle$ to describe its memory accesses, which has the starting address of array `A` as the *base*, the byte offset of the `b` field in the structure, 4, as the *offset*, and the byte size of the structure array element, 8, as the *stride*.

There are many memory descriptors proposed in the past, each with different trade-offs between accuracy and complexity [148] [56] [149] [58] [147] [150] [151]. We choose MAD mainly for the following reasons.

- C programs use pointer and pointer arithmetics extensively.
- We want MAD to be able to describe not only the accessed memory locations but also the access order.
- We want MAD to be simple enough so that we can get a quick prototype to do experiments on real programs.
- We expect MAD to be accurate enough for telecommunication and media applications.

It is always possible to have more sophisticated MAD design at the expense of more engineering effort and more analysis time. Indeed, one of the goal of the prototyping effort is to shed light on how to improve the MAD structure. The design of MAD structure is basically orthogonal to the bottom-up summarization process and the top-down pruning process, which will be explained in the following sections and the next chapter.

6.2 Bottom-up Summarization Process

Figure 6.2 shows the top-level function **Summarize** for summarizing the exposed reads and the exposed writes of the given region R. If R is a *Memory Read* region, **Summarize** will call **new_MAD** to create a MAD structure representing the corresponding memory read, which will be the only element of the set *exposed_reads* of R, and the set *exposed_write* of R is empty (lines 2-4, Figure 6.2). On the other hand, if R is a *Memory*

```
1: function Summarize (R: a region) begin
2:   if R is a Memory Read region then
3:     R.exposed_reads := { new_MAD(R) };
4:     R.exposed_writes := { };
5:   else if R is a Memory Write region then
6:     R.exposed_reads := { };
7:     R.exposed_writes := { new_MAD(R) };
8:   else
9:     // R.subregions is a directed acyclic graph (V, E), with
10:    // V is the set of nodes representing sub-regions of R
11:    // E is the set of edges representing control flow among V
12:    for v ∈ R.subregions do
13:      let r be the corresponding sub-region of v
14:      Summarize (r);
15:    end for
16:    if R is not a Loop region then
17:      R.exposed_reads := FindExposedReads (R.subregions);
18:      R.exposed_writes := FindExposedWrites (R.subregions);
19:    else
20:      let b be the only Loop Body sub-region of R;
21:      InterIterationDependenceTest(b);
22:      R.exposed_reads := ∑_{b.loop} (b.exposed_reads);
23:      R.exposed_writes := ∑_{b.loop} (b.exposed_writes);
24:    end if
25:  end if
26:  for x ∈ (R.exposed_reads ∪ R.exposed_write do
27:    if x is not invariant with respect to R then
28:      x.type := Doomed;
29:    end if
30:  end for
31: end function
```

Figure 6.3 The pseudo-code of **Summarize**

Write region, a new **MAD** will be created for the corresponding memory write, and the *exposed_reads* will be empty (lines 5-7, Figure 6.2).

If R is not a fundamental region, **Summarize** first recursively calls itself to find the exposed reads and writes of its sub-regions (lines 12-15, Figure 6.2), before finding its

own exposed reads and exposed writes (lines 16-21, Figure 6.2). This is the reason why the summarization of exposed memory accesses is a bottom-up process.

If R is a *Loop Body* or a *In-lined Function* region, **Summarize** will call **FindExposedReads** and **FindExposedWrites** to find the exposed reads and exposed writes (lines 16-18, Figure 6.2). The pseudo-codes of **FindExposedReads** and **FindExposedWrites** are shown in Figure 6.6 and Figure 6.8, which will be explained later.

If R is a *Loop* region, it has only one *Loop Body* sub-region, say b. First, **Summarize** will find the inter-iteration producer-consumer relationship between the sub-regions of b (line 21, Figure 6.2). Next, **Summarize** will call **Summation** ($\sum$) to find the exposed reads and the exposed writes of R by expanding the exposed reads and writes of b for all the iterations of R (lines 22-23, Figure 6.2).

Finally, **Summarize** will check each exposed memory access x to see whether x is *invariant* with respect to region R (lines 26-30, Figure 6.2). If not, the *type* of x is changed to *Doomed*. Here, x is *invariant* with respect to R if its MAD fields like *base*, *offset* and *displace* are all represented by affine expressions in terms of variables defined *outside* the region R, and thus not changing during the execution of the program region R. Note that the fundamental induction variable associated with a loop is invariant with respect to the corresponding loop body region. During each execution of the loop body, which is corresponding to one loop iteration, the value of the fundamental induction variable remains constant, because it only increments from iteration to iteration.

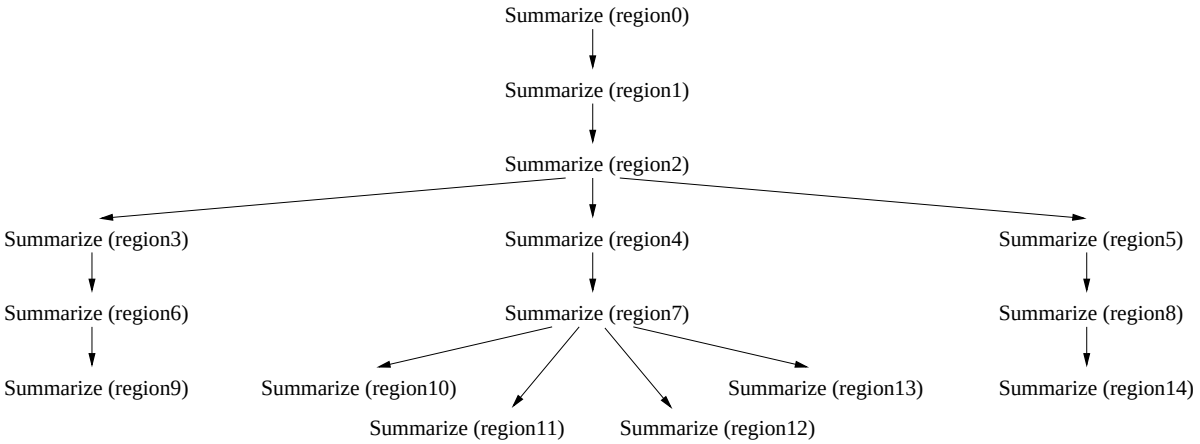


Figure 6.4 Example recursive calls of **Summarize**

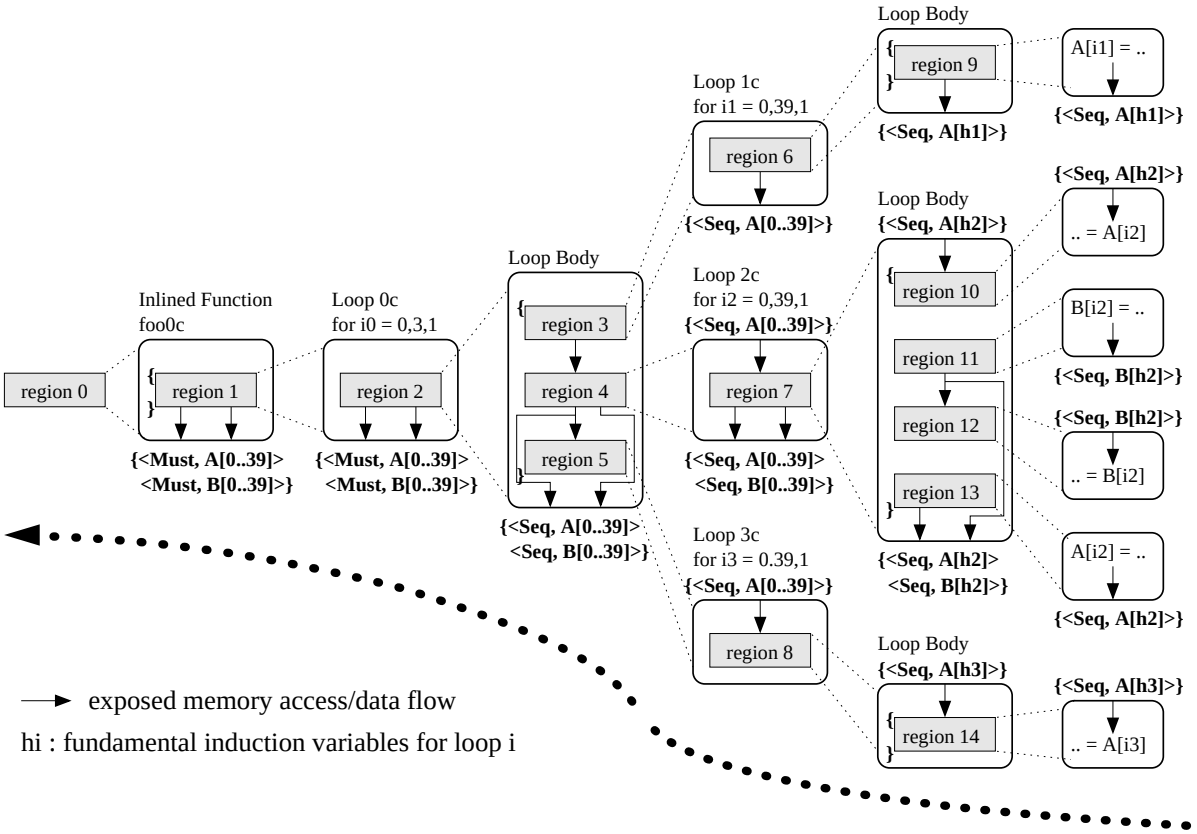


Figure 6.5 Illustration of the bottom-up summarization process

6.2.1 An Example

Taking the `foo0c` in Figure 3.8 as an example, whose program region hierarchy is shown in Figure 5.1. Suppose **Summarize**(region0) is called to summarize the exposed

memory access of `foo0c`. According to lines 12-15, Figure 6.2, **Summarize**(region0) will call **Summarize**(region1) first; which will in turn call **Summarize**(region2) first, and so on. The complete recursive call sequence of **Summarize** in this case is shown in Figure 6.4, and the result of the whole bottom-up process is shown in Figure 6.5.

When **Summarize**(region9) is called, there will be no further recursive call of **Summarize**, because region9 is a *Memory Write* region, which has no sub-region. A new MAD structure will be created to represent the exposed write `A[i1]`. Using the 6-tuple notation $\langle size, alias, base, offset, displace, type\rangle$ for MAD, the exposed writes of region9 is $\{\langle 4, \{ObjID_{\mathtt{A}}\}, \mathtt{A}, h_1, [(0,1)], Seq\rangle\}$, a set with only one MAD [1].

Here we assume the elements of array `A` are 4-byte integers. $\{ObjID_{\mathtt{A}}\}$ is the *may*-alias set given by the pointer analysis. The *offset* is the symbolic value of the array index `i1`, which is h_1, the fundamental induction variable of the enclosing loop `loop1c` in Figure 3.8. The *displace* $[(0,1)]$ indicates that the *stride* is 0, and the *trip-count* is 1, because there is only *one* accessed memory location, $\mathtt{A} + h_1$.

Note that the *type* of the exposed memory access of a fundamental region is always *Seq*. This is because we *define* a *single must*-accessed memory location as a *Seq* access. Also, the symbolic values of the *offset* and *displace must* be defined outside a fundamental region. This means the exposed memory access of a fundamental region R is always invariant with respect to R. Therefore, its *type* will never be down graded to *Doomed* at line 28, Figure 6.2.

[1] For simplicity, in Figure 6.5, the exposed writes of region9 is denoted as $\{\langle Seq, \mathtt{A}[h_1]\rangle\}$

After summarizing the exposed writes of the *Memory Write* region region9, we next summarize the *Loop Body* region region6. Because region9 is the only sub-region of region6, region6 has the same set of exposed writes as region9, and has no exposed reads. One subtlety here is that h_1, the fundamental induction variable of the enclosing loop, is an invariant with respect to the *Loop Body* region, so **Summarize**(region6) will not down grade $\langle Seq, \mathtt{A}[h_1]\rangle$ to *Doomed*.

After calling **Summarize**(region6), **Summarize**(region3) will deduce the exposed memory access of the *Loop* region region3 from the exposed memory access of the *Loop Body* region region3 (line 22-23, Figure 6.2). Basically, given $\{\langle Seq, \mathtt{A}[h_1]\rangle\}$, and the loop trip count of `loop1c`, which is 40, **Summation** ($\sum$) would return $\{\langle Seq, \mathtt{A}[0..39]\rangle\}$ because $h_1 = 0, 1, 2, \cdots, 39$. Recall that $\{\langle Seq, \mathtt{A}[0..39]\rangle\}$ is an abbreviation of the 6-tuple $\langle 4, \{ObjID_{\mathtt{A}}\}, \mathtt{A}, 0, \{[(1, 40)]\}, Seq\rangle$, with 4 being the *size*, $\{ObjID_{\mathtt{A}}\}$ being the *may-alias* set, `A` being the *base*, 0 being the *offset*, $(1, 40)$ being the $(stride, trip\text{-}count)$ pair describing the *displace*, and *Seq* being the *type*.

Similarly, we will summarize the fundamental regions, region10, region11, region12, and region13, and then the *Loop Body* region region7, the *Loop* region region4, and so on. Eventually we will get the summary for the *In-lined Function* region region0, as shown in Figure 6.5.

Note that in Figure 6.5, the exposed reads of region12, $\langle Seq, \mathtt{B}[h_2]\rangle$, is not exposed outside region7. This is because $\{\langle Seq, \mathtt{B}[h_2]\rangle\}$ is *covered* by the exposed writes of region11, which is also $\{\langle Seq, \mathtt{B}[h_2]\rangle\}$. This producer-consumer relationship between region11 and region12 is identified during the execution of **FindExposedReads** (line 17, Figure 6.2).

Figure 6.5 indicates the producer and consumer relation by an arrow from region11 to region12.

Also note that region7 is the corresponding *Loop Body* region of `loop2c`. A producer-consumer relationship may exist between different iterations of `loop2c`, because both the exposed reads and the exposed writes of region7 contain $\langle Seq, \mathtt{A}[h_2] \rangle$. Inter-iteration producer-consumer relationship is identified during the execution of **InterIterationDependenceTest** (line 21, Figure 6.2). In this example, there is no inter-iteration *true* dependence between iterations of `loop2c`. Therefore, there is no arrow from region7 to itself, nor from region13 back to region10, in Figure 6.5.

If a consumer region has only *intra*-iteration dependences, which have dependence distance 0, the consumer region and its producers are all executed in the same loop iteration. In other words, the consumer region and its producers regions are all sub-regions of the same parent loop body region. Therefore, there is no region outside the loop body region and the corresponding loop region to produce the data needed by the consumer region.

On the other hand, if a consumer region has some *inter*-iteration dependences, because the dependence distances must be larger than 0, its data are generated by some producer regions which are executed in previous iterations. For the first iteration of the loop, there is no previous iteration, therefore, the data of the consumer region must be generated by some producer regions outside the loop body region and the corresponding loop region. In other words, the consumer region has producer regions which are in its parent loop body region, and also producer regions outside its parent loop body region.

Memory writes can cover memory reads as well as memory writes. For example, both region3 and region4 have the same expose write $\langle Seq, \texttt{A}[0..39]\rangle$. Because region4 is executed later than region3, as indicated by the control flow in Figure 5.1, so the same exposed write of region3 is *killed* by the exposed write of region4. Writes are killed during the execution of **FindExposedWrites**, line 18, Figure 6.2.

One final point about Figure 6.5, before diving into more detailed explanation of **FindExposedReads** and **FindExposedWrites**, is that the *type* of the exposed writes of region2 is *Must*, instead of *Seq*. This is because for array `A` and `B`, the memory access pattern of `loop0c` is $(0, 1, 2, \cdots, 39, 0, 1, 2, \cdots, 39, 0, 1, 2, \cdots, 39, 0, 1, 2, \cdots 39)$. Strictly speaking, this is not a sequential pattern, because, for a sequential memory access pattern, each memory location can be accessed only once.

6.2.2 Finding Exposed Reads

Figure 6.6 outlines the function **FindExposedReads**. Given the sub-region graph G of region R, it will return the exposed reads of R. Let $G = (V, E)$, where V is the set of nodes representing the sub-regions, and E is the set of edges representing the control flow between sub-regions. Note that G is a direct acyclic graph due to our definition and implementation of program regions.

FindExposedReads visits the nodes in V in *reverse* topological order (lines 5-6, Figure 6.6). A node is visited only after all its successor nodes have been visited. This order can be enforced by performing a topological sort on V [145]. The exposed reads of sub-regions are backward propagated along the control flow until they are covered by the

```
1: function FindExposedReads (G: subregions) begin
2:    // G = (V, E) is a directed acyclic graph for the sub-regions of region R
3:    // V, the set of nodes, represent the sub-regions
4:    // E, the set of edges, represent the control flow between sub-regions
5:    Topological sort V
6:    for v ∈ V in reverse topological order do
7:       if ∃ s0 ∈ v.successors then
8:          R_in := s0.R_out;
9:          for s ∈ (v.successors \ {s0}) do
10:            R_in := R_in ⊔ s.R_out;
11:         end for
12:      else
13:         R_in := {};
14:      end if
15:      // Let r be the corresponding region of v;
16:      R_gen := r.exposed_reads;
17:      W_gen := r.exposed_writes;
18:      v.R_out := R_gen ⊕ (R_in ⊖ W_gen);
19:   end for
20:   // Let v_entry be the entry node of V
21:   return v_entry.R_out;
22: end function
```

Figure 6.6 The pseudo-code of **FindExposedReads**

exposed writes of other sub-regions. Otherwise, they will pass through the entry node v_{entry} and become the exposed reads of region R.

For each node $v \in V$, let r be the corresponding region of v; W_{gen} be the set of exposed writes of r; R_{gen} be the set of exposed reads of r; $v.R_{out}$ be the set of reads that propagate through v. The $v.R_{out}$ can be calculated as follows (lines 7-18, Figure 6.6).

First, the sets of reads that propagate through the successors of v are merged together to form R_{in}, the set of reads entering v (lines 7-14, Figure 6.6). The **Merge** ($\sqcup$) operation basically takes two sets of MADs, merges the MADs that may access the same memory

objects, and produces a set of MADs with disjoint memory accesses. Section 6.2.4.2 will explain **Merge** in more details.

Next, **FindExposedReads** checks whether any reads in R_{in} are (partially) covered by any writes in W_{gen}, ($R_{in} \ominus W_{gen}$, line 18, Figure 6.6). In addition to finding the "difference" between two sets of MADs, the **Subtract** ($\ominus$) operation also helps identify producer and consumer relation. Section 6.2.4.3 will have more detailed explanation of **Subtract**.

Then, $v.R_{out}$ can be obtained by concatenating R_{gen} with ($R_{in} \ominus W_{gen}$). The **Concatenate** ($\oplus$) operation basically takes two sets of MADs, concatenates the MADs that may access the same memory objects, and produces a set of MADs with mutually exclusive memory accesses. **Concatenate** ($\oplus$) differs from **Merge** ($\sqcup$) in that the result of **Concatenate** ($\oplus$) depends on the order of its operands, but the result of **Merge** ($\sqcup$) is independent of the order of its operands. Section 6.2.4.1 will explain **Concatenate** in more details.

Finally, **FindExposedReads** returns $v_{entry}.R_{out}$ as the exposed reads of region R, (line 21, Figure 6.6).

6.2.2.1 An Example

This section will use the example in Figure 6.7 to illustrate **FindExposedReads**. In reverse topological order, **FindExposedReads** may visit the 4 regions [2] in Figure 6.7 in the order of region3 first, then region2, then region1, and finally region0 as follows[3].

[2]For brevity, we do not distinguish between a graph node and its corresponding region.
[3]The order of visiting region2 and region1 is arbitrary.

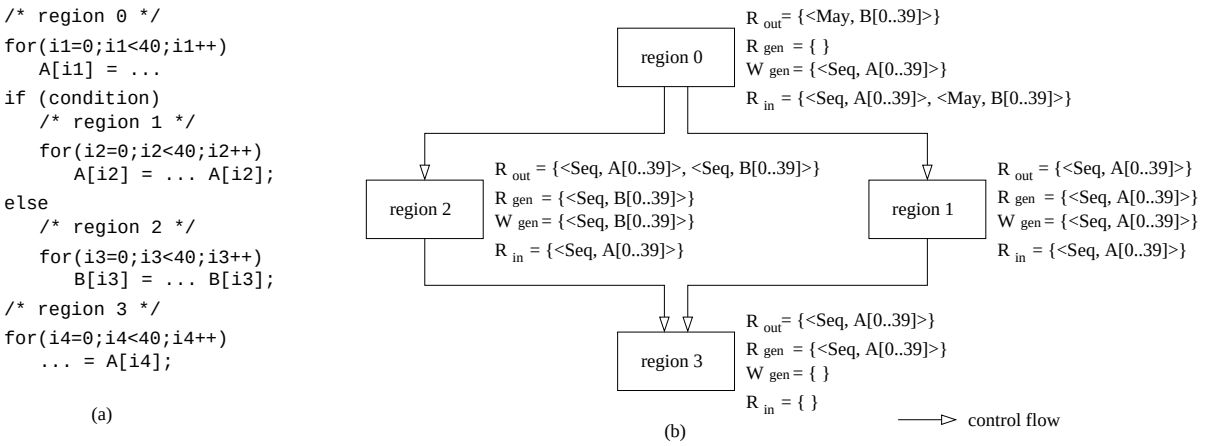


Figure 6.7 Example illustrating **FindExposedReads**

1. v = region3: The R_{in} of region3 is $\{\}$, because region3 has no successor. The exposed reads of region3 is $\{\langle Seq, \texttt{A}[0..39]\rangle\}$ and the exposed writes is $\{\}$. Therefore, the R_{out} of region3 can be calculated as follows.

$$\begin{aligned} \text{region3}.R_{out} &= \text{region3}.R_{gen} \oplus (\text{region3}.R_{in} \ominus \text{region3}.W_{gen}) \\ &= \{\langle Seq, \texttt{A}[0..39]\rangle\} \oplus (\{\} \ominus \{\}) \\ &= \{\langle Seq, \texttt{A}[0..39]\rangle\} \end{aligned}$$

2. v = region2: Because region3 is the only successor of region2, the R_{in} of region2 is the R_{out} of region3. Both the exposed reads and exposed writes of region2 is $\{\langle Seq, \texttt{B}[0..39]\rangle\}$, thus the R_{out} of region2 can be calculated as follows.

$$\begin{aligned} \text{region2}.R_{out} &= \text{region2}.R_{gen} \oplus (\text{region2}.R_{in} \ominus \text{region2}.W_{gen}) \\ &= \{\langle Seq, \texttt{B}[0..39]\rangle\} \oplus (\{\langle Seq, \texttt{A}[0..39]\rangle\} \ominus \{\langle Seq, \texttt{B}[0..39]\rangle\}) \\ &= \{\langle Seq, \texttt{B}[0..39]\rangle\} \oplus (\{\langle Seq, \texttt{A}[0..39]\rangle\}) \\ &= \{\langle Seq, \texttt{A}[0..39]\rangle, \langle Seq, \texttt{B}[0..39]\rangle\} \end{aligned}$$

3. v = region1: This is similar to the case of region2, except that both the exposed reads and the exposed writes of region1 are $\{\langle Seq, \mathtt{A}[0..39]\rangle\}$. The R_{out} of region1 can be calculated as follows.

$$\begin{aligned}
\text{region1}.R_{out} &= \text{region1}.R_{gen} \oplus (\text{region1}.R_{in} \ominus \text{region1}.W_{gen}) \\
&= \{\langle Seq, \mathtt{A}[0..39]\rangle\} \oplus (\{\langle Seq, \mathtt{A}[0..39]\rangle\} \ominus \{\langle Seq, \mathtt{A}[0..39]\rangle\}) \\
&= \{\langle Seq, \mathtt{A}[0..39]\rangle\} \oplus (\{\}) \\
&= \{\langle Seq, \mathtt{A}[0..39]\rangle\}
\end{aligned}$$

Note that although both the R_{out} of region1 and the R_{out} of region2 have the same $\langle Seq, \mathtt{A}[0..39]\rangle$, they are generated by different regions. The $\langle Seq, \mathtt{A}[0..39]\rangle$ in the R_{out} of region2 is the exposed reads of region3. While the $\langle Seq, \mathtt{A}[0..39]\rangle$ in the R_{out} of region1 is the exposed reads of region1, not region3, because the exposed reads of region3 is covered by the exposed writes of region1. The MAD structure can track the originating regions of its memory accesses. More details on this will be discussed later.

4. v = region0: region0 has two successors, region1 and region2, thus the R_{out} of region1 and the R_{out} of region2 will be merged together to form the R_{in} of region0.

$$\begin{aligned}
\text{region0}.R_{in} &= \text{region1}.R_{out} \sqcup \text{region2}.R_{out} \\
&= \{\langle Seq, \mathtt{A}[0..39]\rangle\} \sqcup \{\langle Seq, \mathtt{A}[0..39]\rangle, \langle Seq, \mathtt{B}[0..39]\rangle\} \\
&= \{\langle Seq, \mathtt{A}[0..39]\rangle, \langle May, \mathtt{B}[0..39]\rangle\}
\end{aligned}$$

Note that both the R_{out} of region1 and the R_{out} of region2 have $\langle Seq, \mathtt{A}[0..39]\rangle$, so the R_{in} of region0 includes $\langle Seq, \mathtt{A}[0..39]\rangle$. On the other hand, only the R_{out} of region2 has $\langle Seq, \mathtt{B}[0..39]\rangle$. Therefore, **Merge** ($\sqcup$) will generate a new *May*-type MAD, $\langle May, \mathtt{B}[0..39]\rangle$, to be included in the R_{in} of region0. Here, the type *May* means the memory access $\mathtt{B}[0..39]$ *may* happen, if the control flow actually reaches region2. Because region0 has only exposed writes $\{\langle Seq, \mathtt{A}[0..39]\rangle\}$, but no exposed reads, the R_{out} of region0 can be obtained as follows.

$$\begin{aligned}
\text{region0}.R_{out} &= \text{region0}.R_{gen} \oplus (\text{region0}.R_{in} \ominus \text{region0}.W_{gen}) \\
&= \{\} \oplus (\{\langle Seq, \mathtt{A}[0..39]\rangle, \langle May, \mathtt{B}[0..39]\rangle\} \ominus \{\langle Seq, \mathtt{A}[0..39]\rangle\}) \\
&= \{\} \oplus (\{\langle May, \mathtt{B}[0..39]\rangle\}) \\
&= \{\langle May, \mathtt{B}[0..39]\rangle\}
\end{aligned}$$

Since region0 is the entry subregion in Figure 6.7(b), the R_{out} of region0 becomes the exposed reads of the program segment in Figure 6.7(a), that is $\{\langle May, \mathtt{B}[0..39]\rangle\}$. So **FindExposedReads** has deduced that the program segment in Figure 6.7(a) *may* need $\mathtt{B}[0..39]$ from the outside.

6.2.3 Finding Exposed Writes

The algorithm of **FindExposedWrite** is outlined in Figure 6.8. It is like a "reversed" version of **FindExposedReads**. **FindExposedWrites** visits the nodes of V in topological order (lines 5-6, Figure 6.8). A node in V is visited only after all its predecessors

```
 1: function FindExposedWrites (G: subregions) begin
 2:    // G ≡ (V, E) is a directed acyclic graph
 3:    // V, the set of nodes, represent the regions
 4:    // E, the set of edges, represent the control flow between regions
 5:    Topological sort V
 6:    for v ∈ V in topological order do
 7:       if ∃ p_0 ∈ v.predecessors then
 8:          W_in := p_0.W_out;
 9:          for p ∈ (v.predecessors \ {p_0}) do
10:             W_in := W_in ⊔ p.W_out;
11:          end for
12:       else
13:          W_in := {};
14:       end if
15:       // let r be the corresponding region of v;
16:       W_gen := r.exposed_writes;
17:       v.W_out := (W_in ⊖ W_gen) ⊕ W_gen;
18:    end for
19:    // let v_exit be the exit node of V
20:    return v_exit.W_out;
21: end function
```

Figure 6.8 The pseudo-code of **FindExposedWrites**

have been visited. The exposed writes of sub-regions are forward propagated along the control flow until they are killed by the exposed writes of other sub-regions. Otherwise, they will pass through the exit node v_{exit} and become the exposed writes of region R.

For each node $v \in V$, let r be the corresponding region of v; W_{gen} be the set of exposed writes of r; R_{gen} be the set of exposed reads of r; $v.W_{out}$ be the set of writes that propagate through v. The $v.W_{out}$ can be calculated as follows (lines 7-17, Figure 6.8).

First, the sets of writes propagated through the predecessors of v are merged together to form W_{in}, the set of writes entering v (lines 7-14, Figure 6.8). Next, **FindExposedWrites** checks whether any writes in W_{in} are (partially) killed by the writes in W_{gen} ($W_{in} \ominus W_{gen}$, line 17, Figure 6.8). Then, $v.W_{out}$ can be obtained by concatenating

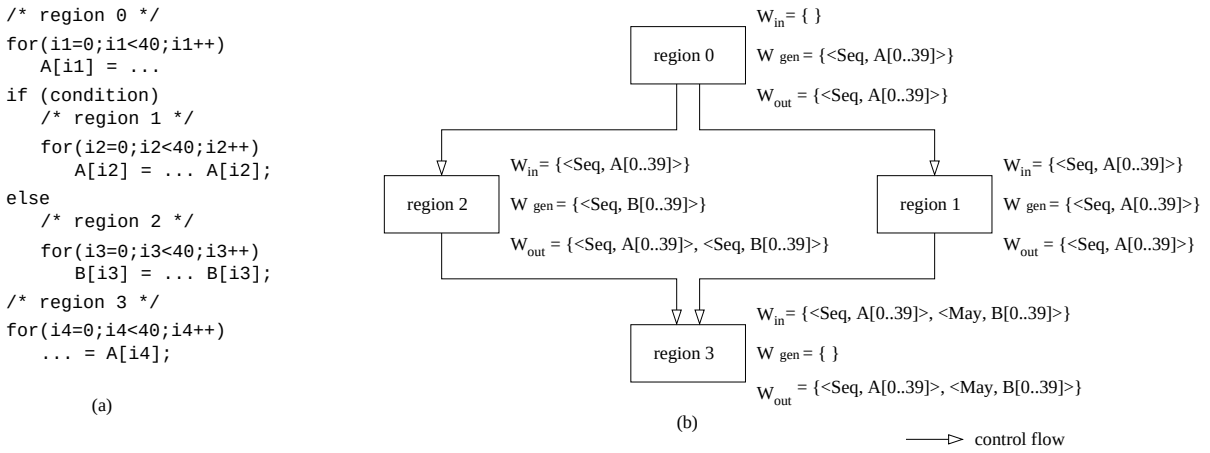


Figure 6.9 Example illustrating **FindExposedWrites**

$(W_{in} \ominus W_{gen})$ with W_{gen}. Finally, **FindExposedWrites** returns $v_{exit}.R_{out}$ as the exposed writes of region R (line 20, Figure 6.8).

FindExposedWrites and **FindExposedReads** apply the same **Merge** ($\sqcup$) and **Concatenate** ($\oplus$) operations. The **Subtract** ($\ominus$) operation is essentially the same, except that when invoked by **FindExposedWrites**, it will not identify any producer-consumer relationship.

6.2.3.1 An Example

Figure 6.9 uses the same example in Figure 6.7(a) to illustrate **FindExposedWrites**. Here the regions in Figure 6.9 will be visited in topological order with region0 first, then region2, then region1, and finally region3[4].

1. v = region0: Because region0 is the entry node in Figure 6.9, the W_{in} of region0 is $\{\}$. Given the exposed writes of region3, $\{\langle Seq, \mathtt{A}[0..39]\rangle\}$, the W_{out} of region0 can

[4]Again, the order of region2 and region1 is arbitrary.

be calculated as follows.

$$\begin{aligned}
\text{region0}.W_{out} &= (\text{region0}.W_{in} \ominus \text{region3}.W_{gen}) \oplus \text{region0}.W_{gen} \\
&= (\{\} \ominus \{\langle Seq, \texttt{A}[0..39]\rangle\}) \oplus \{\langle Seq, \texttt{A}[0..39]\rangle\} \\
&= \{\} \oplus \{\langle Seq, \texttt{A}[0..39]\rangle\} \\
&= \{\langle Seq, \texttt{A}[0..39]\rangle\}
\end{aligned}$$

2. v = region2: Because region0 is the only predecessor of region2, the W_{in} of region2 is the W_{out} of region0. Given the exposed writes of region2, $\{\langle Seq, \texttt{B}[0..39]\rangle\}$, the W_{out} of region2 can be calculated as follows.

$$\begin{aligned}
\text{region2}.W_{out} &= (\text{region2}.W_{in} \ominus \text{region2}.W_{gen}) \oplus \text{region2}.W_{gen} \\
&= (\{\langle Seq, \texttt{A}[0..39]\rangle\} \ominus \{\langle Seq, \texttt{B}[0..39]\rangle\}) \oplus \{\langle Seq, \texttt{B}[0..39]\rangle\} \\
&= (\{\langle Seq, \texttt{A}[0..39]\rangle\}) \oplus \{\langle Seq, \texttt{B}[0..39]\rangle\} \\
&= \{\langle Seq, \texttt{A}[0..39]\rangle, \langle Seq, \texttt{B}[0..39]\rangle\}
\end{aligned}$$

3. v = region1: The W_{in} of region1 is the same as the W_{in} of region2. Given the exposed writes of region1, $\{\langle Seq, \texttt{A}[0..39]\rangle\}$, the W_{out} of region1 can be calculated as follows.

$$\begin{aligned}
\text{region1}.W_{out} &= (\text{region1}.W_{in} \ominus \text{region1}.W_{gen}) \oplus \text{region1}.W_{gen} \\
&= (\{\langle Seq, \texttt{A}[0..39]\rangle\} \ominus \{\langle Seq, \texttt{A}[0..39]\rangle\}) \oplus \{\langle Seq, \texttt{A}[0..39]\rangle\} \\
&= (\{\}) \oplus \{\langle Seq, \texttt{A}[0..39]\rangle\} \\
&= \{\langle Seq, \texttt{A}[0..39]\rangle\}
\end{aligned}$$

Note that, although $\langle Seq, \texttt{A}[0..39]\rangle$ is in both the W_{out} of region1 and the W_{out} of region2, the one in the W_{out} of region1 and the one in the W_{out} of region2 are generated by different regions. The one in the W_{out} of region2 is the exposed writes of region0. However,the one in the W_{out} of region1 is its own exposed writes, because the exposed writes of region0 are killed by the exposed writes of region1.

4. v = region3: The W_{in} of region0 is obtained by merging the W_{out}'s of its predecessors, region1 and region2.

$$\begin{aligned}
\text{region0}.W_{in} &= \text{region1}.W_{out} \sqcup \text{region2}.W_{out} \\
&= \{\langle Seq, \texttt{A}[0..39]\rangle\} \sqcup \{\langle Seq, \texttt{A}[0..39]\rangle, \langle Seq, \texttt{B}[0..39]\rangle\} \\
&= \{\langle Seq, \texttt{A}[0..39]\rangle, \langle May, \texttt{B}[0..39]\rangle\}
\end{aligned}$$

Note that the *type* of memory access $\texttt{B}[0..39]$ in the W_{in} of region3 is *May* because it propagates to region3 only from region2, but not from region1. The W_{out} of region3 can be calculated as follows, given that region3 has no exposed writes.

$$\begin{aligned}
\text{region3}.W_{out} &= (\text{region3}.W_{in} \ominus \text{region3}.W_{gen}) \oplus \text{region0}.W_{gen} \\
&= (\{\langle Seq, \texttt{A}[0..39]\rangle, \langle May, \texttt{B}[0..39]\rangle\} \ominus \{\}) \oplus \{\}) \\
&= (\{\langle Seq, \texttt{A}[0..39]\rangle, \langle May, \texttt{B}[0..39]\rangle\}) \oplus \{\}) \\
&= \{\langle Seq, \texttt{A}[0..39]\rangle, \langle May, \texttt{B}[0..39]\rangle\}
\end{aligned}$$

The W_{out} of region3 becomes the exposed writes of the program segment in Figure 6.9(a), because region3 is the exit sub-region. Finally, **FindExposedWrites** re-

turns $\{\langle Seq, \texttt{A}[0..39]\rangle, \langle May, \texttt{B}[0..39]\rangle\}$, which means the program segment in Figure 6.7(a) writes sequentially to array `A` from element 0 to element 39, meanwhile, it *may* write to array `B` from element 0 to element 39.

6.2.4 Memory Access Descriptor Operations

This section will explain the **Concatenate** ($\oplus$), **Merge** ($\cup$), **Subtract** ($\ominus$), and **Summation** ($\sum$) operations used by **Summarize**, **FindExposedReads**, and **FindExposedWrites**.

6.2.4.1 Concatenate ($\oplus$)

Figure 6.10 shows the top-level algorithm for the **Concatenate** operation. The input operands of **Concatenate** are two sets of MAD structures, $S_{in,1}$ and $S_{in,2}$. The elements in $S_{in,1}$ are disjoint memory accesses in the sense that, for different u and v in $S_{in,1}$, $u.alias$ and $v.alias$ are disjoint sets of memory objects. So are the elements in $S_{in,2}$. Also, the set of MADs returned by **Concatenate**, S_{out}, will also have this property.

Concatenate basically does pair-wise comparison between the elements of $S_{in,1}$ and the elements of $S_{in,2}$ (lines 4-22, Figure 6.10). For $m_1 \in S_{in,1}$ and $m_2 \in S_{in,2}$ which *may* access the same memory objects (line 8, Figure 6.10), **ConcatenateMAD** is invoked to "concatenate" the memory access patterns of m_1 and m_2 (line 10, Figure 6.10). Some examples of concatenating two memory access patterns are shown in Figure 6.13 [5].

[5]Here we overload $\oplus$ for both the operation on two MADs and the operation on two sets of MADs

```
1: function Concatenate (S_in,1, S_in,2) // S_in,1 ⊕ S_in,2 begin
2:    // S_in,1, S_in,2 : set of memory access descriptors (MAD)
3:    S_out := {};
4:    for m1 ∈ S_in,1 do
5:       m1_m2_concatenated := False;
6:       S_tmp := {};
7:       for m2 ∈ S_in,2 do
8:          if m1.alias ∩ m2.alias ≠ {} then
9:             // m1 and m2 may access the same objects
10:            m1 := ConcatenateMAD(m1, m2);
11:            m1_m2_concatenated := True;
12:         else
13:            S_tmp := S_tmp ∪ {m2};
14:         end if
15:      end for
16:      if m1_m2_concatenated = True then
17:         S_tmp := S_tmp ∪ {m1};
18:      else
19:         S_out := S_out ∪ {m1};
20:      end if
21:      S_in,2 := S_tmp;
22:   end for
23:   S_out := S_out ∪ S_in,2;
24:   return S_out;
25: end function
```

Figure 6.10 The pseudo-code of **Concatenate** ($\oplus$)

Figure 6.11 shows how **ConcatenateMAD** generates a new MAD structure m with the "concatenated" memory access pattern from the input MAD operands, m_1 and m_2. The *may-alias* set of m is the union of the *may-alias* sets of m_1 and m_2 (line 4, Figure 6.11). The *components* field of m is generated by **CombineComponents** to keep track of the originating program regions of its constituent MADs. This will be explained in more details.

If the *type* of m_1, or m_2, is *Doomed*, or if m_1 and m_2 have different *bases*, **ConcatenateMAD** will just give up, and create a new *Doomed*-type MAD (lines 6-9, Figure 6.11).

```
1: function ConcatenateMAD (m1, m2) begin
2:    // m1, m2: memory access descriptor MAD
3:    m := new MAD;
4:    m.alias := m1.alias ∪ m2.alias;
5:    m.components := CombineComponents (m1, m2);
6:    if m1.type = Doomed or m2.type = Doomed then
7:       (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
8:    else if m1.base ≠ m2.base then
9:       (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
10:   else
11:      // m1.base = m2.base
12:      m.base = m1.base;
13:      if m1.type ≠ m2.type then
14:         down grade m1 or m2 so that they have the same type;
15:      end if
16:      (m.type, m.offset, m.displace) := ConcatenatePattern (m1, m2);
17:   end if
18:   return m;
19: end function
```

Figure 6.11 The pseudo-code of **ConcatenateMAD**

Otherwise, **ConcatenateMAD** will first adjust m_1 and m_2 so that they have the same *type*, "down grading" one of them if necessary. Then, the adjusted access patterns of m_1 and m_2 will be concatenated as accurately as possible (lines 12-16, Figure 6.11).

ConcatenatePattern, shown in Figure 6.12, essentially compares the *type*, *offset*, and *displace* fields of m_1 and m_2 to determine the *type*, *offset* and *displace* field of the new MAD. First, it will try to generate a new pattern of the same *type* as m_1 and m_2. If this is not possible, it will try a pattern of less accuracy. For example, the current implementation cannot concatenate the two *Seq*-type patterns in Figure 6.13(b) to another *Seq*-type pattern, **ConcatenatePattern** will then concatenate these two patterns into a *Must*-type pattern. If m_1 and m_2 are *May*-type MADs, concatenating them is

```
1: function ConcatenatePattern (m1, m2) begin
2:   // m1, m2: memory access descriptors MAD
3:   // Assume m1.type = m2.type
4:   if m1.type = Seq then
5:     Try to generate a new Seq-type pattern (offset, displace) by concatenating (m1.offset,
       m1.displace) and (m2.offset, m2.displace);
6:     if Succeeded then
7:       return (Seq, offset, displace);
8:     else
9:       down grade m1 and m2 to Must-type MADs;
10:    end if
11:  end if
12:  if m1.type = Must then
13:    Try to generate a new Must-type pattern (offset, displace) by concatenating
       (m1.offset, m1.displace) and (m2.offset, m2.displace);
14:    if Succeeded then
15:      return (Must, offset, displace);
16:    else
17:      down grade m1 and m2 to May-type MADs;
18:    end if
19:  end if
20:  // m1 and m2 are May-type MADs
21:  return MergePattern (m1, m2);
22: end function
```

Figure 6.12 The pseudo-code of **ConcatenatePattern**

the same as merging them (line 21, Figure 6.12). Merging two MADs are explained in Section 6.2.4.2.

Before explaining **Merge**, here we explain the **CombineComponents** function shown in Figure 6.14, which is invoked by both **Concatenate** and **Merge**. **CombineComponents** basically combines the *components* fields of the MAD operands of concatenation or merge operations. The *components* field of the original MAD structures for the exposed reads and the exposed writes of sub-regions is initially set to empty. During the process of backward or forward propagation, MAD structures will be concate-

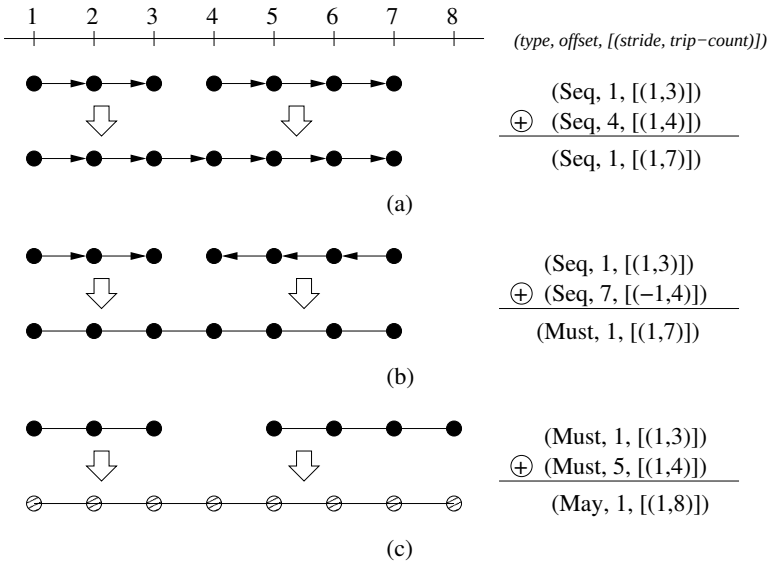


Figure 6.13 Examples of concatenating two memory access patterns

```
1: function CombineComponents (m1, m2) begin
2:    // m1, m2: memory access descriptor MAD
3:    if m1.components = {} and m2.components = {} then
4:       return {m1, m2};
5:    else if m1.components = {} and m2.components ≠ {} then
6:       return {m1} ∪ m2.components;
7:    else if m1.components ≠ {} and m2.components = {} then
8:       return m1.components ∪ {m2};
9:    else
10:      // m1.components ≠ {} and m2.components ≠ {}
11:      return m1.components ∪ m2.components;
12:   end if
13: end function
```

Figure 6.14 The pseudo-code of **CombineComponents**

nated or merged with each other to form new MAD structures. The *components* field of these new MAD structures due to concatenation or merge operations will then keep track of the concatenated or merged MAD structures and their generating sub-regions [6]. The

[6] In addition to the fields describing memory access patterns, the MAD structure also has book-keeping fields, including the generating subg-region.

components field of MAD will be used by **Subtract** when identifying producer-consumer relationship between sub-regions. This will be explained in more details when discussing **Subtract**.

6.2.4.2 Merge (⊔)

Merge and the auxiliary functions, **MergeMAD** and **MergePattern**, are shown in Figures 6.15, 6.16, and 6.17, which have very similar algorithmic structures as the **Concatenate**, **ConcatenateMAD**, and **ConcatenatePattern** shown in Figures 6.10 to 6.12.

The *may*-alias set and the *components* of the merged MAD are obtained in the same way as a concatenated MAD (lines 4-5, Figure 6.16). However, there are still some differences between **Merge** and **Concatenate**, because **Concatenate** is applied when propagating MAD along straight line of code, while **Merge** is applied at the confluence point of control flow. Figure 6.18 shows some examples of merging two memory access patterns.

The major difference between **Merge** and **Concatenate** is that if a *Seq*-type or *Must*-type MAD is not merged with other MADs, it will be down graded to a *May-type* MAD (lines 22-23, 27-31, Figure 6.15). An example of this is shown in Figure 6.7. When calculating the R_{in} of region0, the $\langle Seq, \mathtt{B}[0..39]\rangle$ in the R_{out} of region2 is not merged with any MAD in the R_{out} of region1, and thus it is down graded to $\langle May, \mathtt{B}[0..39]\rangle$.

Like **ConcatenatePattern** (Figure 6.12), **MergePattern** (Figure 6.17) will try to produce, as accurate as possible, a memory access pattern by comparing the *type*, *offset*,

```
1: function Merge (S_in,1, S_in,2) // S_in,1 ⊔ S_in,2 begin
2:    // S_in,1, S_in,2 : set of memory access descriptors (MAD)
3:    S_out := {};
4:    for m_2 ∈S_in,1 do
5:       m_2.merged := False;
6:    end for
7:    for m_1 ∈ S_in,1 do
8:       m_1.merged := False;
9:       S_tmp := {};
10:      for m_2 ∈ S_in,2 do
11:         if m_1.alias ∩ m_2.alias ≠ {} then
12:            // m_1 and m_2 may access the same objects
13:            m_1 := MergeMAD(m_1, m_2);
14:            m_1.merged := True;
15:         else
16:            S_tmp := S_tmp ∪ {m_2};
17:         end if
18:      end for
19:      if m_1.merged = True then
20:         S_tmp := S_tmp ∪ {m_1};
21:      else
22:         down grade m_1 to May-type MAD;
23:         S_out := S_out ∪ {m_1};
24:      end if
25:      S_in,2 := S_tmp;
26:   end for
27:   for m_2 ∈ S_in,2 do
28:      if m_2.merged = False then
29:         down grade m_2 to May-type MAD;
30:      end if
31:   end for
32:   S_out := S_out ∪ S_in,2;
33:   return S_out;
34: end function
```

Figure 6.15 The pseudo-code of **Merge** ($\sqcup$)

and *displace* fields of its input MADs. For the current implementation, two *Seq*-type (*Must*-type) memory access patterns will be merged into a *Seq*-type (*Must*-type) pattern only if they are the same (line 4 and line 11, Figure 6.17), as the example shown in

```
1: function MergeMAD (m1, m2) begin
2:    // m1, m2: memory access descriptor MAD
3:    m := new MAD;
4:    m.alias := m1.alias ∪ m2.alias;
5:    m.components := CombineComponents (m1, m2);
6:    if m1.type = Doomed or m2.type = Doomed then
7:       (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
8:    else if m1.base ≠ m2.base then
9:       (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
10:   else
11:      // m1.base = m2.base
12:      m.base = m1.base;
13:      if m1.type ≠ m2.type then
14:         down grade m1 or m2 so that they are of the same type;
15:      end if
16:      (m.type, m.offset, m.displace) := MergePattern (m1, m2);
17:   end if
18:   return m;
19: end function
```

Figure 6.16 The pseudo-code of **MergeMAD**

Figure 6.18 (b); otherwise, **MergePattern** will generate a pattern of less accurate *type*. The worst scenario is that **MergePattern** totally gives up, and returns a *Doomed*-type memory access pattern (line 21, Figure 6.17).

Note that **Merge** ($\sqcup$) is commutative, but **Concatenate** ($\oplus$) and **Subtract** ($\ominus$) are not.

6.2.4.3 Subtract ($\ominus$)

The **Subtract** operation, shown in Figure 6.19, basically calculates the "difference" between two sets of MADs, $S_{in,1}$ and $S_{in,2}$. Unlike the the input sets of **Concatenate** and **Merge**, which are either both memory read accesses or both memory write accesses, the second input operand $S_{in,2}$ of **Subtract** is always a set of memory *write* accesses.

```
1: function MergePattern (m1, m2) begin
2:    // Assume m1.type = m2.type
3:    if m1.type = Seq then
4:       if (m1.offset, m1.displace) = (m2.offset, m2.displace) then
5:          return (Seq, m1.offset, m1.displace);
6:       else
7:          down grade m1 and m2 to Must-type MADs;
8:       end if
9:    end if
10:   if m1.type = Must then
11:      if (m1.offset, m1.displace) = (m2.offset, m2.displace) then
12:         return (Must, m1.offset, m1.displace);
13:      else
14:         down grade m1 and m2 to May-type MADs;
15:      end if
16:   end if
17:   Try to generate a new May-type pattern (offset, displace) by merging (m1.offset,
      m1.displace) and (m2.offset, m2.displace);
18:   if Succeeded then
19:      return (May, offset, displace);
20:   else
21:      return (Doomed, ⊥, ⊥);
22:   end if
23: end function
```

Figure 6.17 The pseudo-code of **MergePattern**

Like **Concatenate** and **Merge**, **Subtract** also does pair-wise comparison between the elements of $S_{in,1}$ and $S_{in,2}$ (lines 4-5, Figure 6.19). Each m_1 in $S_{in,1}$ is "subtracted" by any m_2 in $S_{in,2}$ which may access the same memory objects (lines 6-8, Figure 6.19). If m_1 is not totally covered by m_2, ($m_1 \neq \perp$, line 13, Figure 6.19), a new MAD describing the remaining memory accesses of m_1 will be included in the returned S_{out} (lines 13-14, Figure 6.19).

SubtractMAD, shown in Figure 6.20, is the function responsible for generating a MAD to describe the memory accesses that are in m_1, but not in m_2. At the beginning,

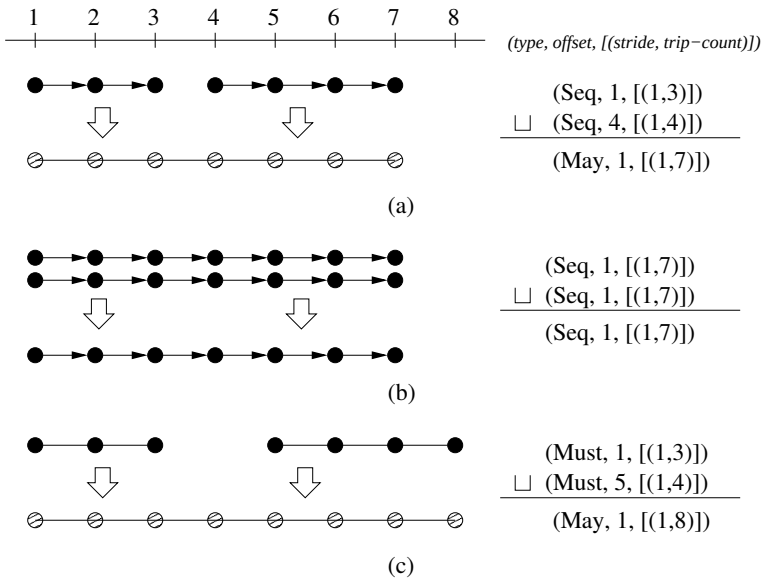


Figure 6.18 Examples of merging two memory access patterns

```
1: function Subtract (S_in,1, S_in,2) /* S_in,1 ⊖ S_in,2 */ begin
2:   // S_in,1, S_in,2 : set of memory access descriptors (MAD)
3:   S_out := {};
4:   for m1 ∈ S_in,1 do
5:     for m2 ∈ S_in,2 do
6:       if m1.alias ∩ m2.alias ≠ {} then
7:         m1 := SubtractMAD(m1, m2);
8:       end if
9:       if m1 = ⊥ then
10:        break;
11:      end if
12:    end for
13:    if m1 ≠ ⊥ then
14:      S_out := S_out ∪ {m1};
15:    end if
16:  end for
17:  return S_out;
18: end function
```

Figure 6.19 The pseudo-code of **Subtract** ($\ominus$)

```
1: function SubtractMAD (m1, m2) begin
2:   // m1: a MAD for memory read or write
3:   // m2: a MAD for memory write
4:   comps := {x : ∃c ∈ m1.components, x = SubtractMAD(c, m2) ∧ x ≠ ⊥};
5:   if m1.components ≠ {} and comps = {} then
6:     m := ⊥;
7:   else
8:     if m1.type = Doomed then
9:       m := m1; m.components := comps;
10:    else if m2.type = Doomed or m1.base ≠ m2.base then
11:      m := m1 down graded to May-type; m.components := comps;
12:    else // m1.base = m2.base
13:      if IntersectPattern (m1, m2) = False then
14:        m := m1;
15:      else
16:        if m2.type = May then
17:          m := m1 down graded to May-type; m.components := comps;
18:        else // m2.type = Seq or Must
19:          if PatternCovered (m1, m2) = True then
20:            m := ⊥;
21:          else
22:            m := new MAD;
23:            (m.components, m.alias, m.base) := (comps, m1.alias, m1.base);
24:            (m.type, m.offset, m.displace) := Pattern_subtract (m1, m2);
25:          end if
26:        end if
27:      end if
28:    end if
29:    if m ≠ m1 and m1 is memory read and m1.components = {} then
30:      // identified a producer-consumer relationship
31:      m1.producer := m1.producer ∪ {m2};
32:      m2.consumer := m2.consumer ∪ {m1};
33:      m2.Consumed := True;
34:    end if
35:  end if
36:  return m;
37: end function
```

Figure 6.20 The pseudo-code of **SubtractMAD**

SubtractMAD recursively calls itself to subtract the components of m_1 by m_2 (line 4, Figure 6.20). This is because SubtractMAD is also responsible for identifying the

```
1: function Pattern_subtract (m1, m2) begin
2:    // m1: a May-,Must- or Seq-type MAD for memory read or write
3:    // m2: a Must- or Seq-type MAD for memory write
4:    Try to describe the memory locations which are in m1 but not in m2, using a
      memory access pattern (m1.type, offset, displace);
5:    if Succeeded then
6:       return (m1.type, offset, displace);
7:    else
8:       return (May, m1.offset, m1.displace);
9:    end if
10: end function
```

Figure 6.21 The pseudo-code of **Pattern_subtract**

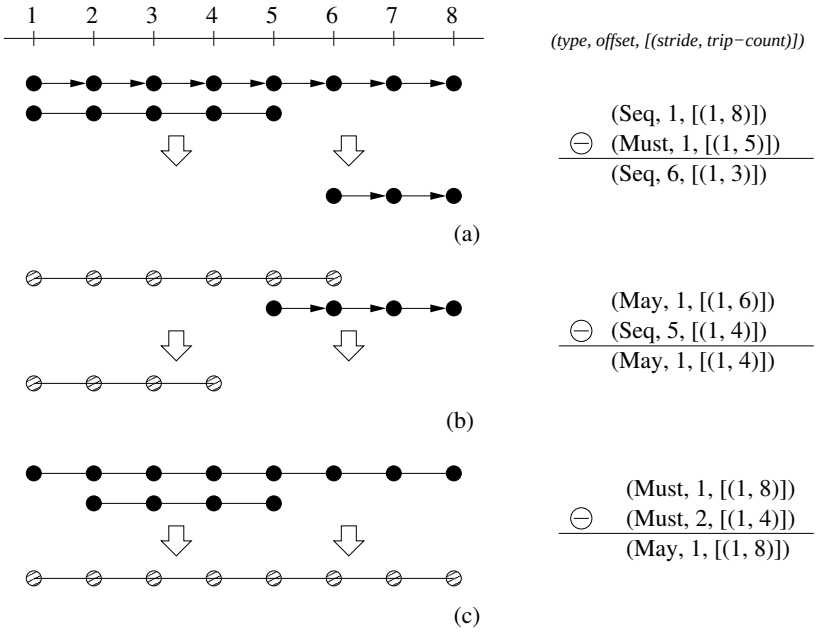


Figure 6.22 Examples of subtracting two memory access patterns

producer-consumer relationship between program regions, and the components of a MAD may be generated by different program regions. If m_1 is a composite MAD, and it has no component MADs left after the subtraction, **SubtractMAD** will return $\perp$, meaning that m_1 is totally covered or killed by m_2.

1: **function IntersectPattern** (m_1, m_2) **begin**
2: // m_1: a MAD for memory *read* or *write*
3: // m_2: a MAD for memory *write*
4: Let $m_1.displace = [(s_{1,1}, T_{1,1}) \dots (s_{1,D_1}, T_{1,D_1})]$.
5: Let $m_2.displace = [(s_{2,1}, T_{2,1}) \dots (s_{2,D_i}, T_{2,D_2})]$.
6: // where s: stride, T: trip count.
7: **return** *True*, if the following proposition holds; otherwise, *False*.

$$\begin{gathered} \exists i_{1,j}, 0 \leq i_{1,j} < T_{1,j}, j = 1 \dots D_1 \\ \exists i_{2,k}, 0 \leq i_{2,k} < T_{1,k}, k = 1 \dots D_2 \\ m_1.offset + \sum_{j=1}^{D_1} i_{1,j} \cdot s_{1,j} = m_2.offset + \sum_{k=1}^{D_2} i_{2,k} \cdot s_{2,k} \end{gathered}$$

8: **end function**

Figure 6.23 The pseudo-code of **IntersectPattern**

1: **function PatternCovered** (m_1, m_2) **begin**
2: // m_1: a MAD for memory *read* or *write*
3: // m_2: a MAD for memory *write*
4: Let $m_1.displace = [(s_{1,1}, T_{1,1}) \dots (s_{1,D_1}, T_{1,D_1})]$.
5: Let $m_2.displace = [(s_{2,1}, T_{2,1}) \dots (s_{2,D_i}, T_{2,D_2})]$.
6: // where s: stride, T: trip count.
7: **return** *True*, if the following proposition holds; otherwise, *False*.

$$\begin{gathered} \forall i_{1,j}, 0 \leq i_{1,j} < T_{1,j}, j = 1 \dots D_1 \\ \exists i_{2,k}, 0 \leq i_{2,k} < T_{1,k}, k = 1 \dots D_2 \\ m_1.offset + \sum_{j=1}^{D_1} i_{1,j} \cdot s_{1,j} = m_2.offset + \sum_{k=1}^{D_2} i_{2,k} \cdot s_{2,k} \end{gathered}$$

8: **end function**

Figure 6.24 The pseudo-code of **PatternCovered**

If m_1 is a *Doomed-type* MAD, the result of subtraction will still be still a *Doom*-type MAD (lines 8-9, Figure 6.20). **SubtractMAD** has a chance to figure out exactly which part of m_1 is subtracted, only if m_2 is not a *Doomed-type* MAD and m_1 and m_2 have the same *bases* (lines 12-28, Figure 6.20). Otherwise, at best it can return a down graded version of m_1 (lines 10-11, Figure 6.20).

If the memory accesses of m_1 and m_2 have no overlap, as determined by **Intersect-Pattern**, which is shown in Figure 6.23 and will be explained later, the same m_1 can be returned intact (lines 13-14, Figure 6.20). If the memory accesses of m_1 and m_2 do intersect, but m_2 is a *May*-type MAD, **SubtractMAD** can at best figure out which part of m_1 *may* be subtracted, and thus a *May*-type m_1 is the best possible MAD that **SubtractMAD** can generate (lines 16-17, Figure 6.20).

If m_2 is *Must*- or *Seq*-type MAD, **SubtractMAD** first invokes **PatternCovered**, which is shown in Figure 6.24 and will be explained later, to check whether m_1 is totally covered by m_2. If so, **SubtractMAD** will return $\perp$ (lines 19-20, Figure 6.20). If m_1 is only partially subtracted by m_2, **SubtractMAD** will call **Pattern_subtract** to determine the *type*, *offset*, and *displace* of the remaining memory accesses of m_1 subtracted by m_2. Figure 6.22 shows some examples of the special cases which can be handled by **Pattern_subtract** in the current implementation.

Figure 6.23 outlines the problem formulation for determining whether memory access m_1 intersects with memory access m_2. Basically it is an integer programming problem. If the system of inequalities in Figure 6.23 has solution, m_1 and m_2 will have intersection. This thesis work relies on the Omega test package [101] for solving the integer programming problem.

Figure 6.24 formulates the problem of whether memory access m_1 is a subset of memory access m_2. It requires the evaluation of Presburger formula, which consists of affine equality and inequality constraints on integer variables, combined with logical operators $\wedge$, $\vee$, $\neg$ and existential quantifiers $\forall$, $\exists$. In general, it is a much more difficult

```
1: function Summation (S_in, L) // ∑_L (S_in) begin
2:     // S_in : set of memory access descriptor (MAD)
3:     // L: loop
4:     S_out := {};
5:     for m ∈ S_in do
6:         S_out := S_out ∪ SummationMAD(m, L);
7:     end for
8:     return S_out;
9: end function
```

Figure 6.25 The pseudo-code of **Summation**

problem than the integer programming problem shown in Figure 6.23. For special cases like the one in Figure 6.24, a solver based on the Omega test can solve the problem quickly [136] [152] [153].

Finally, if m_1 and m_2 have intersection, or equivalently $m \equiv (m_1 - m_2) \neq m_1$, line 29, Figure 6.20, a producer-consumer relationship between the generating program region of m_2 and the generating program region of m_1 is found. **SubtractMAD** will record this relation by including the generating program region of m_2 in the producer set of m_1, and the generating program region of m_1 in the consumer set of m_2, and marking $m_2.Consumed$ as *True* (lines 31-33, Figure 6.20).

6.2.4.4 Summation ($\sum$)

The function **Summation** ($\sum$), shown in Figure 6.25, is for finding the set of exposed memory accesses of a *Loop* region, given the exposed memory accesses of the enclosed *Loop Body* region S_{in}, and the corresponding loop L. As suggested by the name, the functionality of **Summation** can be implemented as concatenating the exposed memory accesses of all the iterations, as illustrated in Figure 6.28.

```
1: function SummationMAD (m_i, L) begin
2:    // m_i : memory access descriptor (MAD)
3:    // L: loop
4:    m := new MAD;
5:    if m_i.type = Doomed then
6:       (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
7:    else if m_i.offset is an unknown induction expression then
8:       (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
9:    else
10:      Find T, the trip count of L;
11:      if T is unknown then
12:         (m.type, m.base, m.offset, m.displace) := (Doomed, ⊥, ⊥, ⊥);
13:      else
14:         if T is an upper bound then
15:            down grade m to a May-typed MAD;
16:         end if
17:         m.base := m_i.base;
18:         (m.type, m.offset, m.displace) := SummationMAD (m_i, T);
19:      end if
20:   end if
21:   return m;
22: end function
```

Figure 6.26 The pseudo-code of **SummationMAD**

For example, in Figure 6.28(a), the loop body has a *Seq*-type exposed memory access with $\mathit{offset} = (1 + h)$, and $\mathit{displace} \equiv [(\mathit{stride}, \mathit{trip\text{-}count})] = [(0, 1)]$, where h is the fundamental induction variable of some loop that iterates 8 times. As indicated by $\mathit{stride} = 0$ and $\mathit{trip\text{-}count} = 1$, for a particular iteration h, the loop body accesses only one memory location. The relative address of accessed location, with respect to the *base*, is given by the *offset*, $h + 1$. From iteration 1 to iteration 8, for $h = 0, 1, 2, \ldots 7$, the whole loop will access the memory locations from 1 to 8 [7]. Therefore, the memory access

[7]Precisely, we should say "memory location of relative address 1 with respect to the *base*."

```
1: function SummationMAD (m_i, T) begin
2:   // m_i : a MAD descriptor
3:   // T : trip count of loop L
4:   Let m_1.offset = c + s_L · h_L,
5:   // where h_L is the fundamental induction variable of loop L.
6:   if m_i is a read with inter-iteration data dependence then
7:     down grade m_i to May-type;
8:   end if
9:   if m_i.type = Seq then
10:    Try to generate a Seq-type pattern (offset, displace), comparing the relationship
       between c, s_L, m_i.displace, and T.
11:    if Succeeded then
12:      return (Seq, offset, displace);
13:    else
14:      down grade m_i to Must-type
15:    end if
16:  end if
17:  if m_i.type = Must then
18:    Try to generate a Must-type pattern (offset, displace), comparing the relationship
       between c, s_L, m_i.displace, and T.
19:    if Succeeded then
20:      return (Must, offset, displace);
21:    else
22:      down grade m_i to May-type
23:    end if
24:  end if
25:  if m_i.type = May then
26:    Try to generate a May-type pattern (offset, displace), comparing the relationship
       between c, s_L, m_i.displace, and T.
27:    if Succeeded then
28:      return (May, offset, displace);
29:    else
30:      return (Doomed, ⊥, ⊥);
31:    end if
32:  end if
33: end function
```

Figure 6.27 The pseudo-code of **SummationMAD**

pattern of the whole loop is $(Seq, 1, [(1, 8)])$, using the $(type, offset, [(stride, trip\text{-}count)])$ notation.

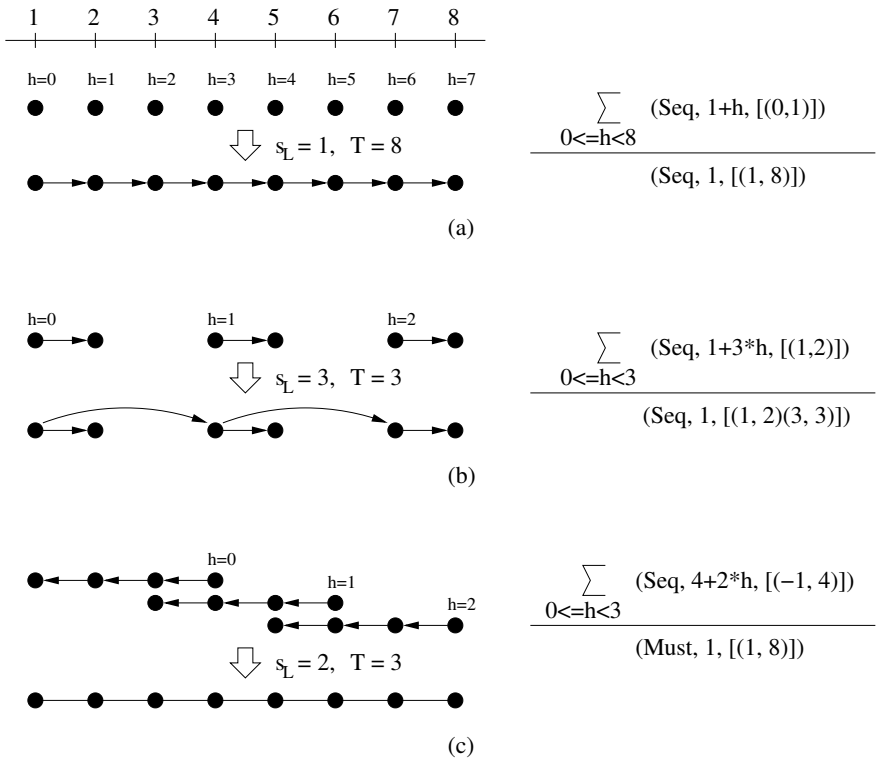


Figure 6.28 Example illustrating **Summation** ($\sum$)

For nested loops, we may need more than one pairs of (*stride*, *trip-count*) to describe the memory access pattern of the whole loop. For example, in Figure 6.28 (b), the memory access pattern of the inner loop has *offset* = $(1+3h)$, where h is the fundamental induction variable of the outer loop. This means, when the outer loop iterates, the starting memory access location of the inner loop will shift to the right by 3, the coefficient of h in the *offset*. Since the loop trip-count of the outer loop is 3, as indicated by $0 \leq h < 3$, the *displace* of the memory access for the whole loop will be $[(1, 2)(3, 3)]$, where $(1, 2)$ describing the *stride* and the *trip-count* of the memory accesses of inner loop, and $(3, 3)$ describing the

stride of the starting point of the inner loop memory access and the *trip-count* of the outer loop.

For the examples in Figure 6.28(a) and (b), the *type* of the memory access of the loop body is preserved by summation. It is not always possible to describe the memory accesses of a *Loop* region as accurate as describe the memory accesses of the corresponding *Loop Body* region. For example, the *Loop Body* region in Figure 6.28(c) has a *Seq*-type MAD, but the memory accesses of the *Loop* region can be only described using *Must*-type MAD in the current implementation.

Implementing **Summation** by concatenating the memory accesses of the loop body for all iterations is not efficient. **SummationMAD**, Figure 6.26, outlines how to do summation, given a MAD m_i and the corresponding loop L. As illustrated in Figure 6.28, the key is to figure out the trip count of L, and the coefficient of the fundamental induction variable of L in the *offset* of m_i.

If m_i is a *Doomed*-type MAD, or if the *offset* of m_i is an induction expression which can not be represented as an affine expression in terms of fundamental induction variables, or if the trip count of loop L is unknown, **SummationMAD** just returns a *Doomed*-type MAD (lines 5-12, Figure 6.26). If we know the loop trip count T, but T is just an upper bound, m_i is conservatively down graded to *May*-type (lines 14-16, Figure 6.28), because L may iterate less than T times. If T is the exact loop trip count of L, **SummationMAD** in Figure 6.27 will try to find the *type*, *offset*, and *displace* of MAD as accurate as possible.

SummationMAD first separates the *offset* of m_i to two terms, $s_L \cdot h_L$ and c, where h_L is the fundamental induction variable of loop L (line 4, Figure 6.27). Then, **Sum-**

mationMAD will try to generate a pattern with the same *type* as m_i by checking the relationship between c, s_L, m_i, *displace*, and T. If this is not possible, **SummationMAD** will try less accurate descriptors until it gives up, and returns a *Doomed type* pattern (lines 9-32, Figure 6.27).

There is one subtlety in **SummationMAD**. If m_i is a memory read access and has some inter-iteration dependence, some of its data will come from previous iterations, instead of from outside the loop. Thus, the summation of the exposed reads of loop body for all the iterations should be calculated as follows. Note that the exposed reads R and the exposed writes W of the loop body are functions of the fundamental induction variable h of loop L.

$$\underbrace{R(0)}_{\text{1st iteration}} \underbrace{\oplus \sum_{h_L^r=1}^{T} (R(h_L^r) \ominus \sum_{h_L^w=0}^{h_L^r-1} (W(h_L^w))}_{\text{the rest iterations}}$$

The calculation of this formula is complicated. A conservative but quick approximation is to down grade m_i to *May*-type if it has any inter-iteration true dependence (lines 6-8, Figure 6.27).

6.3 Related Work

People have developed techniques to summarize the side effects of procedures in order to perform dependence tests across procedure boundary [64] [154] [155] [149]. A summary

of side effects describes the sets of memory locations that the procedure reads and writes, called the *use*-set and the *modify*-set in literature.

The summaries are propagated in the program call graph from bottom up. The summary of a procedure is generated by combining the memory access information of its own loops with the summary information propagated from its callees. For conservatively identifying possible dependencies between procedures, this approach is efficient and effective enough.

However, the information provided by pair-wise dependence tests between program regions is too conservative for more advanced parallelization techniques, which require more accurate information about the data-flow between program regions [156] [56]. The array data-flow problem is first addressed by Feautrier [57], who developed a technique called parametric integer programming [157] to derive, for each memory read, the corresponding memory write which generates the data. Array data-flow analysis based on the parametric integer programming method has two major problem. First, the complexity of parametric integer programming could be high[8]. Second, it can handle only control structures like the Fortran `DO`-loop, but not arbitrary control flow. To address the first problem, researchers developed a more efficient, but less general, method that can handle most of the common cases [60], which, however, still can not handle arbitrary control flow.

[8]While the original paper claimed parametric integer programming method is practical [57], other authors claimed it is not practical [60].

In general, it is very difficult, if not impossible, to exactly describe the producer-consumer relation for programs with arbitrary control flow. For some parallelization purposes, for example, array privatization, exact producer-consumer relation is not necessary. Array privatization could enable more loop parallelization by eliminating false dependences between loop iterations. It replicates the arrays so that each iteration gets its own private copy. Array privatization can be applied to a loop as long as we can prove that every read in the loop gets its data from a write in the same loop iteration. This is a weaker condition than knowing the producer of each read.

Based on this observation, researchers have developed array data-flow analysis techniques which are capable of handling arbitrary control flow, and also efficient and effective enough for array privatization [61] [62] [58] [71] [63] [147] [59] [158]. Essentially all these works follow the same approach of partitioning the program into regions, summarizing the memory accesses for each region, propagating and combining the summary information in the control flow graph and the program call graph. They differ from each other mainly in the data structures that represent the memory accesses of each region, and the complexities of the operations that manipulate these data structures.

The concepts and techniques developed in these works laid the foundation for this work. Because of different target languages and different type of parallelisms exploited, this work differs from the previous works in the following aspects.

- Previous works on parallelizing compilers mainly target scientific applications written in Fortran. On the other hand, this work targets the programs written in C. While scientific Fortran programs use arrays as their main data structures, C

programs can have more complicated data structures referenced through pointers. The data structure MAD used in this work for describing the memory accesses in C programs must incorporate pointer informations, and the operations on the MAD data structure must manipulate the included *may*-alias set. This is not necessary in the previous works.

- In the previous works, the summary of each program region can only tell the *set* of accessed memory locations, but not the *order* of accessing these memory locations. This is sufficient for array data-flow analysis to identify candidate loops for array privatization. However, for the potential optimizations shown in Figure 3.6, we need to know not only the accessed memory locations, but also the memory access order. The MAD data structure used in this work is designed for a memory data flow analysis whose lattice values also contain the memory access order information. The operations on MAD will try to preserve the memory access order information before moving up the data flow value lattice.

- For the array data flow analysis designed for array privatization, the goal is to exploit coarse-grained data parallelism in the outer loop, so it is not a concern for them to identify the producer-consumer relations between program regions. On the other hand, the goal of this thesis work is to uncover coarse-grained function parallelism, so the bottom-up process not only summarizes the memory accesses for each program region, but also identifies the possible producer-consumer relations between program regions. Because of this, the MAD data structure used in this

work not only records the memory access pattern, but also tracks the generting progrom regions of memory accesses.

The main focus of this work is identifying the producer-consumer relations between program regions, which is also the fundamental cause of the differences between this work and the previous works. Next chapter will discuss how the producer-consumer relations identified by the bottom up process can be refined by a top-down process.

CHAPTER 7

Producer-Consumer Relation Analysis

The producer-consumer relations among program regions are identified in two phases. During the bottom-up summarization process discussed in Chapter 6, we constructs a conservative producer-consumer relation. This producer-consumer relation is then refined by an ensuing top-down pruning process. These two phases are explained in the following sections.

7.1 Bottom-up Phase

During the bottom-up summarization process, to summarize the exposed reads of region R, we forward propagates the exposed reads of its sub-regions along the edges in the sub-region graph of R. When propagating the exposed reads of sub-region R_r through sub-region R_w, we subtract the exposed writes of R_w from the exposed reads of R_r. If the `SubtractMAD` operation deduces that an exposed write of R_w and an exposed read of R_r access some common memory locations, it will record this new identified producer-consumer relation between R_w and R_r, and mark the exposed write of R_w as *Consumed* (line 33, Figure 6.20).

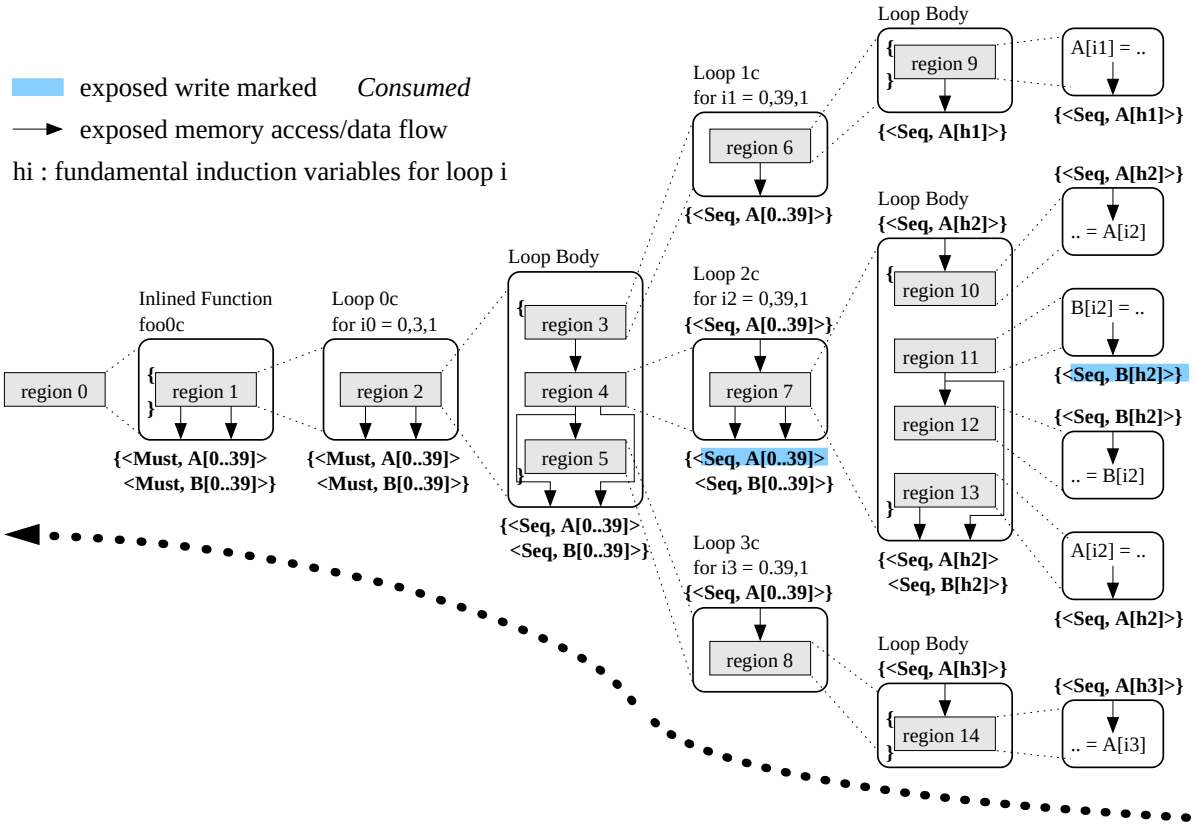


Figure 7.1 Illustration of the bottom-up phase

For example, when summarizing the exposed reads of region7, in Figure 7.1, we will propagate the exposed reads of region12 through region11. Because both region11 and region12 access the same memory, $\mathtt{B}[h_2]$, there exists a producer-consumer relationship between region11 and region12, and the exposed write $\langle Seq, \mathtt{B}[h_2]\rangle$ of region11 will be marked as *Consumed*. Similarly, when summarizing the exposed reads of region2, we will identify the producer-consumer relationship between region4 and region5, and the exposed write $\langle Seq, \mathtt{A}[0..39]\rangle$ of region4 is marked as *Consumed*.

Although the bottom-up phase can identify the producer-consumer relationship between the sub-regions of region R, the exposed writes of the sub-regions may or may not

be consumed outside the region R. For example, in Figure 7.1, both region11 and region13 are sub-regions of region7. The exposed write $\langle Seq, \mathtt{A}[h_2]\rangle$ of region13 is consumed by region5, a region outside region7, but the exposed write $\langle Seq, \mathtt{B}[h_2]\rangle$ of region11 has no consumer outside region7.

Being confined within the scope of region7, the bottom-up phase does not know whether any region outside region7 will consume the exposed writes of region11 or not, so it must conservatively included both $\langle Seq, \mathtt{A}[h_2]\rangle$ and $\langle Seq, \mathtt{B}[h_2]\rangle$ in the exposed writes of region7. To prune the spurious exposed writes like $\langle Seq, \mathtt{B}[h_2]\rangle$, we need a top-down phase after the bottom-up phase.

7.2 Top-down Phase

Figure 7.2 illustrates the top-down pruning process using the same example in Figure 7.1. The top-down pruning process starts from the top-level region, region0 in this case. Since region0 is the top-level region, no other region will consume the exposed writes of region0. So we can prune all the exposed writes of region0, as indicated by crossing the exposed writes with red lines in Figure 7.2. Next, we prune the exposed writes of the sub-regions of region0.

If an exposed write w of region0 is pruned, which means it has no consumer outside region0, none of the components of w will be consumed outside region0. Note that the components of w of region0 are the exposed writes of some sub-regions of region0. So, if a component c of w is an exposed write of sub-region R, and c is not marked as *Consumed*

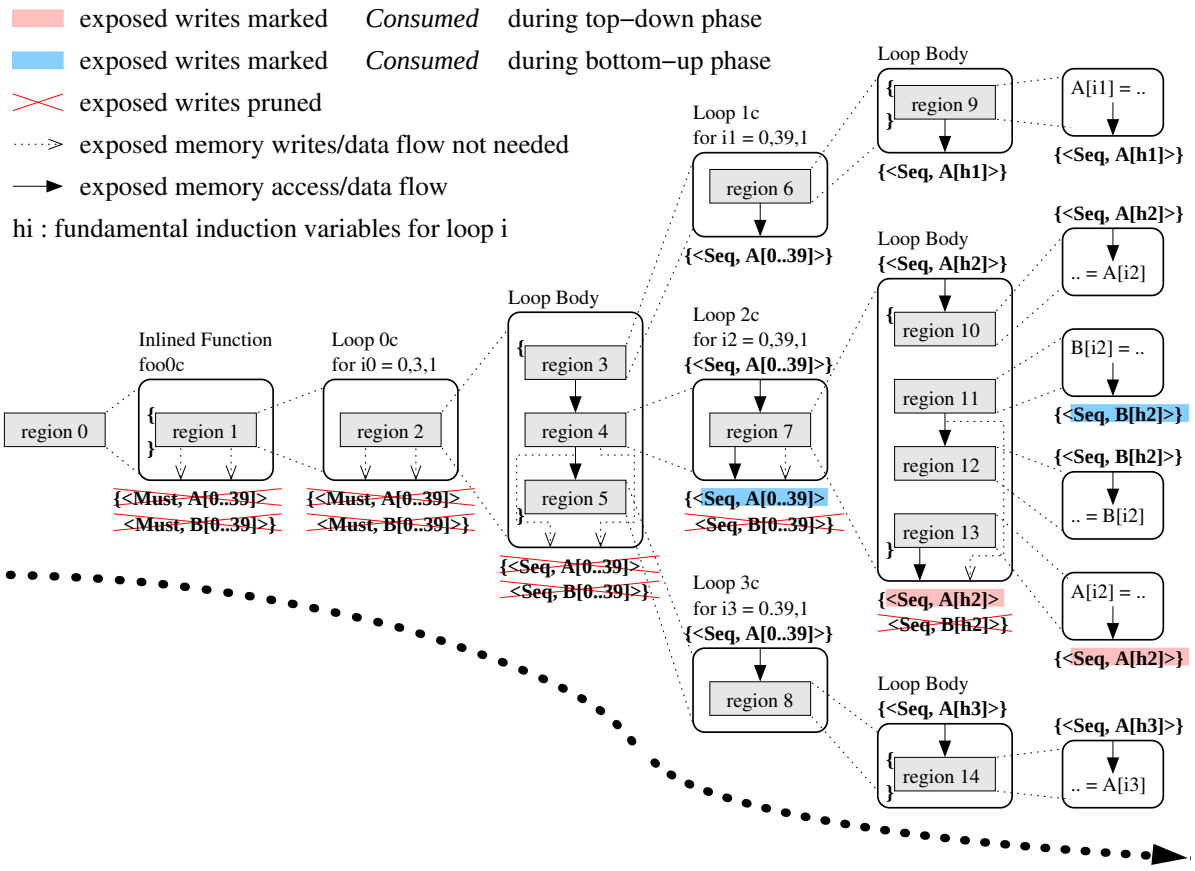


Figure 7.2 Illustration of the top-down phase

when summarizing the exposed memory accesses of region0, that mean c is not consumed by any sub-region of region0, either. Therefore, we can prune c from the exposed writes of R. So, we can prune the exposed writes of region1, the only sub-region of region0, then we can similarly prune the exposed writes of region2, the only sub-region of region1.

Note that, in Figure 7.2, only $\langle Seq, \mathtt{B}[0..39]\rangle$ is pruned from the exposed writes of region4, but not $\langle Seq, \mathtt{A}[0..39]\rangle$. This is because $\langle Seq, \mathtt{A}[0..39]\rangle$ is consumed by region5, and thus marked as *Consumed* during the bottom-up phase.

The components of the exposed writes of region4 are the exposed writes of its sub-region region7. For similar reason, we prune $\langle Seq, \mathtt{B}[h_2]\rangle$, but not $\langle Seq, \mathtt{A}[h_2]\rangle$ marked as

```
1: function PruneExposedWrites (R: a region) begin
2:   for w ∈ R.exposed_writes do
3:     if w.Consumed = True then
4:       for c ∈ x.components do
5:         c.Consumed := True;
6:       end for
7:     end if
8:   end for
9:   for r ∈ R.subregions do
10:    PruneExposedWrites(r);
11:  end for
12: end function
```

Figure 7.3 The pseudo-code of **PruneExposedWrites**

Consumed. While the pruning of the exposed write $\langle Seq, \mathtt{B}[h_2]\rangle$ of region7 is correct, the preservation of $\langle Seq, \mathtt{A}[h_2]\rangle$ is still a conservative approximation. This is because only a *subset* of A[0..39] can be consumed, but by marking $\langle Seq, \mathtt{A}[h_2]\rangle$ as *Consumed*, we are still making a conservative assumption that *every* elements of A[0..39] are consumed. This should be the common case in practice.

The function **PruneExposedWrites** in Figure 7.3 outlines the top-down pruning process. Instead of explicitly pruning exposed writes, **PruneExposedWrites** marks those exposed writes which should not be pruned as *Consumed*. For any exposed write which is marked *Consumed*, **PruneExposedWrites** marks its components as *Consumed* (Lines 2-8, Figure 7.3). Then, the pruning process will continue for the sub-regions of R (Lines 9-11, Figure 7.3).

7.3 Related Work

The top-down pruning phase basically refines the live range of variables. Accurate live range information also benefits parallelization techniques like array privatization. A privatized array must be written back to the global memory, only if is read after the privatized loop. Researchers have proposed another phase of analysis backward propagating the memory access summary of loops in the control flow graph to extend scalar liveness analysis for array liveness analysis [62] [159].

This work is different from the previous works in the following aspects.

- For the purpose of array privatization, live range information is only needed for privatized arrays in privatized loops. For our purpose, we need to do liveness analysis for the exposed writes of every program region, not just for the exposed writes of loop which can be privatized.

- Instead of having another compiler pass for liveness analysis as suggested by previous works, the liveness analysis in this work is partly done during the bottom-up phase by marking the exposed writes of program regions as *Consumed*. This greatly simplifies the top-down phase which essentially refines the live ranges of exposed writes.

The next chapter will discuss the experiment results of prototyping the memory data-flow analysis system, consisting the bottom-up process discussed in the previous chapter and the top-down process discussed in this chapter.

CHAPTER 8

Prototyping and Experiment Result

We implemented the memory data-flow analysis algorithms presented in the previous chapters on top of the IMPACT compiler infrastructure [160], which supports the needed software modules for the in-lining of whole program, the construction of control flow graph from abstract syntax tree, a flow-insensitive and context sensitive pointer analysis [102] [103], and the interface to the Omega library [101]. We tried the prototype program analysis system on extracting coarse-grained data-flow from several benchmark programs in the MediaBench suite [161] and the open-source programs of G.724 coder and decoder. This chapter will present the experiment results on the efficiency and effectiveness of the prototype memory data-flow analysis system.

8.1 Modification of Benchmark Programs

We made the following modifications on the benchmark programs to work around the limitations of the current prototype memory data-flow analysis system.

- The intrinsic functions `left_shift` and `right_shift` used in the G.724 coder and decoder are modified to remove recursion.

- The intrinsic functions used in G.724 coder and decoder are also simplified to eliminate the unnecessary details. These intrinsic functions are written for bit-accurate function simulation. However, the detailed modeling of bit-level operations only significantly increases the analysis time, with no improving on the analysis accuracy. In practice, the templates modeling the memory access behaviors of library functions are accurate enough for the purpose of memory data-flow analysis.

- The multi-entry loops in the MediaBench gsmdec and gsmenc programs are converted to single-entry loops.

- The call-sites of functions with variable number of arguments are renamed to functions with fixed number of arguments.

- Indirect function calls are converted to multiple direct function calls to enable whole program in-lining.

Section 5.2 has more detailed discussion on these modifications.

8.2 Verification and Visualization

For verification purpose, a graphical user interface is built to visualize the memory data-flow between program regions. For each program region, the visualization system could display its exposed reads, exposed writes, and sub-region graph.

Figure 8.1 demonstrates a sample output of the visualization system. By clicking on a grey box on the top, the visualization system will display the memory access pattern

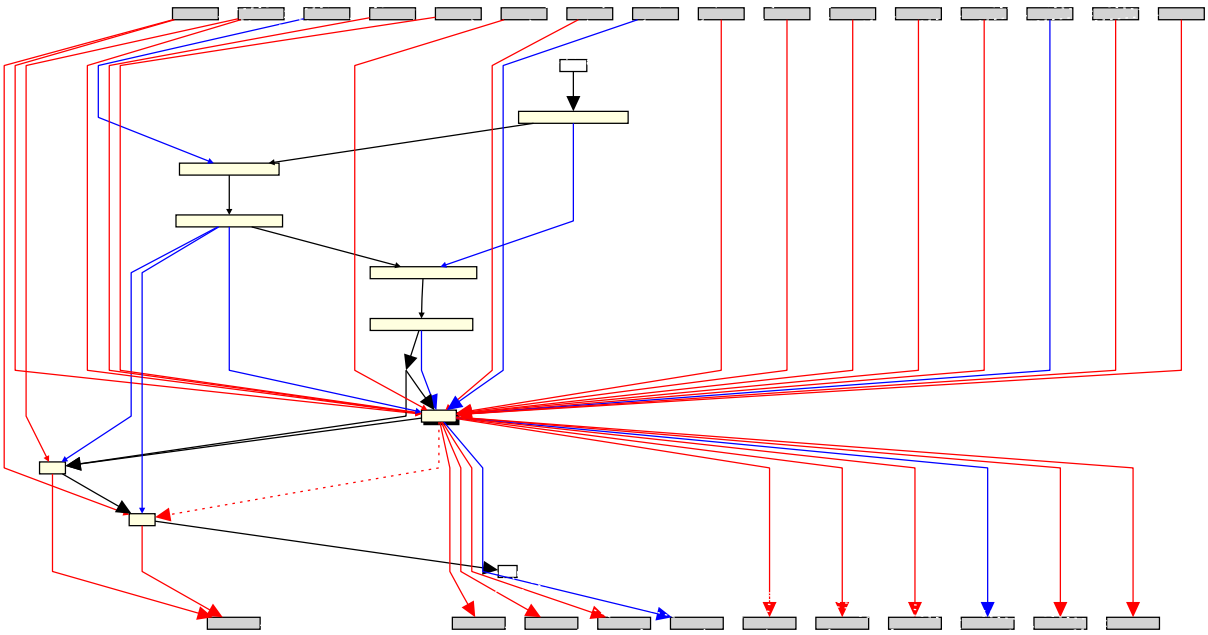


Figure 8.1 Demonstration of the memory data-flow visualization system

of the corresponding exposed reads. The grey boxes on the bottom are corresponding to the exposed writes. The yellow boxes in the middle are corresponding to the sub-regions. In addition to the control flow (black edges) between the sub-regions, the visualization system also displays the memory data-flow (red and blue edges) among the sub-regions. By clicking on a memory data-flow edge, the user can inspect the memory access pattern of the producer, which is the source node of the memory data-flow edge, and the memory access pattern of the consumer, which is the destination node of the memory data-flow edge. By clicking on a sub-region node in the sub-region graph, the user can navigate down the program region hierarchy[1].

[1]There is also a way for the user to navigate up the program region hierarchy.

Table 8.1 Breakdown of the execution time of the prototype memory data-flow analysis system

	adpcm D	adpcm E	g721 D	g721 E	g724 D	g724 E	gsm D	gsm E
I	0.121	0.119	2.160	2.000	9.408	31.377	34.484	19.967
II	0.104	0.117	1.201	1.092	3.348	11.424	15.549	19.180
III	0.035	0.049	2.289	7.881	9.629	161.12	92.057	119.510

Using this visualization tool, we manually check the memory data-flow analysis result of g724dec. We found the prototype system works as expected and generates satisfactory memory data-flow analysis result.

The visualization system is built on top of uDraw(Graph) [162] and Tcl/Tk [163]. During the bottom-up and top-down processes of the memory data-flow analysis, we retains all the necessary data structures and the analysis results which may be used by the visualization system. When the analysis is done, the visualization system will interact with uDraw(Graph) and Tcl/Tk to accept user requests. It will then retrieve the requested analysis results from the retained data structures, and send the reformatted data back to uDraw(Graph) and Tcl/Tk for display.

Although the visualization system is originally created for verification purpose, potentially we can enhance it to a full-fledged program visualization system serving other software engineering purposes.

8.3 Efficiency

The execution times of the prototype memory data-flow analysis system on the tested benchmark programs are listed in Table 8.1, which break down the execution time to three major components: (I) the in-lining time, (II) the pointer analysis time, and (III) the memory data-flow analysis time. For the benchmark programs used in this study, the memory data-flow analysis takes less than 3 minutes. However, these benchmarks are not very large programs. For large programs like JPEG or MPEG, the current in-lining based implementation may not be efficient, as suggested by comparing in-lining based pointer analysis with inter-procedural pointer analysis [103]. The major problem with the in-lining approach is that it may cause code bloat and increase the problem size exponentially. This may significantly increase the memory footprint and the execution time of the memory data-flow analysis.

It is very common that a function is invoked at different call-sites, and thus the same function is in-lined several times. However, these in-lined versions of the same function often have isomorphic memory data-flow analysis results. Therefore, for each function, we could potentially analyze its memory data-flow just once, then derive the memory data-flow analysis result at each call-site based on the calling context, without re-analyzing the same function. A potential implementation of an inter-procedural memory data-flow analysis will be discussed in the Chapter 9.

Figure 8.2 Example for illustrating spurious data producers

8.4 Effectiveness

The goal of memory data-flow analysis is to figure out an accurate producer-consumer relationship among program regions by eliminating false dependences. Therefore, we would like to understand whether there exist false dependences in real programs and whether our prototype memory data-flow analysis system can eliminate them. If there exists false dependences among program regions, which means some program regions have spurious data producers, our memory data-flow analysis system should filter out these spurious data producers.

When summarizing the exposed reads of a program region, the memory data-flow analysis will backward propagate the exposed reads of its sub-regions along the edges in the sub-region graph. During the backward propagation, the exposed reads of the sub-regions will be subtracted by the exposed writes of the sub-regions which could be their data producers. If an exposed read r of a program region R_r is totally covered by the exposed write of another program region R_w, the exposed read r will not be propagated, and R_w will be the last found data producer of R_r, if the basic block of R_w dominates the basic block of R_w. On the other hand, if we do not subtract r with w, and keep

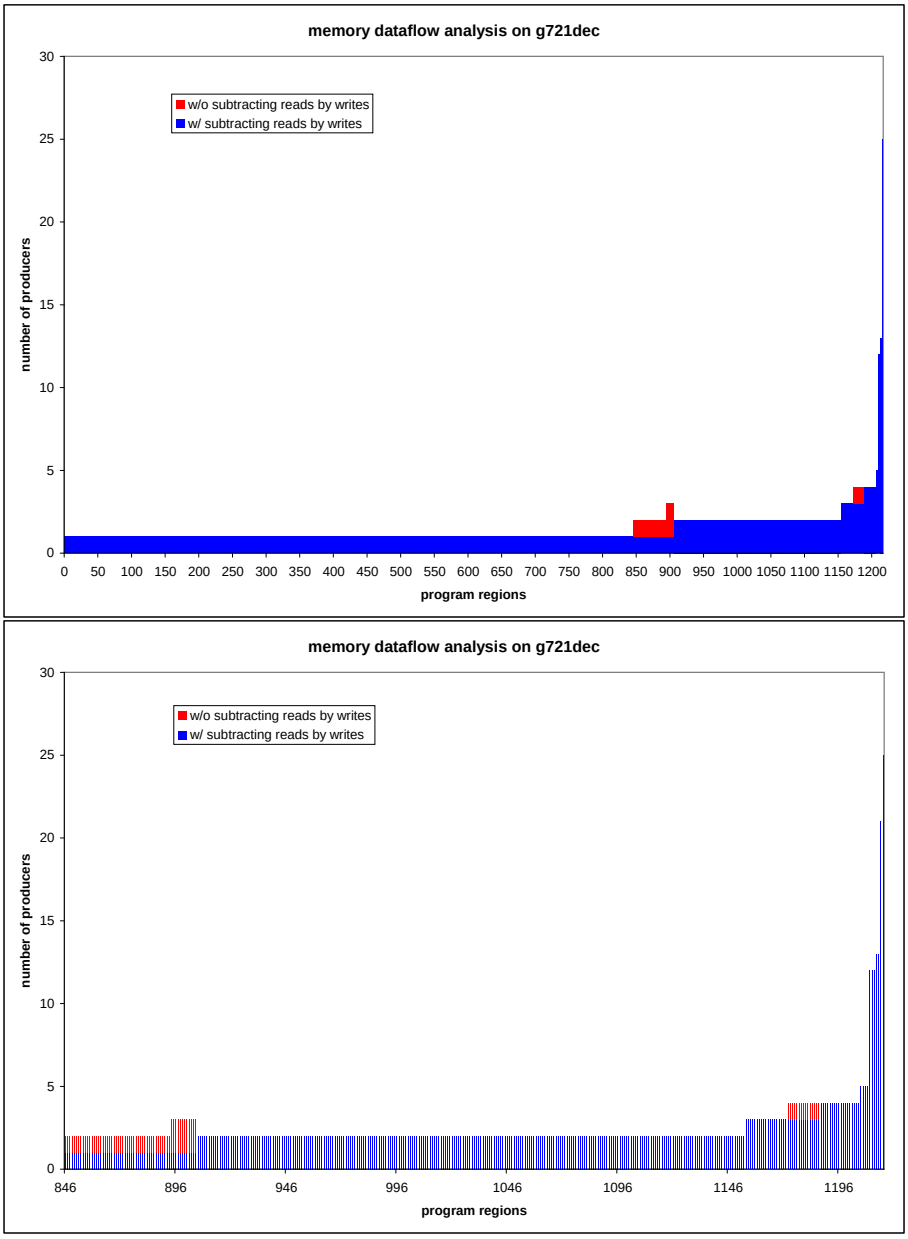


Figure 8.3 Eliminated spurious data producers (false dependences) in g721dec

propagating r beyond R_w, we could find spurious data producers for R_r, if there is other regions before R_w which write to the same memory locations as R_w, even though the basic block of R_w dominates the basic block of R_r.

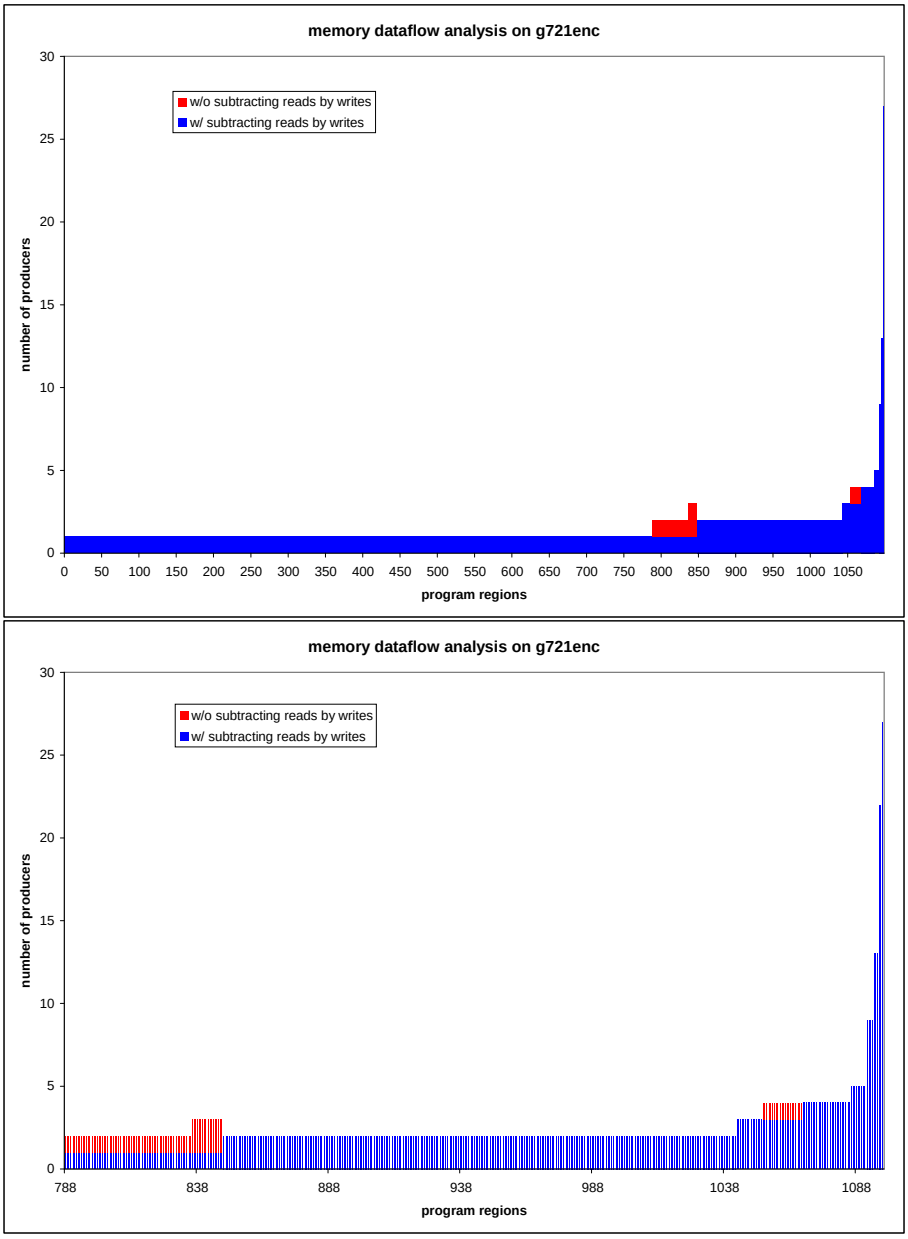


Figure 8.4 Eliminated spurious data producers (false dependences) in g721enc

For example, in Figure 8.2(a), when the exposed read of `region3`, `A[i]`, is propagated to `region2`, it is totally covered by the exposed write of `region2`, which is also `A[i]`. Therefore, `A[i]` will not be propagated further, and `region2` is the only data

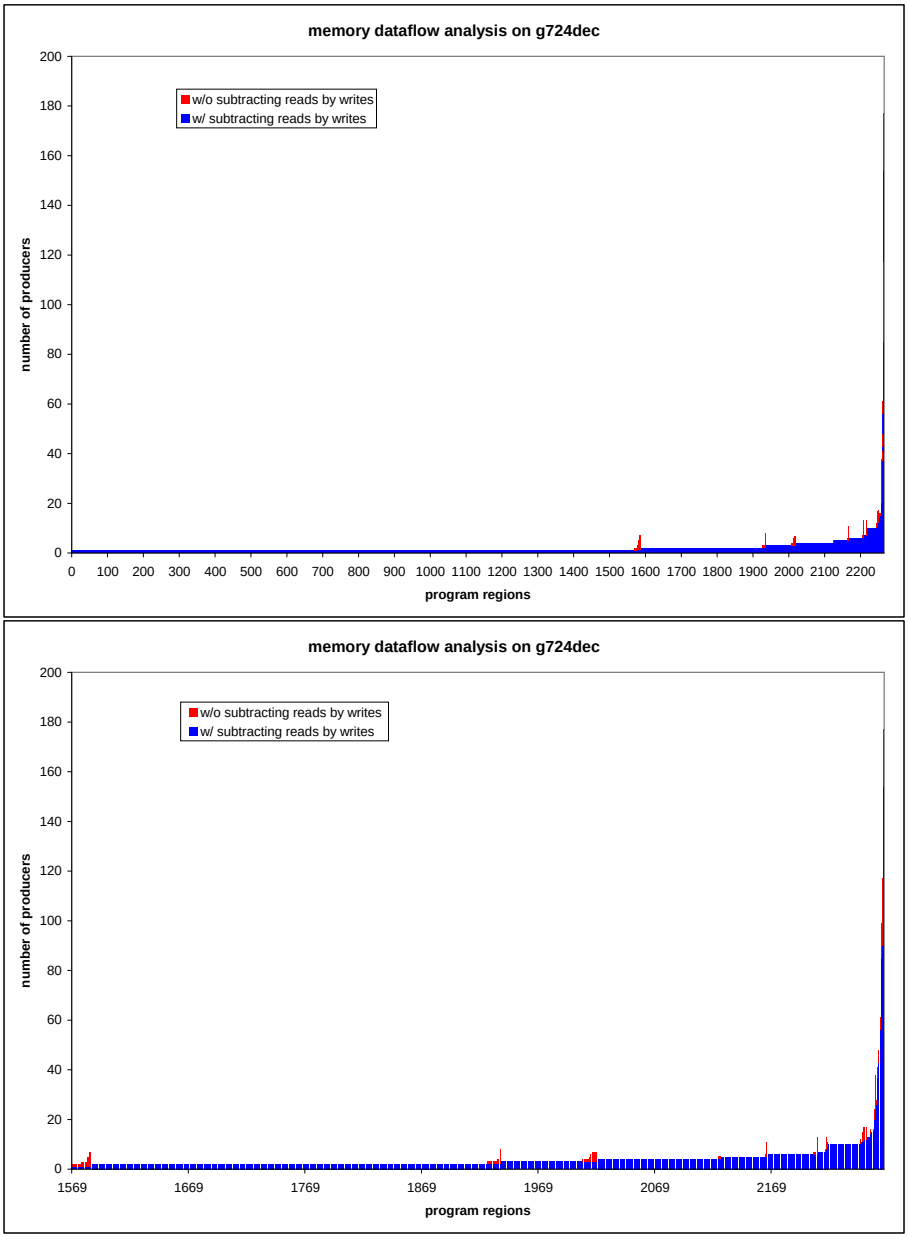


Figure 8.5 Eliminated spurious data producers (false dependences) in g724dec

producer of `region3`, even though `region1` also writes to `A[i]`. On the other hand, if we do not subtract the exposed read of `region3` by the exposed write of `region2`, and

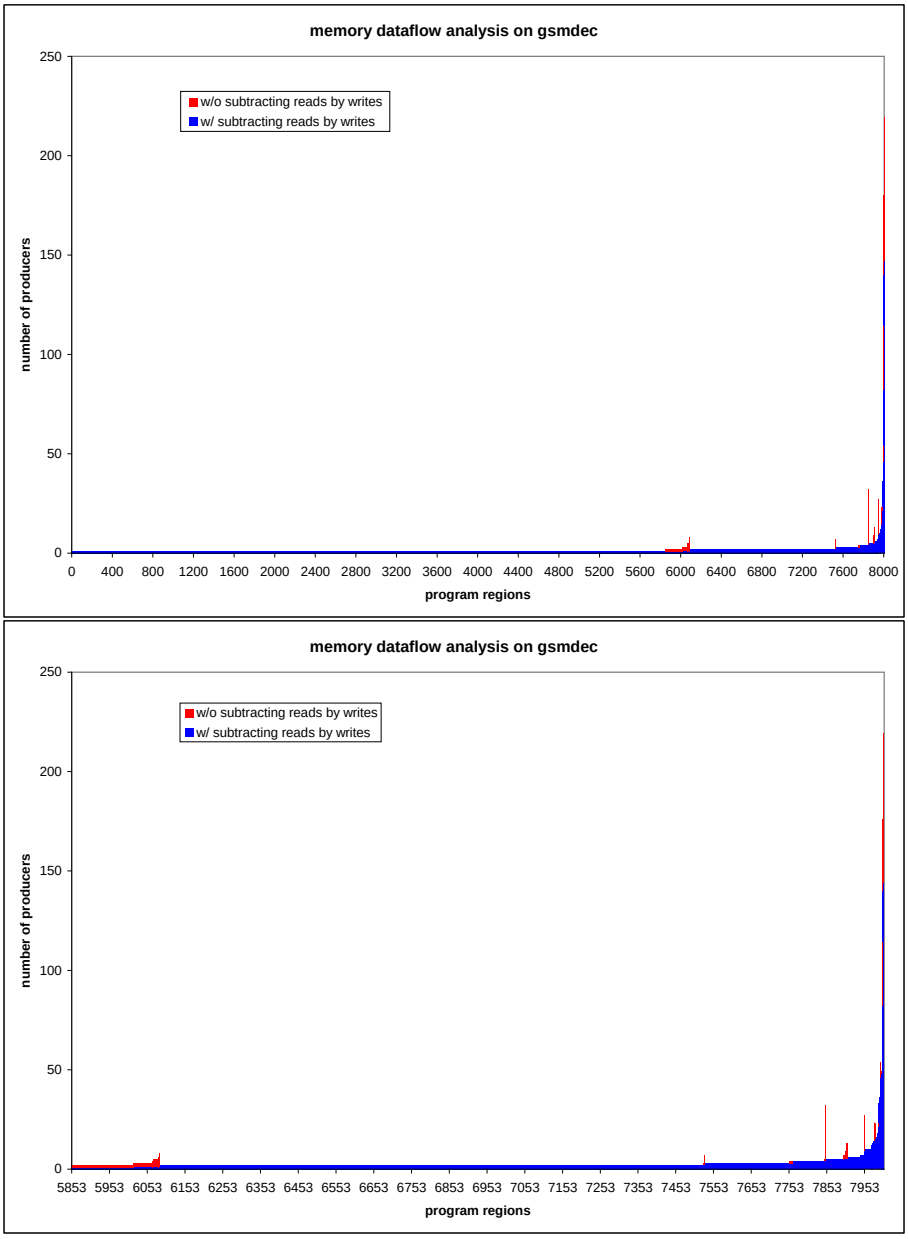


Figure 8.6 Eliminated spurious data producers (false dependences) in gsmdec

keep propagating it to `region1`, `region3` will have another data producer, `region1`, as illustrated in Figure 8.2(b).

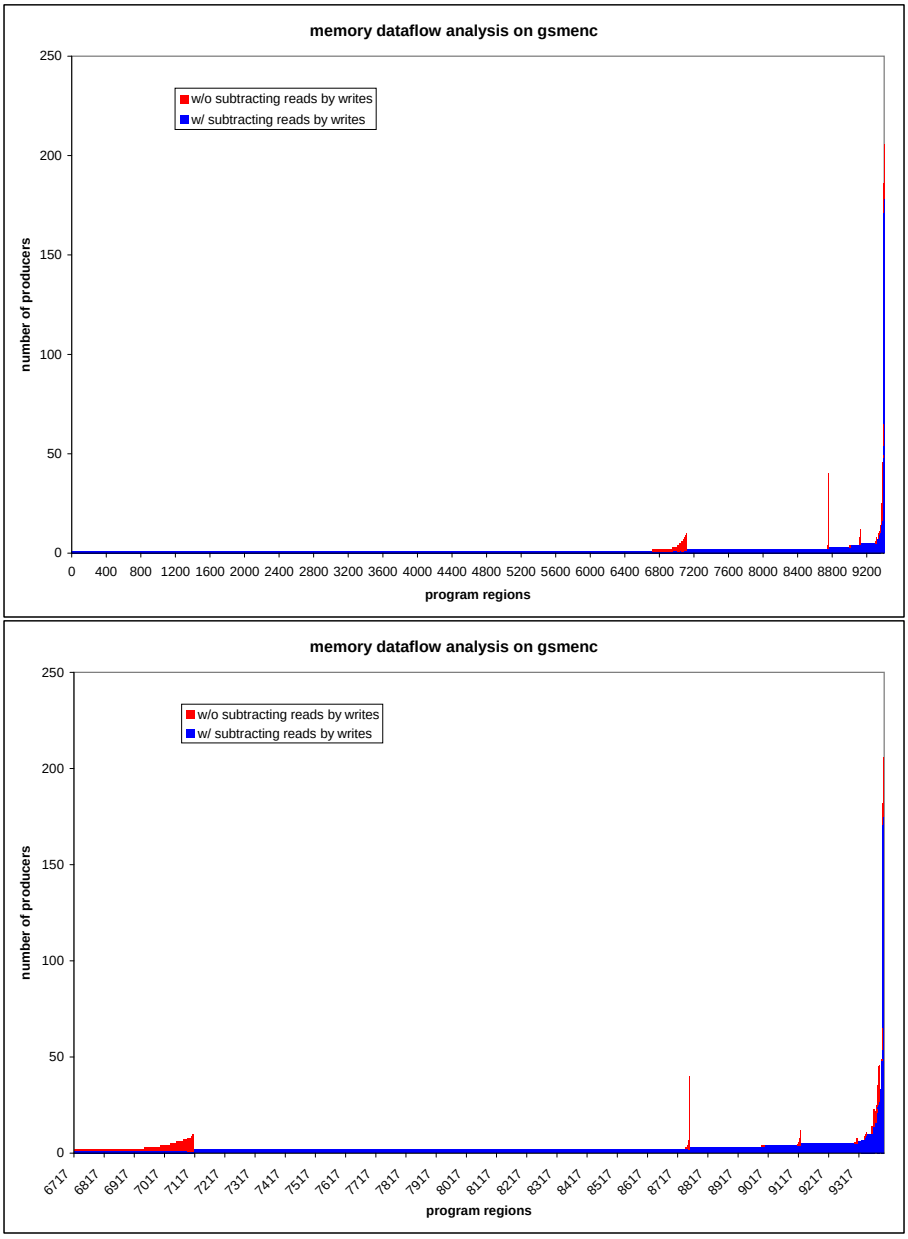


Figure 8.7 Eliminated spurious data producers (false dependences) in gsmenc

So, without subtracting the exposed reads of the sub-regions by the exposed writes of other sub-regions during the bottom-up summarization phase, we can identify the false dependences or spurious data producers eliminated by the prototype memory data-flow

analysis system. Figures 8.3 to 8.7 show the spurious data producers eliminated by the prototype memory data-flow analysis system for some of the benchmark programs. The X-axis corresponds to all the program regions. The Y-axis is the number of data producers for each program region. For each program region, the blue dots are corresponding to the number of data producers identified by the memory data-flow analysis system *with* exposed reads subtracted by exposed writes. The red dots are corresponding to the number of its data producers identified *without* having exposed reads subtracted by exposed writes. If a red dot is above the blue dot of the same program region, it means the memory data-flow analysis eliminates some spurious data producers, or false data dependences. Note that, for the same program region, the blue dot is never above the red dot.

As demonstrated in Figures 8.3 to 8.7, there are indeed false dependences existing in real programs, due to writing to the same variables, which are eliminated by the prototype memory data-flow analysis systems. However, it is hard to tell whether the prototype memory data-flow analysis system eliminates *all* the false memory dependences. It is even harder to tell what benefit the client of the memory data-flow analysis will get by eliminating the false dependences. The ultimate test of the effectiveness of the memory data-flow analysis system is how the extracted coarse-grained data flow can enable better mapping of applications onto multi-core architectures. However, an end-to-end mapping from C programs to multi-core architectures is not available in our compiler infrastructure at this moment.

Table 8.2 Breakdown of the type of MADs for exposed reads

	adpcm D	adpcm E	g721 D	g721 E	g724 D	g724 E	gsm D	gsm E
Seq	10	12	111	90	625	1644	517	653
Must	0	0	6	6	33	61	24	42
May	5	5	172	112	141	233	305	698
Doomed	33	31	86	95	336	948	1544	1582

Table 8.3 Breakdown of the type of MADs for exposed writes

	adpcm D	adpcm E	g721 D	g721 E	g724 D	g724 E	gsm D	gsm E
Seq	3	1	16	20	420	807	206	390
Must	0	0	0	0	19	45	0	12
May	6	6	0	2	141	307	434	192
Doomed	7	9	10	17	167	447	39	100

In addition to counting the number of eliminated false dependences, we can also assess the effectiveness of the prototype memory data-flow analysis system by counting the types of the MADs for exposed reads and exposed writes. If most of the exposed reads and exposed writes are *Doom*-typed or *May*-typed, the prototype memory data-flow analysis system may not be effective in summarizing the memory access patterns of program regions for the tested programs. On the other hand, if many of the exposed reads and exposed writes are *Seq*-typed or *Must*-typed, the prototype memory data-flow analysis system can be considered effective in capturing accurate memory access patterns for test programs.

Tables 8.2 and 8.3 show the breakdown of the types of the exposed reads and the exposed writes of all program regions. Not to exaggerate the effectiveness of the prototype memory data-flow analysis system, Table 8.2 and Table 8.3 exclude the exposed scalar

Table 8.4 Breakdown of the percentages of the causes of *May-type* MAD

Cause	adpcm D	adpcm E	g721 D	g721 E	g724 D	g724 E	gsm D	gsm E
I	92.86	90.32	80.77	80.17	78.18	80.65	96.7	92.01
II	0	3.23	0	1.24	7.5	5.31	0.06	0.27
III	0	0	0	0	2.95	2.63	0.06	1.56
IV	3.57	3.23	13.64	11.16	0.94	0.68	0.92	0.89
V	0	0	4.2	4.96	2.14	5.48	1.43	4.2
VI	0	0	1.05	1.24	1.07	1.2	0.12	0.16
VII	3.57	3.23	0.35	1.24	6.16	3.08	0.71	0.86
VIII	0	0	0	0	1.07	0.97	0	0.05

variable reads and writes, and the exposed reads and exposed writes of fundamental memory access regions, which are always *Seq*-typed.

As shown in Tables 8.2 and 8.3, the prototype memory data-flow analysis system can capture the sequential memory access patterns of many exposed reads and exposed writes using the simple MAD structure. An more important implication of this is there are indeed many sequential memory accesses in the tested programs. If our memory access descriptor can only describe the set of accessed memory locations, but not the access order, we may miss many opportunities for the optimization shown in Figure 3.6. An interesting observation is that Table 8.3 has higher percentage of *Seq*-type memory accesses than Table 8.2. This means memory writes have more regular access patterns than memory reads.

However, Table 8.2 and Table 8.3 also show that there are many *May*-type and *Doomed*-type exposed reads and exposed writes. These *May*-type and *Doomed*-type MADs will result in less accurate producer and consumer relation. Therefore, the first step in improving the accuracy of the prototype memory analysis system is to find out

why *May*-type and *Doomed*-type MADs are generated in the first place. We insert probes in the prototype memory data-flow analysis system to profile the causes of *May*-type and *Doomed*-type MADs.

Table 8.4 shows the breakdown of percentages of the 8 causes of *May-type* MAD, where Cause I is that the corresponding memory access of the MAD is in a conditional statement, and thus may or may not occur; Cause II is that a MAD is subtracted by a *Doom*-type MAD; Cause III is that when subtracting a MAD by another MAD, we can not determine the relation between the *base* of these two MADs; Cause IV is that when summarizing the exposed memory accesses of a loop, we can only know an upper bound of the loop trip count, because of early exit of the loop; Cause V is the inaccuracies of the **Concatenate** operation; Cause VI is the inaccuracies of the **Merge** operation; Cause VII is the inaccuracies of the **Subtract** operation, other than cause II and cause III; Cause VIII is the inaccuracies of the **Summation** operation, other than cause IV.

Apparently, cause I is the most common reason why a *May*-type MAD is generated. This is due to the characteristics of the applications, and we can not replace *May*-type MADs of this cause with more accurate MADs to improve the accuracy of the prototype memory data-flow analysis system.

Cause IV is the second common cause, which is also due to application characteristics. Therefore, we cannot replace *May*-type MADs of this cause with more accurate MADs.

What is surprised is that only a small fraction of *May*-type MADs are due to the inaccuracies of MAD operations (causes V, VI, VII, and VIII), except for g724dec and g724enc. For these two benchmarks, some fraction of the *May*-type MADs are also due

Table 8.5 Breakdown of the percentages of the causes of *Doomed-type* MAD

Cause	adpcm D	adpcm E	g721 D	g721 E	g724 D	g724 E	gsm D	gsm E
I	33.33	33.33	50	45.45	3.23	0.94	2.11	1.99
II	0	0	0	0	33.87	22.54	2.11	1.99
III	0	0	0	0	1.61	0	0	0.66
IV	33.33	33.33	45	40.91	27.42	27.7	93.66	88.74
V	0	0	0	0	0	3.29	0	0
VI	33.33	33.33	5	4.55	9.68	16.9	0	1.32
VII	0	0	0	0	11.29	11.74	2.11	4.64
VIII	0	0	0	0	0	0.47	0	0.66
IX	0	0	0	0	0	0	0	0
X	0	0	0	9.09	12.9	16.43	0	0

to cause II. This means we can potentially replace some *May*-type MADs with more accurate MADs if we can replace some Doomed-type MADs.

Table 8.5 shows the breakdown of the percentages of the 10 causes of *Doomed*-type MAD, where Cause I is that, when performing some operation on two MADs, we found they have different access sizes in bytes; Cause II is that we cannot resolve the relation between two scalar variables, using the current implementation of symbolic scalar variable evaluation, when performing operations, other than **Subtract**, on two MADs; Cause III is that, when summarizing an exposed memory access for some loop, we found an induction variable of the loop cannot be represented in close form, using the current implementation of symbolic scalar variable evaluation; Cause IV is that, when summarizing an exposed memory access for some loop, we do not know the loop trip count, not even an upper bound; Cause V is that, when summarizing an exposed memory access for some loop, we found an induction variable of the loop has variable stride; Cause VI is that, when summarizing an exposed memory access for some region, we found the description of

the exposed memory access is not invariant with respect to that region; Cause VII is other inaccuracies of the **Concatenate** operation; Cause VIII is other inaccuracies of the **Merge** operation; Cause IX is other inaccuracies of the **Subtract** operation; Cause X is other inaccuracies of the **Summation** operation.

Unlike *May*-type MAD, there is no single dominating cause of *Doom*-type MAD. Across all programs, a high percentage of *Doomed*-type MADs are due to cause IV. For the current implementation of the symbolic evaluation, if the exit condition of a loop cannot be represented as an affine induction expression, the loop will have unknown trip count, not even an upper bound. So, we can potentially replace some of the *Doom*-type MADs of this cause with more accurate MADs by improving the symbolic scalar variable evaluation. However, it is a difficult problem to deduce the trip count for arbitrary loops. To some extent, this should also be considered as due to application characteristics.

For g724dec and g724enc, a high percentage of *Doomed*-type MADs are due to cause II. This means there is definitely room in improving the symbolic scalar variable evaluation.

For adpcmdec and adpcmenc, a significant percentage of *Doomed-type* MADs are due to cause V. Usually this means the program region is doing some table lookup using some dynamically generated index, which cannot be figured out at compile time. To some extent, this should also be considered as due to the application.

For adpcmdec/adpcmenc and g721dec/g721enc, a significant fraction of *Doomed*-type MADs are due to cause I. This is a known limitation of the current implementation, and

Doomed-type MADs of this type will be replaced by more accurate MADs after we improve the MAD operations.

Table 8.5 shows that the other inaccuracies of **Concatenate** and **Summation** cause a fair amount of *Doomed*-type MADs for g724dec and g724enc. Therefore, we can potentially improve the effectiveness of the memory data-flow analysis system by enhancing these two operations.

From the experiment results, we have identified some inefficiencies in the prototype memory data-flow analysis system. However, some inaccuracies of the analysis results are due to application characteristics. The next chapter will conclude this dissertation with the insights obtained from the experiment of prototyping the memory data-flow analysis system.

CHAPTER 9

Conclusion and Future Work

In the last chapter of this dissertation, I would like to reflect on my work on memory data-flow analysis, and discuss my thoughts on some future works.

9.1 Conclusion

To efficiently utilize the emerging heterogeneous multi-core architecture, it is essential to exploit the inherent coarse-grained parallelism in applications. In addition to data parallelism, applications like telecommunication, multimedia, and gaming can also benefit from the exploitation of coarse-grained function parallelism. To exploit coarse-grained function parallelism, the common wisdom is to rely on programmers to explicitly express the coarse-grained data-flow between coarse-grained functions using data-flow or streaming languages.

This work is set to explore another approach to exploiting coarse-grained function parallelism, that is to rely on compilers to extract coarse-grained data-flow from imperative programs. I believe imperative languages and the von Neumann programming model will still be the dominating programming model in the future. For this exploration, this research accomplishes the following.

- It developed a memory data-flow analysis framework to extract coarse-grained data-flow from C programs, or imperative programs in general. First, the memory data-flow analysis system partitions a C program into a hierarchy of program regions. It then traverses the program region hierarchy from bottom up, summarizing the exposed memory accesses for each program region. During this bottom-up summarization process, it also constructs a conservative producer-consumer relation between the program regions. After the bottom-up process, a top-down traversal of the program region hierarchy refines the producer-consumer relation by eliminating exposed memory writes which have no consumers.

- It built a prototype of the memory data-flow analysis system. The efficiency and effectiveness of the prototype are studied using real C programs from the the MediaBench suite and open-source G.724 coder and decoder. It also built a visualization system to displace the memory data-flow analysis results. In addition to the original purpose of verification, the memory data-flow visualization system can potentially be enhanced for other software engineering purposes.

- Experiment results show that the prototype memory data-flow system performs reasonably well for the tested C programs. However, the in-lining based prototype memory data-flow analysis system may not be efficient for larger programs. Also, we can still improve the prototype to obtain more accurate memory data-flow analysis results. Root cause analysis of the memory data-flow analysis inaccuracies shows that the memory data-flow analysis can potentially be more accurate by

improving the symbolic scalar variable evaluation, the memory access descriptor and the associated operations used by the memory data-flow analysis. However, some of the inaccuracies are due to the application characteristics, and cannot be eliminated by improving the memory data-flow analysis.

This study shows that it is possible to build a program analysis system to extract coarse-grained data-flow from C programs. However, we found it is difficult to extract accurate coarse-grained data-flow from "spaghetti" code or programs with complicated control flow and extensive accesses of dynamically allocated memory objects. Programmers can improve the effectiveness of the memory data-flow analysis by writing more structured code, grouping related code into functions, and using statically allocated variables as much as possible.

In my opinion, reasoning about complicated control flow and dynamically allocated memory objects will remain the two main challenges of memory data-flow analysis. On the other hand, it is also not clear how successful the programming model of data-flow or streaming languages will be in handling complicated control flow and dynamically allocated memory objects. Unfortunately, as the applications become more and more complicated, it is very unlikely that we can avoid complicated control flow and dynamically allocated memory objects will be .

I believe, most likely, we can partition any application into a data-flow part and a von Neumann part which is either impossible or inefficient to fit into the data-flow model. The data-flow part will be implemented in ASICs, accelerators, or other unconventional architectures, while the von Neumann part will still be executed in von Neumann archi-

tectures. Hopefully, the "80-20" rule will put most of the computation in the data-flow part for efficient execution. Indeed, this is how people design their systems today, but in an ad hoc way. The question is "Can we do this partition systematically and automatically ?".

We can re-phrase this question as "Is it necessary to extend imperative languages with data-flow or streaming language constructs?". Of course, to reply "no", we need a compiler to demonstrate the following.

- For any imperative program that the compiler cannot sort out its data-flow, it is also difficult, if not impossible, to re-write the program in data-flow or streaming language constructs.

- For any imperative program that can be re-written in data-flow or streaming language constructs, the compiler can also extract its data-flow.

For extracting scalar data-flow from imperative programs, researchers have already developed the needed compiler techniques. For extracting coarse-grained data-flow from imperative programs, this work has made an attempt. Although I cannot say I have solved this problem in this work, I think it is an interesting problem for intellectual challenge, and an important problem for practical purposes, worthy of further investigation. Next, I will sketch some future works.

```
int A[10];
foo()
{
   p1 = A ;
   q1 = A + 1;
   bar(p1, q1); /* callsite 1 */
   p2 = A + 2;
   q2 = A + 4;
   bar(p2, q2); /* callsite 2 */
}
```

```
bar (int *x, int *y)
{
   *x = ...;
   ... = ... *y ...;
}
```

Figure 9.1 Example of function with the same summary at two call-sites

9.2 Future Work

There is always more works to be done than has been done. This section will outline some future works on improving the efficiency and effectiveness, and on the evaluation, of our memory data-flow analysis system.

9.2.1 Inter-procedural Memory Data-flow Analysis

For large applications, we need to develop an inter-procedural memory data-flow analysis. By avoiding re-analyzing the same function at different call-sites, inter-procedural memory data-flow analysis can be more efficient than in-lining based approach. The question is how to determine whether we should re-analyze a function or not.

Figure 9.1 shows the example code segment, where the function `bar` is called twice by the function `foo`. To summarize the exposed memory accesses of `bar`, we need to know the relation between its pointers `x` and `y`. If `x` has the same value as `y`, the memory read `*y` will be covered by the memory write `*x`, and thus `bar` will have no exposed read. If the value of `x` is different from the value of `y`, then `bar` will have exposed read `*y`.

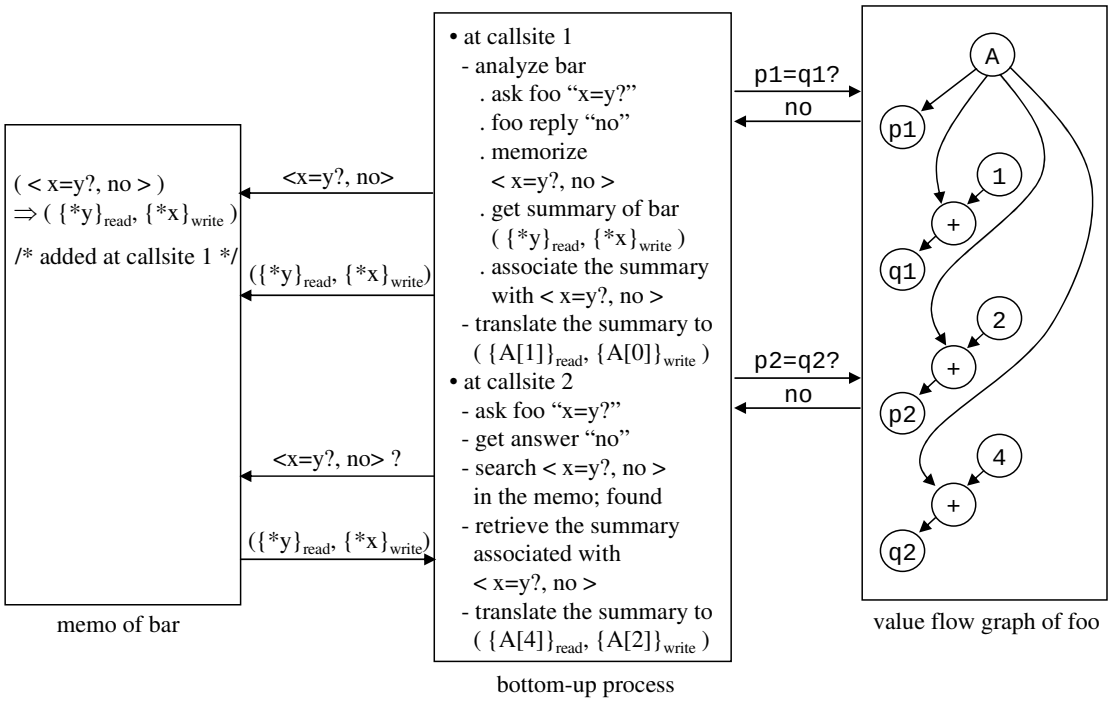


Figure 9.2 Illustration of function calls with isomorphic memory data-flow analysis results

The relation of `x` and `y` may be different at different call-sites of `bar`. If `bar` has the same relation between `x` and `y` at two call-sites, `bar` will have the same summary of exposed reads and exposed writes at these two call-sites. The The exposed reads (exposed writes) of these In other words, if we can know a function will have the same summary of exposed reads and exposed writes at two call-sites, we only need to analyze the function once. The exposed reads (and writes)

Figure 9.2 illustrates memoization based approach to determine whether a function will have the same summary of exposed reads and exposed writes at different call-sites. When the memory data-flow analysis reaches call-site 1, because this is the first call-site of `bar`, we go analyze the exposed reads and exposed writes of `bar`. During the analysis

```
int A[10];
foo2()
{
   p1 = A ;
   q1 = A + 1;
   bar(p1, q1); /* callsite 1 */
   p2 = A + 2;
   q2 = A + 2;
   bar(p2, q2); /* callsite 2 */
}
```

```
bar (int *x, int *y)
{
   *x = ...;
   ... = ... *y ...;
}
```

Figure 9.3 Example of function with different summaries at two call-sites

of `bar`, we need to know whether `x` has the same value as `y`. This depends on the calling context and cannot be resolved by only looking at the code `bar`. So, we query the value flow graph of `foo`, the caller of `bar`, "`x = y ?`". After translating the formal parameters, `x` and `y`, to the corresponding actual arguments, `p1` and `q1`, we can infer from the value flow graph of `foo` that `p1` $\neq$ `q1`, and the answer to the query is "no". The tuple of query and answer, $\langle$`x = y ?`, no$\rangle$, is then recorded in a memo for `bar`.

After resolving the relation between `x` and `y`, we continue the analysis of the exposed reads and exposed writes of `bar`, and eventually obtain the summary of exposed reads and exposed writes of `bar` at call-site 1, (`*y`, `*x`). In the memo for `bar`, we then associate this summary of expose reads and exposed writes with the corresponding list of query-answer tuples, shown as (`*y`, `*y`) $\rightarrow$ ($\langle$`x = y ?`, no$\rangle$) in Figure 9.2. After substituting `p1` with `A`, and `q1` with `A` + 1, the exposed memory reads and memory writes of the function call to `bar` at call-site 1 are `A[0]` and `A[1]` respectively.

When the memory data-flow analysis reaches call-site 2, if we re-analyze `bar` again, we will again ask the same question "`x = y ?`". Instead of blindly re-analyzing `bar`, we first evaluate the query "`x = y ?`" at call-site 2. After translating `x` and `y` to `p2` and `q2`,

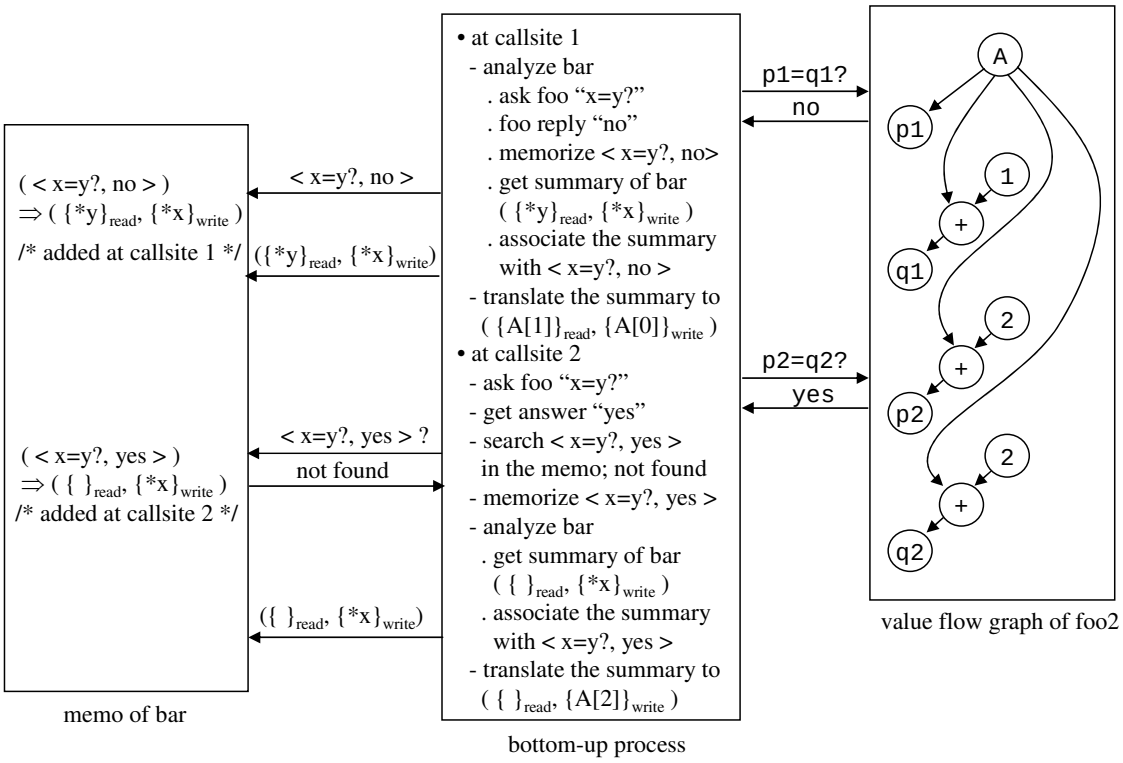


Figure 9.4 Illustration of function calls without isomorphic memory data-flow analysis results

we can infer from the value flow graph of `foo` that `p2` $\neq$ `q2`, and the answer to query is "no". Then, in the memo for `bar`, we search the query-answer tuple $\langle$`x = y ?`, no$\rangle$ generated at call-site 2, and will find that it has already been associated with a summary of exposed reads and exposed writes. This means that we have analyzed `bar` at other call-sites, call-site 1 in this case, and `bar` will have the same summary of exposed reads and exposed writes at call-site 1 and call-site 2. Therefore, without re-analyzing `bar`, we can obtain the pair of exposed reads and exposed writes of `bar` at call-site 2 by retrieving the associated (`*y`, `*y`) from the memo for `bar`.

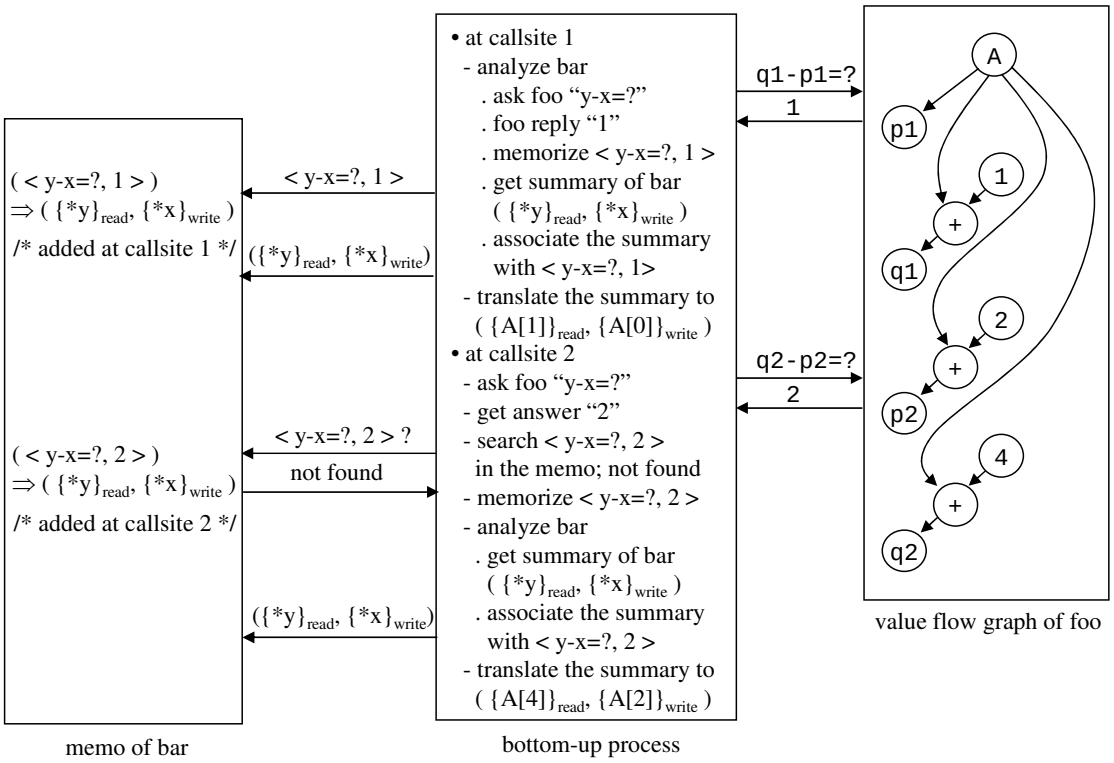


Figure 9.5 Illustration of inefficient queries to value flow graph

After substituting `x` with `p2` (= `A` + 2), and `y` with `q2` (= `A` + 4), we can obtained the exposed reads and exposed writes of the function call to `bar` at call-site 2, `A[2]` and `A[4]`.

Figure 9.3 shows an example that a function has different summaries of exposed reads and exposed writes at two call-sites. Figure 9.3 is different from Figure 9.1 only in the value of `q2`. In Figure 9.3 `q2` is equal to `A` + 2, while in Figure 9.3 `q2` is equal to `A` + 4. When we reach the call-site 2 in Figure 9.3, we will ask the query "`x = y ?`". After translating `x` to `p2`, and `y` to `q2`, we can infer from the value flow graph in Figure 9.3 that `p2` = `q2` = `A` + 2. Therefore, the answer to the query "`x = y ?`" is "yes" at call-site 2 in Figure 9.3.

In Figure 9.3, we cannot find the query-answer tuple $\langle$`x = y ?`, yes$\rangle$ in the memo for `bar`. This means that the summary of exposed reads and exposed writes of `bar` at call-site 2 *may*[1] be different from the summaries at previous call-sites. Therefore, we must re-analyze `bar` at call-site 2, and eventually find the summary of exposed reads and exposed writes to be $(\{\}, *x)$[2], which is indeed different from $(*y, *x)$, the summary of exposed reads and exposed writes of `foo` at call-site 1.

The efficiency of this memoization based inter-procedural memory data-flow analysis will be affected by the queries we ask. If we do not design the queries carefully, we may have the situation that a function has different list of query-answer tuples at two call-sites, even though the function has the same summary of exposed reads and exposed writes at these two call-sites. For example, if the queries we ask in Figure 9.2 were "`y - x = ?`", instead of "`x = y ?`", we will have the situation shown in Figure 9.5. Note that `bar` still has the same summary of exposed reads and exposed writes at call-site 1 and call-site 2 in Figure 9.5. However, the answer to the query "`y - x = ?`" at call-site 1 is "1", while the answer to the same query at call-site 2 is "2". This will mislead us to assume `bar` has different summary of exposed reads and exposed writes at call-site 1 and call-site2, and result in re-analyzing `bar` at call-site 2.

Like the in-lining based approach, the effectiveness of this inter-procedural memory data-flow analysis is also affected by the accuracy of the symbolic evaluation of queries. If the queries generated when analyzing a function can always be resolved at the value

[1]Next paragraph will explain why it is "*may*", instead of "*must*". It depends on the query.

[2]Note that, if `x` and `y` in `bar` are equal, `*y` will be covered by `*x`. Therefore, the exposed reads of `bar` will be empty.

flow graphs of its ancestor functions, inter-procedural symbolic query evaluation is not difficult. Inter-procedural symbolic query evaluation will become difficult if the resolution of the queries generated for analyzing a function cannot be done at its ancestors, but also need information from its child, sibling, or any other functions.

For this kind of queries, a quick and dirty solution is just to say "I don't know", and have a conservative summary for the querying function. Although this may affect the effectiveness of the memory data-flow analysis, it could work very efficiently, and reasonably well if this kind of queries are rare. Just like other program analysis problems, we often need to make a trade-off between efficiency and effectiveness.

9.2.2 Improving Versatility and Effectiveness

We can improve the versatility and effectiveness of the memory data-flow analysis in the following fronts.

- The memory data-flow analysis would be more versatile, if we can eliminate the limitations discussed in Section 5.2. Among these limitations, indirect function calls, recursive functions, and functions with variable number of arguments are common in ordinary programs, and should be considered along with the design of inter-procedural memory data-flow analysis.

- For the current prototype, we partition a C program into functions and other predefined program regions. We could try more sophisticated approaches to partitioning a C program. For example, we can try some iterative partitioning method, which

starts from some fixed partition, and then iteratively refines the partitioning to minimize the communication between program regions.

- We can potentially improve the accuracy of the memory data-flow analysis by improving the symbolic evaluation of scalar variables, as shown in Section 8.4, For example, we can implement full-fledged gated SSA for more accurate evaluation of scalar variables by taking predicates into consideration. Another direction is to perform symbolic evaluation beyond scalar variables. Programmers also use array elements or structure fields to index another array. Without knowing the relation between the values stored in arbitrary memory locations, we cannot have accurate memory data-flow analysis results for general applications.

- We can also potentially improve the operations used in memory data-flow analysis. For example, in Section 6.2.4.4, the current implementation will down grade an exposed read of a loop to a less accurate memory access descriptor, if the exposed read has inter-iteration dependence. Potentially, for some special cases, we can use the dependence distance information to refine the exposed read of the loop, by excluding those memory accesses which are generated inside the loop.

9.2.3 Evaluation

To evaluate the effectiveness, and to show the real benefit, of memory data-flow analysis, we need to connect the memory data-flow analysis to the back-end of the tool

chain in order to form a complete compilation path from C program to hardware, as illustrated in Figure 3.1.

One possibility is to connect the memory data-flow analysis to a high-level synthesis tool. Given the memory data-flow analysis result, we can select a set of program regions for synthesis, based on some cost mode. We can then do source-to-source translation of these program regions using the native language of the high-level synthesis tool. For example, we can translate the selected program regions into concurrent tasks, and specify the communication between these tasks based on the producer-consumer relation between the corresponding program regions.

For each program region, we also need to specify an inter-process communication interface for each of its exposed memory accesses. For a *Seq*-type exposed memory access, we can specify a FIFO interface for streaming data access. For a *Must*-type or *May*-type exposed memory access, we can allocate a memory buffer, or even use double buffering, to store the accessed data. For a *Doomed*-type memory access, we need to allocate enough memory to hold all the possibly accessed memory objects. This may be inefficient, which should be reflected in the cost model. If a task accesses the system memory, we can specify an address generator, or instantiate a DMA, which uses the *base* and *offset* of the corresponding memory access descriptor to determine the starting address, and the *displace* to determine the access stride and access count.

This ends the documentation of my works and my thoughts on extracting coarse-grained data-flow from C programs for the exploitation of coarse-grained function parallelism. Looking back, it is really fascinating to me that researchers have made so much

effort and so many innovations to map applications onto parallel architectures. We have come a long way. Looking forward, I believe there is still a long way to go, but, no matter which road we will take, I believe the journey will be interesting and we will eventually reach our destination.

REFERENCES

[1] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, Apr. 1965.

[2] "International Technology Roadmap for Semiconductors." http://public.itrs.net.

[3] R. Wilson and B. Fuller, "Soaring mask costs roil fine-geometry asics," *EE Times*, 03/26/1999.

[4] D. Lammers, "Shift to 65 nm has its costs," *EE Times*, 07/11/2005.

[5] D. Lammers, "Guide to success: fear and loathing at next node," *EE Times*, 11/14/2005.

[6] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, pp. 490–504, Apr. 2001.

[7] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *Proceedings of Annual International Symposium on Computer Architecture*, 2000.

[8] P.E.Gronowski, W.J.Bowhill, R. Preston, M. K. Gowan, and R. L. Allmon, "High-performance microprocessor design," *IEEE Journal of Solid-state Circuits*, vol. 33, pp. 676–686, May 1998.

[9] T. Mudge, "Power: A first-class architectural design constraint," *IEEE Computer*, Apr. 2001.

[10] N. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. Hu, M.J.Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *IEEE Computer*, pp. 68–75, Dec. 2003.

[11] M. Horowitz and W. Dally, "How scaling will change processor architecture," in *International Solid-State Circuits Conference*, 2004.

[12] C. Maxfield, "Asics find new 'structure'," *EE Times*, 09/03/2003.

[13] F. McMillan, "Best practices for structured-asic design," *EE Times*, 10/17/2005.

[14] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, Mar. 2002.

[15] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *IEEE Computer*, Aug. 2003.

[16] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with streams," in *Proceedings of International Conference on Supercomputing*, 2003.

[17] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proceedings of Annual International Symposium on Computer Architecture*, 2003.

[18] D. Burger and J. R. Goodman, "Billion-transistor architectures: There and back again," *IEEE Computer*, Mar. 2004.

[19] E. F. Moltzen, "Barrett: Intel to migrate to dual-core platforms by year-end," *EE Times*, 03/01/2005.

[20] R. Merritt, "Path holes seen in latest Intel road map," *EE Times*, 03/07/2005.

[21] D. Dunn, "AMD debutes dual-core Opteron," *EE Times*, 04/21/2005.

[22] H. P. Hofstee, "Power efficient processor architecture and the Cell processor," in *Proceedings of International Symposium on High-Performance Computer Architecture*, 2005.

[23] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano, "The microarchitecture of the streaming processor for a cell processor," in *International Solid-State Circuits Conference*, 2005.

[24] ETSI TC-SMG, "Digital cellular communications system; Enhanced Full Rate (EFR) speech transcoding (GSM 06.60)," European Telecommunications Standards Institute, Tech. Rep. ETS 300 726, Mar. 1997.

[25] A. Peleg, S. Wilkie, and U. Weiser, "Intel mmx for multimedia pcs," *Communications of the ACM*, no. 1, pp. 24–38, 1997.

[26] A. Bik, *The Software Vectorization Handbook - applying multimedia extensions for maximum performance*. Intel Press, 2004.

[27] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of AFIPS Spring Joint Computer Conference*, 1967.

[28] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[29] K. Kennedy and R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[30] W.-M. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhall, and D. I. August, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1640, Dec. 1995.

[31] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[32] D. J. Kuck, Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in fortran-like programs and their resulting speedup," *IEEE Transactions on Computers*, pp. 1293–1310, Dec. 1972.

[33] U. Banerjee, "Data dependence in ordinary programs," M.S. thesis, University of Illinois at Urbana-Champaign, 1976.

[34] U. Banerjee, "Speedup of ordinary programs," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1979.

[35] D. J. Kuck, *The Structure of Computers and Computations, Volume 1*. Wiley and Sons, 1978.

[36] J. R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Transactions on Programming Languages and Systems*, pp. 491–542, Oct. 1987.

[37] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *International Computer Software and Applications Conference*, 1980.

[38] L. Lamport, "The parallel execution of do loops," *Communications of the ACM*, pp. 83–93, Feb. 1974.

[39] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1981.

[40] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, pp. 1184–1201, Dec. 1986.

[41] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1984.

[42] M. J. Wolfe, "Advanced loop interchanging," in *Proceedings of International Conference on Parallel Processing*, 1986.

[43] M. J. Wolfe, "Loop skewing: The wavefront method revisited," *International Journal of Parallel Programming*, pp. 279–293, Aug. 1986.

[44] D. A. Padua, "Multiprocessors: Discussion of some theoretical and practical problems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1979.

[45] R. Allen and K. Kennedy, "Vector register allocation," *IEEE Transactions on Computers*, vol. 41, Oct. 1992.

[46] J. R. Allen, D. Callahan, and K. Kennedy, "Automatic decomposition of scientific programs for parallel execution," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1987.

[47] C. D. Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design," *IEEE Transactions on Computers*, pp. 991–1004, Aug. 1988.

[48] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferreant, "An overview for the ptran analysis system for multiprocessing," *Journal of Parallel and Distributed Computing*, pp. 617–640, Oct. 1988.

[49] A. Rogers and K. Pingali, "Compiling for distributed memory architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, Mar. 1994.

[50] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The Paradigm compiler for distributed-memory multicomputers," *IEEE Computer*, vol. 28, Oct. 1995.

[51] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel programming with polaris," *IEEE Computer*, vol. 29, Dec. 1996.

[52] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and E. Bu, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, Dec. 1996.

[53] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath, "Collective loop fusion for array contraction," in *Workshop on Languages and Compilers for Parallel Computing*, 1992.

[54] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[55] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, Oct. 1991.

[56] T. Gross and P. Steenkiste, "Structured dataflow analysis for arrays and its use in an optimizing compiler," *Software-Practice and Experience*, vol. 20, pp. 133–155, Feb. 1990.

[57] P. Feautrier, "Dataflow analysis of scalar and array references," *International Journal of Parallel Programming*, vol. 20, pp. 23–53, Feb. 1991.

[58] J. Gu, Z. Li, and G. Lee, "Symbolic array dataflow analysis for array privatization and program parallelization," in *Proceedings of International Conference on Supercomputing*, 1995.

[59] J. Gu, Z. Li, and G. Lee, "Experience with efficient array data flow analysis for array privatization," in *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1997.

[60] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array data-flow analysis and its use in array privatization," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1993.

[61] Z. Li, "Array privatization for parallel execution of loops," in *Proceedings of International Conference on Supercomputing*, 1992.

[62] P. Tu and D. Padua, "Automatic array privatization," *Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science 768, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (Eds.)*, 1994.

[63] D. J. Palermo, E. Su, E. W. H. IV, and P. Banerjee, "Compiler support for privatization on distributed-memory machines," in *Proceedings of International Conference on Parallel Processing*, 1995.

[64] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1986.

[65] Z. Li and P.-C. Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, 1988.

[66] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the pips project," in *Proceedings of International Conference on Supercomputing*, 1991.

[67] M. W. Hall, K. Kennedy, and K. S. McKinley, "Interprocedural transformations for parallel code generation," in *Proceedings of International Conference on Supercomputing*, 1991.

[68] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Interprocedural compilation of fortran d for mimd distributed-memory machines," in *Proceedings of International Conference on Supercomputing*, 1992.

[69] A. Carle, M. Hall, J. Mellor-Crummey, and R. Rodriguez, "Fiat: A framework for interprocedural analysis and transformation," Rice University, tech. rep., march 1995. CRPC-TR95522-S.

[70] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Proceedings of International Conference on Supercomputing*, 1995.

[71] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," in *Workshop on Languages and Compilers for Parallel Computing*, 1995.

[72] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Exploiting task and data parallelism on a multicomputer," in *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1993.

[73] T. Gross, D. R. O'Hallaron, and J. Subhlok, "Task parallelism in a high performance fortran framework," *IEEE Parallel & Distributed Technology*, vol. 2, no. 3, 1994.

[74] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A convex programming approach for exploiting data and functional parallelism on distributed memory multcomputers," in *Proceedings of International Conference on Parallel Processing*, 1994.

[75] S. Ramaswamy and P. Banerjee, "Processor allocation and scheduling of macro dataflow graphs on distributed memory multicomputers by the paradigm compiler," in *Proceedings of International Conference on Parallel Processing*, 1993.

[76] S. Ramaswamy and P. Banerjee, "Simultaneous allocation and scheduling using convex programming techniques," *Parallel Processing Letters*, Dec. 1995.

[77] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Research and Development*, no. 1, pp. 25–33, 1967.

[78] W.-M. Hwu and Y. Patt, "HPSm, a high performance restricted data flow architecture having minimum functionality," in *Proceedings of Annual International Symposium on Computer Architecture*, 1986.

[79] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, pp. 1609–1624, Dec. 1995.

[80] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of Annual International Symposium on Computer Architecture*, 1981.

[81] A. Smith and J. Lee, "Branch prediction strategies and branch-target buffer design," *IEEE Computer*, pp. 6–22, Jan. 1984.

[82] T. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of Annual International Symposium on Computer Architecture*, 1993.

[83] S. MacFarling, "Combining branch predictors," *WRL Technical Note, Digital Western Research Laboratory*, 1993.

[84] E. Rotenberg, J. Smith, and S. Bennett, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of International Symposium on Microarchitecture*, 1996.

[85] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative fetch and issue policies for the trace cache fetch mechanism," in *Proceedings of International Symposium on Microarchitecture*, 1997.

[86] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of Annual International Symposium on Computer Architecture*, 1986.

[87] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, Jan. 1989.

[88] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of Annual International Symposium on Computer Architecture*, 1995.

[89] W.-M. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, pp. 1496–1514, Dec. 1987.

[90] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Transactions on Computers*, pp. 562–573, May 1988.

[91] G. S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Transactions on Computers*, pp. 349–359, Mar. 1990.

[92] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-M. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," 1994.

[93] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, no. 7, pp. 478–490, 1981.

[94] W.-M. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: an effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, no. 1, pp. 229–248, 1993.

[95] M. Lam, "Software pipelining: An effective scheduling technique for VLIW processors," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[96] B. Rau, M. Schlansker, and P. Tirumalai, "Code generation schema for modulo scheduled loops," in *Proceedings of International Symposium on Microarchitecture*, 1992.

[97] B. Rau, "Iterative modulo scheduling," *International Journal of Parallel Programming*, Feb. 1996.

[98] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of International Symposium on Microarchitecture*, 1992.

[99] W.-M. Hwu, D. I. August, and J. W. Sias, "Program decision logic optimization using predication and control speculation," *Proceedings of the IEEE*, pp. 1660–1675, Nov. 2001.

[100] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: a model for compiler-controlled speculative execution," *ACM Transactions on Computer Systems*, pp. 376–408, Nov. 1993.

[101] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," *Communications of the ACM*, Aug. 1992.

[102] B.-C. Cheng and W.-M. Hwu, "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[103] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *Proceedings of the Static Analysis Symposium*, 2004.

[104] M. J. S. Smith, *Application-Specific Integrated Circuits*. Addison-Wesley Publishing Company, 1998.

[105] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1507–1522, Dec. 2000.

[106] R. Domer, "The SpecC system-level design language and methodology, part 1," in *Embedded Systems Conference*, 2002.

[107] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the International Conference on VLSI Design*, 2003.

[108] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[109] "CoCentric SystemC Compiler." Synopsys Inc.

[110] "Catapult C." Mentor Graphics Corporation.

[111] Impulse Accelerated Technologies. http://www.impulseC.com.

[112] Celoxica Limited. http://www.celoxica.com.

[113] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD distributed-memory machines," *Communications of the ACM*, vol. 35, Aug. 1992.

[114] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. S. Jr., and M. E. Zosel, *The High Performance Fortran Handbook*. The MIT Press, 1994.

[115] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI, the complete reference*. The MIT Press, 1994.

[116] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, pp. 48–56, Nov. 1980.

[117] A. H. Veen, "Data flow machine architecture," *ACM Computing Surveys*, no. 4, pp. 365–396, 1986.

[118] Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on Computers*, no. 3, pp. 300–318, 1990.

[119] P. G. Whiting and R. S. V. Pascoe, "A history of data-flow languages," *IEEE Annals of the History of Computing*, no. 4, pp. 38–59, 1994.

[120] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, no. 1, pp. 1–34, 2004.

[121] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[122] P. Tu and D. Padua, "Gated-ssa based demand-driven symbolic analysis for parallelizing compilers," in *Proceedings of International Conference on Supercomputing*, 1995.

[123] R. Ballance, A. Maccabe, and K. Ottenstein, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[124] D. D. Gajski, D. A. Padua, D. J. Kucle, and R. H. Kuh, "A second opinion on data-flow machines and languages," *IEEE Computer*, no. 2, pp. 58–69, 1982.

[125] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, Sept. 1987.

[126] E. A. Lee and T. Parks, "Data-flow process networks," *Proceedings of the IEEE*, vol. 83, pp. 773–799, May 1995.

[127] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, Jan. 1987.

[128] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for dsp applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 1995.

[129] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor dsp systems," *IEEE Transactions on Signal Processing*, June 1997.

[130] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for signal processing systems," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 47, Sept. 2000.

[131] S. S. Bhattacharyya, N. Bambha, M. Khandelia, and V. Kianzad, "Mapping dsp applications onto self-timed multiprocessors," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 2001.

[132] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the International Conference on Compiler Construction*, 2002.

[133] "Brook project web page." http://brook.sourceforge.net.

[134] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2004.

[135] C. Moore, "Cmps: Now and into the future." Keynote Presentation, Workshop on Design, Architecture and Simulation of Chip Multi-Processors 2005.

[136] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the Omega test," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[137] K. K. Parhi, *VLSI Digital Signal Processing Systems, design and implementation*. John Wiley and Sons, 1999.

[138] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependency graph and its uses in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, June 1987.

[139] P. Tu and D. Padua, "Efficient building and placing of gating functions," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[140] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1996.

[141] B.-C. Cheng, "Compile-time memory disambiguation for c programs," Ph.D. dissertation, University of Illinois, 2000.

[142] M. Das, "Unification-based pointer analysis with directional assignment," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[143] M. P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 85–122, Jan. 1995.

[144] M. Wolfe, "Beyond induction variables," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[145] R. Sedgewick, *Algorithms*. Addison-Wesley.

[146] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company.

[147] S.P.Amarasinghe, "Parallelizing compiler techniques based on linear inequalities," Ph.D. dissertation, Stanford University, 1997.

[148] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism-enhancing transformations," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[149] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, pp. 350–360, July 1991.

[150] Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of array access patterns for compiler optimizations," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[151] S. Moon and M. W. Hall, "Evaluation of predicated array data-flow analysis for automatic parallelization," in *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1999.

[152] W. Pugh and D. Wonnacott, "Constraint-based array dependence analysis," *ACM Transactions on Programming Languages and Systems*, pp. 635–678, May 1998.

[153] D. Wonnacott, "Constraint-based array dependence analysis," Ph.D. dissertation, University of Maryland, 1995.

[154] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[155] K. D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[156] T. Brandes, "The importance of direct dependences for automatic parallelism," in *Proceedings of International Conference on Supercomputing*, 1988.

[157] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opẽrationnelle*, vol. 22, pp. 243–268, Sept. 1988.

[158] S. Moon, M. W. Hall, and B. R. Murphy, "Predicated array data-flow analysis for run-time parallelization," in *Proceedings of International Conference on Supercomputing*, 1998.

[159] P. Tu, "Automatic array privatization and demand-driven symbolic analysis," Ph.D. dissertation, University of Illinois, 1995.

[160] "Impact web page." http://www.crhc.uiuc.edu/impact.

[161] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of International Symposium on Microarchitecture*, 1997.

[162] "uDraw(graph) web page." http://www.informatik.uni-bremen.de/uDrawGraph/en/home.html.

[163] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

AUTHOR'S BIOGRAPHY

Chien-Wei Li was born in Kaohsiung, Taiwan, on April 24, 1968, to a military family. His grandparents and parents migrated from Shandong, China to Taiwan in 1949. He grew up in Taipei, and attended the National Taiwan University, where he obtained the B.S. degree in 1990, and the M.S. degree in 1992, both in computer science. His M.S. thesis work is the design and implementation of a distributed file system which essentially implemented a disk array using commodity personal computer hard disks and token ring networks. After serving in the Navy from 1992 to 1994, he was enrolled in the computer science PhD program at the University of Illinois at Urbana-Champaign, and worked as a research assistant at the Coordinated Science Laboratory, doing researches on the design of processor array and coprocessor for computing recurrence.

In 1996, he worked as a summer intern at Rockwell Semiconductor Inc., Newport Beach, California. For the functional verification of the instruction pipeline control of a digital signal processor, he designed and implemented a test vector generator, which surprised the designers by catching several obscure bugs. During his second summer intern, in 1997, he synthesized the instruction pipeline control to investigate the feasibility of a synthesis based approach for Rockwell's next generation digital signal processor. From 1998 to 2001, he worked as a full-time design engineer at Conexant Inc., formerly Rockwell Semiconductor Inc., doing performance evaluation, micro-architecture design, functional verification, and RTL design and synthesis of the instruction pipeline control

of Conexant's next generation digital signal processor. In November, 2001, he returned to school to resume his PhD research.

After defending his PhD thesis, he joined the platform ingredient architecture group of Intel, at Hillsboro, Oregon, in August, 2005. His general interests are general-purposed or special-purposed processor design and implementation, compilers, operating systems, and digital signal processing, digital communication and computer graphics applications.