INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313 761-4700 800 521-0600

Order Number 9136566

Compiler support for multiple-instruction-issue architectures

Chang, Po-hua, Ph.D.

University of Illinois at Urbana-Champaign, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

COMPILER SUPPORT FOR MULTIPLE-INSTRUCTION-ISSUE ARCHITECTURES

BY

PO-HUA CHANG

B.A., University of California, Berkeley, 1987
M.S., University of Illinois, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MAY 1991

WE HEREBY RECOMMEND THAT THE THESIS BY

PO-HUA CHANG

ENTITLED COMPILER SUPPORT FOR MULTIPLE-INSTRUCTION-ISSUE ARCHITECTURES

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

O-517

COMPILER SUPPORT FOR MULTIPLE-INSTRUCTION-ISSUE ARCHITECTURES

Po-hua Chang, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1991
Wen-mei Hwu, Advisor

This dissertation demonstrates that substantial speedup over that for conventional single-instruction-issue architectures can be achieved by multiple-instruction-issue architectures with the support of an optimizing compiler. We have constructed a full-scale C compiler that can learn the dynamic behavior of user programs by profiling, apply the profile information to guide various code improving techniques, and map the program parallelism onto the parallel architecture. Our base code optimization technology is comparable to today's best commercial C compilers. In addition, we have developed aggressive code generation techniques that are tailored to multiple-instruction-issue architectures. Using our compiler, we have characterized the performance of a large class of multiple-instruction-issue architectures with many important application programs and realistic input data.

DEDICATION

To My Parents

ACKNOWLEDGMENTS

I wish to thank my parents for many years of love and support. They have always introduced me to the best educational environment at every stage of my life.

I would like to express my sincere gratitude to all the people who have made my school life enjoyable and productive. Professor Wen-mei Hwu, my thesis advisor, has been my role model ever since my undergraduate years. In the five years that we have worked together, we have persevered through some hard times. But his vision and optimism have always made our struggle less painful. Professors Michael Loui, Janak Patel, and Prithviraj Banerjee have served on my doctoral committee and provided me with many valuable suggestions that improved my research work. Andy Glew, Sadun Anik, Tomas Conte and all my colleagues in the Center for Reliable and High-performance Computing (CRHC) gave me valuable suggestions and references to further my study. I would like to thank Professor Yale Patt for introducing me to the field of computer architecture.

I would like to acknowledge many people who have contributed and are still working on various components of the IMPACT-I C compiler. Scott Mahlke has contributed a large portion of the code optimizer. William Chen has implemented a code generator for the MIPS-R2000 microprocessor. Roland Ouellette has implemented a code generator for the Sparc microprocessor. Roger Bringmann has implemented a code generator for the AMD-29K microprocessor. Nancy Warter has implemented a code generator for the i860 microprocessor on an Alliant-2800 system. Grant Habb is building an array subscript memory dependence analyzer. Rick Hank is constructing a more aggressive register allocator. John Holm is implementing more machine-dependent code optimizations. These people have spent many weekends and sleepless nights with me in the laboratory throughout the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer engineers have been striving to improve uniprocessor performance since the invention of computers. Recently, many designers have demanded the use of the most powerful microprocessors in their embedded controller and workstation applications. Designers of multiprocessors have also become accustomed to using the most powerful microprocessors that are available on the market as the node processors of multiprocessor architectures. To achieve high performance in a microprocessor, conventional wisdom suggests exploiting concurrency and using the best circuit technology. Advances in circuit technology have reduced the time to perform basic hardware functions. With instruction pipelining and overlapping [Kogge 81], the basic machine cycle time has been greatly reduced over the years. By optimizing a simple instruction pipeline structure, current RISC (Reduced Instruction Set Computer) processors achieve an instruction execution rate of nearly one operation per cycle [Hennessy 81]. A natural extension to instruction pipelining is to design microprocessors that can execute multiple operations per cycle. To consistently perform at this level, these processors must be able to fetch, decode, issue, execute, and commit more than one operation per cycle. Such a processor has been called a *superscalar processor*, a *very long instruction word (VLIW) processor*, and a *multiple-instruction-issue processor* in recent literature.[1] Superscalar processor architectures differ from VLIW architectures in the instruction fetch/decode/issue pipeline stages. In a superscalar processor, the hardware decodes multiple operations simultane-

[1] In this dissertation, an *operation* denotes the basic execution unit. Therefore, we also use the term *multiple-operation-issue processor*.

ously and decides which operations may be issued to the execution stage as a group. For a VLIW processor, the compiler decides which operations can be issued to the execution stage as a group at compile time, and packs these operations into a wide instruction word. In a VLIW processor, the hardware issues one wide instruction word to the execution stage at a time. In this dissertation, we present many compiler techniques that are applicable to both superscalar and VLIW processor architectures. We refer to both superscalar and VLIW processors as multiple-instruction-issue processors. We will make a distinction between superscalar and VLIW architectures when we present a technique that pertains to only one of the two architectures.

In this dissertation, tables and figures always appear at the end of each chapter. Figure 1.1 shows the timing diagram of the execution of four operations by a non-pipelined processor, a pipelined processor, and a multiple-instruction-issue processor. A major task of the compiler for a VLIW processor is to detect a sufficient number of independent operations to saturate the instruction pipeline. The compiler arranges operations in the instruction memory in such a way that when operations have been fetched and decoded, the values of all source operands are available, and these operations can immediately move to the execution stage of the instruction pipeline. The compiler packs independent operations into wide instruction words. The hardware can issue at most one wide instruction word to the execution hardware per cycle [Fisher 81], [Ellis 86], [Colwell 87], [Howland 87].

Alternatively, the detection of independent operations can be performed by the hardware, as in a superscalar processor. The hardware fetches and decodes one or more operations per cycle. After operations have been decoded, the hardware detects operations that can be executed concurrently and whose source operand values are available. The hardware prevents the execution of operations whose source operand values are not available. Therefore, the order in which operations are issued to the execution stage may be different from the order in which these operations are fetched from the instruction memory. The hardware can issue multiple independent operations to the execution stage per cycle [Acosta 86], [Sohi 87], [Weiss 87].

Another method is to use a combination of compile-time and run-time scheduling techniques. The compiler groups independent operations into wide instruction words. The hardware can fetch and decode one wide instruction word per cycle and allow operations from different wide instruction words to execute out of the order in which these operations are fetched [Hwu 87], [Patt 85].

It is unclear how much performance the combined compiler and hardware scheduling method can achieve beyond the improvement by either method alone. It is unclear how close the research community has come to the performance limit of multiple-operation-issue architecture with existing compiler and hardware techniques. Complete answers to these questions would require many experimental research projects that propose new compiler and hardware techniques, measure the effectiveness of existing compiler and hardware techniques on important application programs that exist today and on programs that are written in explicitly parallel languages which promote the use of parallel data structures and algorithms.

Many hardware and software techniques for using multiple function units and supporting multiple-operation-issue architectures have been studied [Fisher 81], [Patt 85], [Smith 85a], [Acosta 86], [Ellis 86], [Sohi 87], [Hwu 87], [Howland 87], [Weiss 87]. Recent interest in applying these techniques to low-cost microprocessor and microsystem designs has grown dramatically [Colwell 87], [Hwu 88a], [Hwu 88b], [Pleszkun 88a], [Jouppi 89b], [Smith 89], [Sohi 89], [Cohn 89], [Intel 89], [IBM 90]. We will discuss the results from some of these studies in Chapter 2.

1.1 The Dissertation

In this dissertation, we focus on improving the performance of some important application programs that were written in the C programming language. These application programs exhibit complex control flow and use complex data structures. We evaluate the effectiveness of existing compiler and hardware techniques on these programs, and

show experimentally that multiple-operation-issue processors can outperform by large amounts processors that issue one operation per cycle.

This research has three major objectives. The first objective is to characterize the performance of multiple-operation-issue architectures using an optimizing compiler. The second objective is to characterize the effectiveness of code optimizations that are designed specifically for multiple-operation-issue processors. The third objective is to provide a modular compiler framework, in which new code optimizations can be quickly implemented, evaluated, and transferred to common use.

An optimizing compiler plays an essential role in processor architecture studies for two important reasons. First, existing application programs are written primarily in high-level languages. To measure the execution time of a large set of existing application programs on a new architecture, a compiler for that architecture must be available. Second, a naive compiler can translate the application programs into inefficient code that may not exercise all hardware functions. A naive compiler can also generate many redundant computations that show unrealistic parallelism. To conduct a fair study of processor performance, the best compiler support should be provided for each processor architecture.

We have implemented a full-scale optimizing C compiler from scratch. This compiler, which we named the IMPACT-I C compiler, can learn the dynamic behavior of the object program prior to compilation, and use that knowledge to guide a large number of code improving techniques. The IMPACT-I C compiler has been ported to a few existing commercial machines. The IMPACT-I C compiler can generate code for the MIPS-R2000, SPARC, i860, and AMD29K microprocessors. In 1991, we plan to construct code generators for the i486, i960, and IBM-RS6000 microprocessors. We distributed the first beta test version of the IMPACT-I C compiler to NCR in February 1991. We plan to release the IMPACT/AMD29K C compiler in April 1991 and the next beta test version of the IMPACT-I C compiler in May 1991.

We will show that the quality of the code emitted by the IMPACT-I C compiler is comparable to that from today's best commercial C compilers. From a sound base

compiler technology, we have further developed aggressive code transformation, register allocation, and code scheduling strategies that are tailored for multiple-operation-issue machines. We have extracted more instruction-level parallelism and have achieved a speedup ratio that is much greater than that reported by previous studies [Tjaden 70], [Smith 89], [Sohi 89], [Jouppi 89b].

1.2 Organization of the Dissertation

This dissertation is organized into ten chapters.

Chapter 2 provides necessary background information, defines important terms, and surveys related works on multiple-operation-issue architectures.

Chapter 3 describes the IMPACT architectural framework of multiple-instruction-issue processors.

Chapter 4 gives an overview of the IMPACT-I C compiler and describes briefly the functions of its major components. The IMPACT-I C compiler uses two levels of intermediate code to communicate between various tools and compiler components. Based on the two levels of intermediate code, two major programming environments have emerged.

Chapter 5 describes machine-independent code optimizations that have been implemented in the IMPACT-I C compiler. Traditional local and global code optimizations, function inline expansion, instruction placement, and profile-based code optimizations are all part of the machine-independent code optimizer. We compare the object code quality against leading commercial C compilers. The measurement data show that the IMPACT-I C compiler generates highly optimized object code.

Chapter 6 describes machine-dependent code optimizations, including constant preloading, register allocation, and code scheduling. We describe how these optimizations may degrade the performance of each other. An integrated register allocation and code scheduling strategy is described in this chapter.

Chapter 7 describes some code transformation techniques that enlarge the scope of compile-time code scheduling and reduce the lengths of critical paths.

Chapter 8 presents measurement data demonstrating the speedup ratio of many multiple-operation-issue processor architectures over a fixed base processor architecture. We also compare the performances of compile-time and run-time code scheduling.

Chapter 9 describes a branch architecture which allows multiple branch operations to be issued per cycle and from branch slots. We show that, by selectively allocating branch slots, the code expansion penalty due to branch slots is small.

Chapter 10 offers concluding remarks and future directions.

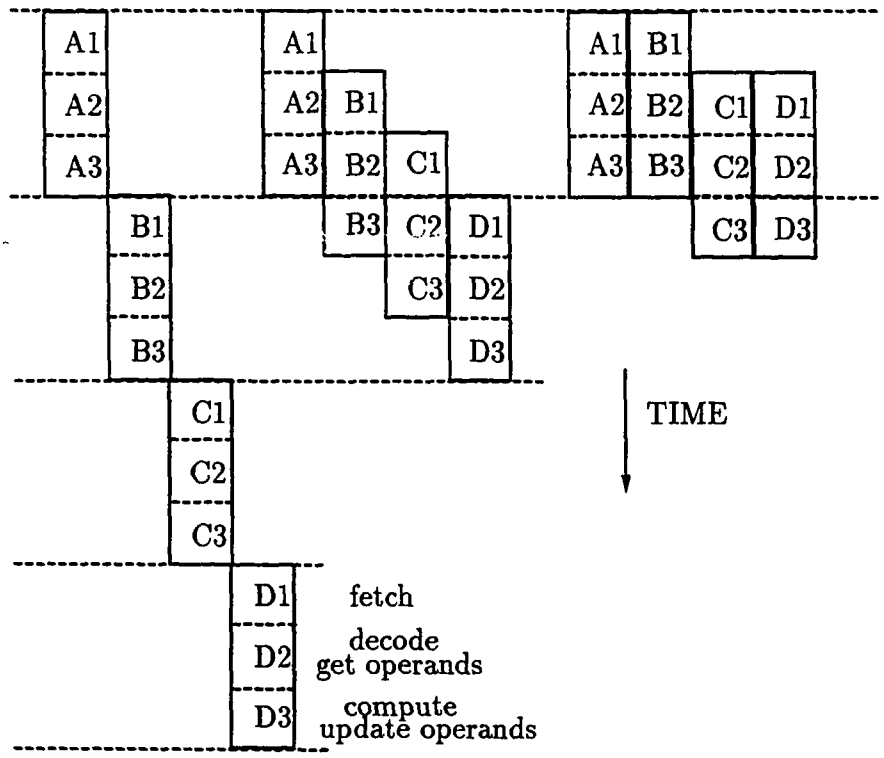


Figure 1.1 Timing diagram of the execution of four operations.

CHAPTER 2

BACKGROUND

2.1 Fundamental Concepts

We define the *state* of a programmed machine as the collection of the values of all of its memory elements. Thus, the memory elements can be regarded as state variables whose values belong to a well-defined range. A *state change* occurs if there is a change in the value of any one memory element.

An *operation* is denoted by a quadruple *(OP, SR, DS, RS)* where *OP* indicates a primitive hardware-defined function, *SR* (source) and *DS* (destination) are sets of memory elements, and *RS* is the hardware resource that is required to carry out the hardware-defined function. The execution of an operation consists of the following phases. 1) The operation is fetched from the instruction memory. 2) The values of all source operands (*SR*) are obtained. 3) A value is produced by the primitive hardware-defined function (given values of all source operands). 4) The destination operand (*DS*) is assigned the resultant value. All of the above phases are implemented in hardware.

A *purely sequential machine* executes a single operation at a time. Therefore, given a certain program and a specific input, an execution sequence of operations is derived. Let one such sequence be $< op_1, op_2, ..., op_n >$ (n a finite number). We define *observation points* as cuts at several operation boundaries, where users are allowed to probe a subset of the *state* of the program. For a total of m ($m < n$) such observation points, the *result of program execution* is denoted by $< s_1, s_2, ..., s_m >$, where s_i corresponds to the subset of the program *state* that is visible at the i_{th} observation point.

All alternative execution models must satisfy the conditions of *determinacy* and *termination* [Karp 66]. Informally stated, given all legal input data to a program, the *result of program execution* must be identical for all execution models that satisfy the *determinacy* property, and the length of the execution sequence must be finite for all execution models that satisfy the *termination* property.

2.1.1 Transformation of execution sequence by software

Compile-time code transformation produces a new version of the object program. Therefore, the sequence of operations that are fetched from the instruction memory could be different from that of the original program. A useful application of compile-time code transformation is to reduce the length of the execution sequence. For example, a multiplication of an integer value by 4, which takes several machine cycles, can be replaced by a single-cycle bit-shift operation. Another useful application of compile-time code transformation is to schedule operations so that once an operation is fetched from the instruction memory, the needed source operands and function unit resources are immediately available to execute the operation.

In general, the behavior of the execution hardware is fixed (e.g., is pipelined into four stages). When hardware parameters, e.g., delay of multiply operation, and organization, e.g., datapath, are specified, compilers can tailor the output object code to maximize the resource utilization and to minimize the execution time.

2.1.2 Transformation of execution sequence by hardware

Figure 2.1 depicts the behavior of the execution hardware. The *fetch-decode* component obtains a finite number of operations from the instruction memory and inserts them into the output queue per cycle. While waiting in the output queue of the *fetch-decode* component, operations gather their source operands. The *issue* component selects a finite number of operations whose source operands have been obtained and whose function unit resources have been reserved, and moves the operations to the output queue

of the *issue* component. The *execute* component takes operations from its input queue and delivers the result to its output queue. Finally, the *commit* component updates the program *state*.

In Figure 2.1, *concurrency detection* is the action to identify all operations that can be moved from the input queue to the output queue of the *issue* component. Due to limited hardware resources, e.g., limited bus bandwidth, not all concurrent operations can be moved to the next stage. *Scheduling* is the process of selecting a subset of concurrent operations to be moved to the next stage. Depending on the scheduling policy used, the operation sequence may be altered in any one component in Figure 2.1. A scheduling policy which does not change the operation sequence is said to be *in-order;* otherwise, it is said to be *out-of-order.*

The *fetch* component implements an in-order process. For example, upon an instruction cache miss, the fetch unit does not try to fetch the subsequent operation. It is necessary to fetch operations in-order to establish a precedence relationship between operations, e.g., assigning a tag to each operation. Operation precedence is an essential piece of information in implementing register renaming, exception handling, and squashing in out-of-order execution machines. Squashing cancels an operation by converting its opcode to no-op or by clearing its valid bit. A squashed operation is prevented from changing the machine state. An operation should be squashed if it is fetched after an incorrectly predicted branch operation or a trapping operation.

If the *issue* component implements an in-order process, the machine is said to be *in-order issue.* Otherwise, it is *out-of-order issue.* If both the *issue* and the *execute* components implement in-order processes, the machine is said to be *in-order execution*; otherwise, it is *out-of-order execution.*

An operation *commits* if it modifies the program state. To support precise interrupt, it is necessary to *commit* in-order. It is not in the scope of this research to address interrupt handling issues. Several techniques for implementing precise interrupts have been proposed [Smith 85a], [Sohi 87], [Hwu 87] . These techniques allow operations to modify the memory before they commit.

2.1.3 Detection of dependent operations

Consider two operations op_i and op_j, where op_i precedes cp_j in sequential mode of execution. The problem is to decide whether we could issue op_i and op_j at the same time, or issue op_j before op_i in a multiple-operation-issue processor.

The dependence graph is a well-known representation of operation precedence relationships [Tjaden 70], [Kuck 81]. Consider two operations op_i and op_j, where op_i precedes op_j in sequential mode of execution. We say that

- op_j is *flow-dependent* on op_i iff the destination operand of op_i is a source operand of op_j, and there is no other operation fetched after op_i but before op_j such that it has the same destination operand as op_i.
- op_j is *anti-dependent* on op_i iff the destination operand of op_j is a source operand of op_i.
- op_j is *output-dependent* on op_i iff they have the same destination operand.
- op_j is *control-dependent* on op_i iff op_i is a branch operation.

A dependence graph constructed by adding dependence arcs to operations in a dynamic operation trace is acyclic. The length of each dependence arc corresponds to the minimum number of cycles between the issue time of the source and destination nodes. For a flow-dependence arc, the length is usually the operation latency of the source node.

A dependence graph constructed from a program graph may be cyclic due to loop structures.

2.1.4 Hazard prevention

Consider an integer multiply operation op_i and another operation op_j which uses the result of op_i. If the delay of multiply operation is 6 cycles, then op_i must be issued to the *execution* component at least 6 cycles before op_j can be issued to the *execution* component. If this enforcement is accomplished by the hardware, then the mechanism

for enforcing data dependencies is called *hardware interlocking.* Detailed descriptions of various ways to implement hardware interlocks have been surveyed by Kogge [Kogge 81].

Dependence distance can be reduced by adding hardware features. Consider two operations op_i and op_j, where op_i precedes op_j in sequential mode of execution. If hardware register renaming allows op_i and op_j to write the same destination operand, then the length of this output-dependence arc is zero, and op_i and op_j may be issued at the same time. For another example, the length of flow-dependencies may be reduced by 1 cycle with a data forwarding circuit.

Software techniques to prevent hazards are called *software interlocking.* Dependencies can be enforced by reordering operations and inserting no-ops [Hennessy 83].

2.2 Processor Architecture

The term *processor architecture* refers to what the machine language programmers see of a computer system. For example, the VAX, IBM-360, IBM-370, MC68000 architectures allowed families of compatible computers. The definition of a processor architecture usually includes an instruction set, a virtual memory management policy, and an exception and interrupt mechanism.

The term *processor microarchitecture* refers to a particular implementation of a processor architecture. Each microarchitecture is fine-tuned according to specific cost and performance objectives. For example, the width of the internal datapaths, the number of buses, the number of translation buffer entries, the degree of pipelining, the sizes of caches, and many other design choices are never directly visible to users. These microarchitectural choices strongly affect the delivered speed of the processor system, however.

Some functions can be implemented in hardware or in software. To replace a hardware function by a software function, some microarchitectural parameters must be specified to the software designers. For example, by exposing instruction timing information, instruction scheduling schemes that prevent hazards [Kogge 81] can be implemented in software. For example, the MIPS project at the Stanford University used optimizing

compiler technology to exploit a pipelined microprocessor without hardware interlocks [Hennessy 83], [Hennessy 82], [Hennessy 81], [Chow 87]. Gross and Lam have also shown that a compiler can schedule operations for a systolic array computer whose execution timing is deterministic [Gross 86]. Even if there is support for hardware interlocking, compilers can schedule operations to minimize the occurrences of run-time interlocks to achieve higher utilization of the parallel hardware.

2.2.1 Instruction pipelining

We define *instruction* as a number of *operations* that are fetched from the instruction memory, decoded, and issued to the execution unit at a time, in lock-step. For single-operation-issue machines, there is exactly one operation in an instruction; therefore, the two terms can be used interchangeably when referring to a single-operation-issue machine.

An accepted performance measure of executing a benchmark using a particular input is the execution time *(N * C * T)*, where N is the number of instructions that need to be executed, C is the number of cycles per instruction, and T is the cycle time.

The number of instructions that are required to complete a task depends on the instruction set definition and the quality of the code generated by the compiler. Given a fixed instruction set that is designed for efficient streamlining (at a maximal rate of issuing one instruction per cycle), reducing N is one of compiler's major responsibilities.

A goal in designing a fast microarchitecture is to minimize the cycle time and to minimize the number of cycles per instruction. However, once the degree of instruction pipelining has been determined, the cycle time is very much a technology dependent parameter. The length of the instruction pipeline is limited by the data and control dependencies between instructions; therefore, it cannot be arbitrarily increased. The maximum throughput is achieved when the processor completes one instruction per cycle ($C = 1$).

A typical breakdown of the instruction pipeline consists of the following stages. 1) (*fetch*) Fetch one instruction. 2) (*decode*) Decode the instruction, and access source operands. 3) (*issue*) Move operations to function unit input latches. 4) (*execute*) Execute

operations. 5) (*distribute*) Forward results to function unit input latches. Write result to the reordering buffer or future file. [1] 6) (*commit*) Commit the instruction, permanently affecting the program state. Each stage may require several cycles. In the best condition, instructions flow through these stages without blocking, and, effectively, one instruction is executed per cycle.

Figure 2.2 shows the overall organization of a pipelined processor. In Figure 2.2, connections between components should be interpreted as multiple buses. The uppermost component is the instruction fetch stage, in which operations are fetched from the instruction memory. Operations pass from the fetch stage to the decode stage, in which they obtain their source operands (or at least tags for obtaining the values of the source operands later). According to the operation code and the position of the operation in the instruction, each decoded operation is sent to a function unit. The function units should be pipelined for operations that may take more than one cycle to execute.

To extend a single-operation-issue architecture into a multiple-operation-issue architecture, some components in Figure 2.2 should be replicated. 1) The fetch stage should be able to fetch more than one operation per cycle. 2) The decode stage should be able to decode all fetched operations simultaneously. 3) There can be more than one register file to provide more register read and write ports. 4) Some function units should be replicated. For example, we may want to execute multiple branch operations per cycle and multiple memory load operations per cycle. 5) The distribution buses must be able to deliver all results back to the register files.

All forms of hardware concurrencies must be increased in a balanced manner, since the throughput of the instruction pipeline is determined by the slowest stage of the pipeline.

[1] Reordering buffer and future file are hardware data structures used to implement precise interrupts [Smith 85a]. Alternative hardware data structures can be used for the same purpose [Sohi 87], [Hwu 87].

2.2.2 Instruction format

If the compiler can decide which operations are always fetched and issued as a group (an instruction), the compiler can schedule operations (pack operations into an instruction) to avoid function unit and distribution bus conflicts. For example, if the machine has only one floating-point arithmetic unit, it makes no sense to issue two floating-point operations per cycle.

For binary compatibility reasons, we let the hardware decide what operations should be fetched as a group. Therefore, a program compiled for issuing two operations per cycle can also run on a machine that issues a different number of operations per cycle. The disadvantage is that the compile-time management of the function unit resource may not be as efficient.

In both cases, we can use a variable-length instruction format to reduce the number of no-ops in program regions where there are few concurrent operations. For example, Multiflow [Colwell 87] uses a variable-length memory representation.

For *Very Long Instruction Word* machines, an important question is whether we should make all function units powerful enough to handle every operation code. Doing so would require multiple memory read and write ports, and multiple floating-point units. An alternative is to use many heterogeneous function units and let the compiler limit the number of each type of operation in an instruction. Sohi and Vajapeyam have studied the feasibility of this method using small numerical kernels [Sohi 89].

2.2.3 Instruction-fetch limitations

A fixed-length instruction (containing a number of operations) is fetched by indexing the program counter into the instruction memory, in the form of a control store, a cache, or instruction buffers. To achieve near single-cycle execution, it is necessary to fetch at least one instruction per cycle. Instruction and data memory access conflicts can be greatly reduced by the use of separate instruction and data caches [Matick 84]. It is

important to align each instruction word at an instruction cache block boundary so that one wide instruction word can be fetched per cycle.

With a variable-length instruction format, an instruction (containing a number of concurrent operations) may not be properly aligned at an instruction cache block boundary. A possible solution is to provide an instruction buffer which is at least two times larger than an instruction cache block. Except for the first misaligned instruction in a sequential run of instructions, the instruction buffer can combine pieces of data from two instruction cache blocks to form an instruction per cycle. Therefore, one cycle penalty is incurred when branching into a nonaligned instruction. A better but more costly solution is to design an instruction cache memory that can extract and align data from two consecutive cache blocks in a single cycle.

There is a problem with taken branches in multiple-operation-issue architectures. Suppose there are three concurrent operations, including the taken branch. In order to reach an execution rate of four operations per cycle, one operation must be fetched from the taken path. This problem can be solved by using an extremely fast branch target buffer [Lee 84], or by using compile-time branch prediction and squashing branch [McFarling 86]. Hwu, Conte, and Chang [Hwu 89b] have made a direct comparison of the branch target buffer scheme and the squashing branch scheme for a set of C application programs, and reported that both are effective, but to achieve high prediction accuracy, a large number of entries need to be used in the branch target buffer scheme. Smith et al. have shown that instruction fetching is the most severe bottlneck in a superscalar processor [Smith 89].

2.2.4 Instruction-decode limitations

To achieve near single-cycle execution, the instruction decode stage must be able to decode one instruction (multiple operations) per cycle. This is not very difficult if operations have a fixed size format. The decoder simply needs to parse concurrently all operations into several fields: fields for controlling the execution hardware, fields for acquiring source operands, fields for destination operands, and fields for affecting

the control flow. On the other hand, decoding variable-length operations can be very complex and time-consuming [Clark 87], [DeRosa 85].

2.2.5 Branch handling

For a taken branch to redirect the control flow, it must first calculate the *target address* and order the fetch logic to fetch sequentially from the *target path.* Many cycles may be needed to reload the instruction pipeline. To reduce the length of the instruction pipeline that needs to be reloaded, branch prediction techniques can be used to allow reloading without waiting for the condition code to be computed. Several compile-time and run-time branch prediction schemes have been studied [Lee 84], [Smith 81], [McFarling 86], [Hwu 89b].

A good branch predictor does not solve the branch problem. A taken branch can redirect the control flow only after it has been decoded, because the *target address* is encoded in the instruction. A solution is to fetch and decode few instructions which are located subsequent to the branch instruction, while reloading the pipeline from the target address. The delayed branch scheme [Gross 82] always executes a fixed number of instructions subsequent to a branch, regardless of the direction of the branch. The squashing branch scheme [McFarling 86], [Chang 89b] executes the first few instructions from the predicted path as sequential instructions, by code copying, while reloading the instruction pipeline. If the prediction is incorrect, the instructions that are executed from the predicted path are squashed.

2.2.6 Operand-fetch limitations

To issue multiple operations to the execution hardware in every cycle, the decode stage must be able to fetch source operands of all these operations at the same time. If we consider only a load/store architecture, the decode stage must be able to read many register entries at the same time. One approach is to provide one multiple-read-port and multiple-write-port register file for all function units. Another approach is to provide

several register files, each register file having fewer read and write ports than the first approach. To fetch source operands and to modify destination operands every cycle, it is necessary to finish the read operation in a half-cycle. Due to limited current budget and the operation time requirement, there is a limit on the number of read and write ports that can be implemented in the current technology. Increasing issue parallelism to a certain point will require the use of multiple register files.

2.2.7 Resource conflict

Increasing the instruction issue rate from one operation to two operations per cycle does not require all function units to be replicated. For example, adding another floating-point function unit may produce insignificant speedup, because nonnumerical programs rarely need floating-point computation. The effects of varying the number of function units on the performance of multiple-operation-issue architectures have been studied for small numerical kernels [Hwu 88b], [Pleszkun 88a], [Sohi 89].

2.2.8 Cache memories

The performance of a processor depends greatly on how fast the memory system can supply instructions and data. One way to improve the performance of the cache memory subsystem is to increase its size and/or set-associativity [Smith 82], [Hill 85]. This approach is limited because the cache cycle time increases as the size and set-associativity increase and because only a limited amount of chip space is available [Eickenmeyer 88], [Mitchell 88], [Alpert 88], [Przybylski 88], [Hill 88].

From the software side, the performance of the memory system can be improved by program transformation and data placement techniques. Ferrari examined the potential of restructuring programs to improve program paging behavior [Ferrari 83]. Hartley described a function-level program restructuring technique to improve the page-level locality of references and to reduce the number of page faults, using the call graph [Hartley 88]. In array and VLIW processors, multiple memory banks are needed to supply instructions

and data to all processing units. In order to access several pieces of data concurrently, it is necessary to place them in different memory banks. Lawrie published a data alignment technique which allows parallel and conflict-free access to various slices of data for an array processor [Lawrie 75]. Ellis discussed several memory-bank disambiguation methods, which distribute memory accesses evenly to each memory bank [Ellis 86]. Data alignment methods based on data dependence analysis for highly iterative scientific codes have been observed to improve the performance of cache and local memory organizations [Lawrie 75], [Gannon 88], [Breternitz 88]. J.E. Smith and J.R. Goodman have reported the effectiveness of various instruction cache replacement policies and organizations [Smith 85b]. McFarling showed that, by using profile information and excluding certain instructions from the instruction cache, his program restructuring algorithm significantly increased the performance of direct-mapped instruction caches [McFarling 89] Hwu and Chang have proposed another profile-based program restructuring algorithm, independent from McFarling's work, to achieve good performance on small direct-mapped instruction caches [Hwu 89a].

2.3 Scheduling

2.3.1 Hardware scheduling

The freedom to concurrently execute multiple operations is constrained by various forms of dependencies, namely *flow-dependence*, *anti-dependence*, *output-dependence*, and *control-dependence*. If operation op_j is flow-dependent on operation op_i, then the execution of op_j must be postponed until op_i has completed execution and has forwarded the result to op_j. Therefore, the only way to reduce the waiting time is to reduce the execution time of op_i and/or to reduce the data forwarding time. When the outcome of op_i is highly predictable, the execution of op_j may be initiated early with a predicted value of the outcome of op_i. If operation op_j is anti-dependent on operation op_i, then op_j is not allowed to modify its destination operand before op_i has obtained the original value of that operand. Therefore, anti-dependence does not pose any problem in an in-order

issue machine. In an out-of-order issue machine, op_i must keep the original value of its source operands after it is issued. If op_i fails to acquire the value of a source operand, the hardware must guarantee that the operand value will eventually be forwarded to op_i, considering that a later issued operation op_j may intend to write that operand. If operation op_j is output-dependent on operation op_i, then op_i may not write to its destination operand after op_j has written to that operand. Furthermore, op_i must be able to forward its result to all operations that need the value and were fetched after op_i but before op_j. If operation op_j is control-dependent on operation op_i, then op_j is not allowed to commit before op_i has generated the condition code. To achieve more concurrency, hardware scheduling schemes that support out-of-order issuing often can dynamically rename registers and issue ahead of several pending branch operations.

Scoreboarding: A scoreboard is a centralized hardware controller for coordinating the concurrent execution of independent operations [Thornton 70]. The main features of this method are

(1) Issue logic is limited to one operation per cycle.

(2) An operation can be issued even when its source operands are not available. Until all source operands have become available, the operation is said to be *pending*.

(3) Issue logic is blocked when it needs to issue an operation that is output-dependent on a *pending* operation.

(4) Issue logic is blocked when it needs to issue an operation to a busy function unit.

(5) Concurrent execution of anti-dependent operations is allowed. But the dependent operation stays *pending* in the function unit, until the first operation completes execution.

(6) All function units communicate through the scoreboard.

Tomasulo algorithm: The Tomasulo algorithm [Tomasulo 67] was first implemented in the IBM 360/91 system. The main features of this algorithm are

(1) Issue logic is limited to one operation per cycle.

(2) Each function unit has a few *reservation stations* where operations are held pending.

(3) An operation can be issued even when its source operands are not available. Until all source operands have become available, the operation is held pending in a reservation station.

(4) Each register entry and reservation station source operand entry contains a *busy bit* and a *tag* indicating the location of the pending operation which will produce the value, when the busy bit is set.

(5) When a result is produced, the *common data bus* broadcasts the value to all register entries and all reservation stations, which use associative *tag* match to read the result off the bus.

(6) The decode logic assigns the value of a register to an operation when the register busy bit is not set; otherwise, the *tag* of the register is assigned to the operation.

(7) Dynamic register renaming reduces anti-dependency and also output-dependency.

(8) The issue logic blocks when it needs to issue an operation to a function unit which has no more available reservation station.

Several derivatives of the Tomasulo algorithm have been proposed [Weiss 84], [Hwu 86], [Sohi 87].

2.3.2 Software scheduling

The code scheduling problem has been studied in many different contexts such as inventory control and manufacturing systems. A survey of scheduling techniques prior

to 1977 can be found in [Gonzalez 77]. Here, we will discuss only a small subset of the published results in software scheduling that are directly relevent to this research.

Local microcode compaction works on a straight-line code without branches [Kleir 71], [Davidson 81]. The problem with local microcode compaction is that basic blocks typically contain very few operations to work with. Trace scheduling extends straight-line code compaction by grouping several basic blocks into a trace [Fisher 81], [Ellis 86], [Howland 87], [Colwell 87]. The global microcode compaction technique works on an entire function at a time [Tokoro 81]. Code scheduling for other architectures is very similar to microcode compaction. Bruno, Jones, and So [Bruno 80] have described techniques of deterministic scheduling for pipelined processors. Hennessy and Gross have described a postpass code reordering scheme to ensure software interlocking [Hennessy 83]. Sahni has studied the problem of scheduling multipipelined and multiprocessor computers [Sahni 84]. Arya [Arya 85] has described an optimal instruction-scheduling model for a class of vector processors. Gibbon and Muchnick have studied instruction scheduling for a pipelined architecture [Gibbons 86]. Davidson has described a retargetable instruction reorganizer [Davidson 86]. Gross and Lam have described an instruction scheduling scheme for systolic arrays [Gross 86]. Granski, Koren, and Silberman have measured the effect of code scheduling on the performance of a dataflow computer [Granski 87]. Eisenbeis has studied the code compaction of loops [Eisenbeis 88]. Lam has studied software pipelining for VLIW machines [Lam 88]. Most recently, code scheduling has appeared in compilers for superscalar microprocessors [Warren 90], [Golumbic 90]. The following paragraphs will provide more discussion of some of the research that has been mentioned above.

Software interlocking: One extreme point of instruction scheduling is to enforce all dependencies by software scheduling at the compile-time; it is called *software interlocking.* A software interlock is provided by reordering operations and inserting no-ops to prevent hazards. Because at most one instruction is fetched per cycle, inserting a no-op between two operations ensures that the fetch times of the two operations are at least two cycles

apart. The objective of a code reorganizer is to minimize the length of the schedule while enforcing software interlock. Hennessy and Gross have shown in [Hennessy 83] that the complexity of this problem is *NP-complete* and have proposed a heuristic algorithm. The scope of code reordering of this heuristic algorithm is limited to within a basic block. Hennessy and Gross have shown empirical data that their heuristic algorithm performs well in practice [Hennessy 83].

Trace scheduling: Trace scheduling [Fisher 81], [Ellis 86] has been a popular technique among VLIW (Very Long Instruction Word) machines [Colwell 87], [Fisher 83]. VLIW machines have the following features: 1) There is a central controller issuing a single long instruction word per cycle. 2) Each instruction word contains many independent operations. 3) Each operation requires a statically predictable number of cycles to execute. 4) Each operation may be pipelined.

VLIW compilers are totally responsible for controlling all datapaths and functional units. The scope of code scheduling can be increased by function inline expansion, loop unrolling, and trace scheduling. A trace is a loop-free sequence of operations that are likely to be executed contiguously for most input data. Trace scheduling consists of a loop of three steps: *trace selection*, *code compaction for a trace*, and *generation of repair code*. Trace selection can be based on static analysis of the program structure or on profile information. Several selection heuristics have been studied in [Chang 88]. Code compaction of a trace is identical to that of a local microcode compaction algorithm. Code motion across branch operations may cause logical inconsistencies when branching off from the middle of a trace, or entering a trace from its middle. Therefore, some repair code has to be generated for these off-trace branches.

Percolation scheduling: Unlike trace scheduling in which code is compacted only in one trace at a time, Nicolau's percolation scheduling allows operations to *percolate* from the various parts of the program graph towards the start node [Nicolau 85]. Code motion is accomplished by repeatedly applying a small set of primitive program transformations

between adjacent operations. Nicolau uses a set of rules to decide when and where to apply the primitive program transformations. After code motion has been stablized, a list scheduling algorithm [Coffman 76] is used to map the program graph onto the hardware.

Microcode compaction: Code generation for a multiple-operation-issue machine is very much like horizontal microprogramming. The only difference is that horizontally microprogrammed machines are often more irregular in structure and more complex in timing than multiple-operation-issue machines, such as VLIWs and superscalars. The code scheduling model for horizontal microprogramming is thus more complex. Early microprogramming techniques have been summarized in a number of survey papers [Agerwala 76], [Landskov 80], [Rauscher 80]. Previous works on microcode optimization have treated several different objectives: minimizing the control memory, minimizing the control word complexity, minimizing the schedule length, and minimizing the programming effort. Background information on all these topics can be found in [Kleir 71], [Tsuchiya 76].

Local code compaction means that the scope of code compaction is limited to within a basic block. A realistic machine model for local code compaction can be found in [Davidson 81]. Davidson et al. have compared four local code compaction methods: first-come first-serve, critical path, branch and bound, and list scheduling. *Global code compaction* allows code motion across basic block boundaries. Tokoro et al. have described an extension to a critical-path-based local code compaction algorithm that allows moving operations on the critical paths across basic blocks [Tokoro 81]. Code motion across a basic block requires data flow analysis to maintain logical consistency and resource analysis to avoid contention. Isoda et al. have described a global code compaction scheme based on the generalized data dependency graph [Isoda 83]. A special case of global code compaction techniques is *trace scheduling*, which limits code motion within a linear sequence of basic blocks [Fisher 81].

To simplify the work of a compiler to detect concurrent operations, one can develop a high-level language that is most suitable for expressing the intricate timing and concur-

rency constraints, and program in that language. Ramamoorthy and Tsuchiya have described such a language, which is based on the single assignment concept [Ramamoorthy 74]. Dasgupta has surveyed work in high-level microprogramming [Dasgupta 80].

Loop unrolling and software pipelining: Trace scheduling and global microcode compaction techniques may not be useful for inner loops that contain very few operations. Two loop transformation techniques have been commonly applied to enlarge the scope of code scheduling. Loop unrolling replicates the loop body a number of times, removes all intermediate conditional branch operations, and combines all index increment operations into one increment operation. A loop preheader may be required to handle an odd number of iterations. An advantage of loop unrolling is the elimination of some index computations and some conditional branches. Another advantage of loop unrolling is that the scope of code scheduling has been enlarged several times [Weiss 87], [Ellis 86], [Dongarra 79.2]. Software pipelining initiates iterations of a loop before the preceding iteration completes, so that loop bodies of several consecutive iterations can be overlapped. Lam has provided a hierarchical scheme to make loop pipelining applicable to many loops, including those with conditional operations [Lam 88]. Weiss and Smith have shown for small numerical kernels that loop unrolling achieves a 1.7 speedup, and software pipelining achieves a 1.28 speedup for the CRAY-1S scalar architecture [Weiss 87].

Expression-tree height reduction: Kuck et al. have described in detail various ways to reduce the height of expression trees [Kuck 72] by exploiting the associativity, commutativity, and distributivity of arithmetic operations. For example, $(((a + b) + c) + d)$ may be computed in two parallel steps as $((a + b) + (c + d))$. The actual tree rewriting process is straightforward. The major difficulty is in detecting when a rewriting rule is beneficial and should be applied. Reducing the height of expression trees can eliminate some critical paths and allow more concurrent operations.

2.3.2.1 Guarded instruction

Hsu and Davidson have described a decision-tree scheduling algorithm to benefit from using guarded instructions [Hsu 86]. A *decision tree* is a set of basic blocks, in which each interior node is a basic block that terminates in a conditional branch, and each exterior node is a basic block that terminates in an unconditional branch. A *guarded instruction* is a normal instruction plus an additional Boolean *guard expression.* If the guard expression is evaluated to false, the instruction is squashed from the instruction pipeline and effectively becomes a no-op. Using guarded instructions, instructions from a high probability path can be scheduled early to make efficient use of the delayed part of a conditional branch.

Register allocation and code scheduling: Applying register allocation (including assignment [Aho 86]) before code scheduling may sometimes introduce artificial data dependencies due to recycling registers. Code scheduling increases the time between a write to a register and reads of the register after the write. Therefore, code scheduling increases the number of variables that are simultaneously live. It has been found that code scheduling before register allocation (prepass code scheduling) may use more registers than necessary [Goodman 88].

Hwu and Chang have proposed an integrated prepass scheduling method and measured its effectiveness on small numeric kernels [Hwu 88b]. That method consists of three steps: prepass code scheduling, register allocation, and final code scheduling. The effect of using an integrated prepass scheduling method on a pipelined superscalar (issuing 2 operations per cycle, 32 registers) is about a 40% reduction in execution cycle count.

Goodman and Hsu have proposed two methods to integrate the register allocation and code scheduling in large basic blocks [Goodman 88]. Their first method is also an integrated prepass scheduling method. The effect of using this method for a heavily pipelined processor can be as much as a 100% reduction in instruction count when the register resource is constrained (15 registers). When the register resource is scarce, register spilling when the next issuing operation has long interlock with previously issued

operations can be profitable. Their second method is a DAG-driven register allocator, using the dependence graph to guide the register assignment. This method has also been shown to be effective for large basic blocks.

2.4 Comparison with Similar Works

This research explores many compiler and hardware techniques that may affect the performance of a multiple-operation-issue processor. Measurement data are derived from some realistic C programs that are in common use. Table 2.1 lists the benchmark programs that are used in this research.

Most previous research work in multiple-function-unit and multiple-operation-issue architectures has focused on numerical programs that have large amounts of instruction-level parallelism in the original source code. This dissertation addresses control intensive C programs, which are substantially more difficult to parallelize because branch operations occur frequently, and the number of loop iterations is usually small. Many classic code optimizations, such as loop unrolling and software pipelining, are less effective for nonnumeric C programs than for numeric Fortran programs. In general, loop optimizations may introduce extra operations to set up a more efficient or more parallel version of the loop body. For software pipelining, several iterations are executed prior to reaching the software-pipelined loop body. In the C programs that we have studied, many loops iterate only a few times. For these loops, the software-pipelined loop body would rarely be executed. Loops that iterate very many times usually involve memory accesses through pointers. Without very powerful memory disambiguation analysis, very limited code motion can be performed for the unrolled version of the loop. Because we insist on implementing a fully automatic C compiler, we have implemented a comprehensive suite of code optimization and analysis programs, instead of treating only one or two code optimizations. The measurement data that we present in this dissertation belong to the category of automatic program parallelization. We compile the benchmark programs in their original form.

Hwu and Patt have designed and measured the performance of the HPSm microprocessor that can issue several operations per cycle and can dynamically schedule operations [Hwu 86]. In this dissertation, we have provided a much larger set of code optimization techniques and have measured the performance of both in-order and out-of-order execution architectures. We report the performance of a large class of multiple-instruction-issue architectures, instead of one processor implementation.

Smith, Johnson, and Horowitz have studied the performance of out-of-order execution architectures and have derived many interesting design points [Smith 89]. Using the commercial MIPS C compiler that schedules code specifically for the single-operation-issue MIPS processor architecture, Smith, Johnson, and Horowitz have not used more powerful code transformation and static scheduling techniques. We have implemented and applied many powerful code transformation and static scheduling techniques in our study of out-of-order execution architectures.

Smith, Lam, and Horowitz have proposed an in-order execution architecture that totally relies on static code scheduling [Smith 90]. They have provided special hardware support for *boosting* (moving) operations above a branch operation and have obtained a performance level that is comparable to that for a purely dynamic scheduling architecture. They have used only local code scheduling. Part of this dissertation also compares the performance of static and dynamic code scheduling methods. We have implemented aggressive code transformation and global code scheduling algorithms. We show that instruction boosting provides insignificant performance beyond a good global code scheduling algorithm.

Table 2.1 Benchmarks.

name	*description*
cccp	GNU C preprocessor
cmp	compare files
compress	compress files
ditroff	text formatter and typesetter
eqn	typeset mathematical formulas for troff
eqntott	Boolean minimization
espresso	Boolean minimization
grep	string search
lex	lexical analysis program generator
li	Lisp interpreter
mpla	pla generator
pic	format pictures for troff
qsort	quick sort
tbl	format tables for troff
wc	word count
yacc	parsing program generator

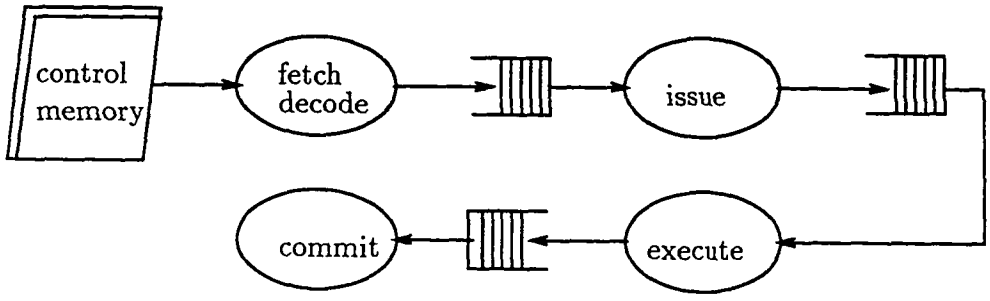


Figure 2.1 Behavior diagram of the execution hardware.

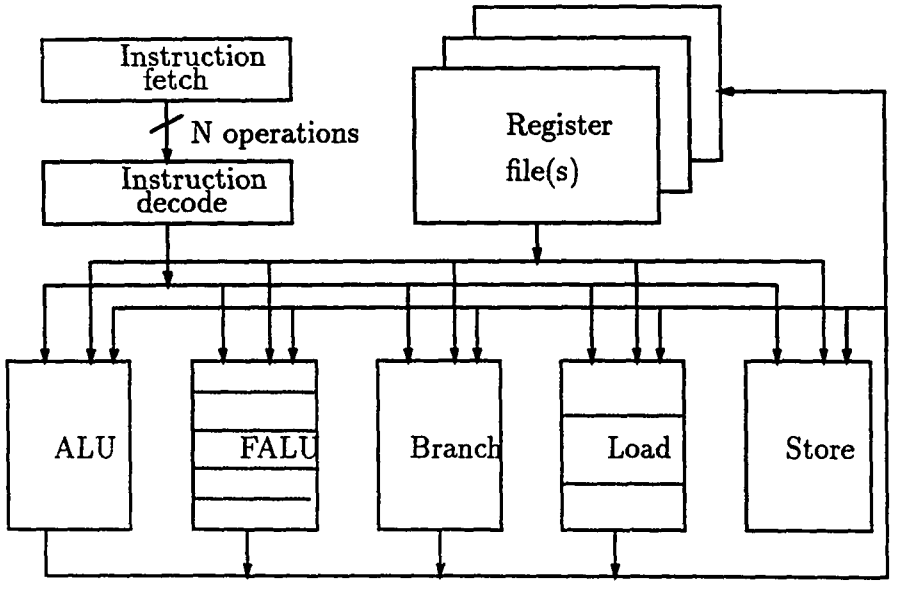


Figure 2.2 Block diagram of processor model.

CHAPTER 3

THE IMPACT ARCHITECTURAL FRAMEWORK

In this chapter, we describe a parameterized processor architecture that is fully supported by the IMPACT-I C compiler. The objective for developing this architecture is to provide a simple and cost-effective hardware design. In this dissertation, with our compiler support, we show experimentally that this simple architecture performs as well as the most aggressive architecture. The problem of allocating resource and scheduling operations are primarily treated in the compilation process. Unlike VLIW architectures in which the compiler is responsible for total control of the hardware, we require some hardware support to the compilation model.

The machine description language is described in Appendix A.

Figure 3.1 shows a top-level block diagram of the IMPACT processor architecture. The control unit manages a single instruction stream. In the ideal case, one instruction is fetched and decoded per cycle, and is forwarded to the function units. The control unit issues instructions to the function units in the order in which the instructions are fetched. The number of operations that can be packed into an instruction is an architectural parameter. In Figure 3.1, we assume that each instruction contains four operations. Let $op[i]$ denote the i_{th} operation in an instruction. There is an implicit precedence ordering between operations $op[i]$, $i = 0...3$. Because all operations

in an instruction obtain their source operands prior to execution, it is illegal to have anti-dependent operations in the same instruction. Even though we permit two or more operations in an instruction to modify the same destination register, the hardware ensures that only the last operation in the implicit ordering eventually writes the register. Therefore, output-dependence between operations in an instruction is automatically enforced by the hardware. The compiler schedules operations to ensure that there is no flow-dependence (essential-dependence) between operations in an instruction. Operations of an instruction are processed in lockstep within the control unit. After an instruction is fetched, all of its operations are decoded at the same time. If any one operation fails to obtain a source operand, the control unit stalls until a function unit returns the needed result back to the register file. Upon an instruction cache miss, the control unit stalls until the requested instruction is obtained from the secondary instruction memory. The control unit forms a rigid pipeline.

The output of the control unit is fed directly into several independent function units. Figure 3.1 shows four function unit groups. Each function unit group consists of a set of function units, such as a group for memory operations and a group for fixed-point arithmetic operations. The functionality of each function unit group is given to the compiler in a technology file. The compiler needs to schedule operations in such a way that $op[i]$ can always be executed by the i_{th} function unit group. To simplify the design, the resource contention problem will be ignored by providing fully pipelined function units and enough distribution buses to ensure that each function unit can accept a new operation per cycle. Except for the memory load operation latency, operation latencies are deterministic.

Hardware interlocking and register renaming are provided. Therefore, it is not necessary that operation latencies be deterministic. However, for simplicity, external events that may prolong operation latencies cause the instruction pipeline to stall.

3.1 Function Unit Resource

When several operations are issued to the execution unit per cycle, it is necessary to provide multiple function units. This includes multiple load, store, integer, and floating-point operations per cycle. In the worst case, all function units are replicated for each operation slot. For integer programs, we can speculate that the floating-point unit is not frequently used and does not need to be duplicated. In this case, we can issue at most one floating-point operation per cycle. In a later chapter, we present experimental data that show the effect of limiting some function unit resources.

3.2 Function Unit Delay

Concurrent execution of scalar code is often constrained by flow-dependencies between operations that form critical paths. For example, the condition code of a branch is often generated by first loading one or more memory variables into registers, and then executing an arithmetic operation on the registers. It is not always possible to find independent operations that can be executed after the memory load operations. The only way to alleviate this problem is to reduce the operation latency of certain operations that often appear in critical paths, such as memory load operations. Other long latency operations include integer multiply, integer divide, and floating-point operations. Operation latency can be reduced by improving the circuit design and by providing a bypass circuit. The problem with long operation latency can also be alleviated by using aggressive code motion that computes operations on the critical paths as early as possible. In a later chapter, we show that memory load operations often appear on critical paths. We recommend that the operation latency of memory operations be kept as small as possible.

3.3 Branch Handling

Increasing the instruction fetch bandwidth alone is not an adequate solution to the problem of instruction supply. Hardware support such as a branch target buffer or

squashing branch must be provided to maintain a contiguous instruction stream when branch operations are frequently taken. We have developed *inline target insertion*, a variant of the squashing branch scheme [McFarling 86]. Inline target insertion allows scheduling multiple branch operations into an instruction word, and allows filling branch slots with branch operations. Inline target insertion requires the compiler to decide for each branch operation whether it is likely to be taken and whether branch slots should be allocated for it. Formal proofs of its correctness are provided in Chapter 9.

Figure 3.2 shows the branch architecture. After an instruction has been decoded and all source operands have been obtained, the integer ALU units compute the branch condition codes and branch target addresses. The fetch pipeline and the first stage of the function units form a closed loop. If any one stage stalls, all stages in the closed loop stall. The semantics of the branch operation in an instruction can be defined as follows:

```
for (i=0..N-1) {  # for issue bandwidth = N operations
  if (op[i] is a branch) {
    if (op[i] is taken) {
      if (op[i] is incorrectly predicted)
        flush the fetch pipeline;
      squash(op[i+1..N-1]);
      pc = target(op[i]);
    } else {
      if (op[i] is incorrectly predicted)
        flush the fetch pipeline;
      pc = pc + 1;
    }
  }
}
```

According to inline target insertion, there can be at most one branch operation that is predicted taken. If there is a predicted-taken branch operation in an instruction, branch slots are allocated immediately after the instruction and are filled with the first

few instructions of the target path. The hardware must implement the above sequential algorithm in a parallel form, exploiting parallel datapaths in VLSI. The algorithm specifies that the first taken branch squashs later operations in the implicit operation ordering of an instruction. If an instruction contains an incorrectly predicted branch, subsequent instructions in the instruction fetch pipeline are removed. Therefore, the cost of a mis-predicted branch is the time to refill the instruction fetch pipeline.

3.4 Register Interlocking

The decode stage assigns a unique instruction tag (an integer field) to each instruction. Dynamic register renaming can be implemented by attaching an instruction tag field and a Boolean valid bit to each register. If an instruction intends to write a register, it clears the valid bit of the register and writes the instruction tag into the instruction tag field of the register. Because an operation may be squashed by a taken branch operation, the write permission must be reserved after branch operations have been verified.

The valid bit of a register is zero if the value of that register is unknown and will be defined by an instruction in execution. An instruction can move to the execution unit if the valid bits of all of its register source operands are set.[1]

It is desirable to allow several operations in an instruction to write to the same register. For example,

```
    r1 = r2;
    beq (r2, 0) to L1;
    r1 = r3;
L1:
```

can be scheduled into one instruction After all branches have been verified, the last operation in the implicit operation ordering of an instruction is allowed to write the register; previous writes are squashed.

[1] A load/store architecture is assumed. Therefore, we do not consider memory source operands.

3.5 Lockstep Execution

The property that all source operands of an instruction must be obtained prior to issuing the instruction to the function units enables the following code optimization.

```
for (i=N; i>0; i--)
    .....
```

is translated to

```
    i = N;
L0:
    .....
    i--;
    if (i>0) goto L0;
L1:
```

A flow-dependence exists between the last two operations of the inner loop. The code scheduler can transform the code into a parallel form without considering the flow-dependence.

```
    i = N;
L0:
    .....
    i--;  if (i>1) goto L0;
L1:
```

Because the two operations obtain their source operands before they are issued to the execution hardware, the branch condition expression can be adjusted to use the old value of the variable i.

Lockstep execution is valid only for VLIW architectures and not for superscalar architectures. For superscalar processors, the compiler does not know what operations the hardware will issue to the execution unit in a cycle. The IMPACT-I C compiler can generate code for architectures with and without lockstep execution.

3.6 Special Operations

Some flow-dependencies can be eliminated when two interdependent operations can be combined into a compound operation. For example, when manipulating integer arrays, the following code segment to load the value of an array element into a register is often seen:

```
(mul (r0) (index 4))     # r0 = index * sizeof(int)
(ld_i (r1) (base r0))    # r1 = memory[base + r0]
```

There exists a flow-dependency between the two operations. One cycle can be saved if a special memory load operation is provided that automatically multiplies a source operand by 4. Similar extension can be made for memory store operations. Multiplication by 4 can be implemented as a logical shift of a two's complement number by 2 bit positions to the left. The additional delay is at most that of a multiplexer and is not likely to significantly prolong the machine cycle time.

Similarly, some control-dependencies can be eliminated when two interdependent operations can be combined into a compound operation. For example,

```
    if (r0<>0) goto L1;
    r1 = 5;
L1:
```

can be converted into a guarded operation $((r1 = 5)if(r0 <> 0))$.

Because the focus of this research is on general-purpose computation, we do not apply this optimization.

3.7 Silent Exceptions

For each operation code, the IMPACT processor architecture provides a functionally equivalent operation code that signals neither exception nor trap. Using the nontrapping operation code, the code scheduler may move division and memory load operations from

below to above branch operations. When a division operation divides by zero, the result is not specified. When a memory load operation causes a memory access violation, the result is also unspecified. If a load operation that has been moved from below to above a branch operation causes a page fault, the page fault can be handled in the usual way. The working set of the program may be increased because of the additional page faults. However, we do not expect these infrequent page faults to degrade the overall system performance significantly.

We will show in Chapter 8 that nontrapping operations provide substantial speedup.

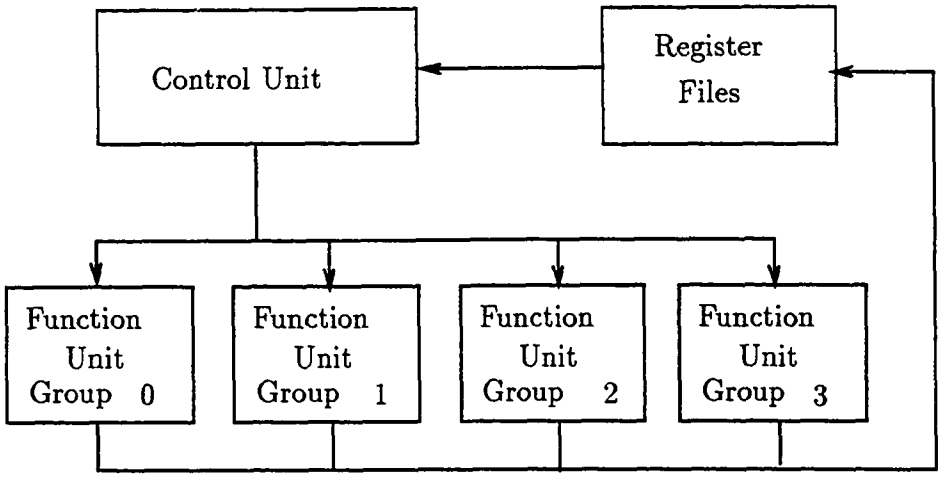


Figure 3.1 Top-level block diagram of the processor architecture.

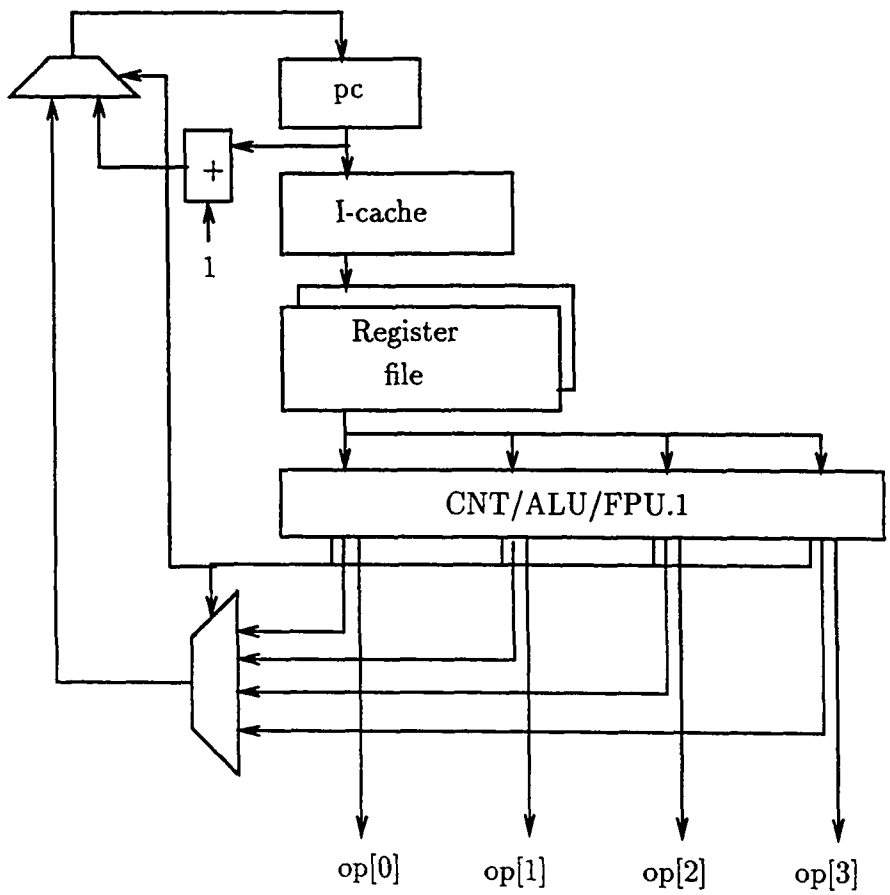


Figure 3.2 Branch architecture.

CHAPTER 4

ESSENTIAL FEATURES Of THE IMPACT-I C COMPILER

Figure 4.1 shows a block diagram of the IMPACT-I C compiler. The compiler supports the full C programming language. The compiler has a portable frontend that performs lexical, syntactic, and semantic analyses. The organization of the IMPACT-I C compiler is mainly traditional. However, there are three features that distinguishs the IMPACT-I C compiler from a typical commercial C compiler. This chapter will address each of the three main features.

4.1 Open Compiler Architecture

An open compiler architecture simplifies the task of adding components to and deleting components from the compiler.

(1) A compiler is an evolving program. Reducing the time to test and verify the performance of new code optimizations enables us to transfer technology to end users more quickly.

(2) Because of the large number of components that are required to make a compiler functional, it is not likely that all components can be implemented using the best technology in the beginning. Therefore, it is desirable to be able to replace old components by better replacement parts.

(3) With the ability to delete (or at least disable) some components, fault identification can be accomplished with less effort. This reduces the time that is needed to introduce a new code optimization technique. By restoring existing code optimizations one by one, one can identify coupling faults between a new component and old components.

To achieve an open compiler architecture, it is best to make each compiler component independent of others, by reducing the number of implicit information channels among components. Our approach to achieving an open compiler architecture is to organize compiler components around two major intermediate codes. An intermediate code is a program representation which is easier for the compiler to operate on than the original source code. For example, a three-address intermediate code can be easily operated on by code optimizers. An example of a three-address intermediate code can be found in [Aho 86]. Each intermediate code has a well-defined file representation and internal data structure representation. The basic functions to read in and write out the intermediate code from and to external files have been implemented as standard library functions. In addition, there are functions that check the integrity of the internal data structures. Primitive functions for manipulating the internal data structures have also been provided. These library functions have been tested carefully to make each intermediate code a comfortable environment for component designers. All major compiler components are implemented as tools. Each tool is connected to an intermediate code environment. When a tool is invoked, it takes input from the intermediate code internal data structure, allocates some private data structures if necessary, performs some computations, and finally, updates the intermediate code internal data structure. After the invocation of a tool, the library function that checks the integrity of the intermediate code data structure can be invoked to detect bad components. All information sharing between tools is through the intermediate data structure.

4.2 Two-level Intermediate Code

The IMPACT-I C compiler uses two levels of intermediate code representation. The reason for using two representations is that some program analysis and code optimizations require source code information and others require simple intermediate code representation. The high-level intermediate representation is called Hcode. Hcode representation preserves complete source code information, including data structure definitions, variable definitions, and function definitions. The low-level intermediate representation is called Lcode. Lcode representation uses a very simple RISC-like instruction set. All variable accesses are converted into operations on registers and memory locations. All function calls are converted into explicit operation sequences to pass parameters, to jump to subroutine, and to store the result.

Some program analyses and code optimizations can be more easily implemented using Hcode. For example, memory disambiguation requires source code information about data structure declarations. Memory accesses to different C data structures (except the *union* data structure in C) and fields can be considered to be independent memory operations. Memory accesses to different variable classes, e.g., static and global, can also be considered to be independent memory operations. Such information cannot be derived from assembly language such as the Lcode representation. For another example, function inline expansion can be easily performed in Hcode by replacing a call statement by the body of the function. On the other hand, identifying all operations that are part of a calling sequence (after code motion) is already a difficult task, discounting the actual expansion steps, at the Lcode level.

Traditional code optimizations work on simple three-address forms. Therefore, Lcode is a better candidate for implementing traditional code optimizations. Machine-dependent code optimizations such as constant preloading and code scheduling require accurate mapping from the intermediate form to the target assembly or machine language. Therefore, most machine-dependent code optimizations belong to the Lcode level.

An alternative to using two levels of intermediate forms is to use a single intermediate form whose complexity is somewhere between the Hcode and the Lcode. For example, parameter passing can be represented by push-args and pop-args psuedo operations. Most existing compilers have resorted to using one intermediate form. However, we like to specialize the functionalities of tools surrounding the intermediate forms, by making information as explicit as possible.

4.2.1 The Hcode environment

Figure 4.2 shows a block diagram of the Hcode environment. Hcode has a well-defined text representation, which is also a high-level program language. The semantic and expressive power of Hcode is the same as the C programming language, for Hcode can preserve all source code information of a C program. Hcode text format, however, uses an Lisp-like grammar, which is easy to parse and to generate automatically. Hcode also has a well-defined internal data structure representation. The functions to convert between the text and the internal representations have been provided.

Three major tools have been constructed. The first tool is an execution profiler that collects run-time information about the source program. The second tool is a profile-based interfile function inline expander. The third tool is a profile-based instruction placement algorithm. Each of the three tools directly modifies the Hcode data structure, and the tools do not communicate with each other.

The Hcode data structure can be written out to external files in three different styles. The first style is the Hcode text representation, the second style is the C programming language, and the third style is the next level of intermediate form, Lcode. The Hcode output style has greatly assisted in the debugging of Hcode tools, and is essential for connecting Hcode tools that cannot be accomplished in a single pass. The C output style has allowed us to implement a machine-independent profiler and also to debug the Hcode environment on any machine. Generating Lcode is a machine-dependent process: a set of machine specific functions is written for each target machine. Machine-dependent parameters include the sizes and alignment requirements of various data types, the layout

of data structures, the parameter passing convention, the activation stack convention, and global/local variable space allocation and placement schemes.

The ability to translate Hcode into C is important for three reasons. First, the Hcode representation preserves all information in the original C source code. Code optimizations based on Hcode can exploit all source code level knowledge. Second, Hcode optimizations can be debugged by translating Hcode into C, and compiling the C program using a stable compiler. Third, some optimizations, such as function inline expansion, can be easily done at the Hcode level. After code optimizations have been applied at the Hcode level, the Hcode intermediate form is translated to the Lcode intermediate form.

4.2.2 The Lcode environment

Figure 4.3 shows the block diagram of the Lcode environment. Like Hcode, Lcode has a well-defined text representation and an internal data structure representation. Functions for conversion between the external and the internal formats, for manipulating the internal format, and for checking the correctness of the internal format have been provided to tool designers. Lcode tools include a set of local code optimization functions, a set of global code optimization functions, and a set of machine-dependent optimizations (register allocation, constant preloading, code scheduling).

After Lcode transformations, the result can be written to external files in Lcode text representation. When compiling for a specific machine, the corresponding code generator can be invoked. We have constructed code generators for MIPS R2000 [Kane 87], SPARC [Sparc 87], Intel 860 [Intel 89], and AMD29K [Amd].

The most important features of the Lcode intermediate form can be summarized as follows:

(1) It has infinite number of virtual registers.

(2) It assumes a load/store architecture. The only addressing modes are constants and register operands.

(3) It supports basic integer, single-precision, and double-precision arithmetic operations.

(4) It supports memory operations for unsigned characters, signed characters, unsigned short integers, signed short integers, integers, single-precision floating-point and double-precision floating-point data types.

(5) It supports a spectrum of branch architectures.

(6) It provides a minimal set of synchronization operations.

Hcode and Lcode documents are available as internal reports. Because they are long, they will not be included in this dissertation. Appendix B shows some Hcode and Lcode files.

4.3 Profiling

Mapping a computation to a hardware with limited resources requires allocating resources to the most important code section first. For example, the most frequently used variables should be kept in registers. The traditional approach is to identify loop structures and assume that the code section within a loop body is most important. However, a better approach is to implement a profiler in the compilation process. Using a profiler to obtain the run-time behavior of a source program before code optimization has been reported to be very effective [McFarling 86], [Wall 86], [Wall 88], [Chang 89a], [Chang 89b], [Chang 89c], [Hwu 89a], [Hwu 89b], [Hwu 89c]. Integrating a profiler with a compiler has been shown to be feasible. More research work is needed in applying the profile information in various code optimization techniques. In this research, we have implemented a profile-based function inline expansion algorithm, a profile-based branch prediction algorithm, a profile-based instruction placement algorithm, profile-based global code optimizations, and a profile-based code generation algorithm. Detailed descriptions of these techniques will be presented in Chapters 5 and 6.

Figure 4.4 shows a block diagram of the integrated profiler. To profile a C program, the IMPACT-I C profiler converts the program into a functionally equivalent C program with all the probes inserted. This new C program can then be compiled by the C compilers of different systems and executed on these systems to collect profile information in parallel.

Portability is an important issue in the IMPACT-I C compiler design because it is an experimental compiler for many possible processor configurations and different instruction sets. Because the IMPACT-I C compiler will be ported to various systems, the compiler and profiler interface must also be completely system-independent.

The IMPACT-I profiler is system-independent for the following reasons.

(1) The profiler itself can execute on different systems.

(2) The program with profiling probes can execute on different systems.

(3) The profile information accumulated on a system can be directly used by the IMPACT-I C compiler and architecture design tools running on a very different system.

(4) The profile information accumulated on an existing system can be used to guide the architecture design and code optimization for a nonexisting system.

One problem we have encountered is that the library functions of different operating systems are different and are not portable. This prevents the library functions from being profiled if the user insists on machine-independent profiling. On the hand, if the user is willing to accept system-dependent profiling, then the library functions can also be profiled along with the user application program.

4.3.1 Definition of a weighted control graph

To make the profile information useful to the compiler, the profile information must be presented in a structure which can be easily understood by the compiler. The *weighted*

control graph defined below is a structure through which the profile information can be presented to the compiler.

A *control graph* is a directed graph in which every node is a basic block and every arc is a branch path between two basic blocks. There is an arc from node A to node B if and only if the final branch operation in basic block A can potentially cause a control flow to basic block B. The *node weight* is the average execution count of the corresponding basic block over many inputs. The *arc weight* is the average number of times the corresponding branch path is taken over many inputs. A *weighted control graph* is a control graph in which all of the nodes and arcs are labeled with their weights.

Let us assume that there are two basic blocks which are uniquely labeled A and B, and are connected by a branch path from A to B. The arc (A,B) is said to be an *outgoing arc* of node A, and an *incoming arc* of node B. Node A is said to be the *source*, and node B is the *destination* of the arc (A,B). A node may have several incoming and outgoing arcs.

If we further assume that node A has been executed 50, 60, and 40 times in three separate runs of the program, the node weight of A is 50, the average of the three runs. If in the same three runs the arc (A,B) has been taken 40, 45, and 35 times, respectively, the arc weight of (A,B) is 40, the average of the three runs. Then the probability of the arc (A,B) will be taken, given that the program control is already in node A, and can be estimated to be 40/50 (80%).

4.3.2 Construction of a weighted control graph

There are 8 major steps to generate profile information.

(1) The compiler frontend converts the C source program into a control graph.

(2) Constant folding and (block-level) dead code removal eliminate unreachable blocks from the control graph. Jump optimizations merge basic blocks which are connected by unconditional branch operations.

(3) The compiler inserts probes into the control graph.

(4) The compiler converts the control graph into a functionally equivalent C program.

(5) The functionally equivalent C program with probes is then compiled and installed into the system.

(6) The program is run many times with realistic input data. Each run produces a profile file. All profile files are summarized into a single profile file.

(7) The compiler constructs an identical control graph by repeating step 1 and step 2, or by saving the control graph from step 2. Then the compiler asks the profiler to supply the node and arc weight information. A weighted control graph is formed by assigning weights to the nodes and arcs of the control graph.

(8) A weight consistency check program verifies that all weights have been gathered and assigned consistently.

4.3.3 Probe insertion

After jump optimizations, probes are placed at various places of the control graph. First, the compiler assigns a unique identifier to each basic block in the program. For each basic block, the compiler inserts a probe to determine basic block execution count and the transition count. To derive the transition count, the profiler has to keep track of the previous basic block during execution. A state variable *last-tag* is initially set to 0 and is modified to contain the identifier of the previous basic block during execution of the program. A probe is inserted in every basic block.

```
static int last-tag = 0;
basic-block-probe(current-id) {
    increment-node-weight(current-id);
    increment-arc-weight(last-tag, current-id);
    last-tag = current-id;
}
```

```
function-entry-probe(function-id) {
    push-tag(last_tag);
    last-tag = special ENTRY tag for function (function-id);
}
function-exit-probe() {
    last-tag = pop-tag();
}
```

A stack structure, which we call *tag-stack*, is provided to store and recover the *last-tag* value across function calls.[1] In the beginning of a function, a probe is inserted to push the *last-tag* value onto the stack. Right before returning from a function, a different probe is inserted to move the top entry of the *tag-stack* back to *last-tag.*

The C programming language contains two special library functions, setjmp() and longjmp(), which must be handled differently from other functions. The compiler has to recognize these two functions and replace setjmp() with a probe which marks the top of the *tag-stack* and longjmp() with another probe to return the *tag-stack* to the marked position. Setjmp() and longjmp() are called only indirectly from the two special probes.

4.3.4 Input data

The profile code can be compiled and installed in a public system. In our case, we have a university research environment in which most jobs are CPU intensive CAD programs, text editing and formatting programs, and program compilations. Inputs from various users in selected computer environments can be profiled and averaged. Inputs come from various people and represent the general system usage.

[1] In C, a procedure is a function whose return type is *void*. Therefore, we do not distinguish between a function call and a procedure invocation.

4.3.5 Profile data representation

A node weight attribute and a list of outgoing arc weight attributes are attached to each control graph node.

```
struct arc {
    int destination;
    double weight;
    struct arc *next;
} ;
struct node{
    double weight;
    struct arc *outgoing-arcs;
} NodeTable[MAX-NUMBER-OF-NODES];
```

The *destination* field of the arc structure specifies the unique node identification number of the destination block. The *weight* field of the arc structure is the number of times the arc has been taken. The *next* field of the arc structure is a pointer to the next outgoing arc. The *weight* field of the node structure is the number of times the node has been visited. The *outgoing-arcs* field stores a pointer to a linked list of arc elements whose weights are nonzero. This data structure is maintained and constantly updated by the monitor probes inserted in the profile code. Memory spaces for storing the node and arc structures are allocated statically by declaring two large arrays which are appended to the user program that is being monitored.

To maintain the profile information over many runs, the user specifies a file in which the profile information should be stored. At the end of a profile run, the profiler first reads in the accumulated information stored in the data file, adds in the new information, and then stores the final data back to the data file.

4.3.6 Profile data maintenance

The number of profile runs is also stored in the data file. Each run of the program generates a new set of node and arc weights. The profiler adjusts the profile data with the statements: *W.permanent = (W.permanent * N/(N+1)) + (W.new / (N+1)); N=N+1*, where N is the number of times the program has been profiled, *W.new* is the new node (arc) weight, and *W.permanent* is the accumulated node (arc) weight.

To combine two accumulated sets, the profiler adjusts the profile data according to *W.total = (W.N * N / (N+M)) + (W.M * M / (N+M)); total=N+M*, where N and M are the number of runs made by the two systems, respectively. With these flexible rules, we can concurrently profile a program on a network of heterogeneous machines and combine the results. The combined profile data can then be used by the IMPACT-I C compiler and the IMPACT-I architecture design tools executing on different machines in the network.

4.3.7 Reconstruction of control graph

The IMPACT-I profiler and the IMPACT-I C compiler share the same frontend. Therefore, they share a consistent view in naming the basic blocks and control transfers. To generate the profile information, the profiler labels the node and arc weights by their corresponding unique basic block identifiers. To use the profile information, the compiler constructs an identical control graph and uses the unique identifiers to assign weights to the nodes and arcs. After weight assignment, the compiler generates the Hcode intermediate code. The control graph can be further optimized, and the node and arc weights are also modified consistently.

4.3.8 Node and arc weight assignment

The probe and query functions have been renamed here to simplify our discussion. The actual names in the real implementation are long and complex in order to avoid declaration conflicts with existing user and system defined functions and variables.

To access the profile information, the compiler calls a set of functions which are defined by the profiler.

```
double NodeWeight(id);
double ArcWeight(src-bb-id, dest-bb-id);
```

The NodeWeight() function takes one argument which identifies a basic block and returns the weight associated to the basic block. The ArcWeight() function takes two arguments. The first argument specifies the source of a control arc. The second argument specifies the destination of a control arc.

Any arc can be uniquely identified by its two terminal basic blocks. The ArcWeight() function returns the weight of a specified control arc.

A simple algorithm is used to assign node and arc weights. It is combined into the compiler frontend processing and, therefore, does not require a separate pass.

```
WeightAssignment(P) {
    for (all nodes Ni of P) {
        Ni.weight = NodeWeight(Ni.id);
        for (all outgoing arcs Aj of Ni) {
            D = destination of Aj;
            Aj.weight = ArcWeight(N.id, D.id);
        }
    }
}
```

4.3.9 Weight consistency verification

Since a node can be entered only from one of its incoming arcs and exit only through one of its outgoing arcs, the *node weight = sum of the weights of all incoming arcs = sum of the weights of all outgoing arcs.* The control graph of a large integer program usually consists of thousands of nodes and arcs. The weight consistency check is a nice way to detect errors in the profile data. This check function will detect most errors due

to nonunique basic block ID assignment or inconsistent basic block ID assignment due to source code change.

4.3.10 Separate compilation

Separate compilation can not be done when it is necessary to assign each function (or basic block) a unique identifier. However, the labeling process does not require the entire program to be present at once, and, thus, one can still keep a program across a large number of files. The IMPACT-I C compiler reads in files in an order that is specified by the user and labels each basic block with a unique integer number. The particular order specified by the user is recorded in a log file maintained by the IMPACT-I C compiler. The recorded file sequence is used again by the compiler to construct the control graph after the profiling process.

Except for providing the initial file sequence, the user does not need to know how basic blocks are labeled, how the probes are inserted, and how the profile information is mapped to the source code.

4.3.11 Lcode profiling

Except for the function inline expansion and the instruction placement optimization, code optimizations are performed at the Lcode level. Therefore, instead of an Hcode-level profiler, an Lcode-level profiler can effectively guide most code optimizations.

Another reason for constructing an Lcode-level profiler is that some code optimizations can decrease the accuracy of the profile information. Although approximate profile information is generally sufficient for guiding later code optimizations, it is not sufficient to derive performance statistics.

We have implemented an Lcode-level profiler that maintains weighted control graphs as described in the above sections. The implementation involves changing the code generator to insert additional code to measure the execution frequencies of every basic block and the direction of every branch operation.

4.3.12 Profile-based code optimization

Most traditional code optimizations can be easily modified to take advantage of the profile information. For example, classical loop optimizations such as induction variable elimination and loop unrolling may introduce extra operations in a loop preheader in order to set up a more efficient loop body. These optimizations may degrade performance if the number of loop iterations is very small. The average number of loop iterations can be derived from the weighted control graph. For another example, the compiler should allocate the most frequently accessed variables to registers. Static program analysis cannot distinguish a loop that is never executed from one that is frequently executed. On the other hand, execution and access frequencies can be easily derived from the weighted control graph. In addition to extending traditional code optimizations to use the profile information, we have designed more aggressive code optimizations that customize the most frequently executed program regions and expand the scope of code scheduling. Detailed descriptions and analyses of these code optimization techniques are provided in Chapters 5, 6, and 7.

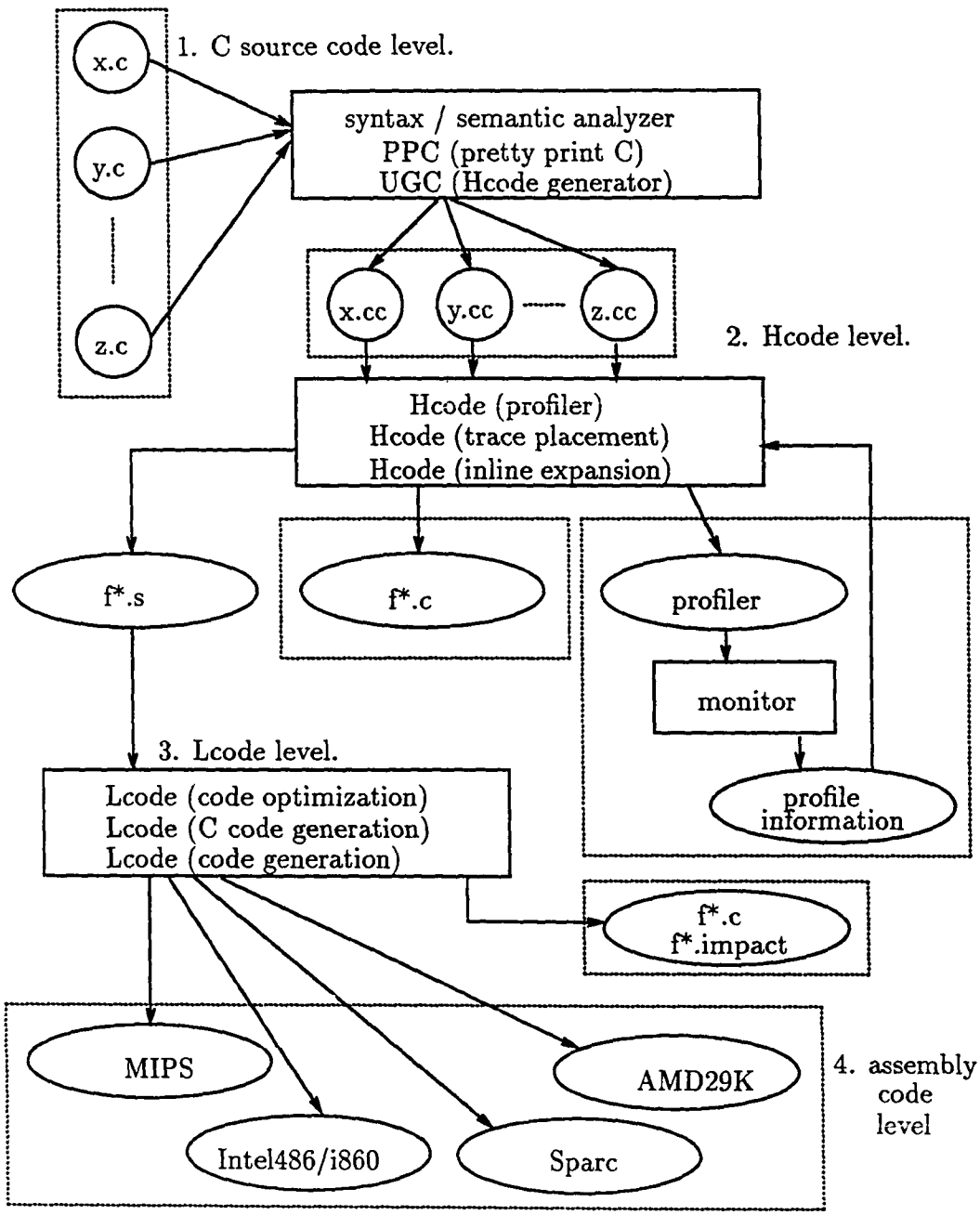


Figure 4.1 Framework

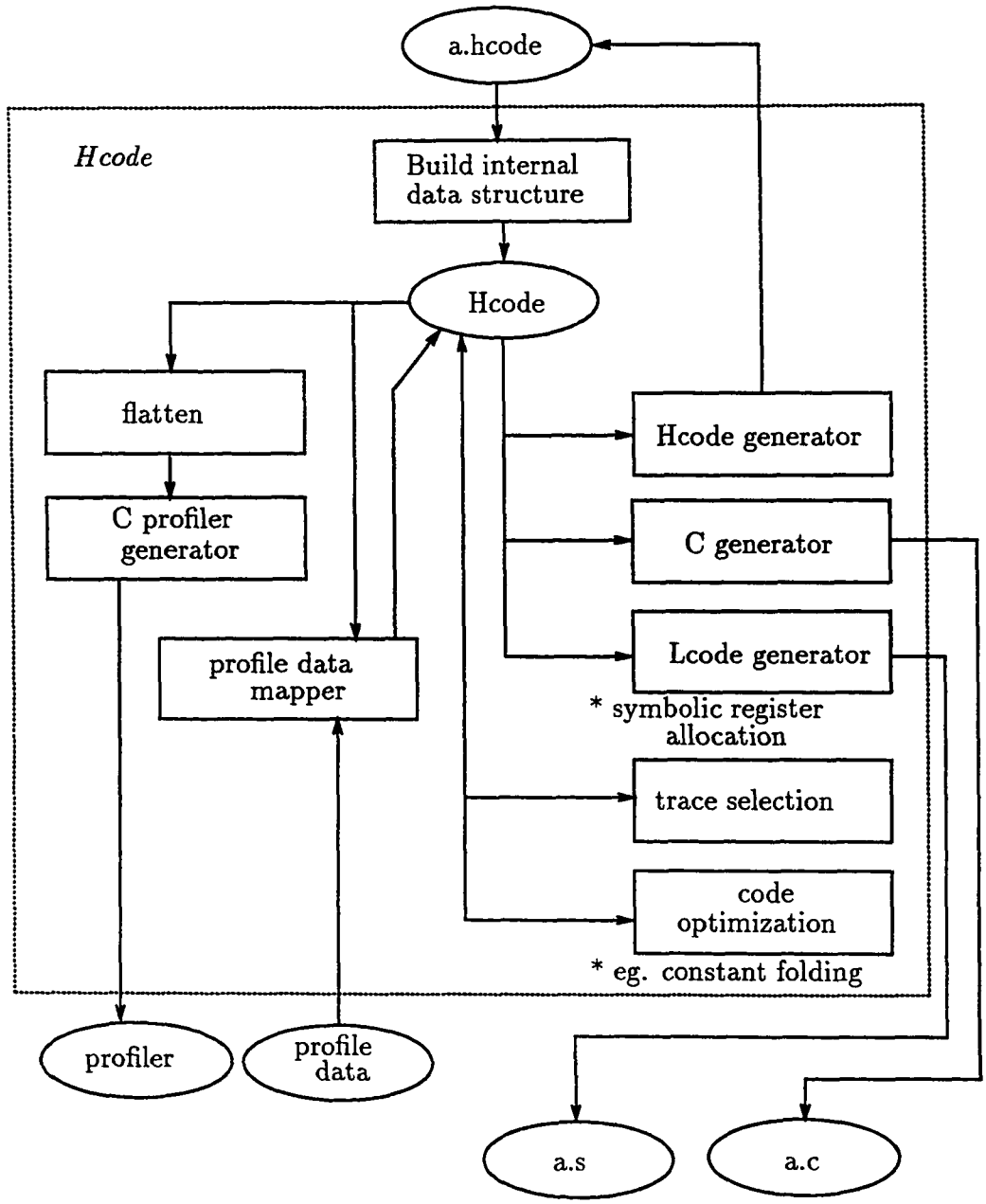


Figure 4.2 Hcode

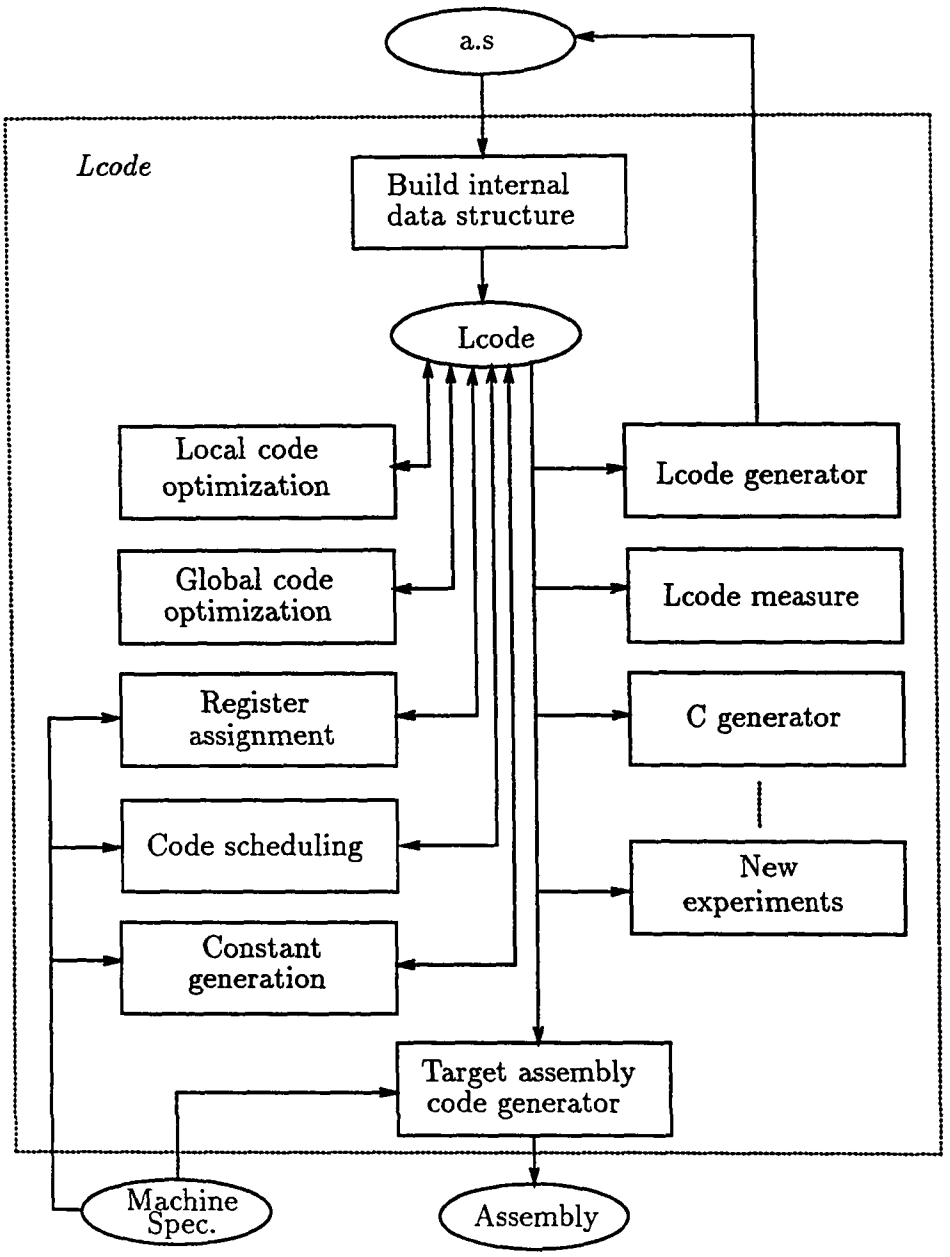


Figure 4.3 Lcode

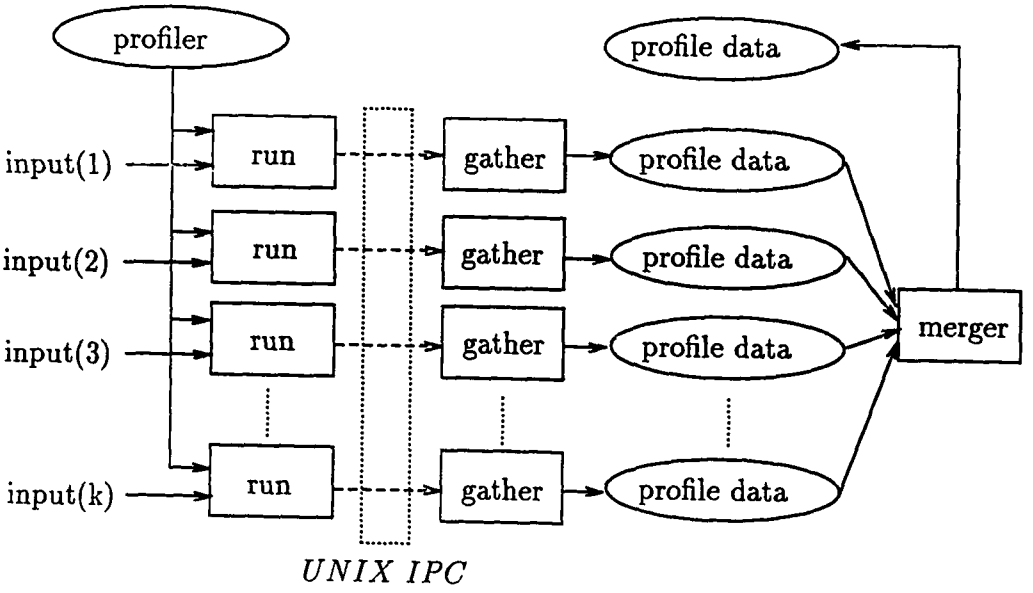


Figure 4.4 Profiler

CHAPTER 5

MACHINE-INDEPENDENT CODE OPTIMIZATION

The set of code optimizations in the IMPACT-I C compiler can be partitioned into three major groups. The first group is a set of code optimizations that are applicable to all scalar/multiple-instruction-issue processors. The objectives of these code optimizations are to make the code more efficient by eliminating redundant operations and by moving operations from frequently executed program regions to infrequently executed program regions. In processor architectural studies, it is important to evaluate performance with highly optimized benchmarks, because redundant operations may show deceptive parallelism. Machine-independent code optimizations are described in Chapter 5. The second group is a set of code optimizations that are machine-dependent and whose objectives are to exploit machine features such as a register window. In processor architectural studies, it is important to apply machine-dependent optimizations to the benchmarks that are being evaluated, because the true performance of a processor architecture can be shown only if the benchmarks are optimized for that processor architecture. Machine-dependent code optimizations are presented in Chapter 6. The third group is a set of code transformations that enlarge the scope of code scheduling and reduce some dependencies between operations to improve the performance of code scheduling. These transformations are specific to processor architectures that can execute many operations concurrently, such as multiple-instruction-issue architectures. These code transformations are presented in Chapter 7.

The decision components of many code optimizations are interdependent and are customized for different target machines. To explain why a code optimization should be applied for a MIPS R2000 machine and not for a SPARC machine, we need to describe the architectural features of the MIPS R2000 and SPARC architectures. It is not possible to describe in a dissertation the implementation issues and decisions of the code optimizations that we have implemented in the IMPACT-I C compiler to such a point that the reader can reproduce the implementation. Therefore, we will provide the reader with only an intuitive understanding of the code optimization functions. We will describe few code optimizations, e.g., inline expansion, in detail to show the reader how to design and implement a code optimization. If the reader is interested in reproducing the results, the IMPACT-I C compiler can be obtained through a University of Illinois license.

In this chapter, we describe the machine-independent code optimizations that have been included in the IMPACT-I C compiler. In the Hcode level, we have implemented function inline expansion, instruction placement, and control flow optimization. In the Lcode level, we have implemented many classical code optimizations and extended them to trace-based code optimizations.

In Section 5.1, we describe the function inline expansion technique, which was presented in [Hwu 89c]. In Section 5.2, we describe the instruction placement technique, which was presented in [Hwu 89a]. In Section 5.3, we describe the branch optimizations, which we presented in [Chang 89c]. In Section 5.4, we describe a large number of classic code optimizations that we have implemented in the IMPACT-I C compiler. In Section 5.5, we present an extension to classic code optimizations to use profile information. Formulations and detailed discussions of these code optimizations can be found in [Chang 91b].

5.1 Function Inline Expansion

5.1.1 Introduction

Large computing tasks are often divided into many smaller subtasks which can be more easily developed and understood. Function definition and invocation in high level languages provide a natural means to define and coordinate subtasks to perform the original task. Structured programming techniques therefore encourage the use of functions. Unfortunately, function invocation disrupts compile-time code optimizations such as register allocation, code compaction, common subexpression elimination, constant propagation, copy propagation, and dead code removal. The decreased effectiveness of these optimization techniques increases memory accesses, decreases pipeline efficiency, and increases redundant computation.

Emer and Clark reported, for a composite VAX workload, 4.5% of all dynamic instructions are function calls and returns [Emer 84]. If we assume equal numbers of call and return instructions, the above number indicates that there is a function call instruction for every 44 instructions executed. Eickemeyer and Patel reported a dynamic call frequency of one out of every 27 to 130 VAX instructions [Eickenmeyer 88]. Gross and Hennessy reported a dynamic call frequency of one out of every 25 to 50 MIPS instructions [Gross 82]. Berkeley RISC researchers have reported that a function call is the most costly source language statements [Patterson 82]. All these previous results argue for an effective approach to reducing function call costs.

Some recent processors provide hardware support for minimizing the extra memory accesses due to function calls. For example, the Berkeley RISC processors provide overlapping register windows to reduce the number of memory accesses required to save/restore registers and to pass parameters [Patterson 82]. Another example is the CRISP processor that uses stack buffers to capture the memory accesses to local variables so that register allocation crossing function calls can be simulated in hardware [Ditzel 87]. These hardware approaches consume a significant amount of hardware, stretch the processor cycle time, and provide little assistance for enlarging the scope of compiler code optimization.

In the software realm, interprocedural register allocation schemes reduce the register save/restore cost across function call boundaries [Chow 88]. Callers and callees can also communicate parameters and results through a small number of registers [Sherburne 83]. Wall has shown that link-time register allocation that is guided by profile information is comparable in performance to hardware register window schemes [Wall 86], [Wall 88]. Interprocedural analysis is effective in reducing the negative effects of function calls on the code scheduling and other code optimization techniques [Allen 74], [Allen 76], [Hecht 75], [Barth 78], [Li 88]. These software remedies assume that frequent function calls can not be avoided. If most of the function calls can be eliminated, these complicated remedies would be unnecessary.

Inline function expansion (or simply *inlining*) replaces a function call with the function body. Inline function expansion removes the function call/return costs and provides enlarged and specialized functions to the code optimizers. With automatic inline function expansion, the advantages of using functions in software development remain, and the costs are reduced. In a recent study, Allen and Johnson identified inline expansion as an essential part of an optimizing C compiler. They gave a few critical reasons for implementing inline expansion. First, the variable aliasing problem becomes less onerous after inline expansion. Second, the code optimizer can work on the real effects of the callee after inlining. Third, inlining function calls contained in loops may increase the opportunities for vectorization [Allen 88]. Scheifler formulated the problem of inline expansion as a knapsack problem. An inline expander which takes advantage of run-time statistics in making inlining decisions was implemented for the CLU programming language. Experimental results, including function invocation reduction, execution time reduction, and code size expansion, were reported based on four programs written in CLU [Scheifler 77].

Several code improving techniques may be applicable after inline expansion. These include register allocation, code scheduling, common subexpression elimination, constant propagation, and dead code elimination. Richardson and Ganapathi have discussed the effect of inline expansion and code optimization across functions [Richardson 89].

Many optimizing compilers can perform inline expansion. For example, the IBM PL.8 compiler does inline expansion of all leaf-level functions [Auslander 82]. In the GNU C compiler, the programmers can use the keyword *inline* as a hint to the compiler for inline expanding function calls [Stallman 88]. In the MIPS C compiler, the compiler examines the code structure, e.g., loops, to choose the function calls for inline expansion [Chow 84]. Parafrase has an inline expander based on program structure analysis to increase the exposed program parallelism [Huson 82]. It should be noted that the careful use of the macro expansion and language preprocessing utilities has the same effect as inline expansion, when inline expansion decisions are made entirely by the programmers.

The IMPACT-I C compiler expands function calls to increase the effectiveness of compiler code optimization [Chang 88], [Hwu 89a], [Hwu 89b]. Inline expansion reduces the number of function calls so that hardware mechanisms such as register windows and stack buffers become unnecessary. For compiler code optimization, inline expansion serves to enlarge the scope of register allocation, code scheduling, and other optimizations. The IMPACT-I Profiler-to-C-Compiler interface allows the profile information to be automatically used by the IMPACT-I C Compiler. The inline expansion is based on execution profile information to ensure that only the important function calls are expanded. It is critical that the inputs used for executing the equivalent C program are representative. Therefore, this approach is more suitable for characterizing realistic programs for which representative inputs can be easily collected.

5.1.2 Critical issues

The basic idea of inline expansion is simple. Most of the difficulties are due to hazards, missing information, and reducing the compilation time. We have identified the following critical issues of inline expansion:

(1) Where should inline expansion be performed in the compilation process?

(2) What data structure should be employed to represent programs?

(3) How can hazards be avoided without incurring excessive compilation cost?

(4) How should the sequence of inlining be controlled to reduce compilation cost?

(5) What are the essential operations for inlining a function call?

(6) What are the desirable optimizations to reduce the undesirable effects of inline expansion?

In the following discussions, the term *function* corresponds to both procedures and functions defined in the programming languages such as C and Pascal. A *static function call site* (or simply *call site*) refers to a function invocation specified by the static program. A *function call* is the activity of invoking a particular function from a particular call site. If a call site can potentially invoke more than one function, the call site has more than one function call associated with it. This is usually due to the use of the call-through-pointer feature provided in some programming languages. The *caller* of a function call is the function which contains the call site of that function call. The *callee* of a function call is the function invoked by the function call. An example is shown in the C program below. There are three static function call sites in the main() function; two invoke function a() and one invokes function b(). Since each call site in this example invokes a unique function, each has only one function call associated with it. The caller of all the function calls is main() and the callees are a() and b().

```
main () {
int i, j;
    i = a() + b();
    ...
    j = a();
}
int a() { ... }
int b() { ... }
```

The first issue regarding inline function expansion is where inlining should be performed in the translation process. In most traditional program development environments, the source files of a program are separately compiled into their corresponding object files before being linked into an executable file (see Figure 5.1). The *compile time* is defined as the period of time in which the source files are independently translated into

object files. The *link time* is defined as the period of time in which the object files are combined into an executable file. Most of the optimizations are performed at compile time, whereas only a minimal amount of work to link the object files together is performed at link time. This simple two-stage translation paradigm is frequently referred to as the *separate compilation paradigm.*

The advantage of the separate compilation paradigm is that when one of the source files is modified, only the corresponding object file needs to be regenerated before linking the object files into the new executable file, leaving all the other object files intact. Because most of the translation work is performed at compile time, separate compilation greatly reduces the cost of program recompilation when only a small number of source files are modified. Therefore, the two-stage separate compilation paradigm is the most attractive for program development environments in which programs are frequently recompiled and usually a small number of source files are modified between each recompilation. There are special tools such as the UNIX make program to exploit this advantage.

Because the caller and callee functions may reside in different source files, inline function expansion and global optimization in general increase the coupling of the source files involved. Inline function expansion could be performed either at compile time or at link time. In either case, separate compilation is no longer possible to perform interfile inline expansion. The GNU C Compiler has a limited inline expansion feature which requires the caller and callee to be in the same source file for expansion. With this limitation, the simple separate compilation paradigm remains intact.

An extension to the separate compilation paradigm to allow inlining at compile time is illustrated in Figure 5.2. Performing inline function expansion at compile time provides four major advantages. First, inline function expansion enlarges the scope of code optimization and thus increases the opportunities for the optimization techniques such as constant propagation, common subexpression elimination, and dead code removal. Performing inline function expansion at the early stage of the compile time (before the code optimization steps) ensures that these code optimization steps benefit from inlin-

ing. Second, functions are often created as generic modules to be invoked for a variety of purposes. Inlining a function call places the body of the corresponding function into a specific invocation, which eliminates the need to cover the service required by the other callers. Therefore, constant propagation, constant folding, and dead code removal can be expected to reduce the code size expansion due to inlining. Third, by inlining the frequently executed function calls, inlining reduces the coupling between functions. This reduces the need for complex interprocedural analysis to support optimizations. Fourth, being applied before system-dependent code generation, inline expansion can be included in a portable frontend.

Performing inline function expansion at compile time requires the callee function source (or intermediate) code to be available when the caller is compiled. Note that the callee functions can reside in source files different from the caller's. As a result, the caller and callee source files can no longer be compiled independently. In addition, whenever a callee function is modified, both the callee and caller source files must be recompiled. This coupling between the caller and callee source files reduces the advantage of the two-step translation process.

In practice, some library functions are written in assembly languages; they are available only in the form of object files to be integrated with the user object files at link time. These library functions are not available for inline function expansion at compile time. One can argue, however, that since these library functions are already hand-optimized by the assembly programmers, they need not be involved in the inline function expansion whose major objective is to improve the effectiveness of compile-time optimizations. Dynamically linked libraries represent a step further in the direction of separating the library functions from the user programs invoking them. Since the dynamically linked library functions are not available for inline function expansion at all, they are not in the scope of this paper.

Inline function expansion can also be performed at link time. A translation process which employs inlining at link time is illustrated in Figure 5.3. Because all functions are available at link time, inline expansion can be naturally performed without sacrificing

separate compilation. The problem is that many compile-time optimizations should be performed after inline function expansion but can not be if the inline expansion is done at link time. There are two alternative solutions to this problem. One is to exclude the compile-time optimizations from the benefit of inline function expansion. This solution eliminates most of the advantages of the inline expansion: to enlarge the scope of compile-time optimizations. There are, however, important code restructuring techniques which can still benefit from link-time inline expansion [Hwu 89a].

The other solution is to defer the compile-time optimizations to link time, after the inline expansion is performed. In fact, register allocation has been performed at link time in Wall's work [Wall 86], [Wall 88]. The problem with this approach is that it eliminates most of the advantages of separate compilation. Since most of the optimizations are performed at link time, modifying a single source file incurs the cost of optimizing the entire program. Note that this is worse than performing inline expansion at the compile time where modifying a callee function source file requires only the recompilation and optimization of the corresponding callers. Also, performing optimization at link time often requires the symbol information to be passed from the compiler to the linker. This adds to the amount of information stored in the object files.

Inline function expansion is performed at compile time in the IMPACT-I C Compiler. Two major considerations led to this design decision. First, all of the compile-time optimizations can naturally benefit from inline expansion. These compile-time optimizations include register allocation, common subexpression elimination, constant propagation, constant folding, dead code removal, and program restructuring. Performing inline function expansion at compile time is compatible with most of the existing compiler structures. This makes it more feasible to incorporate the IMPACT-I inlining mechanism into the existing compilers.

Second, the inline expander in the IMPACT-I C Compiler is designed as a part of the program optimization mechanism for mature programs. It is designed for compiling production quality programs such as operating systems, text processing tools, engineering design tools, program development tools, and user interfaces. It is not recommended for

programs at their early stage of development. The general philosophy is that programs should be tuned only after they start working. This is consistent with the existing software development practice: make a program work before making it efficient.

The IMPACT-I C Compiler program optimization mechanism is designed as the last stage of the program tuning process, applied after the programmers have finished debugging and tuning at the coding level. Programs compiled with these optimizations are expected to run many times before they are revised; trading compilation time for execution efficiency is a desirable tradeoff. Therefore, separate compilation is not an important issue for the IMPACT-I inline expander; the primary goal is to have as many optimizations as possible to benefit from the inline expansion. This leaves us the choice of either performing inline expansion at the compile time or deferring the inline expansion and all the optimizations to link time. A major advantage of performing compile-time rather than link-time inline expansion is that it makes it possible to incorporate the inline expander into a system-independent compiler frontend. As a result, the IMPACT-I inline expansion is performed at compile time.

5.1.3 Program representation

The second issue regarding inline function expansion is what data structure should be employed to represent the program. To support efficient inlining, the data structure should have two characteristics. First, the data structure should conveniently capture the dynamic and static function calling behavior of the represented programs. Second, efficient algorithms should be available to construct and manipulate the data structure during the whole process of inline function expansion. Weighted call graphs, as described below, exhibit both desirable characteristics.

A weighted call graph captures the static and dynamic function call behavior of a program. A weighted call graph (a directed multigraph), $G = (N, E, main)$, is characterized by three major components: N is a set of nodes, E is a set of arcs, and *main* is the first node of the call graph. Each node in N is a function in the program and has associated with it a weight, which is the number of invocations of the function by all

callers. Each arc in E is a static function call in the program and has associated with it a weight, which is the execution count of the call. Finally, *main* is the first function executed in this program. The node weights and arc weights may be determined either by program structure analysis or by profiling.

An example of a weighted call graph is shown in Figure 5.4. There are eight functions in this example: main, A, B, C, D, E, F, and G. The weights of these functions are indicated beside the names of the functions. For example, the weights of functions A and E are 70 and 4, respectively. Each arc in the call graph represents a static function call whose weight gives its expected dynamic execution count in a run. For example, the main function calls G from two different static locations; one is expected to execute once and the other is expected to execute twice in a typical run.

Inlining a function call is equivalent to duplicating the callee node, absorbing the duplicated node into the caller node, eliminating the arc from the caller to the callee, and possibly creating some new arcs in the weighted call graph. For example, inlining B into D in Figure 5.4 involves duplicating B, absorbing the duplicated B into D, eliminating the arc going from D to B, and creating a new system call arc. The resulting call graph is shown in Figure 5.5.

Detecting recursion is equivalent to detecting cycles in the weighted call graph. For example, a recursion involving functions A and E in Figure 5.4 can be identified by detecting the cycle involving nodes A and E in the weighted call graph. Identifying functions which can never be reached during execution is equivalent to finding unreachable nodes from the *main* node. For example, Function B is no longer reachable from the main function after it is inline expanded into Function D (see Figure 5.5). This can be determined by identifying all of the unreachable nodes from the main node in the weighted call graph. Efficient graph algorithms for these operations are widely available [Tarjan 83].

When the inline expander fails to positively determine the internal function calling characteristics of some functions, there is missing information in the call graph construction. The two major causes of the missing information are calling external functions

and calling through pointers. Calling external functions occurs when a program invokes a function whose source file is unavailable to the inline expander. Examples include privileged system service functions and library functions distributed without source files. Because these can perform function calls themselves, the call graphs thus constructed are incomplete. Practically, because some privileged system services and library functions can invoke user functions, a call to an external function may have to be assumed to indirectly reach all nodes whose function addresses have been used in the computation in order to detect all recursions and all functions reachable from *main*.

Calling through pointers is a language feature which allows the callee of a function call to be determined at the run time. Theoretically, the set of potential callees for a call through pointer can be identified using program analysis. In practice, calling through pointers occurs so rarely that it may be assumed to reach all functions without significant penalty. Whenever there is any uncertainty, it is important to capture all the potential callees in order to detect all recursions and all functions reachable from *main*.

Each node in the weighted call graph contains three pieces of information: 1) the body of the function, 2) the node weight, and 3) a set of outgoing arcs to the callees. The node for a callee function is duplicated and absorbed by a caller during each inline expansion. The body of a function gives all the program declarations and statements of the function. The node weight gives the expected invocation count of the function. The outgoing arcs identify all static function calls in the present function.

Each arc in the weighted call graph contains five pieces of information: 1) a unique identifier, 2) the name of the caller, 3) the name of the callee, 4) the arc weight, and 5) a status. It is necessary to assign each arc a unique identifier because there may be several arcs between the same pair of caller and callee; the combination of the caller and callee information can not uniquely identify a static function call. The caller attribute identifies the function in which the corresponding call site is located. The callee attribute identifies the function invoked by the function call. The arc weight attribute indicates the expected execution frequency of the corresponding function call. The *status* attribute indicates

whether this arc is to be considered for inline expansion, rejected for inline expansion, or already inline expanded.

A weighted call graph is constructed in two steps. The first step generates all the nodes and arcs according to static program analysis. A node is generated for each function and an arc is generated for each call site. The function body and the outgoing arcs of each node are generated at this step. The unique identifier, the caller, the callee, and the status of each arc are also generated at this step. The second step is to fill in the weights for the nodes and the arcs.

A system-independent profiler has been integrated into the IMPACT-I C compiler. The profiler accumulates the average run-time statistics over many runs of a program. From the profile information, the IMPACT-I C compiler can determine the execution counts of all instructions and the frequencies of each of the possible directions of branch instructions. From the execution and branch frequencies, the node weights and arc weights of the call graph can be derived. Each node weight is simply the number of times a function is called in a typical run of the program. Each arc weight is the execution count of a function call.

A special node, &&&, is created to represent all the external functions. A function which calls external functions requires only one outgoing arc to the &&& node. In turn, the &&& node has many outgoing arcs, one to each function whose address has been used in the computation to reflect the fact that these external functions can potentially invoke every such function in the call graph. One arc to the &&& node sufficiently represents the effect of calling external functions, because calls to external functions can not be inlined, and, since an external function call is assumed to indirectly reach all nodes whose function addresses have been used in the computation, all the potential recursions and all the functions reachable from the *main* can be safely detected.

Similarly, a special node, ###, is used to represent all the functions which may be called through pointers. Calls through pointers are not considered for inlining in the IMPACT-I implementation. Rather than assigning a node to represent the potential callee of each call through pointer, ### is shared among all calls through pointers.

In fact, ### is assumed to reach all functions whose addresses have been used in the computation. This again ensures that all of the potential recursions and all of the functions reachable from the *main* can be safely detected. Experimental data indicate that function calls to external functions and function calls through pointers occur so rarely that this conservative approach reduces complexity at little cost in effectiveness.

5.1.4 Hazard prevention

The third issue regarding inline function expansion is how the hazardous function calls should be excluded from inlining. Three hazards have been identified in inline expansion: unavailable callee function bodies, multiple potential callees for a call site, and activation stack explosion. A practical inline expander has to address all these hazards. All the hazardous function calls are excluded from the weighted call graph and are not considered for inlining by the sequence controller.

The bodies of external functions are unavailable to the compiler. External functions include privileged system calls and library functions that are written in an assembly language. In the case of privileged system calls, the function body is usually not available regardless of whether the inline expansion is performed at compile time or link time. In fact, inlining privileged system calls is usually not desirable due to security reasons. Therefore, privileged system calls should be considered as not inline expandable.

Multiple potential callees for a call site occur due to calling through pointers. Because the callees of calls through pointers depend on the run-time data, there is, in general, more than one potential callee for each call site. Note that each inline expansion is equivalent to replacing a call site with a callee function body. If there is more than one potential callee, replacing the call site with only one of the potential callee function bodies eliminates all the calls to the other callees by mistake. Therefore, function calls originating from a call site with multiple potential callees should not be considered for inline expansion. If a call through pointer is executed with extremely high frequency, one can insert if statements to selectively inline the most frequent callees.

Parameter passing, register saving, local variable declarations, and returned value passing associated with a function can all contribute to the activation stack usage. A summarized activation stack usage can be computed for each function. A recursion may cause activation stack overflow if a call site with a large activation record is inlined into one of the functions in the recursion. For example, a recursive function m(x) and another function n(x) are defined as follows.

```
m(x) { if (x > 0) return(m(x-1)); else return(n(x)); }
n(x) { int y[100000]; ..... }
```

For the above example, two activation stacks are shown in Figure 5.6, one with inline expansion and one without. Note that inlining n(x) into the recursion significantly increases the activation stack usage. If m(x) tends to be called with a large x value, expanding n(x) will cause an explosion of activation stack usage. Programs which run correctly without inline expansion may not run after inline expansion. To prevent activation stack explosion, a limit on the control stack usage can be imposed for inline expanding a call into a recursion.

The calls to external functions and the calls through pointers are excluded from inline expansion. Because the IMPACT-I inline expansion is performed at compile time, any function calls whose callee source code (or intermediate code) is unavailable are excluded from inlining. A parameter to the compiler specifies the limit on the activation stack usage of a function to be inlined into a (potential) recursion. Any functions which require more activation stack usage are excluded from being inlined into a (potential) recursion. All the arcs corresponding to these hazardous function calls are excluded from the consideration of inline expansion. The experimental data indicate that this conservative approach has little negative impact on the effectiveness of the expander.

5.1.5 Sequence control

The fourth issue regarding inline function expansion is how the sequence of inlining should be controlled to minimize unnecessary computation, source file access, and code

expansion. In this step, we do not consider the hazardous function calls. The sequence control in inline expansion determines the order in which the arcs in the weighted control graph, i.e., the static function calls in the program, are inlined. Different sequence control policies result in different numbers of expansions, different numbers of file accesses, different code size expansions, and different reductions in dynamic function calls. All of these considerations affect the cost-effectiveness of inline expansion, and some of them conflict with one another.

The sequence control of inline expansion can be naturally divided into two steps: selecting the function calls for expansion and actually expanding these functions. The goal of selecting the function calls is to minimize the number of dynamic function calls subject to a limit on code size increase. The goal of actual expansion control is to minimize the computation cost incurred by the expansion of these selected function calls. Both steps will be discussed in this section.

In this section, we will limit the discussion to a class of inline expansion with the following restriction. If a function F has a callee L and L is to be inlined into F, then all functions absorbing F will also absorb L. Note that this restriction can cause some extra code expansion, as illustrated in the following example. Function F calls L (100 times) and is called by A (990 times) and B (10 times) (see Figure 5.7). In this call graph, there is not enough information to separate the number of times F calls L when it is being invoked by A and by B. Assume F is to be absorbed into both A and B. If F calls L 99 times when it is invoked by A and 1 time when by B, then L should be absorbed into A but not B (see Figure 5.8). With our restriction, however, L will be absorbed into both A and B (see Figure 5.8). Obviously absorbing L into B is not cost-effective in this case.

The problem is, however, that there is not enough information in the call graph to attribute the F→L weight to A and B separately. Therefore, the decision to absorb L only into A would be based on uncertain information. To accurately break down the weights, one needs to duplicate each arc as many times as the number of possible paths through which the arc can be reached from the main function. This will cause an exponential explosion of the number of arcs in the weighted call graph.

Because all the hazards due to recursion have been handled by the Hazard Prevention step, the call graph can be simplified by breaking all the cycles. The cycles in the call graph can be broken by excluding the least important arc from each cycle in the call graph. If the least important arc is excluded from inlining to break a cycle involving N functions, one can lose the opportunity to reduce up to $1/N$ of the dynamic calls involved in the recursion. This is usually acceptable for N greater than 1.

If N is equal to 1, breaking the cycle will eliminate all of the opportunity of reducing the dynamic calls in the recursion. If the recursion happens to be the dominating cause of dynamic function calls in the entire program, one would lose most of the call reduction opportunity by breaking the cycle. There is, however, a simple solution to this problem (see Figure 5.9). One can inline the recursive function call I times before breaking the cycle. In this case, one loses only $1/I$ of the call reduction opportunity by breaking the cycle.

The weighted call graph becomes a directed acyclic graph after all of the cycles are broken. All of the following discussions assume this property.

It is desirable to expand as many frequently executed function calls (heavily weighted arcs in the call graph) as possible. However, unlimited inline expansion causes code size expansion. To expand a function call, the body of the callee must be duplicated and the new copy of the callee must be absorbed by the caller. Obviously, this code duplication process increases program code size in general. Therefore, it is necessary to set an upper bound on the code size expansion. This limit may be specified as a fixed number and/or as a function of the original program size. The problem with using a fixed limit is that the size of the programs handled varies so much that it is very difficult to find a single limit to suit all of the programs. Setting the upper limit as a function of the original program size tends to work better for virtual memory and favor large programs. It may be true that many C functions are called once, and thus the original copies of these call-once functions can be eliminated by finding unreachable nodes from the *main* node after inline expansion. This issue will be addressed in the Desired Optimizations Section.

Code size expansion increases the memory required to accommodate the program and reduces instruction memory hierarchy performance. Precise costs can not be obtained during inline expansion because the code size depends on the optimizations to be performed after inline expansion. The combination of copy propagation, constant propagation, and unreachable code removal will reduce the increase in code size. A rough estimate of the code size increase can be derived from the intermediate code size of each function. Because the sizes of the functions change during inline expansion, it is important to keep track of the up-to-date size of each function.

Accurate benefits of inline expansion are equally difficult to obtain during inline expansion. Inline expansion improves the effectiveness of register allocation and algebraic optimizations, which reduces the computation steps and the memory accesses required to execute the program. Because these optimizations are performed after inline expansion, the precise improvement of their effectiveness due to inline expansion can not be known during inline expansion. Therefore, the benefit of inline expansion will be judged only by the reduction in dynamic function calls, which in turn reduces execution time of the program for each computer architecture. Using call frequency reduction rather than execution time reduction allows the inline expander to be independent of architectures.

The problem of selecting functions for inline expansion can be formulated as an optimization problem that attempts to minimize dynamic calls given a limited code expansion allowance. In terms of call graphs, the problem can be formulated as collecting a set of arcs whose total weight is maximized while the code expansion limit is satisfied. It appears that the problem is equivalent to a knapsack problem defined as follows: There is a pile of valuable items each of which has a value and a weight. One is given a knapsack which can hold up to only a certain weight. The problem is to select a set of the items whose total weight fits in the knapsack and whose total value is maximized. The knapsack problem has been shown to be NP-complete [Garey 79]. However, this straightforward formulation is unfortunately incorrect for inlining. The code size of each function changes during the inlining process. The code size increase due to inlining each function call depends on the decision made about each function call. The decision made

about each function call, in turn, depends on the code size increase. This dilemma is illustrated in Figure 5.10.

If L is to be inlined into F, the code expansion due to inlining F into A is the total size of F and L. Otherwise, the code expansion is simply the size of F. The problem is that the code increase and the expansion decision depend on each other. Therefore, inline expansion sequencing is even more difficult than the knapsack problem. Nevertheless, we will show that a selection algorithm based on call reduction achieves good results in practice.

The arcs in the weighted call graph are marked with the decision made on them. These arcs are then inlined in an order which minimizes the expansion steps and source file accesses incurred.

Different inline expansion sequences can be used to expand the same set of selected functions. For example, in Figure 5.11, Function D is invoked by both E and G. Assume that the selection step decides to absorb D, B, and C into both E and G. There are at least two sequences which can achieve the same goal. One sequence is illustrated in Figure 5.11, where E→D and G→D are eliminated first. Note that by absorbing D into both E and G (and therefore eliminating E→D and G→D in two expansion steps), four new arcs are created: E→B, E→C, G→B, and G→C. It takes four more steps to further absorb B and C into both E and G to eliminate all of these four new arcs. Therefore, it takes a total of 6 expansion steps to achieve the original goal.

A second sequence is illustrated in Figure 5.12, where B and C are first absorbed into D, eliminating D→B and D→C. Function D, after absorbing B and C, is than absorbed into E and G. This further eliminates E→B and E→C. Note that it takes a total of only 4 expansion steps to achieve the original goal.

The general observation is that if a function is to be absorbed by more than one caller, inlining this function into its caller before absorbing its callees can increase the total steps of expansion. The observation is illustrated in Figure 5.13. If a function, F, is to be inlined into one caller, there is no difference whether the calls in F are inlined

before F itself is inlined. Therefore, we need to consider only the situation in which F is to be inlined into more than one caller.

For the class of inlining algorithms considered in this dissertation, the rule for minimizing the expansion steps can be stated as follows: If a function F is absorbed into more than one caller, all of the callees to be inlined into F must be already inlined. It is clear that any violation against this rule will increase the number of expansions. It is also clear that an algorithm conforming to this rule will perform N expansion steps, where N is the number of function calls to be inlined. Therefore, an algorithm conforming to the rule is an optimal one as far as the number of expansion steps is concerned.

In a directed acyclic call graph, the optimal rule can be realized by an algorithm manipulating a queue of terminal nodes. The terminal nodes in the call graph are inlined into their callers if desired and eliminated from the call graph. This produces a new group of terminal nodes which are inserted into the queue. The algorithm terminates when all of the nodes are eliminated from the call graph. The complexity of this algorithm is $O(N)$, where N is the number of function calls in the program (arcs in the call graph) eligible for inlining.

Different inline expansion sequences to achieve the same goal may also incur different numbers of source file accesses. Due to the limited main memory size, only a limited number of function bodies can reside in the main memory at any time. A natural way to utilize this limited resource is to cache the function bodies. At any time, a number of function bodies reside in the main memory. If the inline expander finds the required function bodies in the main memory, the expansion can be performed without any file access. Otherwise, file access is performed and new function bodies may replace some existing ones in the main memory. As in any other cache organization, the locality of the function body is critical for this caching scheme to reduce the file access frequencies.

A function body is read when it is inlined into its callers; it is written when it absorbs its callees. Therefore, each inline expansion sequence can be reduced to a sequence of read and write accesses to the function bodies. To maximize the locality of these accesses, all of the accesses to a function body should be as temporally close as possible. That is,

after the callees of a function are inlined, that function should be inlined into its callers as soon as possible.

A queue-based algorithm which minimizes the expansion steps also exhibits good locality. As soon as a function absorbs its callees, it becomes a terminal node in the call graph. Because only the terminal nodes are processed in each iteration, the algorithm tends to inline the functions as soon as their callees are inlined. The optimal algorithm to achieve the maximal locality is yet to be derived. In fact, a precise definition of locality is yet to be introduced.

The selection of function calls for inlining is based mainly on dynamic call reduction. All of the arcs in the call graph are sorted according to their weights. The selection process then goes through the list starting from the heaviest arc. The arcs will be accepted for inlining until the code increase reaches the predetermined limit. Each time an arc is selected for inlining, its impact on the code size is immediately reflected in the call graph.

Because the order of consideration is independent of the code size increase, the decision process is somewhat simplified. However, the algorithm is not guaranteed to be optimal in dynamic call reduction. This is illustrated in Figure 5.14. The relative sizes of the functions A, F, L, and M are 4, 4, 2, and 2, respectively. Assume that the limit on code expansion is 40%. Because inlining F into A is the single step which decreases the largest number of dynamic function calls, it will be selected by the IMPACT-I expander. However, inlining both L and M (in two steps) into F actually reduces more dynamic function calls while incurring the same code increase.

The general observation is that inlining some function calls may incur too much code increase and thus prevent some cost-effective inlining steps from being selected. We will show, in the experimentation section, that this problem is not significant in the real programs examined.

To simplify the control for actually expanding the function calls, inline expansion is constrained to follow a linear order. The functions (nodes in the call graph) are first sorted into a linear list according to their weights. The most frequently executed function

leads the linear list. A function X can be inlined into another function Y if and only if X appears before Y in the linear list. Therefore, all inline expansions pertaining to function X must already have been accomplished before function Y is processed. The rationale is that functions which are executed frequently are usually called by functions which are executed less frequently. Therefore, this simple heuristic approximates the effect of the optimal queue-based algorithm. We will show, in the experimentation section, that this simple heuristic does approximate the optimal algorithms in practice.

5.1.6 Essential operations

The fifth issue regarding function inline expansion concerns the nature of the essential operations for inlining a function call. This task consists of three parts: 1) callee duplication, 2) variable renaming, and 3) parameter handling. The work required to duplicate the callee is trivial. The actual implementation difficulty is in caching the definitions of the most frequently inlined functions in memory to reduce the number of file reads.

To avoid conflicts with the caller's local variables, the callee's local variables must be renamed before inserting the code into the caller. This could be achieved by introducing a new scope for these local variables. This is especially easy in the modern structure languages such as Pascal and C where provisions have been made to allow multiple scopes within each function.

The callee's formal parameters must also be renamed before code insertion. This again could be achieved by introducing a new scope for these formal parameters. The renamed formal parameters can then receive the actual parameter values. The return value has to be buffered by new local temporary variables so that it can be used by the caller.

5.1.7 Desirable optimizations

The sixth issue regarding function inline expansion is what kind of code optimization techniques should be applied after inlining. On the one hand, inlining provides an en-

larged scope for code optimization techniques and makes them more effective. On the other hand, code optimization reduces the undesirable effects of inlining such as code size increase.

Functions are often created as generic modules to be invoked for a variety of purposes. Different callers may supply different flags to request different services. This is illustrated in the code segment below, where function F can be invoked by both A and B. The function can return either 3 or 1000 depending on the value of a flag. In this example, A and B will pass flag values 1 and 0, respectively.

```
A() {
    ...
    i = F(1);
}
B() {
    ...
    j = F(0);
}
F(flag) {
int flag;
    if (flag) return(3); else return(1000);
}
```

Inlining a function call places the body of the callee function into a specific invocation, which eliminates the need to cover the service required by the other callers. This is illustrated in the code segment below, where function F is inlined into both A and B. Note that the formal parameter *flag* has been renamed by introducing new scopes in both A and B. Also the actual parameters and the return value has been buffered. Function F is not shown because it is no longer important after expansion.

```
A() {
    ...
    {   int flag, temp;
        flag  = 1;
        if (flag) temp = 3; else temp = 1000;
        i = temp;
    }
}
```

```
B() {
    ...
    {   int flag, temp;
        flag = 0;
        if (flag) temp = 3; else temp = 1000;
        j = temp;
    }
}
```

With constant propagation, the constant value assigned to *flag* is propagated to the condition of the if statement. The resulting program is illustrated in the code segment below. The condition of the if statement in A becomes constant 1 and that in B constant 0.

```
A() {
    ...
    {   int flag, temp;
        flag = 1;
        if (1) temp = 3; else temp = 1000;
        i = temp;
    }
}
B() {
    ...
    {   int flag, temp;
        flag = 0;
        if (0) temp = 0; else temp = 1;
        j = temp;
    }
}
```

A simple analysis identifies one of the branches of the if statements as unreachable code. In our example, the *else* part in A and the *then* part in B are identified as unreachable code. These parts can be eliminated from the program as the result of unreachable code removal. The resulting program is illustrated in the following code segment.

```
A() {
    ...
    {   int flag, temp;
```

```
        flag = 1;
        temp = 0;
        i = temp;
    }
}
B() {
    ...
    {   int flag, temp;
        flag = 0;
        temp = 1;
        j = temp;
    }
}
```

Another pass of constant propagation will propagate the constant value assigned to *F_renamed_temp* to the subsequent assignment statement. The resulting program is illustrated as follows:

```
A() {
    ...
    {   int flag, temp;
        flag = 1;
        temp = 0;
        i = 0;
    }
}
B() {
    ...
    {   int flag, temp;
        flag = 0;
        temp = 1;
        j = 1;
    }
}
```

Finally, another analysis identifies the assignments to *flag* and *temp* as dead code because these variables are not used after these assignments. The corresponding declaration can be removed because these variables are neither defined nor used in A and B. The resulting program is as follows:

```
A() {
```

```
    ...
    {   i = 0;  }
}
B() {
    ...
    {   j = 1;  }
}
```

The above example illustrates that the callee function body can be inlined into a specific invocation in which the callee is free from the other obligations. On the one hand, standard optimizations such as copy propagation, constant propagation, constant folding, and unreachable code can be applied in a straightforward manner to improve the program efficiency. Without inline expansion, sophisticated interprocedural analysis would have to be performed to achieve similar effects. Similarly, register allocation and common subexpression elimination benefit from inlining. On the other hand, the code increase due to inline expansion can be significantly reduced using these optimizations.

Because programs always start from the *main* function, any function which is not reachable from the *main* function will never be used and can be removed. A function is reachable from the *main* function if there is a (directed) path in the call graph from the *main* function to the function, or if the function may serve as an exception handler, or be activated by some external functions. In the C language, this can be detected by identifying all functions whose addresses are used in computations.

Therefore, if a function is not explicitly reachable after inlining and its address is not used in any computation, that function can be eliminated. This rule can be applied to most system and user programs. In some special cases, such as real-time programs, there may be hidden paths where functions can be invoked through interrupts. Because these special cases occur rarely, an option to turn off the feature of eliminating unreachable functions is sufficient for handling them.

5.1.8 Experiments

We choose to evaluate the IMPACT-I inline expander with experiments on real programs. The purpose of these experiments is to answer the following questions:

(1) How many call sites are free of hazards and have significant benefits when inlined?

(2) For all call sites which are considered for inline expansion, how many dynamic calls can be eliminated?

(3) How much code expansion is incurred by inline expansion?

(4) Do most programs have similar static and dynamic function call characteristics?

(5) How frequently are the function calls executed before and after inline function expansion?

This experiment consists of four major steps. First, we select a benchmark suite of fourteen real UNIX programs. Most of the UNIX library functions such as printf() are included. Second, a variety of inputs for each benchmark are applied to establish reliable profile information. For example, we select from many sources 20 files of C programs, ranging from 100 to 3000 lines, as inputs for *cccp*, the GNU C language preprocessor. We also make special effort to exercise as many program options as possible. Third, the benchmarks are recompiled using profile information. Finally, the effects of inline expansion are measured.

Table 5.1 summarizes several important characteristics of our benchmarks. The *runs* column gives the number of different inputs used in the experiment. The *IL* column gives the average dynamic code sizes of the benchmark programs, measured in the number of thousands of intermediate instructions executed in a typical run of the programs.[1] There are about 3 billion intermediate instructions in the experiments. The *CT* column gives the average dynamic count of thousands of control transfers, other than function call/return,

[1]The static code size of a program is the number of instructions in the program. The dynamic code size of a program is the number of instructions that are executed in a single run of the program.

executed in a typical run of the programs. The *input* column describes the nature of the inputs used in the experiment.

Note that we use the dynamic counts of intermediate instructions rather than those of any specific machine instructions in an effort to keep the results general. The benchmark programs exhibit very different code sizes, control structures, and applications. There is no direct correlation between the static and dynamic code sizes of these benchmark programs.

Table 5.2 shows the static function call characteristics. The *total* column gives the number of different function calls in the static program. We categorize the static function calls into four types. The *external* column gives the percentages of static function calls to functions whose bodies are unavailable to inline expansion and to system functions (syscall). The *pointer* column gives the percentage of static function calls through pointers. Function calls through pointers cannot be inlined. The *avoided* column gives the number of static function calls which would either introduce function bodies into recursive paths and could cause activation stack explosion, or have an estimated execution count less than 10. The *candidate* column gives the percentage of the static function calls which are candidates for inline expanded. Only the *candidate* function calls are considered for inline expansion.

There are a total of 6,722 static function calls in all of the benchmarks. Dividing the total number of C lines in all of the benchmarks (53,617) by this number gives a static function call frequency of one in every 8 C lines. All benchmarks show large percentages of *avoided* functions (average about 65%). Only very small percentages of static calls are considered *candidate* (average about 10%). As a result, after the Hazard Prevention step, the sequence controller needs to examine only a small number of static function calls in typical programs.

Note that *tee* and *wc* contain no candidate function calls for inlining. As for *tee*, all of the frequently executed function calls are privileged system calls. We included this benchmark to show that programs with extremely high system frequencies exist. As for *wc*, there is very little function call activity. A possible explanation is that the program

is so small that its author decided to inline all the important function calls by hand. We included this benchmark to show that automatic inline expansion may not be necessary for some small programs.

Table 5.3 presents the dynamic behaviors of function calls. A static function call can correspond to many dynamic function calls. Only those static call sites corresponding to a large number of dynamic function calls should be considered for inline expansion. The small percentage of *avoided* dynamic calls indicates that the conservative IMPACT-I hazard prevention mechanism is very effective. Note that more than half of the function calls in *cmp*, *tee*, and *wc* are to external functions (mostly privileged system calls). Techniques to reduce the frequency of system calls need to be devised to reduce the function call frequency in these benchmarks.

Although the percentages of static *candidate* calls are small, *candidate* call sites correspond to large percentages of dynamic calls (about 70%). This means that by expanding a few static call sites, a large number of dynamic calls can be eliminated. One exception is *wc*, where function calls are unimportant because they are invoked very infrequently. The other exception is *tee*, where almost all the functions calls are to privileged system functions; the trapping overhead in these privileged system calls makes the function call overhead unimportant.

Table 5.4 offers the most important results of inline expansion. The *code inc* column gives the percentages of increase in static code sizes due to inline expansion. This number is measured without any optimization after inlining. The *call dec* column gives the percentage of dynamic function calls eliminated by inline expansion. The *IL per call* column gives the average number of dynamic intermediate instructions executed between dynamic function calls after inline expansion. The *CT per call* column gives the average number of dynamic control transfers executed between dynamic function calls after inline expansion.

Note that the inline expansion mechanism eliminates a large percentage of dynamic function calls for function call intensive programs. For programs with few dynamic function calls, the inline expansion mechanism does not eliminate large percentages of

dynamic function calls. This is a desirable behavior because the overall goal is to ensure infrequent function calls rather than to achieve high elimination percentages.

After inline expansion, function calls account for only a very small percentage of the control transfers (see the *CT per call* column). Therefore, function calls become much less important in the hardware design tradeoffs. Large scopes for compiler optimizations can be expected for the critical parts of the programs. The code expansion, on the average, is about a 17% increase in static code size. Because the code size increase is measured without optimizations after inlining, it is expected to be lower after optimization. In Figure 5.15, there are two bars associated with each benchmark: the left one shows the percentage of code size increase and the right one the percentage of call reduction.

The inline expander is not able to eliminate more than 80% of the candidate dynamic function calls for *cccp*, *espresso*, and *make*, because a large percentage of dynamic function calls were distributed among a large number of static calls. Inlining many of these function calls results in only a very small marginal improvement in the dynamic call reduction. As a result, the inline expander terminates after all the cost-effective static function calls have been expanded. We would like to point out that an optimal algorithm would also terminate under these conditions. In all of these benchmarks, the function call reductions achievable by an optimal algorithm have been achieved by the IMPACT-I heuristic. It should be noted, however, that an optimal algorithm might incur less code size increase to achieve the same result.

After inline expansion, the dynamic *external*, *pointer*, *avoided*, and *candidate* calls correspond to 56%, 3%, 18%, and 23% of all dynamic calls, respectively. Therefore, better ways to handle *external* functions are desirable. Since most *external* function calls in this experiment are system calls, new techniques to reduce the number of system calls should be studied.

5.1.9 Summary

We have identified six critical issues which have to be addressed by realistic inline expanders: the role of inlining, program representation, hazard prevention, sequence con-

trol, program modification, and desirable optimizations. Both theoretical and practical considerations for addressing these issues are presented. Optimal algorithms are provided whenever possible and heuristics are suggested whenever desirable. The IMPACT-I C Compiler inline expander has been implemented and is used to illustrate the design decisions involved in a practical inline expander.

We have shown, for fourteen realistic programs, that inline expansion can substantially reduce the function call frequencies. The heuristic algorithms adopted in the IMPACT-I inline expander approximate the optimal algorithms closely for these benchmarks. Inline expansion also results in enlarged optimization scopes for critical sections of the programs. We conclude that inline expansion is an extremely cost-effective alternative and/or supplement to other software and hardware interprocedural optimization techniques.

We have also pointed out problems with system calls, which become the major cost of function calls after inline expansion. Further study to reduce system calls is necessary.

The art of using profile information to make inlining and other compilation decisions in general is still in its infancy. The critical issue is how reliable run-time information can be derived from the profile data. A hybrid methodology combining program analysis and statistical analysis is being developed in the IMPACT project. A major breakthrough in this area will lead to the extensive use of run-time information to perform optimizations not possible in the present generation of compilers.

5.2 Instruction Placement

5.2.1 Introduction

The instruction memory hierarchy (on-chip caches, off-chip secondary caches, memory) has received only moderate attention due to the low instruction bandwidth requirements of conventional machines with a high microcycle count per instruction. In VAX-11/780, it takes 10.5 microcycles to execute every 3.8 bytes of instructions [Emer 84]. An 8-byte instruction buffer which prefetches instructions during idle cache cycles pro-

vides enough instruction bandwidth for the VAX-11/780 microengine. In response to the increasing demand for processor speed, performance improving techniques such as pipelining have been widely used to implement processors which requires much higher instruction bandwidth. For example, the VAX 8600 implementation requires 3.8 instruction bytes every 6 microcycles. Further reducing the number of microcycles per instruction will further increase the instruction memory bandwidth requirement, making the performance of the instruction memory access an important issue. Many processor architectures have adopted instruction formats and semantics to allow the instruction units to be efficiently pipelined [Russell 78], [Hennessy 81], [Chow 87], [Patterson 82]. To simplify instruction decoding, these processor architectures specify fixed instruction formats, for which the conventional encoding techniques cannot be applied. To simplify instruction sequencing, these processors specify instructions whose functions are close to the microinstructions of the microprogrammed processors. The instruction set does not include powerful opcodes, e.g., block move, that encode sequences of microinstructions. These two policies make the instruction unit pipelining more efficient, and therefore match the speed of the instruction unit pipeline to that of the execution pipeline. However, these policies increase dynamic code size and increase the instruction bandwidth requirement.

Compiler code improving techniques often increase code size. Inline expansion reduces function call overhead at the cost of increased code size. Loop unrolling increases code scheduling flexibility at the cost of increased code size. Trace scheduling extracts the program parallelism at the cost of increased code size. These techniques rely on the instruction memory hierarchy to absorb the increased code size so that the program execution speed can be improved. This puts further demand on the instruction memory hierarchy performance.

One conventional approach to improving the memory hierarchy performance is to increase the size and/or set-associativity of the top level cache memory [Smith 82], [Smith 87]. For example, the MIPS-X processor uses a 2048-byte, 8-way set-associative instruction cache with 8-byte blocks. This approach is limited because the cache cycle

time and the chip space increase as the size and set-associativity increase [Eickenmeyer 88], [Flynn 85], [Alpert 88]. To make the situation worse, if the compiler generates code with little spatial locality and/or many cache mapping conflicts, no cache of reasonable size and set-associativity can provide enough instruction bandwidth. The previous research results on the instruction cache design, however, did not consider the compiler's instruction placement algorithms.

We have designed and implemented an instruction placement algorithm to improve the performance of the instruction memory hierarchy. Spatial locality is maximized by placing the instructions executed near each other in time into consecutive memory locations. Cache mapping conflicts are minimized by placing the functions with overlapping lifetimes into memory locations which do not contend with each other in cache. This algorithm improves both caching and paging performance.

Using trace-driven simulation, we have demonstrated that the instruction layout algorithm can efficiently exploit small, direct-mapped instruction caches with large blocks. Good performance is achieved due to a low miss ratio, low memory traffic ratio, and fast hardware. The effect of varying the cache design parameters (cache size, block size, block sectoring, partial and loading) has been presented. Experiment data and algorithms can be found in our published papers [Chang 88], [Hwu 89a].

We will first present the trace selection algorithm, which is the heart of our instruction placement algorithm. Then we will describe an outline of our instruction placement optimization.

5.2.2 Trace selection

A trace is an ordered set of basic blocks that tend to execute in a sequence. The program control is likely to enter a trace from its first basic block. Once the program control enters a trace, it is likely that all basic blocks in the trace are executed. A trace selection algorithm identifies traces in a weighted control graph. The objective of trace selection is to minimize the number of times the program control enters and exits from the middle of traces, and to maximize the trace lengths.

Trace selection was first proposed by Fisher as a systematic approach to global microcode compaction [Fisher 81]. Since then, improvements and implementations of optimizations based on trace selection techniques have been reported [Linn 83], [Su 84], [Ellis 86], [Howland 87]. These techniques are useful for generating efficient code for application programs which are too large and too complicated to be hand-optimized. However, most of the experimental results reported on using trace selection to assist optimizing large application programs have been based on small benchmarks with simple control structures. For different trace selection algorithms, we report the distribution of control transfers categorized according to their potential impact on the microcode optimizations. The experimental results are based on ten C application programs which exhibit large code size and complicated control structure. The measured data for each program are accumulated across a large number of input files to ensure the reliability of the result. All experiments are performed automatically using our IMPACT C compiler which contains integrated profiling and analysis tools.

Trace Scheduling: We refer readers who are unfamiliar with trace scheduling to the original paper by Fisher [Fisher 81]. Trace scheduling consists of three major functions : *trace selection*, *local compaction*, and *bookkeep*. First, the trace selection function selects the most likely to be executed program path. Then, local compaction is applied to schedule the trace. And finally, the bookkeep function inserts patch code at the *split* and *rejoin* points to preserve correctness. The three functions are described in great detail in Ellis's thesis [Ellis 86].

Trace scheduling permits the patch code created during the bookkeep phase of a trace to be selected and compacted as part of later traces. However, we do not allow the additional basic blocks generated by the bookkeep function to be considered when forming later traces, unless they can be absorbed by jump optimization. This requirement allows us to apply trace selection independently of the local compaction and bookkeep functions. Code motion moves critical instructions on the program critical paths up to the earliest point at which they can be executed. The usefulness of the code motion

and the cost of the bookkeeping on the total program execution time depend on the program structure and on the underlying microarchitecture. For example, code motion applied to a section of a program with large fine-grain parallelism will tend to do well due to the large code movement freedom. In a pipelined processor, code motion allows the execution of multicycle operations to overlap with the issuing and execution of less critical operations when there is no data dependence. Similarly in a processor capable of issuing multiple instructions per cycle, code motion reduces execution time by packing operations into fewer instructions.

Trace scheduling guides global code motion by favoring most frequently executed program paths. Therefore, the goal of the trace selection function is to identify when forming longer traces is desirable and how all basic blocks should be partitioned into various traces. It would be grossly complicated for the trace selection function to deal with microarchitecture-dependent factors such as degree of hardware parallelism. Disregarding the hardware limitations, the trace selection function tries to form the longest possible traces, limited only by program-dependent factors.

The question is what program-dependent factors must the trace selection function consider. The program control flow, local program parallelism, and the code mobility as determined by data-flow analysis can all be implemented in the trace selector. The program flow analysis, by either loop analysis or dynamic profiling, allows the trace selector to form traces by grouping series of basic blocks which tend to execute together. The local program parallelism and code mobility analysis tell the trace selector when trace expansion should be stopped due to limited code movement freedom. However, the complexity of the analysis, although required in later phases of compilation, hinders the development of a clean selection function. It is best to use only the control flow information and to construct the longest traces.

The problem is how to form traces in such a way that the in-trace transition is maximized and the off-trace transition is minimized. Off-trace transitions can be classified into five different types. Together with in-trace transition, there are a total of six transition types (T1-T6).

(1) T1 connects the last node of a trace to the first node of a different trace.

(2) T2 connects the last node of a trace to a middle node of another trace (maybe the same trace).

(3) T3 connects a middle node of a trace to the first node of another trace (maybe the same trace).

(4) T4 connects two middle nodes of different traces.

(5) T5 connects two consecutive nodes within a trace.

(6) T6 connects the last node of a trace to the start node of the same trace.

Code motion is permitted only for T5 connections. A T2 transition requires bookkeeping at the rejoin location. A T3 transition requires bookkeeping at the branch location. A T4 connection requires bookkeeping at both the branch and the rejoin locations. A T2, T3, or T4 transition may execute longer than the same code without applying trace scheduling. Because code motion is not allowed across T1 and T6 connections, global code motion obtains no speedup over local code compaction for T1 and T6 connections.

Let %a, %b, %c, %d, %e and %f denote the percentages of T1, T2, T3, T4, T5 and T6 transitions, respectively, in a typical program run. The goal of the trace selector is to maximize %e and to minimize %b, %c, and %d. The various percentages allow us to compare different trace selection functions. A trace selection function is better than others if it generates higher %e and lower %b, %c, and %d, for a given control graph.

Selection Algorithm: In his trace scheduling paper, Fisher presented the following trace selection algorithm with node weights as the selection criteria. Later, Ellis in his thesis implemented the same general trace selection algorithm but used arc weights as the selection criteria.

```
algorithm trace_selection
    mark all nodes unvisited;
    while (there are unvisited nodes)
```

```
        /* select a seed */
        seed = the node with the largest execution
          count among all unvisited nodes;
        mark seed visited;
        /* grow the trace forward */
        current = seed;
        loop
            s = best_successor_of(current);
            if (s==0) exit loop;
            add s to the trace;
            mark s visited;
            current = s;
        end loop
        /* grow the trace backward */
        current = seed;
        loop
            s = best_predecessor_of(current);
            if (s==0) exit loop;
            add s to the trace;
            mark s visited;
            current = s;
        end loop
        /* compaction and bookkeep */
        trace_compaction;
        book_keep;
    end while
end algorithm
```

Since we do not consider the additional basic blocks generated by the bookkeep function in the trace selection process, the trace_compaction and the bookkeep functions are not included in the above algorithm.

To ensure that loop headers become the leading nodes of traces, when enlarging traces, crossing loop back-edges is prohibited. To avoid generating too many jump operations, trace selection is turned off for infrequently executed program sections. For example,

```
    branch if (r0>0) to L1;
L0: XXX
L1: YYY
```

is translated to the following code segment if L0 is rarely executed.

```
    branch if (r0<=0) to L0;
L1: YYY
    ......
L0: XXX
    goto L1;
```

The above example shows that trace selection can increase the number of unconditional branches. For machines that require branch slots for unconditional branches, it is better not to perform trace selection for infrequently executed code sections to reduce code size.

The node weight is the execution count of a basic block. This number can be either estimated statically by loop analysis or profiled dynamically by an automatic profiler. In this section, all weights used in the trace selection functions are strictly derived from the average program profile accumulated over many runs. The selection function based on node weights is shown in the following code segment.

```
best_successor_of(x)
    let n be the immediate successor of x
      having the largest execution count;
    if (n is visited) return 0;
    return n;
best_predecessor_of(x)
    let n be the immediate predecessor of x
      having the largest execution count;
    if (n is visited) return 0;
    return n;
```

Each node (basic block) of the control graph can have several incoming and outgoing arcs. Each arc represents a possible branch path connecting two nodes. Trace scheduling yields some performance gain when the program flows through an arc within a trace, and suffers when an off-trace arc is taken. Hence, arc weight is a better selection criterion than node weight. The selection based on arc weights is shown in the following code segment.

```
best_successor_of(x)
    let e be the arc with the largest execution count
      among arcs leaving x;
```

```
    n = the destination of e;
    if (n is visited) return 0;
    return n;
best_predecessor_of(x)
    let e be the arc with the largest execution count
      among arcs entering x;
    n = the source of e;
    if (n is visited) return 0;
    return n;
```

Some nodes have many incoming and outgoing arcs. If there is not a single arc which dominates all others, the performance gain that can be extracted by including the most likely to be taken arc by a trace will be overshadowed by the combined off-trace cost of all other arcs. In such instances, it is better to stop the trace expansion. To detect such cases, a minimum arc probability requirement is added to the selection function.

The probability that an outgoing arc Ai will be taken, given that the program control is already at node Nj which is the source of Ai, is simply (arc_weight(Ai) / node_weight(Nj)). The probability that a node Na is reached through an arc Ab is (arc_weight(Ab) / node_weight(Na)). Adding a minimum branch probability to the selection by arc function results in the following function.

```
best_successor_of(x)
    let e be the arc with the largest execution count
      among arcs leaving x;
    if (probability(e)<=MIN_PROB) return 0;
    n = the destination of e;
    if (n is visited) return 0;
    return n;
best_predecessor_of(x)
    let e be the arc with the largest execution count
      among arcs entering x;
    if (probability(e)<=MIN_PROB) return 0;
    n = the source of e;
    if (n is visited) return 0;
    return n;
probability(e)
    s = source of e;
    d = destination of e;
    return min((weight(e)/weight(s)),
```

```
(weight(e)/weight(d)));
```

With the minimum branch probability requirement, the trace selection algorithm will produce shorter traces, which is undesirable. On the other hand, control flows that enter and exit from the middle of traces will be kept to a very small number, which is desirable. In situations in which the bookkeep cost is large, it is better to add the minimum branch probability requirement.

Experiments: The compiler compiles and profiles the benchmark programs by inserting extra code to record the execution count of basic blocks and branch paths. The compiled programs are installed and tested with many inputs. For each run, the profiler updates the accumulated average execution count of basic blocks and branch paths for a typical run of the program. With the profile information, the compiler constructs the weighted control graph. Then trace selection is applied to the weighted control graph, and the percentages of the six connection types (%a %b %c %d %e %f) are measured.

Ten programs from several application domains are chosen mainly because of their popularity and substantial program size. Each of the ten programs is run at least ten times with realistic inputs. We make a special effort to exercise nearly all program options. In Table 5.5, the *name* column lists the program name. The *runs* column indicates the number of runs under profiler monitoring.

We report the percentage of each of the six transition types executed in a typical run of the benchmark program. The *loop* column in the following tables is the average number of basic blocks in an executed inner loop. The *trace* column is the average number of basic blocks of all traces executed. Table 5.6 corresponds to the selection according to node weight function. Table 5.7 corresponds to the selection according to arc weight function. Tables 5.8 to 5.11 demonstrate the effect of imposing additional minimum branch probability requirement.

As we have expected, arc weight is a better selection criterion than node weight. The additional minimum branch probability requirement further reduces the off-trace cost. As the minimum branch probability requirement increases, %b, %c, and %d decline

slightly. However, as the minimum requirement rises, fewer and smaller traces are formed, leading to low percentages of in-trace transitions. In any case, the in-trace transition percentage (%e) is several times larger than the off-trace transition percentages (%b, %c, %d) combined. This essentially tells us that even a small improvement in in-trace code movement can compensate for much larger bookkeep cost. The off-trace transitions (%b, %c, %d) are low, because benchmark programs have predictable branch behavior. The profile information shows that, on the average, the branch direction of more than 90% of all branch instructions executed can be correctly predicted statically.

A few of the benchmark programs show substantial inner loop back-edge transitions (%f). Loop unrolling can be applied to exploit program parallelism across loop iterations. When N copies of a loop exist, the loop back-edge of the first (N-1) instances can be transformed into normal connections between two distinct nodes. These (N-1) connections between different iterations of the loop can be selected for trace expansion. Since many iterations are usually taken before the program control leaves the loop, the expanded loop structure will form a long trace covering the most important path of all unrolled instances of the loop.

For several benchmarks, the number of function calls is substantial, more than one function call per every six basic blocks executed. The program *tbl* shows the highest function call frequency, about one function call for every two basic blocks executed. The profile result shows that the most frequently executed function in *tbl* consists of only one basic block. Similarly in the other programs, the most frequently executed functions tend to be small and can be easily in-line expanded. Since function in-line expansion not only gives larger traces but also eliminates register saving and restoring around the function boundaries, the potential gain seems to be more substantial than loop unrolling.

Of all the traces actually executed, the average trace size is about three to four basic blocks for various selection functions. The relatively small size is due to control uncertainties and small function body. One can expect some increase in trace length after function in-line expansion. An inner loop as seen by the IMPACT C compiler is a trace whose last node branches back to the trace header. The average size of all inner loops

executed is about three basic blocks. In other words, one can expect two conditional branchs in inner loops. Therefore, loop unrolling and software pipelining techniques for large integer programs must cope with at least two conditional branchs in inner loops.

Since the percentage of off-trace transition (%b, %c, %d) is much smaller than in-trace transition (%e), trace scheduling can tolerate large off-trace cost.

5.2.3 Instruction placement

The goal of the IMPACT-I C Compiler instruction placement optimization is to lay out the target program to maximize spatial locality and to minimize cache mapping conflicts. To maximize spatial locality, instructions are mapped into the same block if they are executed close to each other in time. Therefore, almost all the bytes in a block are used when that block is brought in cache. To minimize mapping conflicts, functions with overlapping lifetimes are mapped into different blocks of the cache. The instruction placement optimization is implemented in five major steps: execution profiling, function inline expansion, trace selection, function layout, and global layout.

Step 1. Execution profiling. A program is represented by a weighted call graph. A call graph is a directed graph in which every node is a function and every arc is a function call. A weighted call graph is a call graph in which all the nodes and arcs are marked with their execution frequencies. Each node of the weighted call graph corresponds to a weighted control graph. A control graph (for a function) is a directed graph in which every node is a basic block, and every arc is a branch path between two basic blocks. A weighted control graph is a control graph in which all the nodes and arcs are marked with their execution frequencies. The IMPACT-I profiler translates each target C program into an equivalent C program with additional probe function calls. When the equivalent C program is executed, these probe function calls record the weights of the nodes and arcs of the call graph for the entire program and the control graph for each function. It is critical that the inputs used for executing the equivalent C program be representative. Therefore, this approach is more suitable for characterizing realistic programs for which

representative inputs can be easily collected. The IMPACT-I Profiler to C Compiler interface allows the profile information to be automatically used by the IMPACT-I C Compiler.

Step 2. Function inline expansion. The function calls (arcs in the weighted call graph) with high execution count are replaced with the function bodies if possible. The goal is to transform all of the important interfunction control transfers into intrafunction control transfers. Inline expansion reduces the dynamic interfunction control transfers to a small percentage (about 1%) of all the control transfers, which provides two major advantages. First, the spatial locality improves because almost all the control transfers are within individual functions. Second, the potential cache mapping conflicts are reduced because the potential conflicts across functions are insignificant.

Step 3. Trace selection. For each function, basic blocks which tend to execute in sequence are grouped into traces. The traces are the units of instruction placement to maximize spatial locality. Note that the inline expansion step provides large functions to enhance the size of the traces selected.

Step 4. Function layout. For each function, traces which tend to execute in sequence are placed in consecutive memory locations. We start with the function entrance trace, and expand the placement by placing the most important descendant after it. We grow the placement until all the traces with nonzero execution count have been placed. Traces with zero execution count are moved to the bottom of the function. This results in a smaller effective function body, allowing more functions to be packed into each page.

Step 5. Global layout. The goal of the global layout algorithm is to place functions which are executed close to each other in time into the same page, so that interfunction cache conflicts are further reduced and the working set for instruction paging can be also reduced.

5.3 Control Flow Optimization

5.3.1 Introduction

Pipelining increases the throughput of the instruction fetch, instruction decode, and instruction execution portions of a high-performance scalar processor. Function call/return and branch instructions disrupt the flow of instructions through the pipeline, degrading the utilization of the pipelined datapaths. The IMPACT-I C compiler performs four optimizations in sequence to improve the control flow:

1) function inline expansion,

2) trace selection,

3) instruction placement, and

4) branch prediction and smart multiway branch implementation.

This section describes the compile-time branch handling issues. We will use the benchmark programs that are listed in Table 5.1. It is assumed that function inline expansion, trace selection, and instruction placement have been applied.

5.3.2 Multiway branch

The distribution of various types of branch instructions is listed in Table 5.12. The *%conditional* column of Table 5.12 indicates the percentage of conditional branch instructions among all the dynamic control transfer instructions. The *%unconditional* column of Table 5.12 indicates the percentage of unconditional branch (including call/return) instructions among all of the dynamic control transfer instructions. Inline expansion has already reduced the number of unconditional branches. The *%multiway* column of Table 5.12 indicates the percentage of multiway branch instructions among all dynamic control transfer instructions. Although the percentage of multiway branch instructions is small, they are nevertheless important due to their long potential execution time.

Each multiway branch (switch statement) can be implemented by a hashing jump or a sequence of conditional branches. The IMPACT-I C compiler implements each

multiway decision as follows. First, the compiler sorts all of the target cases by their probability of execution. Second, the compiler lays out the conditional branches so that the ones with higher branching probability appear before those with lower branching probabilities. An exception to this rule is the *default* case, which has to be placed at the very end as an unconditional jump instruction. Third, the compiler calculates the expected number of comparisons to implement the multiway decision with the sequence of conditional branches formed in the second step. If the expected number of comparisons is beyond a threshold (10 in this measurement), a hashing jump will be used instead. The execution of these hashing jumps involves hashing the input condition into a hash table of explicit and default cases, fetching the corresponding target address, and redirecting the instruction fetch with that target address.

Table 5.13 shows the results of the multiway branch implementation. The *%default* column indicates the percentage of the time the *default* case is reached for all switch statements. For some benchmarks, the *%default* percentage is high due to the low coverage of the explicit cases. Because we must place the *default* case at the end of the branch sequence as an unconditional branch instruction, high *%default* percentage lessens the effectiveness of compiler case layout optimization. The effect is especially pronounced in *eqn*.

The *%hashing* column indicates the percentage of all multiway branches being implemented by hashing jumps. For architectures with long scalar memory access delays, the threshold for adopting the hashing jumps could be increased to much more than the one we used (10 expected comparisons). Therefore, one can expect to see a smaller percentage of hashing jumps for architectures with long scalar memory delays. The *%sequence* column of Table 5.13 indicates the percentage of all switch statements being implemented by branch sequences. The *total* column indicates the average number of cases per multiway branch implemented by branch sequences, excluding the default case. The *expected* column indicates the expected number of comparisons required to resolve a multiway branch implemented as a branch sequence. Note that for most benchmarks, the sequence percentage is close to 100%. For *compress*, *grep*, and *lex*, the high percentage of branch

sequence implementations results from the highly biased distribution of selecting cases. For these benchmarks, the average total number of comparisons is high (10 or more) but the expected number of comparisons is much lower (at most 5). For the other benchmarks, almost all multiway branches are implemented as branch sequences, due to their small numbers of total cases.

If a hashing jump is 10 times more expensive than each conditional branch, the cost of each multiway branch is reduced to about 3.5 conditional branches per multiway branch. Because multiway branches occur infrequently in execution, we conclude that the cost of multiway branches is no longer a major concern.

5.3.3 Branch prediction

We now examine the characteristics of the conditional branches corresponding to the two-way decisions in C programs. These branches are due to *if* statements, the conditional operators (&&, || and ?:), and the loop control structures. The IMPACT-I C compiler uses the profile information to lay out the instruction space to reduce the frequency of taken-branch instructions. For each function, basic blocks which tend to execute in sequence are grouped into traces. Trace selection reduces the number of (dynamic) taken branches.

Table 5.14 shows a detailed breakdown of the statically predicted and actual behavior of branches.[2] Column TT of Table 5.14 indicates the percentage of branches which are predicted to be taken and are actually taken, as a percentage of all conditional branches. Column TN of Table 5.14 indicates the percentage of branches which are predicted to be taken but are actually not taken, as a percentage of all conditional branches. Column NT of Table 5.14 indicates the percentage of branches which are predicted not taken but are actually taken, as a percentage of all conditional branches. Column NN of Table

[2]The precision of the numbers in Table 5.14 and the other tables in this chapter may not comport with their accuracy because the sample sizes are small (e.g., 20). The reader should round off one or two digits when using the results.

5.14 indicates the percentage of branches which are predicted not taken and are actually not taken.

Two observations are worth mentioning. First, about 65% of the dynamic branches are not taken and almost all of them can be correctly predicted at the compile time. Comparing this number with the traditional 35% percentage ([Smith 81], [Lee 84], [Emer 84]) shows that our instruction placement algorithm is effective in reducing taken branches. Second, among the taken branches (which account for about 35% of the dynamic branches), most of them can also be correctly predicted at the compile time. Overall, about 92% of the dynamic branches can be correctly predicted at the compile time.

5.4 Conventional Code Optimization

All of the optimizations that are presented in this section can be formulated as predicates on a set of operations. If all predicates are true, then the set of operations can be replaced by another set of operations that is more efficient. The scope of code optimization is where operations are selected to be tested by the predicates. Local code optimization limits its scope to a basic block at a time. Global code optimization limits its scope to a function at a time.

Table 5.15 shows a list of classical code optimizations that have been integrated into the Lcode optimizer. The *name* column shows the names of the code optimizations. The *local* column is marked *yes* if the optimization has been implemented as a local code optimization, and *no* if otherwise. The *global* column is marked *yes* if the optimization has been implemented as a global code optimization, and *no* if otherwise. The *trace* column is marked *yes* if the optimization has been implemented as a trace-based global code optimization, and *no* if otherwise. In this section, we will briefly describe the functionality of each optimization. The implementation details can be found in most compiler text books [Aho 86]. Trace-based code optimizations are described in the next subsection.

Constant Propagation: Constant propagation involves statements of the form ($a = b$), where b is a constant. After determining where this definition of a reaches,[3] the constant b can be propagated to replace some references to a. This optimization is very effective in propagating constant parameters after function inline expansion.

Copy Propagation: Copy propagation involves statements of the form ($a = b$), where b is a virtual register. After determining where this definition reaches, references to a can be replaced by b if b is not modified, or a new register can be introduced to preserve the value of b. Standard algorithms for performing this copy propagation can be found in [Aho 86].

```
R0 = R1;              R0 = R1;              /* can become dead code */
....          ->      ....          ->      ....
R2 = R0 * 5;          R2 = R1 * 5;          R2 = R1 * 5;
```

In many cases, the original move statement becomes dead code after copy propagation.

Another form of copy propagation merges two virtual registers into one virtual register if their lifetimes do not overlap. For example,

```
R0 = R1 * 5;              R2 = R1 * 5;
R2 = R0;            ->    /* becomes dead code */
....                      ....
if (R0>12) goto L0;       if (R2>12) goto L0;
```

Memory Copy Propagation: Memory copy propagation involves statements of the form ($mem[a] = b$), where b can be a constant or a register. After determining where this definition reaches, references to $mem[a]$ can be replaced by b if $mem[a]$ is not modified. We have implemented a very limited memory disambiguation function to support this optimization. Our memory disambiguation function currently distinguishes different global scalar variables and memory accesses using the same base address and constant offsets.

[3] An instruction x reaches another instruction y if the values of one or more source operands of instruction y can come from instruction x. A formal definition of the *reaching definition* property and an algorithm for detecting reaching definitions can be found in [Aho 86].

```
mem[_a] = R1;          mem[_a] = R1;          mem[_a] = R1;
....              ->   ....              ->   ....
R2 = mem[_a];          R2 = R1;               /* can become dead code */
R2 = R2 - 19;          R2 = R1 - 19;          R2 = R1 - 19;
```

Classical copy propagation optimization includes constant propagation, copy propagation, and memory copy propagation. We classify copy propagation techniques into these types in order to fine-tune and to characterize the importance of each type.

Operation Combining: There are several forms of operation combining. The first type of operation combining combines two operations into a more powerful operation. For example, condition code computation and conditional branch operations can often be combined.

```
R1 = R0 - 5;          -> R1 = R0 - 5;              /* can become dead code */
if (R1 > 0) goto L0;  if (R0 > 5) goto L0; -> if (R0 > 5) goto L0;
```

For another example, some machines support *and_not* and *or_not* operations.

```
R1 = not R0;
R1 = R1 and R2;  ->  R1 = R2 and_not R0;
```

A side effect of this type of operation combining is that it reduces the length of critical paths and improves the code scheduling for a multiple operation issue processor.

The second type of operation combining is similar to tree height reduction of expressions by moving constant operands up in an expression tree. For example,

```
R1 = $SP - 24;     ->  R3 = 10 - 24;
mem[R1+10] = R0;       mem[$SP+R3] = R0;  ->  mem[$SP-14] = R0;
```

A side effect of this type of operation combining is that it may benefit from loop invariant code elimination and loop induction variable elimination. For example,

```
R1 = R0 - 20;      ->  R1 = $SP - 20;     /* becomes invariant code */
mem[$SP+R1] = 0;       mem[R0+R1] = 0;
```

On machines that support guarded instructions, another type of operation combining can combine a conditional branch operation and a data movement operation into a guarded operation. For example,

```
    if (cc) goto L1;          /* becomes dead code */
L0: R1 = 5;              ->   if (!cc) R1 = 5;
L1: ....                      ....
```

Common Subexpression Elimination: Because the conversion from Hcode to Lcode is done one Hcode expression at a time, there can be a lot of redundant computations across Hcode expressions. Common subexpression elimination tries to identify common operations and eliminate redundant work. For example,

```
struct xx B[];        /* sizeof(struct xx) = 40 */
int A, B[], C;
A = B[X] + 5;
C = 4 - B[X];
```

is translated to

```
R0 = X * 40;
R0 = mem[_B + R0];
A = R0 + 5;
R1 = X * 40;          /* redundant */
R1 = mem[_B + 40];    /* redundant */
C = 4 - R1;
```

and can be optimized to

```
R0 = X * 40;
R0 = mem[_B + R0];
A = R0 + 5;
C = 4 - R0;
```

To characterize this optimization more accurately, we distinguish three types of operations: memory load, memory store, and the rest. Common subexpression elimination that involves memory load operations is called redundant load elimination, and common subexpression elimination that involves memory store operations is called redundant store elimination. Redundant load and store elimination can be substantially more difficult than common subexpression elimination due to limited memory disambiguation capability.

Dead Code Removal: Dead code is a collection of operations whose results will not be used by later operations and does not affect the output. In the case of local variables, assignments are to virtual registers and can be eliminated if no more use of the register occurs before the exit point of a function or before another definition of the register. In the case of memory stores, dead code removal can be difficult due to limited memory disambiguation capability. Dead code is traditionally found by determining the liveness of variables. Most other optimizations convert redundant operations into NO_OPs and rely on dead code removal to eliminate NO_OPs. Another application of dead code removal is to remove operations of the form $(a = a)$ and $(mem[a] = mem[a])$ due to binding variables to the same storage location.

Constant Folding: After function inline expansion and constant propagation, many operations will have one or more constant operands. Constant folding is applied if the value of an operation can be determined at compile time. For example, addition by zero can be converted to a move operation. When all operands are constant, most arithmetic operations can be evaluated at compile time. Handling branch operations is substantially more difficult, because constant folding of a conditional branch operation may alter the control graph structure and therefore affect the dataflow information.

Strength Reduction: More expensive operations, such as multiplication and division, can be converted to less expensive operations. For example, multiplication and division by a constant of a power of two can be converted to a shift operation (can shift multiply bit positions). On machines that do not have a hardware multiplier, it is desirable to expand a multiplication into a sequence of shift and add operations. For example,

```
R1 = R0 * 17;  ->  R2 = R0 * 16;  ->  R2 = R0 << 4;
                   R1 = R2 + R0;      R1 = R2 + R0;
```

For another example, modulo operation on a constant of a power of two can be converted into a bitwise AND operation.

Operation Cancellation: On rare occasions, the code optimizer can identify two operations that cancel each other exactly. This optimization is implemented by pattern matching special operation pairs.

Code Reordering: Whereas code scheduling improves instruction pipelining, code reordering enables more copy propagation and operation combining optimizations. For example,

```
R1 = R0;        ->  R1 = R0;       ->  R2 = R0 + 6;
R0 = 1;             R2 = R1 + 6;       R0 = 1;
R2 = R1 + 6;        R0 = 1;
```

Jump Optimization: Jump optimization replaces a frequently executed unconditional jump operation with a copy of the target basic block. This optimization reduces the number of spurious jump operations that are introduced by instruction placement and constant folding, which convert some conditional branches whose source operands are constants into jump operations. This optimization also enlarges the scope of code optimization. However, the drawback is that it modifies the control graph and thus affects the dataflow information.

Dead Block Elimination: Basic blocks that will never be executed can be eliminated. This optimization can be implemented by a simple graph algorithm that detects unreachable nodes of a control graph. All unreachable nodes are dead blocks.

Loop Invariant Code Removal: Operations whose operands are invariant in a loop body can be moved to before the beginning of the loop. The simplest way is to introduce a loop header basic block and insert invariant code in that basic block. The additional control flow paths can be simplified later by jump optimization. By reducing the size of loop bodies, the number of operations that are executed is greatly reduced. Standard implementation techniques can be found in [Aho 86].

Loop Induction Variable Strength Reduction: An induction variable is a variable that appears only in operations of the form ($v = v + constant$), ($v = v - constant$), or ($v = constant - v$) within a loop body. Loop induction variable strength reduction replaces complex operations that are linear functions of an induction variable by simpler operations. Most often, this optimization replaces multiplications between induction variables and constants by simple increments [Aho 86]. For multidimensional arrays and structure arrays, multiplications by constants are always necessary to compute address offsets.

Loop Induction Variable Elimination: Two induction variables of the same form can be combined into one. For example,

```
     R0 = 0;
     R1 = 0;                R0 = 0;
L0:  ....              ->   .... /* change R1 to R0 */
     R0 = R0 + 1;           R0 = R0 + 1;
     R1 = R1 + 1;           if (cc) goto L0;
     if (cc) goto L0;
```

When the initial values of two induction variables are not identical, they can still be combined by appropriately adjusting other operations. For example,

```
     R0 = $SP - 20;         R0 = $SP - 20;
     R1 = 0;                R1 = 0;
L0:  R2 = mem[R0];     ->   R2 = mem[R0 + R1];
     R3 = mem[R1];          R3 = mem[R1];
     ....                   ....
     R0 = R0 + 1;           R1 = R1 + 1;
     R1 = R1 + 1;           if (cc) goto L0;
     if (cc) goto L0;
```

5.5 Trace-Based Code Optimization

The results from the trace selection experiments indicated that traces are generally small, containing only a few basic blocks. To increase the size of traces, we first apply

jump optimization to replace an unconditional jump operation by a copy of the destination basic block. Figure 5.16 shows that by duplicating basic block A, an inner loop that is free of branch operations is formed. Figure 5.17 shows a typical control graph generated from an *(if A then C; B)* statement, where C is not likely executed. By duplicating basic block B, an off-trace into the (A, B) trace is eliminated and the (C) trace is enlarged. Code expansion due to jump optimization can be controlled by inhibiting it at infrequently executed code sections.

Optimizing frequently executed paths: All profile-based code optimizations that will be presented in this section explore a single concept: *optimizing the most frequently executed paths*. We will illustrate this concept by an example. Figure 5.18 shows a weighted control graph which represents a loop program. The execution counts of basic blocks $\{A, B, C, D, E, F\}$ are $\{100, 90, 10, 0, 90, 100\}$, respectively. Clearly, the most important execution path in this example is the $\{A, B, E, F\}$ trace. Because basic blocks in this trace are executed many more times than basic blocks D and C, the code optimizer can apply transformations that reduce the execution time of the $\{A, B, E, F\}$ trace, but may increase the execution time of basic blocks D and C. Nonloop-based classic code optimizations are conservative and do not perform transformations that may increase the execution time of any basic block. Loop-based classic code optimizations consider the entire loop body and do not consider the case in which some basic blocks in the loop are rarely executed because of branch operations that are heavily biased to go to one direction. In the rest of this subsection, we describe several profile-based code optimizations that make aggressive decisions and explore more optimization opportunities. Details can be found in [Chang 91b].

Forming super-blocks: We propose a simple data structure called super-block to represent a frequently executed path. A super-block has the following features. (1) It is a linear sequence of basic blocks $B(i), i = 1...n$, where $n \geq 1$. (2) It can be entered only from $B(1)$. (3) The program control may leave the super-block from any one basic

block. (4) When a super-block is executed, it is very likely that all basic blocks in the super-block are executed.

The formation of super-blocks is a two-step procedure: (1) trace selection and (2) tail duplication. Trace selection identifies basic blocks that tend to execute in a sequence and groups them into a trace. The trace selection algorithm has been shown in a previous subsection. Figure 5.18 shows the result of trace selection. Each dotted-line box represents a trace. There are three traces: $\{A, B, E, F\}$, $\{D\}$, and $\{C\}$.

After trace selection, each trace is converted into a super-block by duplicating the tail part of the trace to ensure that the program control can enter only from the first basic block. The tail duplication algorithm is shown in the following code segment.

```
algorithm tail_duplication(a trace B(1..n)) begin
    if (B(1) is the only basic block, from which program
      control can enter the trace) then
        exit; /* it is already a super-block */
    let B(i) be the second basic block that is an entry
      point to the trace.
    for (k=i..n) begin
        create a trace that contains a copy of B(k);
        place the trace at the end of the function;
        redirect all control flows to B(k), except
          the ones from B(k-1), to the new trace;
    end for
end algorithm
```

After tail duplication, the example in Figure 5.18 becomes the graph in Figure 5.19. Because there are several control paths into F, we duplicate the tail part of the $\{A, B, E, F\}$ trace from basic block F. Each duplicated basic block forms a new super-block and is appended to the end of the function.[4] More code transformations can be applied after tail duplication to eliminate spurious jump operations. For example, the F' super-block in Figure 5.19 can be duplicated and each copy can be combined with

[4]Note that the profile information needs to be scaled accordingly. Scaling the profile information destroys the accuracy. Fortunately, code optimizations after forming super-blocks need only approximate profile information. In order to have accurate profile information (for taking measurements), the transformed program can be profiled again.

the $C(D)$ super-block to form a larger super-block. To control code expansion, we add a basic block to a trace only if the execution count of the basic block exceeds a threshold value, e.g., 100. After forming super-blocks, we optimize only super-blocks whose execution counts are higher than the threshold value.

Examples: Figure 5.20 shows an example of super-block based common subexpression elimination. Common subexpression elimination cannot be applied to the original program in Figure 5.20(a) because opB modifies r2 (a source operand of the common subexpression). Figure 5.20(b) shows the super-blocks that are formed from the original program. In the transformed program, opB no longer affects the value of r2 that is used in opC; therefore, common subexpression can now be applied and result in the program in Figure 5.20(c).

Figure 5.21 shows an example of super-block-based dead code removal. The program is a simple loop that has been unrolled four times. The loop index variable (r0) has been expanded into four registers (r1,r2,r3,r4) whose values can be computed in parallel. If the loop index variable is live after the loop execution, then it is necessary to update the value of r0 in each iteration, as shown in Figure 5.21(a). These update operations, e.g., r0=r1, r0=r2, and r0=r3, are dead code in the super-block, because references to them now refer to r1,r2,r3, and r4. Therefore, we can move these update operations out of the super-block. The result is shown in Figure 5.21(b).

Figure 5.22 shows an example of super-block-based loop invariant code removal. In Figure 5.22(a), opA is not loop invariant (in the traditional sense) because its source operand is a memory variable (buffer.length), and opD is a function call that may modify the memory variable (buffer.length). In super-block-based loop invariant code removal, opA is invariant because opD is not in the super-block. The result is shown in Figure 5.22(b).

Figure 5.23 shows an example of super-block-based global variable migration. The memory variable x[1] cannot migrate into a register in traditional global variable migration because opC may access x[1]. In super-block-based global variable migration, x[1]

can migrate into a register. The result is shown in Figure 5.23(b). Extra operations (opX, opY and opC) are added to the super-block entry and exit points to ensure correctness of execution.

Summary: Nonloop super-block code optimizations are effective because of tail duplication. Loop super-block code optimizations are effective because we optimize only the most important execution path of each loop. Experimental data that show the importance of super-block code optimizations will be presented in Chapter 8.

Table 5.1 Benchmark characteristics.

benchmark	*runs*	*IL*	*CT*	*input*
bison	10	9797K	1944K	grammar for a C compiler, etc.
cccp	20	585K	111K	C programs (100-3000 lines)
cmp	16	135K	30K	similar/dissimilar text files
compress	20	981K	155K	same as cccp
eqn	20	1809K	537K	papers with .EQ options
espresso	20	54496K	8522K	original espresso benchmarks
grep	20	2357K	857K	exercised various options
lex	4	152630K	56295K	lexers for C, Lisp, awk, and pic
make	20	7629K	1620K	makefiles for cccp, compress, etc.
tar	14	809K	104K	save/extract files
tbl	20	581K	137K	papers with .TS options
tee	20	24K	9.5K	same as cccp
wc	20	392K	112K	same as cccp
yacc	8	15668K	3935K	grammar for a C compiler, etc.

Table 5.2 Static function call characteristics.

benchmark	*total*	*external*	*pointer*	*avoided*	*candidate*
bison	1026	40.4%	0.0%	49.9%	9.6%
cccp	393	15.8%	0.2%	74.3%	9.7%
cmp	40	50.0%	0.0%	47.5%	2.5%
compress	183	37.7%	0.0%	61.7%	0.5%
eqn	463	4.1%	0.0%	79.3%	16.6%
espresso	1466	4.7%	0.8%	64.0%	30.4%
grep	90	20.0%	0.0%	73.3%	6.6%
lex	560	8.9%	0.0%	73.2%	17.9%
make	686	15.2%	0.0%	63.5%	21.4%
tbl	797	4.4%	0.0%	74.8%	20.8%
tar	445	31.2%	0.0%	63.8%	4.9%
tee	82	40.2%	0.0%	59.8%	0.0%
wc	27	48.1%	0.0%	51.9%	0.0%
yacc	464	19.2%	0.0%	64.7%	16.2%

Table 5.3 Dynamic function call behavior.

benchmark	*total*	*external*	*pointer*	*avoided*	*candidate*
bison	31104	36.6%	0.0%	1.4%	62.0%
cccp	2569	5.3%	5.4%	6.5%	82.7%
cmp	1001	50.2%	0.0%	0.5%	49.3%
compress	4684	8.1%	0.0%	0.6%	91.3%
eqn	48428	8.2%	0.0%	0.9%	90.9%
espresso	295778	0.1%	9.4%	0.2%	90.3%
grep	17489	1.2%	0.0%	0.1%	98.8%
lex	84648	13.5%	0.0%	0.4%	86.1%
make	48056	9.2%	0.0%	0.2%	90.6%
tar	1442	35.3%	0.0%	12.0%	52.7%
tbl	31987	14.6%	0.0%	3.4%	82.0%
tee	1583	99.1%	0.0%	0.9%	0.0%
wc	21	53.1%	0.0%	46.9%	0.0%
yacc	3935	7.7%	0.0%	0.3%	92.0%

Table 5.4 Inline expansion results.

benchmark	*code inc*	*call dec*	*IL per call*	*CT per call*
bison	17%	50%	630	125
cccp	17%	55%	506	95
cmp	3%	49%	265	58
compress	4%	91%	2324	368
eqn	22%	81%	197	58
espresso	24%	70%	616	96
grep	31%	99%	11214	4071
lex	23%	77%	7807	2880
make	34%	59%	388	82
tar	16%	43%	983	127
tbl	30%	66%	55	13
tee	0%	0%	15	6
wc	0%	0%	18310	5146
yacc	24%	80%	1205	303

Table 5.5 Benchmarks.

name	*runs*	*description*
cpp	34	GNU C preprocessor
eqn	10	typeset mathematics for nroff/ditroff
espresso	18	Boolean minimization
grep	10	pattern search
more	10	browse through a text file
mpla	18	technology independent PLA generator
nroff	10	format documents for display
pic	20	format pictures for nroff/ditroff
tbl	14	format tables for nroff/ditroff
wc	10	word count program

Table 5.6 Selection according to node weight.

benchmark	%a	%b	%c	%d	%e	%f	loop	trace
cpp	13.9	3.5	10.5	1.1	37.6	33.4	1.8	1.8
eqn	4.2	17.4	18.0	0.0	56.3	4.2	4.0	2.6
espresso	26.3	8.13	12.9	7.8	29.4	15.5	2.0	1.9
grep	27.4	9.8	10.8	0.4	43.8	7.8	3.0	2.2
more	9.6	13.7	13.8	1.0	60.0	2.4	4.7	3.8
mpla	10.9	6.1	8.5	12.8	53.1	8.7	3.9	2.8
nroff	2.5	9.8	10.6	1.2	71.6	4.1	5.4	3.7
pic	2.0	10.1	10.9	2.0	71.0	3.9	2.0	3.6
tbl	3.4	8.2	9.1	0.5	70.2	8.6	1.9	2.5
wc	9.4	10.9	13.7	0.0	57.3	8.6	6.0	3.3

Table 5.7 Selection according to arc weight.

benchmark	%a	%b	%c	%d	%e	%f	loop	trace
cpp	12.6	1.0	8.0	2.0	43.0	33.4	1.8	2.0
eqn	19.7	1.0	2.2	2.2	73.1	1.8	1.3	3.1
espresso	14.9	5.7	9.7	18.6	40.3	10.8	2.1	2.2
grep	17.8	2.1	2.9	0.9	68.0	8.5	4.9	3.4
more	20.1	1.6	2.1	0.7	75.1	0.3	3.0	4.4
mpla	12.3	4.7	7.4	12.8	54.2	8.7	3.9	2.8
nroff	5.1	0.8	1.7	1.8	87.1	3.6	6.7	5.1
pic	9.4	1.5	4.1	1.3	79.3	4.4	5.6	3.9
tbl	6.5	0.8	1.8	1.5	81.3	8.1	1.5	2.7
wc	7.0	0.4	2.8	2.4	78.8	8.6	7.0	5.7

Table 5.8 Minimum branch probability = 60%.

benchmark	%a	%b	%c	%d	%e	%f	loop	trace
cpp	33.4	1.0	2.1	2.0	35.5	25.9	1.7	1.6
eqn	21.7	0.7	1.9	0.9	73.0	1.8	1.4	2.9
espresso	23.0	4.7	7.6	16.2	36.7	11.9	1.9	1.8
grep	19.2	1.7	2.4	0.4	67.6	8.7	4.9	3.3
more	20.1	1.6	2.1	0.7	75.1	0.3	3.0	4.4
mpla	29.0	0.9	2.3	12.7	49.4	5.8	3.2	2.1
nroff	5.7	0.7	1.4	1.8	86.8	3.6	6.6	5.0
pic	13.0	1.3	3.1	1.0	78.0	3.6	2.0	3.2
tbl	7.4	0.7	1.5	1.6	80.8	8.1	1.5	2.7
wc	7.0	0.4	2.8	2.4	78.8	8.6	7.0	5.7

Table 5.9 Minimum branch probability = 70%.

benchmark	%a	%b	%c	%d	%e	%f	loop	trace
cpp	35.8	0.9	1.5	1.8	34.1	25.9	1.7	1.6
eqn	23.7	0.5	1.4	0.8	71.9	1.7	1.3	2.7
espresso	56.6	1.5	2.2	8.7	20.2	10.9	1.9	1.6
grep	2.0	1.6	2.4	0.0	67.5	8.7	4.9	3.2
more	20.2	1.6	2.1	0.7	75.1	0.3	3.0	4.4
mpla	29.0	0.9	2.3	12.7	49.4	5.8	3.2	2.1
nroff	5.9	0.7	1.4	1.8	86.7	3.6	6.6	5.0
pic	15.0	1.1	2.6	1.0	76.9	3.5	1.9	2.8
tbl	9.1	0.7	1.1	1.3	79.7	8.1	1.5	2.6
wc	7.0	0.4	2.8	2.4	78.8	8.6	7.0	5.7

Table 5.10 Minimum branch probability = 80%.

benchmark	%a	%b	%c	%d	%e	%f	loop	trace
cpp	40.5	0.6	1.1	1.44	31.0	25.3	1.7	1.5
eqn	26.9	0.1	0.9	0.77	69.7	1.7	1.3	2.4
espresso	67.5	0.8	0.8	5.18	15.7	10.1	1.7	1.4
grep	19.9	1.6	2.4	0.02	67.5	8.7	4.9	3.2
more	20.2	1.6	2.1	0.74	75.1	0.3	3.0	4.4
mpla	32.9	0.8	1.4	12.70	46.5	5.8	3.2	2.0
nroff	8.6	0.5	1.3	7.11	79.4	3.1	2.9	4.2
pic	21.1	0.3	1.2	1.74	72.2	3.5	1.9	2.3
tbl	11.3	0.7	1.0	1.10	77.9	8.1	1.5	2.5
wc	7.0	0.4	2.8	2.41	78.8	8.6	7.0	5.7

Table 5.11 Minimum branch probability = 90%.

benchmark	%a	%b	%c	%d	%e	%f	loop	trace
cpp	44.7	0.4	1.0	0.8	28.8	24.3	1.6	1.4
eqn	28.3	0.1	0.7	0.8	68.7	1.5	1.2	2.4
espresso	76.6	0.1	0.2	0.7	14.2	8.3	1.3	1.2
grep	29.5	0.0	0.8	0.0	61.0	8.7	4.9	2.6
more	29.1	0.1	0.6	0.7	69.1	0.3	3.0	3.3
mpla	39.0	0.6	0.8	12.7	42.7	4.1	3.5	1.8
nroff	17.6	0.1	0.4	7.0	73.0	2.0	2.6	3.3
pic	32.5	0.1	0.2	0.5	64.9	1.7	1.7	2.0
tbl	12.7	0.5	0.8	1.1	76.8	8.0	1.5	2.4
wc	58.0	0.0	0.0	0.0	42.0	0.0	0.0	1.7

Table 5.12 Percentage of various branch types.

name	*%conditional*	*%unconditional*	*multiway*
bison	92.8%	6.8%	0.3%
cccp	69.0%	11.0%	19.9%
cmp	80.5%	19.4%	0.0%
compress	90.5%	9.5%	0.0%
eqn	91.5%	7.4%	1.0%
espresso	85.7%	13.5%	0.9%
grep	82.2%	13.3%	4.4%
lex	98.4%	1.5%	0.1%
make	93.7%	6.0%	0.3%
tar	97.2%	2.8%	0.0%
tbl	81.4%	17.8%	0.8%
tee	79.6%	20.4%	0.0%
wc	91.4%	8.6%	0.0%
yacc	97.1%	2.7%	0.2%

Table 5.13 Multiway branch statistics.

name	*%default*	*%hashing*	*%sequence*	*total*	*expected*
bison	74.7%	9.3%	90.7%	6.96	6.22
cccp	92.8%	51.8%	48.2%	3.36	3.15
cmp	0.0%	0.0%	100.0%	3.00	1.00
compress	0.0%	0.0%	100.0%	10.00	1.00
eqn	84.0%	75.3%	24.7%	6.97	6.13
espresso	66.2%	0.0%	100.0%	2.71	2.00
grep	0.0%	0.0%	100.0%	12.00	1.50
lex	35.9%	0.0%	100.0%	12.72	5.42
make	39.7%	0.0%	100.0%	8.60	4.56
tar	0.0%	0.0%	100.0%	6.38	1.26
tbl	22.4%	0.0%	100.0%	11.99	2.94
tee	0.0%	0.0%	100.0%	3.00	1.00
wc	0.0%	0.0%	100.0%	3.00	1.60
yacc	48.5%	0.0%	100.0%	6.28	4.87

Table 5.14 Conditional branch results.

name	*TT*	*TN*	*NT*	*NN*	*hit − ratio*
bison	33.4%	3.1%	5.4%	58.1%	91.5%
cccp	42.5%	6.0%	5.2%	46.3%	88.8%
cmp	0.0%	0.0%	3.1%	96.9%	96.9%
compress	18.3%	2.8%	11.5%	67.4%	85.7%
eqn	14.2%	3.3%	3.3%	79.2%	93.4%
espresso	26.5%	6.3%	9.2%	58.0%	84.5%
grep	8.3%	0.3%	1.7%	89.7%	98.0%
lex	46.6%	1.1%	1.7%	50.6%	97.2%
make	49.8%	3.3%	2.5%	44.4%	94.2%
tar	90.2%	0.7%	0.6%	8.6%	98.8%
tbl	24.5%	1.7%	3.7%	70.1%	94.6%
tee	12.3%	0.1%	12.7%	75.0%	87.3%
wc	10.6%	2.9%	11.2%	75.3%	85.9%
yacc	38.6%	2.0%	8.1%	51.3%	89.9%

Table 5.15 Classical code optimizations.

name	*local*	*global*	*trace*
constant propagation	yes	yes	yes
copy propagation	yes	yes	yes
memory copy propagation	yes	yes	yes
operation combining	yes	yes	yes
common subexpression elimination	yes	yes	yes
redundant load elimination	yes	yes	yes
redundant store elimination	yes	yes	yes
dead code removal	yes	yes	yes
constant folding	yes	no	yes
strength reduction	yes	no	yes
operation cancellation	yes	no	yes
code reordering	yes	no	yes
jump optimization	no	yes	no
dead block elimination	no	yes	no
loop invariant code removal	no	yes	yes
loop induction variable strength reduction	no	yes	yes
loop induction variable elimination	no	yes	yes
loop unrolling	no	yes	yes

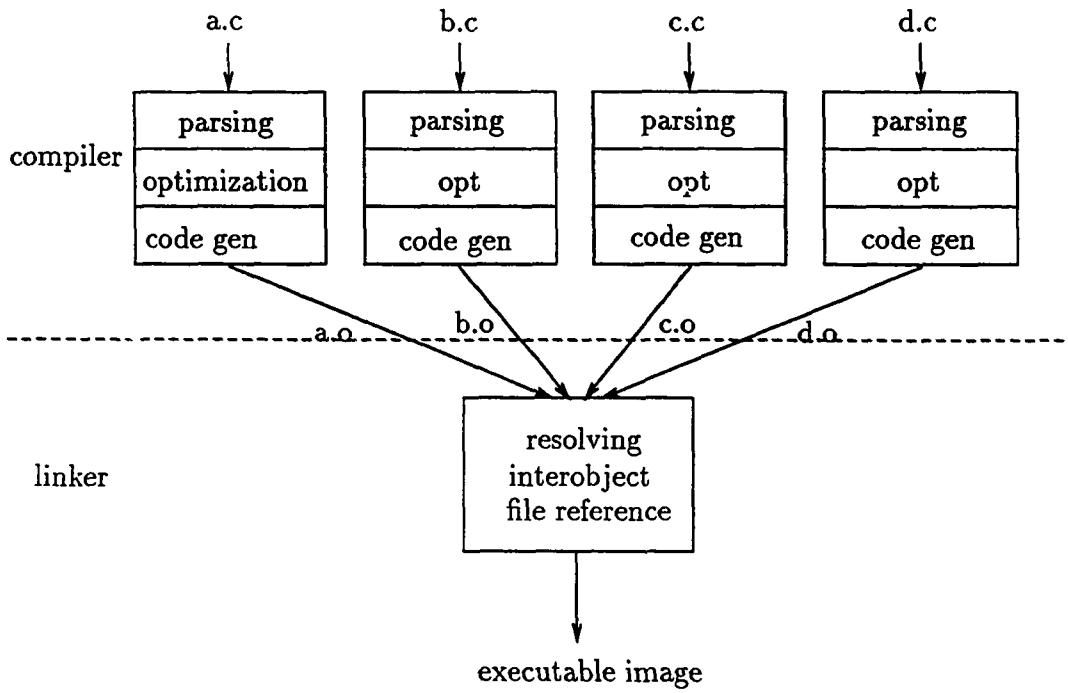


Figure 5.1 Separate compilation paradigm.

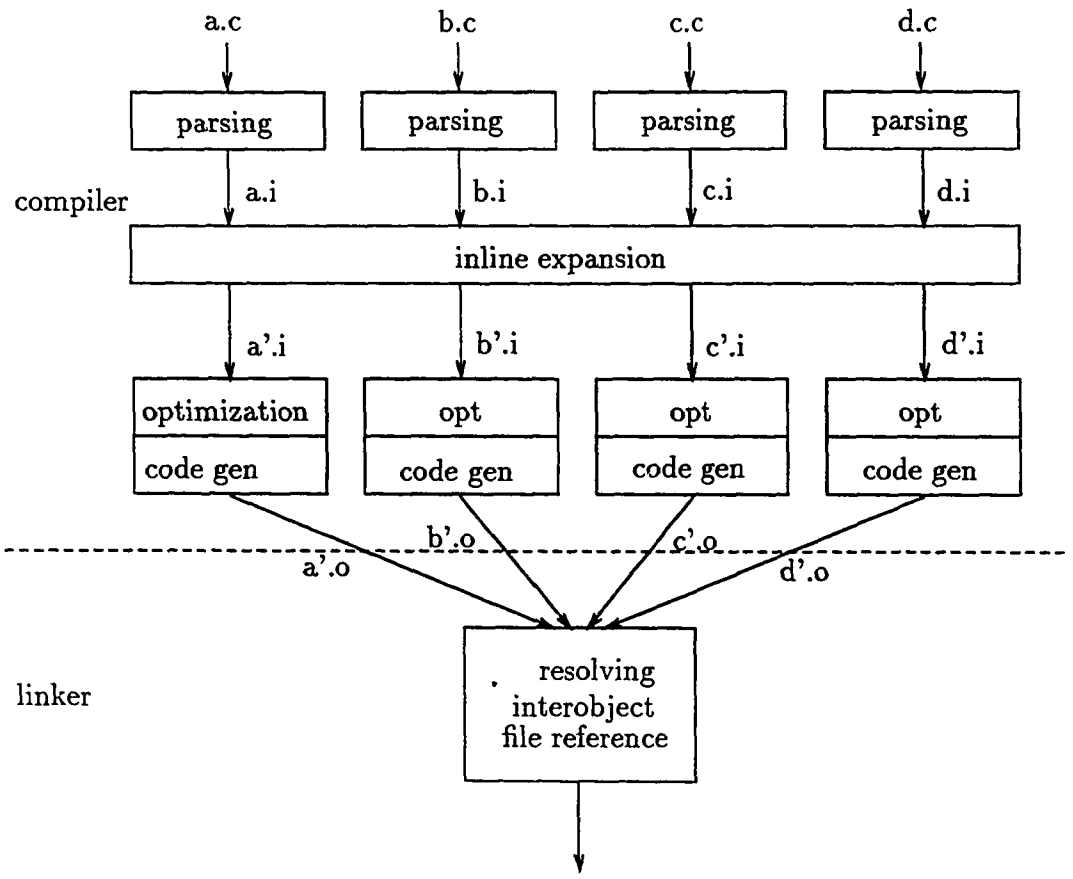


Figure 5.2 Inlining at compile time.

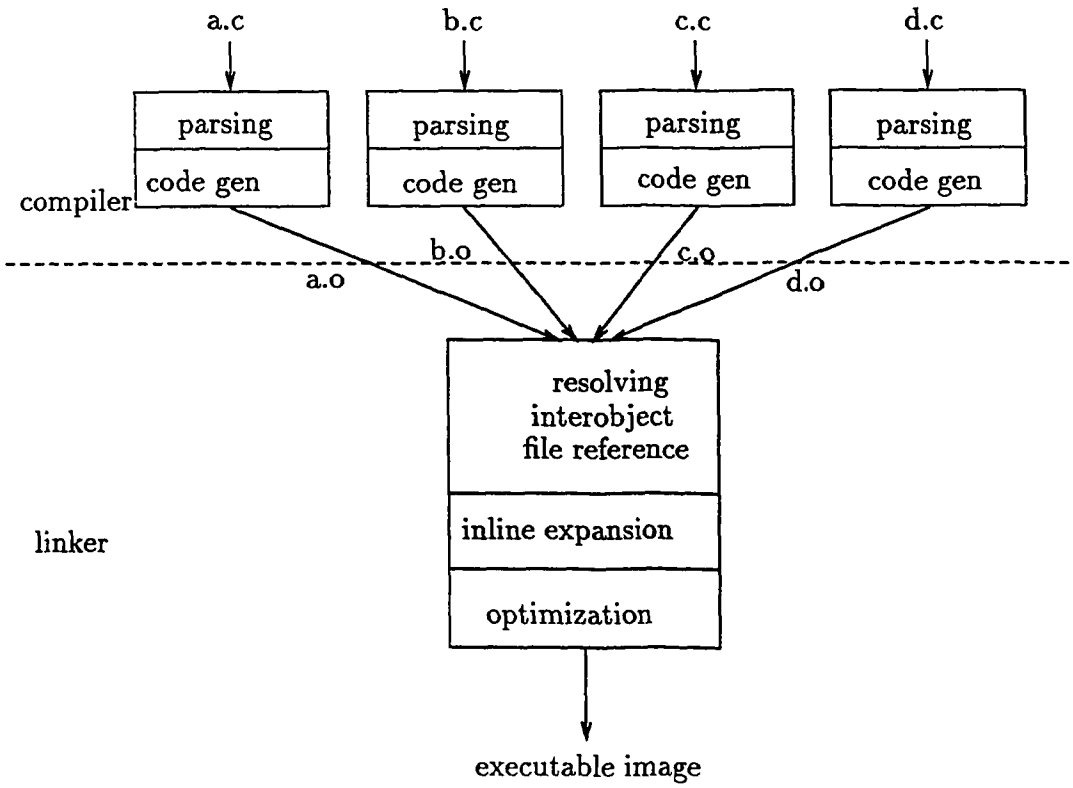


Figure 5.3 Inlining at link time.

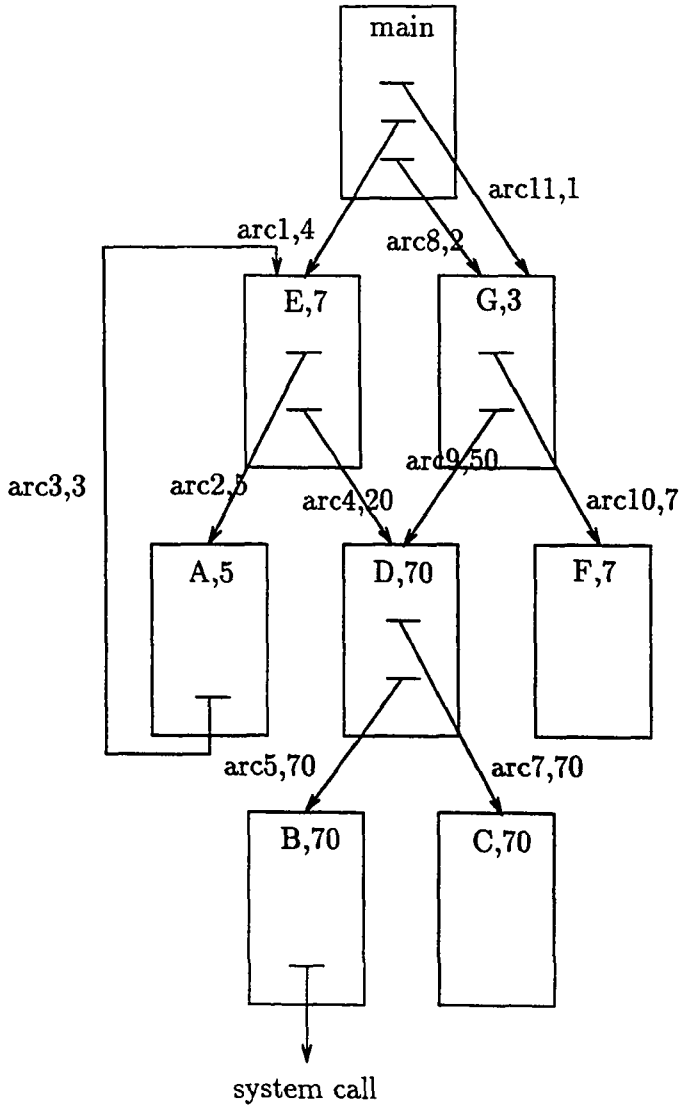


Figure 5.4 A weighted call graph.

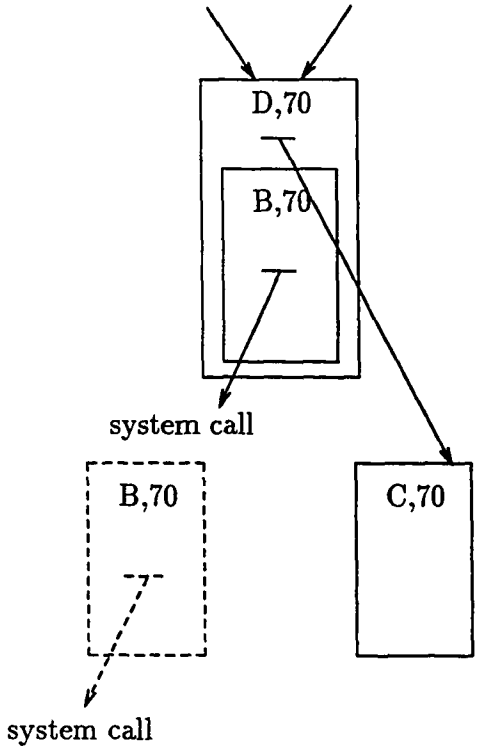


Figure 5.5 An inlining example.

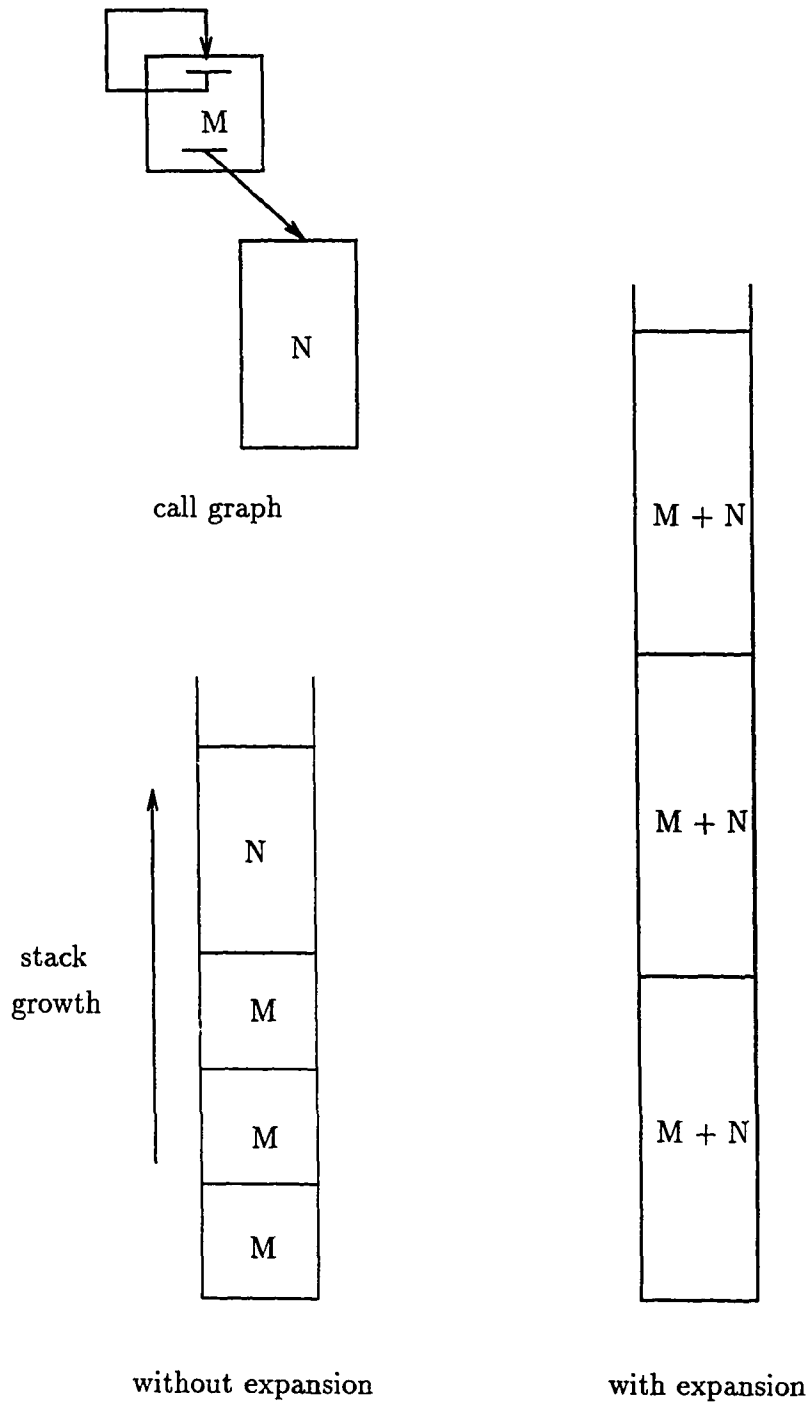


Figure 5.6 Activation stack explosion.

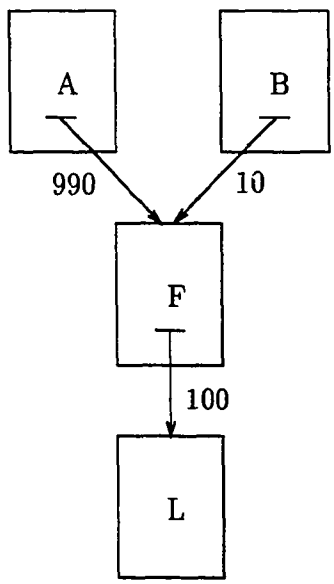


Figure 5.7 An example of restricted inlining.

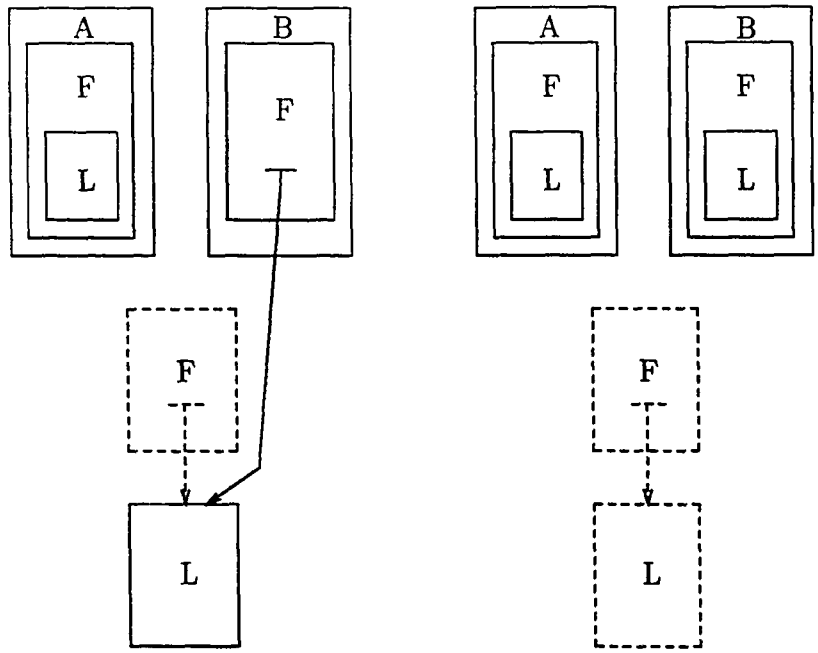


Figure 5.8 Lost opportunity.

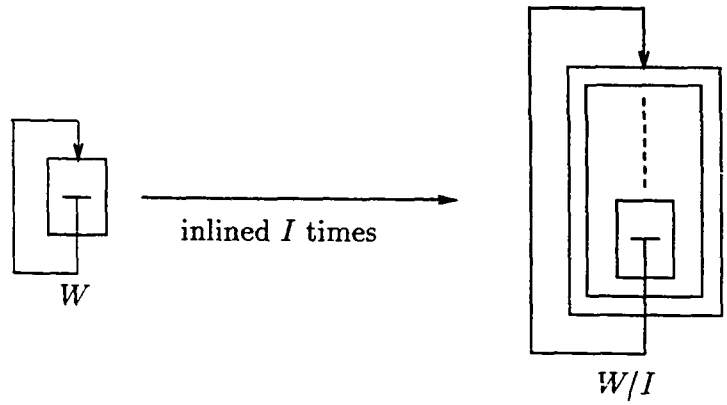


Figure 5.9 Handling single-function recursions.

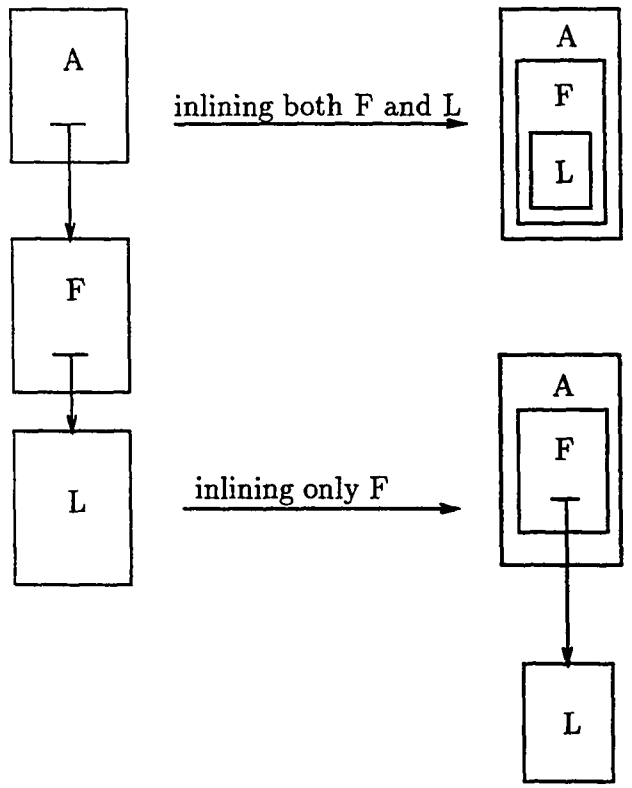


Figure 5.10 Interdependence between code size increase and sequencing.

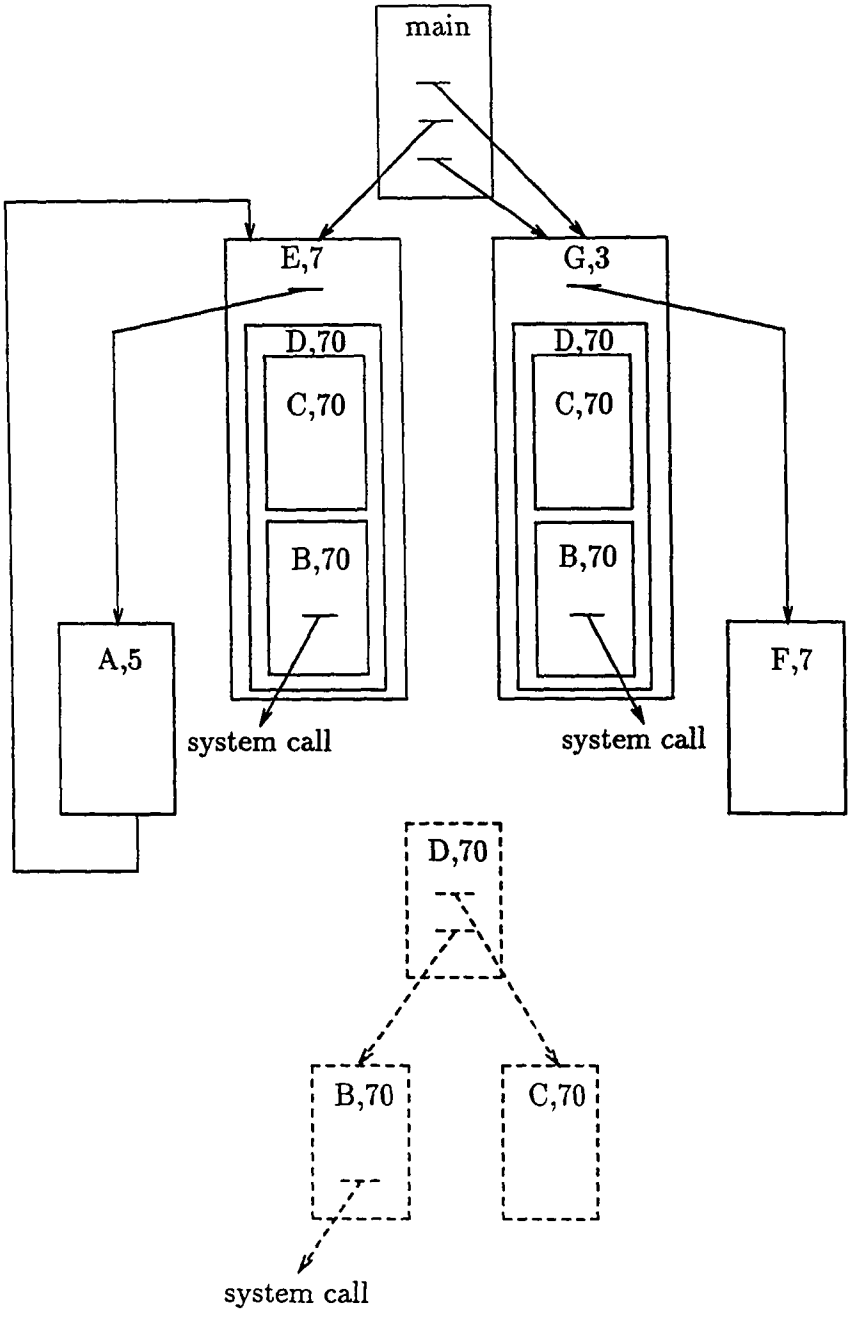


Figure 5.11 Inlining a function before absorbing its callees.

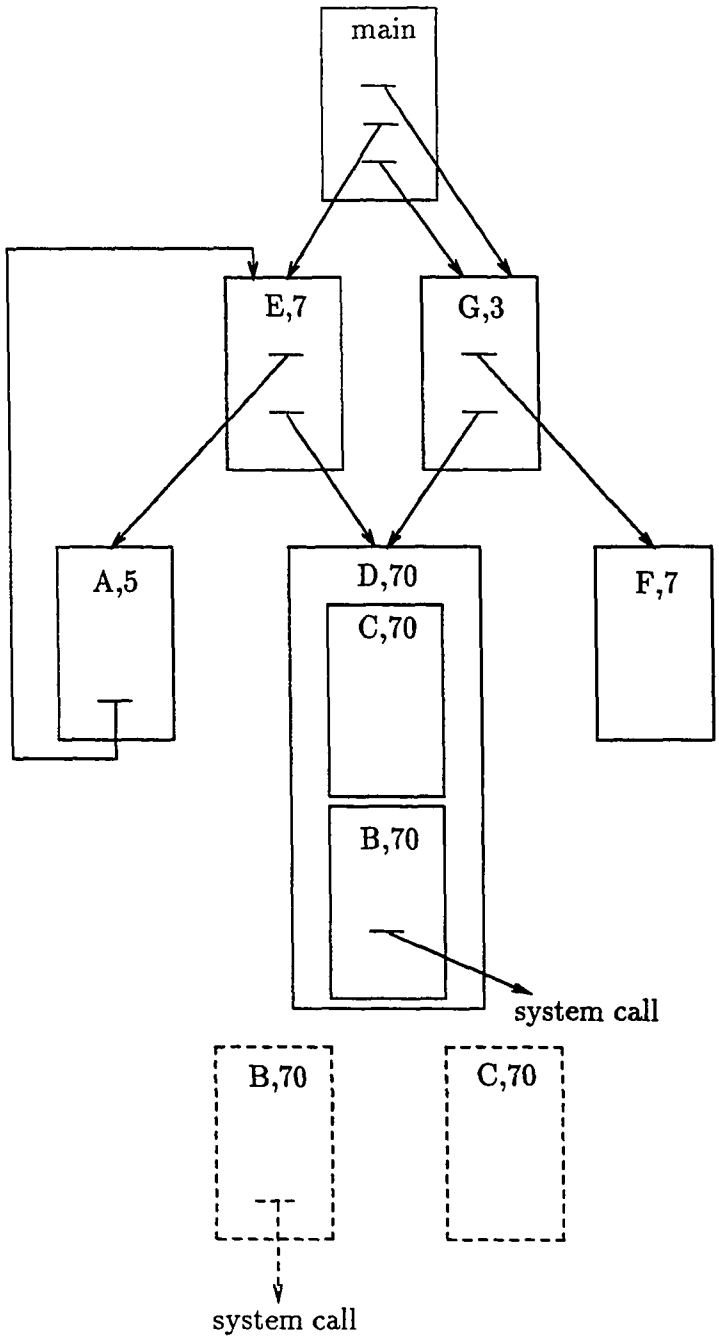


Figure 5.12 Inlining a function after absorbing its callees.

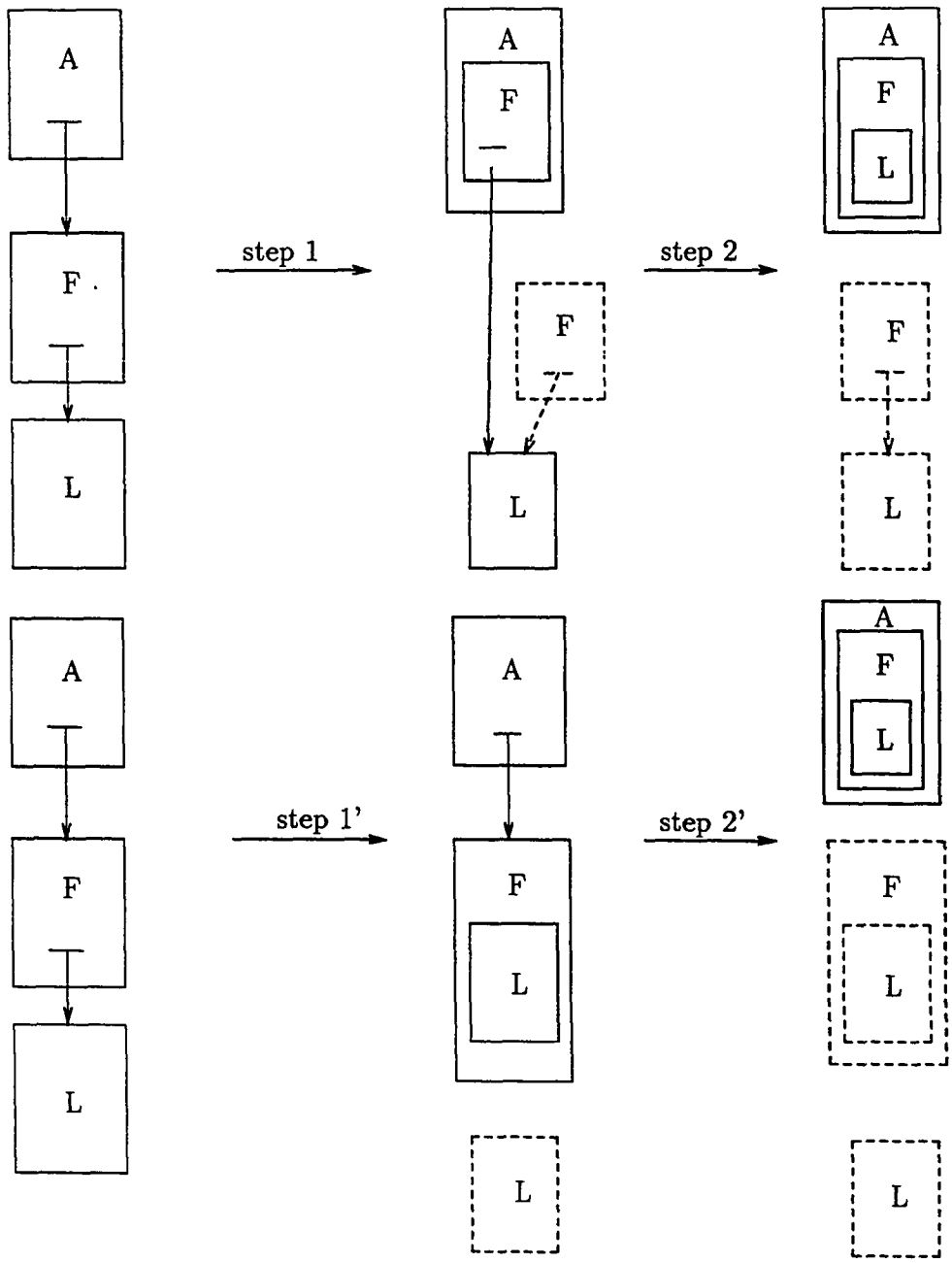


Figure 5.13 Expanding into a single caller.

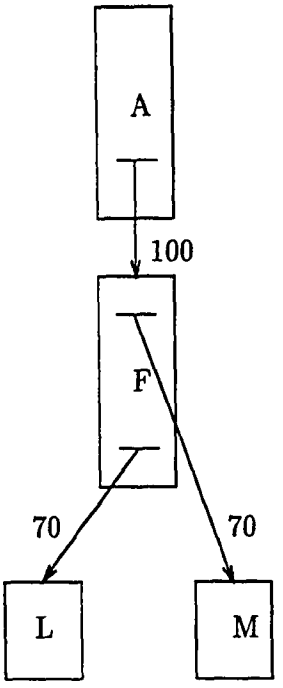


Figure 5.14 Restricted linear sequencing.

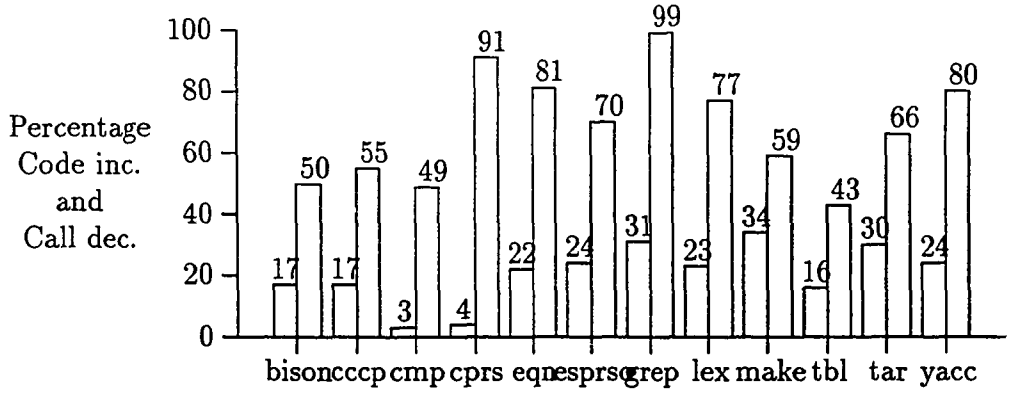


Figure 5.15 Code size increase versus call reduction.

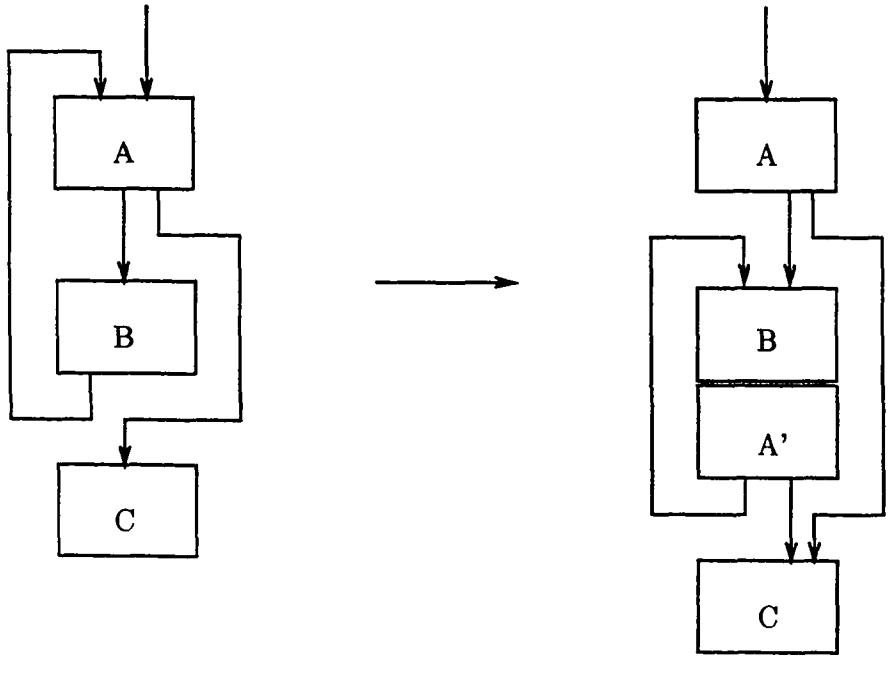


Figure 5.16 Example of jump optimization.

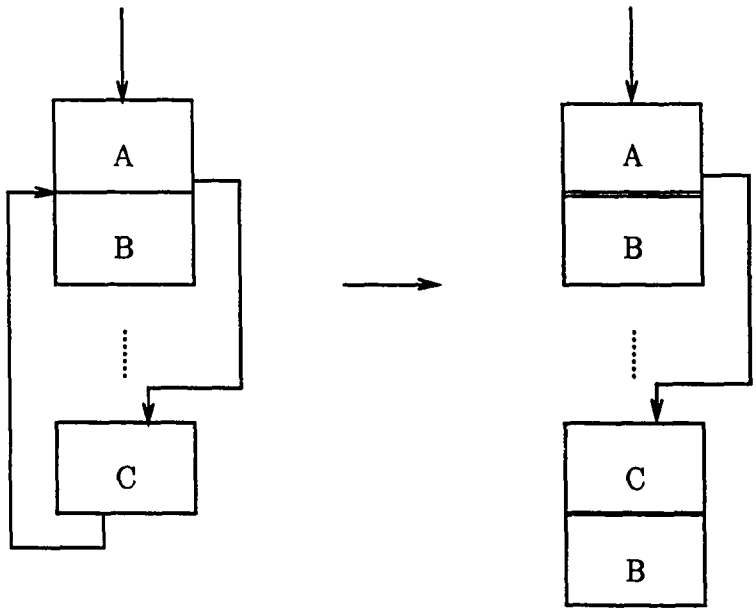


Figure 5.17 Another example of jump optimization.

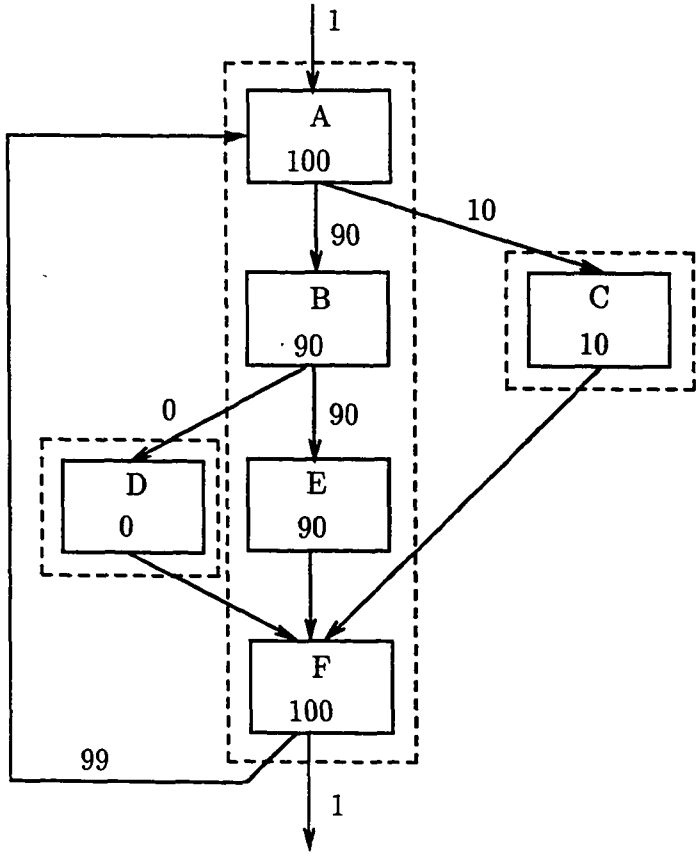


Figure 5.18 A weighted flow graph.

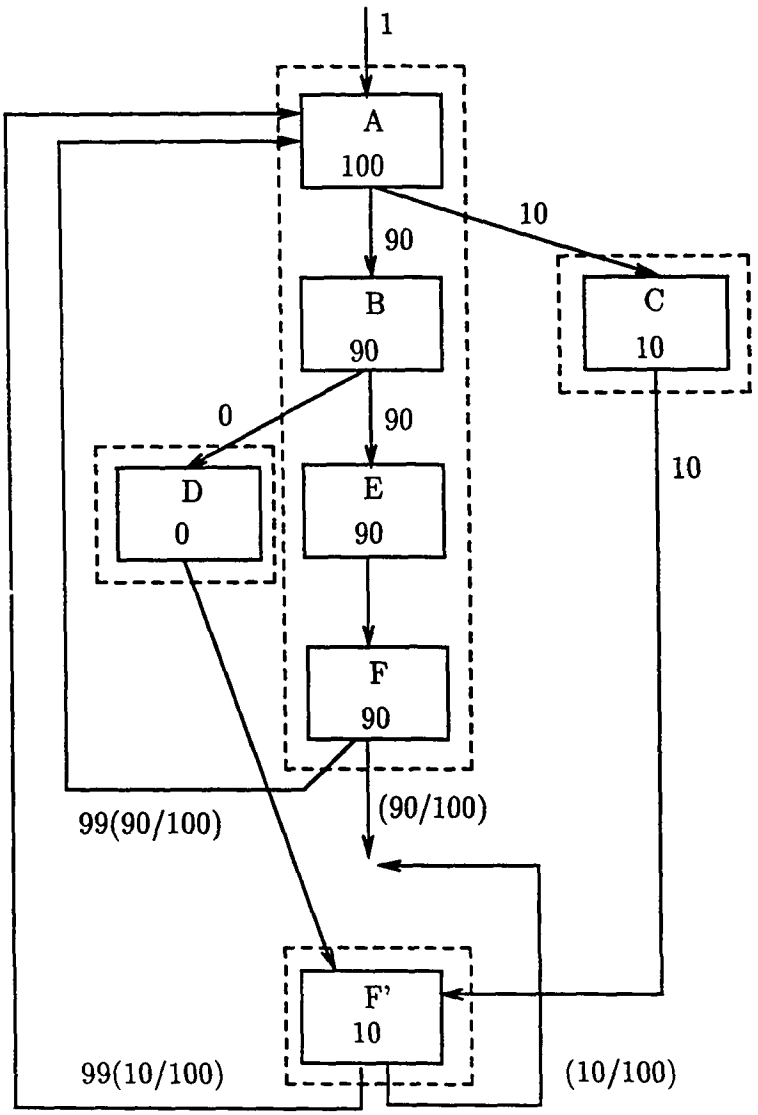


Figure 5.19 Forming super-blocks.

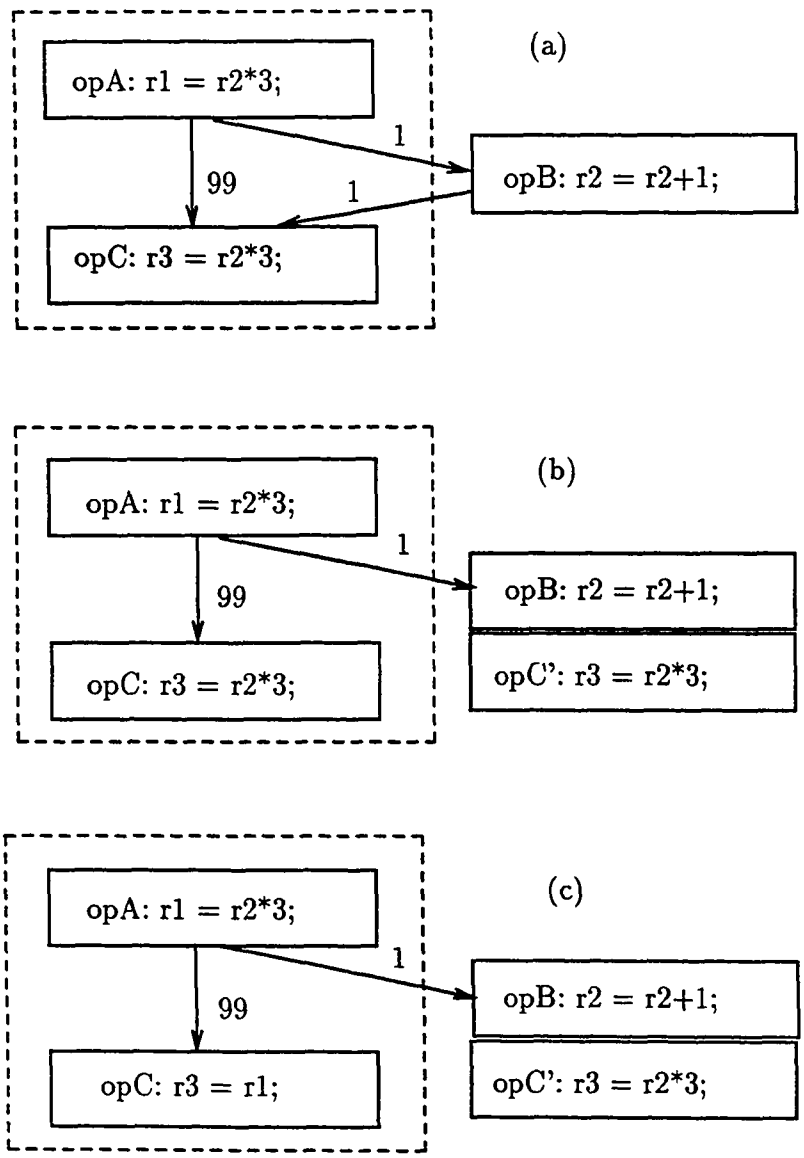


Figure 5.20 An example of common subexpression elimination.

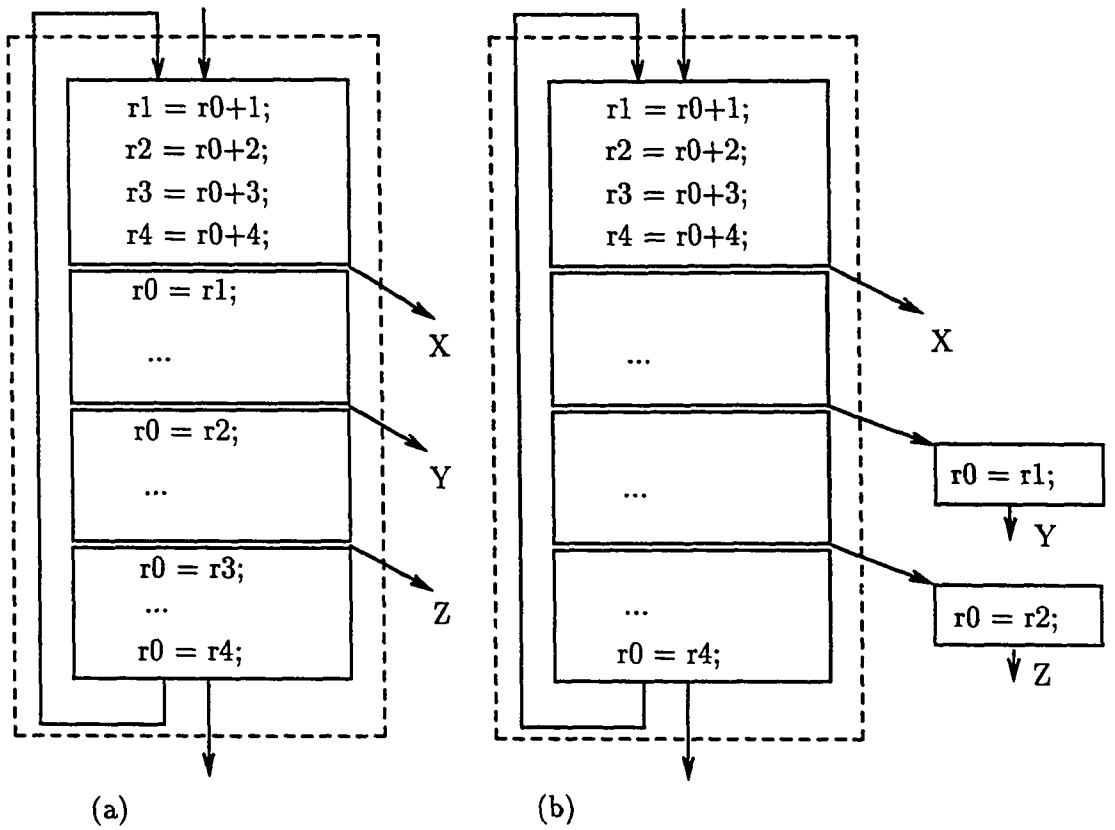


Figure 5.21 An example of dead code removal.

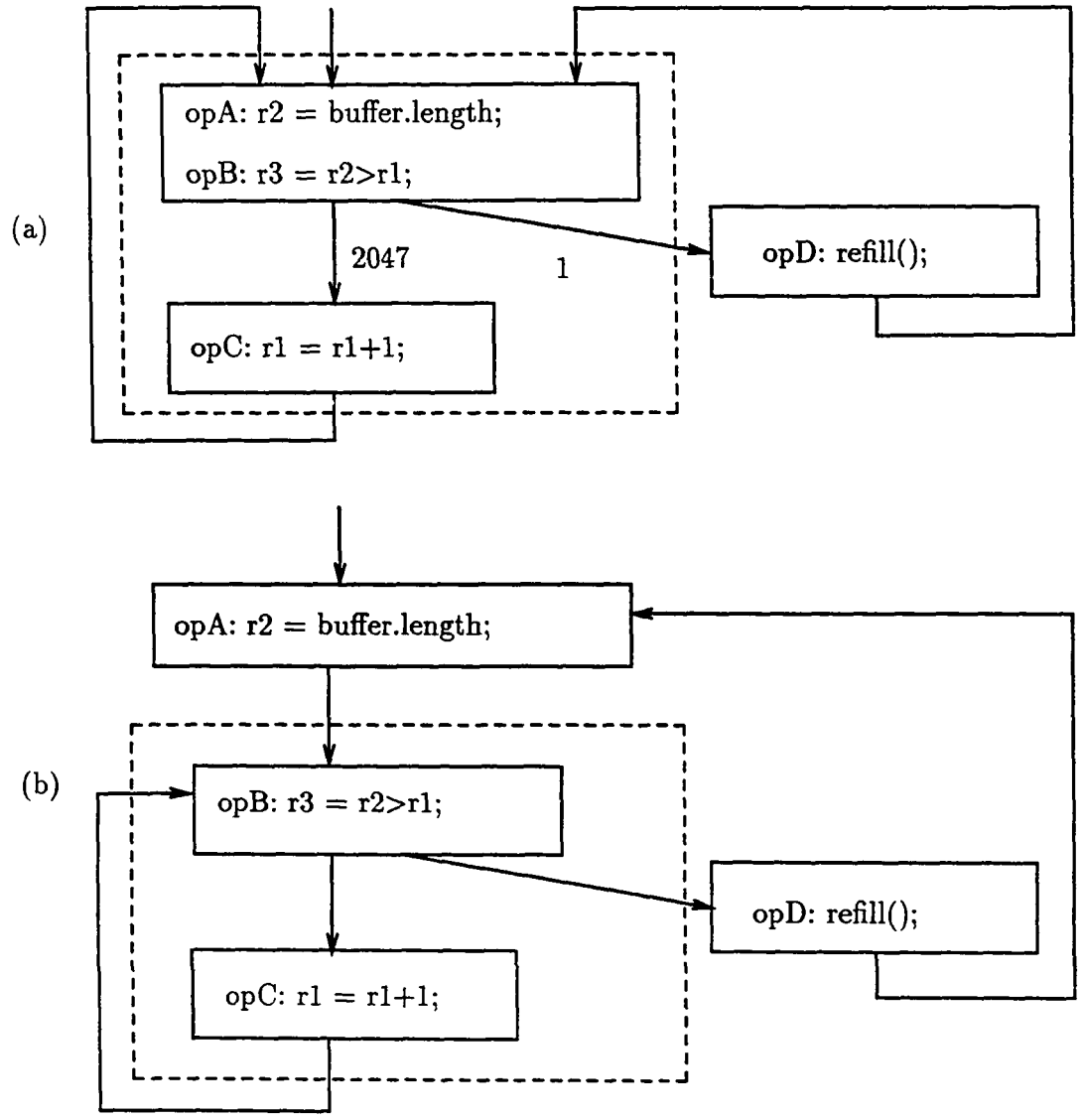


Figure 5.22 An example of loop invariant code removal.

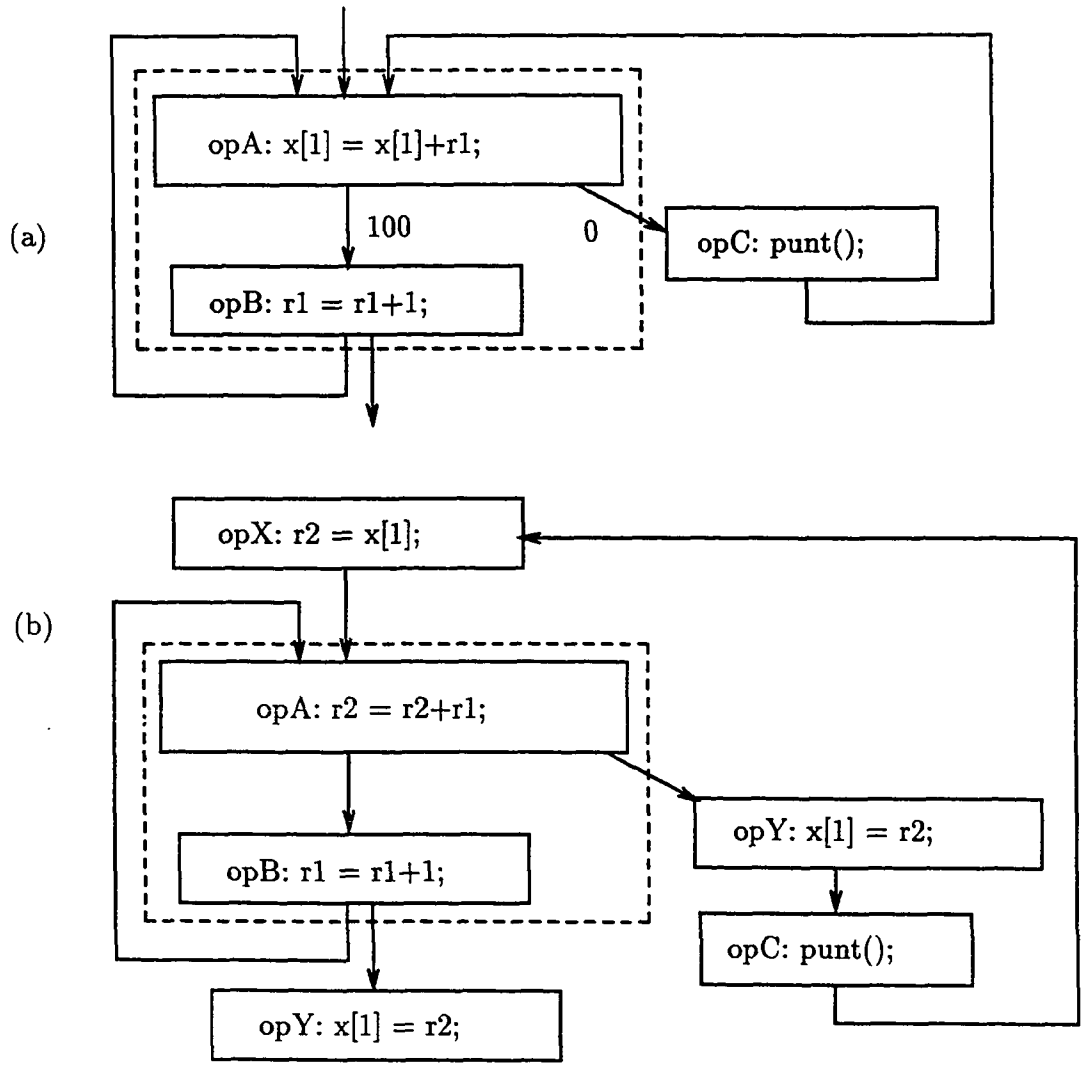


Figure 5.23 An example of super-block global variable migration.

CHAPTER 6

MACHINE-DEPENDENT CODE OPTIMIZATION

A typical IMPACT-I code generator traverses the Lcode data structure four times. For example, the preliminary version of our Sparc code generator has the following four phases:

(1) Replace each Lcode operation with one or more target machine operations.

(2) Extract necessary information for the register allocator.

(3) Modify operand fields according to the result of register allocation.

(4) Generate output.

However, to produce better code, additional passes through the Lcode data structure are made by the code optimizer.

In this chapter, we describe four machine specific optimizations: instruction selection, constant preloading, register allocation, and code scheduling. These optimizations have been implemented in the IMPACT-I C compiler for several target machines. The impact of these code optimizations on the quality of code that the IMPACT-I C compiler produces for the MIPS-R2000 processor is large.

6.1 Instruction Selection

The first step of code generation is to replace a group of Lcode operations by some target machine operations that produce the same effect on the machine state. For the most recent RISC processor architectures, one can almost always identify a one-to-one mapping from every Lcode operation to a target machine operation. Occasionally, an Lcode operation needs to be converted to a sequence of target machine operations. An example is the integer division operation for the Sparc processor, in which the integer division operation is converted into a function call to a highly optimized library routine. On the other hand, several Lcode operations may be equivalent to one target machine operation. For example, memory operations can be eliminated by using complex operand addressing modes in CISC architectures. To derive a good mapping function from the Lcode instruction set to the target machine instruction set, one needs a clear understanding of the cost of each target machine operation.

The code generator should avoid emitting assembly operations that convert to multiple machine operations. For example, an assembler typically supports many types of conditional branch operations (e.g., beq, bne, bgt, bge, blt, ble) for the simplicity of programming. However, the hardware usually supports few branch operations. The assembler macro-expands the unsupported ones to several machine operations.

Because the C programming language does not specify how overflow traps should be handled, the code generator may always emit nontrapping arithmetic operations. The overhead for testing overflow conditions can be eliminated.

Instruction selection may become more important in future machines that support special instructions, such as guarded instructions. For example, accessing an array element $A[X]$ requires a sequence of flow-dependent operations that is shown in the following code segment.

```
r0 = X << 2;          /* suppose sizeof(A[0])=4 */
r1 = load(_A + r0);
```

The direct solution is to provide a hardware function to do the shifting, addition, and then the memory load operation as one machine operation. Shifting an operand by two

bit positions in hardware is not expensive and is unlikely to increase the machine cycle time significantly. Special hardware functions have been widely used in special-purpose processor architectures. For example, hardware accelerators for logic simulation often support bit-field operand modes.

6.2 Constant Preloading

Some constant literals may be encoded into a machine operation. For example, a number of MIPS-R2000 operations allow an integer constant in the second source operand field. Therefore, the code generator should move the constant integer operand field of arithmetic, load, store, and branch operations to the second operand position to take advantage of the short integer addressing mode. For branch operations, changing the order of source operands is legal only if the operation code can be complemented. For example, branch if $0 < r1$ (blt) can be transformed to branch if $r1 > 0$ (bgt).

When a constant literal cannot be encoded into an operand field, an explicit memory load operation is introduced to load the constant into a register. An exception occurs when the processor has special registers that are hardwired to fixed constant values. For example, the MIPS-R2000 processor architecture has a special register whose value is always zero. Therefore, for a MIPS-R2000 processor, preloading zero is never necessary. In the general case, a range is specified for each operand field of each operation. For example, we can specify that the second operand field of an *add* operation can contain an integer in the range [-1024, 1024]. An integer constant that is not in the range needs to be preloaded.

After detecting all constant operands that need to be preloaded, the compiler needs to decide where to insert the preload operation. Suppose that operation X has an operand field that needs to be preloaded, the preload operation must always be executed before operation X from all execution paths, and the register that holds the preloaded constant must not be modified before the execution of operation X. Algorithms that compute the *dominance* relation can be found in [Aho 86]. We want to insert the preload operation

in an infrequently executed basic block that dominates operation X to minimize the execution overhead, and to keep the distance between the preload operation and operation X as short as possible to reduce the lifetime of the register that holds the preloaded value. Often, the two objectives cannot be simultaneously satisfied. For a machine that has an abundant supply of registers (e.g., AMD29K), the preload operations can be introduced as early as at the entry point of a function. For a machine with very few registers, it may be best to preload only constants for operations in loops and to insert preload operations in the loop headers immediately before the loop bodies.

Our constant preloading optimization consists of four steps. The first step is to arrange the source operands to fully utilize the constant literal operand mode of the architecture. The second step is to determine what needs to be preloaded. Because the number of usable registers in MIPS-R2000 is limited, constants are preloaded only in loop preheaders, rather than at entry points of functions. Therefore, the third step is to detect loops and introduce loop preheader basic blocks. A loop preheader basic block is guaranteed to dominate every basic block in the loop. When there are nested loops, we choose a nesting level that has a high iteration count and low loop preheader cost. We also avoid performing constant preloading for large loops in which the number of simultaneously live registers is large. Our decision depends on the loop size, the execution frequencies, and the number of registers that are referenced in the loop. The fourth step is to decide which constant operands should be preloaded. For each constant operand, we calculate the number of times it is referenced in the loop body. The profile information provides an accurate dynamic reference count. Each reference costs one memory operation if the constant is not preloaded. Therefore, the benefit of preloading is (number of references $-$ 1). We preload the most important constants. After constant preloading, nearly every Lcode operation has a one-to-one translation to a MIPS-R2000 machine operation.

Commercial VLIW machines support an elaborate immediate addressing mode [Colwell 87]. A multiple-operation-issue architecture already has sufficient instruction memory bandwidth to support an immediate addressing mode. Providing an immediate addressing

mode is desirable in multiple-operation-issue architectures for two reasons. First, immediate addressing does not increase the length of the critical path, as does loading a constant from memory to a register. Second, registers that are required for constant preloading tend to have long lifetimes and cause more register spilling. With the immediate addressing mode, the constant preloading optimization becomes less significant.

6.3 Register Allocation

Formulations of the register allocation problem can be found in [Chaitin 82] and [Chow 88]. We have implemented a register allocation algorithm based on graph coloring that is similar to Chaitin's original algorithm [Chaitin 82]. Our algorithm can handle both shared and split register files. For example, our register allocator can handle registers in a floating-point coprocessor, and can also handle different register types (e.g., quad-word registers). The algorithm consists of the following steps: 1) construct an interference graph, 2) determine a correct assignment based on the interference graph, and 3) insert spill code where necessary. Determining caller-save and callee-save registers has a large impact on the execution time of our benchmark programs. For leaf-level functions, we want to use the caller-save registers first because they do not need to be saved and restored by leaf-level functions. For top-level functions, we want to use the callee-save registers first because they do not need to be saved and restored around a subroutine call operation. We prefer to put in caller-save registers values that are not live across function calls. We also prefer to put in caller-save registers values that may be easily regenerated, for example, constants and results of simple integer operations, because they do not need to be saved before a subroutine call.

Conventional compilers perform register allocation and assignment prior to code scheduling. This approach has a severe drawback for heavily pipelined and multiple-operation-issue machines. Register allocation can create artificial data dependencies between operations that limit code scheduling. For example, the first two operations of the code segment shown below are independent and their order may be interchanged.

```
r3 = r1 + r2;
r6 = r4 + r5;
r7 = r3 + r6;
```

But if the register allocator assigns r6 and r1 to the same physical register, the first two operations are no longer interchangeable.

```
r3 = r1 + r2;  /* r1 will not be used afterward */
r1 = r4 + r5;
r7 = r3 + r1;
```

Then the code scheduler cannot issue the second operation early, even though r4 and r5 may be available before r1 and r2 become available. We have conducted a study on some numerical kernels and found out that the artificial data dependencies that are introduced by register allocation can degrade the benchmark performance by an average of about 30% for a processor that can issue two operations per cycle [Hwu 88b]. In that paper, we have added a prepass code scheduling to the code generator. After prepass code scheduling, the register allocator can still introduce artificial data dependencies. But the new data dependencies will not severely degrade performance because the postpass code scheduling does not need to change the order of operations that was determined by the prepass code scheduling, except for spill code due to register allocation.

Code scheduling reduces execution time by overlapping the execution of independent expressions. A sideeffect is that it tends to increase register lifetimes and increase the number of simultaneous live registers. Unless properly controlled, prepass code scheduling may cause many registers to spill. Our approach is to schedule operations that release registers (source operands) first. Others have proposed a more intelligent scheme to switch between two scheduling policies, one that minimizes register usage and another that minimizes schedule length, depending on the number of available registers at any given time [Goodman 88].

Constant preloading and register allocation compete for registers. Applying constant preloading before register allocation has the risk of preloading too many constants. Preloaded registers must be kept alive in the duration of the loop where the constants

are used. Because the lifetimes of preloaded constants are long, the decision to preload a constant must be conservative. An integrated constant preloading and register allocation algorithm may be the ideal solution.

6.4 Code Scheduling

Recall our definition of a *super-block* in Section 5.5. Our code scheduling algorithm works on the super-block level. Our approach can be considered as an improvement over trace scheduling [Fisher 81] because branches entering from the middle of traces are eliminated by code duplication. The scope of our scheduler is as large as in trace scheduling, and upward code motion across basic block boundaries is easier in our approach because a super-block can be entered only from the top. Downward code motion across basic block boundaries is unconstrained, but may require copying the operation to the target path.

In Chapter 8, we will describe some code transformation techniques that greatly enlarge the sizes of super-blocks (hence, the scope of static scheduling), and reduce the lengths of critical paths. Classical code optimizations, such as loop invariant code elimination, can also move operations across basic block boundaries. In this chapter, we assume that all code optimizations have been applied and discuss code motion that is due to only code scheduling.

The input to our code scheduler is a function represented as a set of super-blocks and some control flow paths connecting these super-blocks. The code scheduler processes one super-block at a time, neglecting the effects of other super-blocks. The effect of long operation latency across super-block boundaries is very small for our scalar benchmark programs. When operation latency across super-block boundaries becomes an important performance degradation factor, one can start from the most important super-block according to the profile information, and consider the effect of protruding operation latency when scheduling less important super-blocks. We have used this approach in a prototype

code generator [Hwu 88b], but have not yet adopted this approach in the current code scheduler due to its complexity.

The outline of our code scheduling algorithm appears in the following code segment.

```
schedule_function(fn) begin
    compute dataflow information;
    for each super-block sb in fn do
        schedule_super_block(sb);
end
schedule_super_block(sb) begin
    construct an acyclic dependence graph;
    remove redundant dependence arcs;
    apply list scheduling;
end
```

The following subsections describes the major steps of our algorithm.

6.4.1 Dataflow analysis

To move operations across basic block boundaries, we need to know what registers are alive at the basic block boundaries. For each super-block, there is an IN set that specifies the live registers upon entering the super-block. If a variable x belongs to the IN set of a super-block, the value of variable x may be used before it is defined in the super-block. For each exit point of a super-block, there is an OUT set that specifies the live registers upon leaving the super-block through that exit point. Therefore, for each branch operation, there is a corresponding OUT set. If there is a fall-through path, there is also a corresponding OUT set. The standard algorithm for computing the IN and OUT sets can be found in [Aho 86]. Dataflow information is used to decide whether an operation may be moved across a branch operation.

Operation Percolation: Operation percolation refers to moving an operation across a conditional branch operation. For example, the code segment shown below contains three dependent operations.

```
op[1]: if (r0<0) goto L1;
```

```
op[2]: r1 = memory[r0 + 5];
op[3]: if (r1>10) goto L2;
```

The operation op[2] is control-dependent on op[1], and op[3] is flow-dependent on op[2]. However, if $r1$ is not in the OUT set of op[1], then op[2] can be moved above op[1].

```
op[2]: r1 = memory[r0 + 5];
op[1]: if (r0<0) goto L1;    /* r1 not in OUT(op[1]) */
op[3]: if (r1>10) goto L2;
```

When the operation latency of memory load operation is 2 cycles, the original code will have an idle cycle after issuing op[2]. The percolated version completely hides the memory load latency by executing op[1] after op[2].

Code percolation repeatedly exchanges the order of two operations until a good schedule is derived. Suppose that *op*1 precedes *op*2 in the strictly sequential execution order; *op*2 can be scheduled before *op*1 if and only if

(1) *op*2 is not data-dependent on *op*1,

(2) if *op*2 is control-dependent on *op*1, *op*2 does not modify any register that is in the OUT set of the taken path of *op*1, and

(3) if *op*1 is a branch, *op*2 does not cause exceptions.

The first condition means that *op*1 and *op*2 are independent operations. The second condition guarantees that *op*2 will not corrupt the variables that are essential in case *op*1 redirects the control flow. The third condition prevents *op*2 from affecting the system state when *op*1 redirects the control flow.

Most integer operations, except division and remaindering operations, can be considered to be $safe$ operations that do not cause exceptions. On systems that do not cause a trap upon memory access violation, memory load operations can also be considered to be $safe$. Some machines write exception flags to the destination registers and handle exception only until the register is used. For these machines, $unsafe$ operations can also be moved above conditional branches. A code scheduling scheme that does not allow

moving *unsafe* operations above a branch operation is called a **restricted code percolation** scheme. A code scheduling scheme that does allow moving *unsafe* operations above a branch operation is called a **general code percolation** scheme.

Speculative Execution: To further increase the freedom of code motion, the second condition can also be neglected if the machine can automatically convert *op*2 to a no-op when *op*1 redirects control flow. We call this type of hardware support *speculative execution.* The instruction set has to be changed to attach to each operation a tag that specifies the number of branches that have been percolated with this operation. For example,

```
if (r0==0) goto L1;   if (r0==1) goto L2;
r1 = memory[r0 + 24];
if (r1<0) goto L4;
```

can be transformed to

```
r1 = memory[r0 + 24];  (tag=2)
if (r0==0) goto L1;   if (r0==1) goto L2;   if (r1<0) goto L4;
```

Since the tag field of the memory load operation contains the value 2, if the next two conditional branches must redirect control flow, then the original value of r1 must be restored.

Speculative execution has been proposed by Smith, Lam, and Horowitz [Smith 90]. Their method, which they called *boosting*, allows operations to be moved above one branch operation. We compare the performance of this scheme to that of restricted and general code percolation in Chapter 8. We show that the benefit of speculative execution is insignificant beyond general code percolation.

6.4.2 Dependence graph

In addition to ensuring that all code motions are legal, the code scheduling algorithm also needs to consider the allocation of function units. Therefore, code percolation and speculative execution cannot be implemented as separate passes of the code scheduling

algorithm. Our approach is to implement a good code compaction algorithm based on a dependence graph, and to implement optimizations, such as code percolation, as filters that transform the dependence graph prior to code compaction.

Treating a super-block as a linear sequence of operations, we can construct an acyclic dependence graph. The algorithm is shown in the following code segment:

```
construct_dependence_graph(sb) begin
    for i = 1 .. sizeof(sb) do begin
        /* add flow-dependence arcs */
        for each source operand of op[i] do begin
            if (there is at least one operation in op[1..i-1]
                that writes to the source operand) then begin
                op[i] is flow-dependent on the last operation
                  in op[1..i-1] that writes the source operand;
            end
        end
        /* add anti-dependence arcs */
        for j = 1 .. i-1 do begin
             if (op[j] uses the destination operand of op[i]) then begin
                 if (there is no operation in op[j+1..i-1] that
                     writes to that operand) then begin
                     op[i] is anti-dependent on op[j];
                 end
             end
        end
        /* add output-dependence arcs */
        for j = 1 .. i-1 do begin
             if (op[j] writes the destination operand of op[i]) then begin
                 if (there is no operation in op[j+1..i-1] that
                     writes to that operand) then begin
                     op[i] is output-dependent on op[j];
                 end
             end
        end
        /* add control-dependence arcs */
        if (op[i] is a branch operation) then begin
            op[i+1..sizeof(sb)] are control-dependent on op[i];
        end
    end
end
```

To compute memory dependencies conservatively, we assume that all memory operations refer to the same memory variable. Several optimizations can improve the accuracy of the dependence graph, e.g., memory disambiguation.

6.4.3 Dependence arc optimization

Each dependence arc is marked with a *distance* attribute, designating the minimum issue distance between two dependent operations. For example, if *op*2 is flow-dependent on *op*1 and the operation latency of *op*1 is 2 cycles, then the distance of the flow-dependence arc is 2. The code scheduler has to schedule *op*2 at least 2 cycles after *op*1. Typically, the flow-dependence arc distance is the operation latency of the source operation, and all other dependence arcs have a distance of one. Several optimizations can be made to reduce the dependence arc distances.

First, if the machine allows multiple branch operations to be issued per cycle and to provide squashing capability, the distance of control dependence arcs can be reduced to zero. For example,

```
if (r0>0) goto L1;
r1 = r1 - 1;
if (r2<10) goto L2;
```

can be issued at the same cycle as

```
if (r0>0) goto L1;  r1 = r1 - 1;  if (r2<10) goto L2;
```

An implicit ordering of operations is assumed within an instruction word. The first taken branch operation squashes all subsequent operations in the same instruction word. In the above example, if ($r0 > 0$), the next two operations become no-ops.

Second, if the machine allows the compiler to decide what operations are always fetched and decoded as an instruction word, the distance of anti-dependence arcs can be reduced to zero. For example,

```
r0 = r1 - 1;
r1 = 0;
```

can be packed into an instruction word as

```
r0 = r1 - 1;    r1 = 0;
```

The hardware has to guarantee that all source operands of all operations in the instruction word are acquired before any of the operations are allowed to modify the machine state.

Third, if the hardware assumes an implicit ordering of operations when more than one operation of an instruction can write to the same location, the distance of output-dependence arcs can be reduced to zero. For example,

```
r0 = r1;
if (r1>0) goto L1;
r0 = -r1;
```

can be packed into an instruction word as

```
r0 = r1;    if (r1>0) goto L1;    r0 = -r1;
```

If the branch is not taken, the third operation determines the final value of r0. If the branch is taken, the third operation is squashed and the first operation determines the final value of r0.

Fourth, operation percolation and speculative execution can be implemented by removing certain control-dependence arcs. For example,

```
if (r0>0) goto L1;
if (r0==0) goto L2;
r1 = r1 - 5;
```

has three control-dependence arcs. To allow moving the third operation to above the two branch operations, we simply eliminate the control-dependence arc between the first operation and the third operation, and between the second operation and the third operation.

Fifth, some flow-dependence arcs can also be optimized. For example,

```
r0 = r0 - 1;
if (r0==0) goto L1;
```

can be packed into an instruction word as

```
r0 = r0 - 1;  if (r0==1) goto L1;
```

This can be accomplished by converting the dependence arc distance to zero, and let the code compaction algorithm decide whether to pack the two operations into the same instruction word.

Sixth, memory dependencies can be eliminated with information from user pragmas and automatic memory disambiguation tools. We have implemented a tool that performs limited resolution of memory addresses. Each memory address contains a base address part and an offset field. If two memory operations have different base address parts and the base addresses are data labels (global variables), we can assume the two memory operations to be independent operations. If two memory operations have different base address parts and the two base addresses are pointers (an address stored in a register), without pointer analysis, we have to assume that the two memory operations are dependent, unless explicitly declared independent by a user assertion. Some pointer accesses can be distinguished from global variable accesses if the compiler determines that the global variable cannot possibly be accessed through a pointer in the function. If two memory operations have the same base address, we can determine whether the two memory operations access overlapping memory regions from the offsets fields. If the two memory operations access from nonoverlapping memory regions, we can assume that they are independent operations. If we cannot determine whether or not the two memory operations access from nonoverlapping memory regions, the two operations are assumed to be dependent.

6.4.4 List scheduling

The input to the code compaction algorithm is an acyclic dependence graph, where each node is an operation and each arc specifies the minimum distance between the issue time of two dependent operations. Our algorithm is a variant of the popular list scheduling algorithm. The first extension is that we need to handle dependence arcs of

zero distance. The second extension is that our algorithm has to allocate function unit resources.

For each operation, we introduce two attribute fields: *issue_time* and *priority*. The code compaction algorithm will assign the *issue_time* field of each operation to a positive integer value. Operations with equal *issue_time* value are packed into the same instruction word. A special function *issue_urgency*() determines the importance of scheduling each operation. For example, operations that belong to some critical paths are assigned high *priority* because they must be scheduled as soon as possible. We also need to introduce order sets of operations. One such set is called the *ready_set* which contains a set of operations that may be scheduled for execution at a particular time. The operations in *ready_set* are sorted according to their priorities. The *ready_set* is recomputed for every time increment. A special function *ready*() is defined to determine whether an operation can be scheduled at a given time.

In Chapter 3, we introduced the machine model. Each instruction template contains an order set of operation slots. Each operation slot can contain one operation of a given type. For example, for a processor that may issue two operations per cycle, one instruction template may contain a integer ALU operation slot and a floating-point ALU operation slot. The first operation slot can be filled by any integer ALU operation, such as an integer addition or subtraction operation. The second operation slot can be filled by any floating-point ALU operation, such as a double-precision multiplication operation.

The code compaction algorithm is shown in the following code segment:

```
schedule_graph() begin
    /* initailization */
    for each operation op do
        issue_time(op) = 0;
    /* compute (static) priority */
    for each operation op do
        priority(op) = issue_urgency(op);
    /* scheduling */
    time = 0;
    while (there is unscheduled operation) do begin
        time = time + 1;
        /* initialization */
```

```
        for each instruction template tp do
            tp = {};
        /* determine operation status */
        ready_set = {};
        for each unscheduled operation op do
            if (ready(op)) then
                ready_set = ready_set + {op};
        /* pack operations into instruction templates */
        for each operation op in ready_set in priority order do begin
            for each instruction template tp do begin
                if (op can be packed into tp) then
                    tp = tp + {op};
            end
        end
        /* select the best instruction template */
        for each instruction template tp do
            priority(tp) = sum of priority of all operations in tp;
        best = the instruction template with the highest priority(tp);
        /* schedule selected operations */
        for each operation op in best do
            issue_time(op) = time;
    end
end
```

An operation is *ready* to be scheduled when all of its dependencies have been fulfilled. For example, if operation *op*2 is flow-dependent only on operation *op*1 and the operation latency of *op*1 is 1 cycle, then *op*2 can be issued at any time after *op*1 has been scheduled. A special case occurs when the dependence distance equals to zero. For example, if operation *op*2 is only anti-dependent on operation *op*1 and the anti-dependence distance is zero, then *op*2 can be packed to an instruction template *tp* if *op*1 has been scheduled at an earlier time, or if *op*1 has been packed into the instruction template *tp*.

Our approach is to compute the *issue_urgency* of operations statically, that is, to compute once before scheduling all operations. An alternative is to recompute the *issue_urgency* of unscheduled operations at each time step. We have chosen the static approach because *issue_urgency*() is a fairly complex function, and we cannot afford to recompute per each time step.

The objective of our scheduling algorithm is not to produce the shortest schedule for the super-block, due to branch operations that may redirect control flow away from the middle of a super-block. The objective is to minimize the average execution time of the super-block. For example, given the following code segment,

```
....
br to L0 if (cc0)    /* section S0 */
....
br to L1 if (cc1)    /* section S1 */
....
br to L2 if (cc2)    /* section S2 */
....
```

the objective is to minimize the function $(length(Si) * weight(Si))$, for $i = 1...N$, where $length(Si)$ is the length of the schedule for section Si, $weight(Si)$ is the profiled execution count of the section Si, and N is the number of sections that are separated by branch operations in the super-block.

Our *issue_urgency* function considers the following factors:

(1) Section weight: operations should be moved to an earlier section if and only if the schedule of the earlier section is not prolonged.

(2) Latest issue time: operations on critical paths should be scheduled early.

(3) Register liveness: operations that release registers should be scheduled early.

(4) Uncovering: operations that enable more unscheduled operations should be scheduled early.

(5) Function unit resource: operations that compete for a limited function unit should be scheduled early.

CHAPTER 7

MULTIPLE-INSTRUCTION-ISSUE CODE OPTIMIZATION

Previous works on concurrency exploitation of C integer programs have reported very low speedup [Jouppi 89a], [Smith 90]. Assuming a uniform unit-time operation latency and infinite resource, Jouppi has reported an execution rate of about 1.6 operations per cycle for *yacc*, a parser generator program [Jouppi 89a]. Smith, Lam, and Horowitz have reported an execution rate of about 1.4 (1.63/1.18) operations per cycle for a machine that can fetch 4 operations per cycle and can boost operations above one branch operation [Smith 90].

We have identified two limiting factors that prevent the static code scheduler from producing a parallel schedule. First, the scope of scheduling is usually small. The average basic block size is only about 4 operations. Even with trace selection optimization, a trace typically contains 12 to 16 operations [Chang 88]. To achieve an execution rate that is better than 2 operations per cycle, the schedule must be no more than 6 cycles. It is unlikely to find that level of instruction parallelism in a trace. Second, there is generally a long critical path in a trace. It is typical to see code segments that load data into registers, perform some computations on the registers, and then store the result back into the memory. Such code segments are dominated by (essential) flow-dependencies that cannot be reduced.

The first part of this chapter describes several code transformation techniques that we have implemented to enlarge the scope of code scheduling. Even if operations in a trace are inherently sequential, by merging several traces together, the code generator may find more parallelism. The second part of this chapter describes several code transformation techniques that we have implemented to reduce the length of a critical path.

7.1 Expanding the Scope of Static Code Scheduling

Recall our definition of a *super-block* in Chapter 5. A super-block is a linear sequence of basic blocks that has a single entry point from the top of the super-block, and potentially multiple exit points. The scope of our scheduler is a super-block. In the following subsections, we explore ways to increase the number of operations in a frequently executed super-block.

The relationship between the sizes of super-blocks and the instruction-level parallelism is not well understood. However, after studying many realistic programs, we strongly believe that enlarging the sizes of super-blocks can lead to more effective static code scheduling. We provide experimental data to support this argument in Chapter 8.

To enlarge a super-block, one can move or copy operations from another super-block. In the case of copying, the instruction space may increase exponentially in the worst case. In a subsection, we discuss how the amount of instruction space expansion can be controlled.

7.1.1 Function inline expansion

A function call is an unconditional jump operation that terminates a super-block. In order to expand a super-block across a function call, one can expand the body of the called function into the caller function. Many leaf-level functions contain only one or two C expressions that can be nicely expanded into the calling super-blocks. Implementation issues of function inline expansion were discussed in Chapter 5.

7.1.2 Instruction placement

The instruction placement optimization selects groups of basic blocks that tend to execute as sequences, reorders basic blocks in a function, and groups basic blocks that tend to execute sequentially into a trace. A trace is later converted into a super-block by code duplication. Implementation issues of instruction placement can be found in Chapter 5. The method to convert a trace into a super-block has been described in Chapters 5 and 6.

The accuracy of the instruction placement optimization affects the sizes of super-blocks. For example, the code segment

```
if (A) B else C; D;
```

is traditionally translated to

```
    A; goto Lc if .false.;
    B;
    goto Ld;
Lc: C;
Ld: D;
```

When the result of A is almost always true, the instruction placement optimization generates the following code:

```
    A; goto Lc if .false.;
    B;
Ld: D;
    ......
Lc: C;
    goto Ld;
```

Because we expect to go from B to D most of the time, the instruction placement optimization eliminates unnecessary jump operations at the end of the basic block B. Because C is less frequently executed, the penalty of adding an unconditional jump operation to C is small. Then, basic blocks A, B, and D form a trace. After converting traces to super-blocks, the code becomes

```
    A; goto Lc if .false.; B; D;
    ......
Lc: C;
Ld': D;   /* a copy of D */
    ......
```

The result is a large super-block that contains three basic blocks (A, B, D). The effect that has been shown in the above example is amplified in realistic C programs, for instruction placement usually identifies several basic blocks that are likely to be executed as a sequence.

7.1.3 Branch expansion

Instruction placement moves basic blocks that do not belong to a trace to the end of the function. To preserve correct control flow, a jump operation is inserted at the end of the basic block. In order to enlarge the sizes of super-blocks that are formed from these basic blocks, we can replace the jump operations by copies of the target basic blocks. If we continue from the example in the previous subsection,

```
La: A; goto Lc if .false.; B; D;
Le: E; goto La if .true.;
Lf: F;
    .....
Lc: C;
    D; goto Le;
```

becomes

```
La: A; goto Lc if .false.; B; D;
Le: E; goto La if .true.;
Lf: F;
    .....
Lc: C; D; E; goto La if .true.;
    goto Lf;
```

The super-block (C, D) absorbs a new element E by appending a copy after the super-block. Because Le is no longer an entry point, the basic block E can be absorbed into the (A, B, D) super-block. The code becomes

```
La: A; goto Lc if .false.; B; D; E; goto La if .true.;
Lf: F;
    .....
Lc: C; D; E; goto La if .true.;
    goto Lf;
```

The result is a larger super-block (A, B, D, E) that forms an inner loop. The off-trace blocks also form a large super-block (C, D, E).

The first type of branch expansion is to replace an unconditional branch operation by the target basic block. The second type of branch expansion is to copy the target basic block of a conditional branch, place the basic block after the conditional branch operation, and then complement the branch condition.

7.1.4 Loop unrolling

When the number of loop iterations is large, we can expand the body of a loop by unrolling it a number of times. For small loops, e.g., at most 10 operations, the IMPACT-I C compiler unrolls the loop bodies 8 or more times. For larger loops or loops containing several branch operations, the IMPACT-I C compiler typically unrolls the loop bodies 4 times. The IMPACT-I C compiler supports three types of loop unrolling. The first type is performed when the compiler can detect that the number of loop iterations is always a multiple of the number of times the loop is unrolled. For example, in the code segment that follows, the left-side loop can be transformed to the right-side loop because the number of iterations is a multiple of 4.

```
for (i=0; i<120; i++)        for (i=0; i<120; i+=4) {
    x[i] = 0;                    x[i] = 0;
                                 x[i+1] = 0;
                                 x[i+2] = 0;
                                 x[i+3] = 0;
                             }
```

This type of loop unrolling is always preferred because it eliminates some induction variable increments and loop boundary check operations, without incurring any overhead cost.

The second type of loop unrolling is performed when the loop body is fairly simple, but it cannot be determined statically that the number of loop iterations is a multiple of the number of times the loop is unrolled. Therefore, a sequential version is kept to execute the odd number of iterations. For example,

```
for (i=0; i<lim; i++)
    x[i] = 0;
```

can be translated to

```
for (i=0; i<(lim%4); i++)
    x[i] = 0;
for (; i<lim; i+=4) {
    x[i] = 0;
    x[i+1] = 0;
    x[i+2] = 0;
    x[i+3] = 0;
}
```

The third type of loop unrolling is performed when the loop body is fairly complex, perhaps with several branch operations. The body of the loop is duplicated completely, including the loop induction variable increment and the loop boundary test operations. Although the number of operations is not reduced, static code scheduling can benefit from a much larger loop body. For example,

```
La: r1 = memory[_x + r2];
    goto Lb if (r1 == 0);
    r2 = r2 + r1;
    goto La if (r2 < 100);
Lb:
```

can be expanded to

```
La: r1 = memory[_x + r2];     /* iteration 1 */
    goto Lb if (r1 == 0);
    r2 = r2 + r1;
    goto Lb if (r2 >= 100);
    r1 = memory[_x + r2];     /* iteration 2 */
    goto Lb if (r1 == 0);
    r2 = r2 + r1;
    goto Lb if (r2 >= 100);
```

```
    r1 = memory[_x + r2];    /* iteration 3 */
    goto Lb if (r1 == 0);
    r2 = r2 + r1;
    goto Lb if (r2 >= 100);
    r1 = memory[_x + r2];    /* iteration 4 */
    goto Lb if (r1 == 0);
    r2 = r2 + r1;
    goto La if (r2 < 100);
Lb:
```

The result of unrolling is a super-block that contains 16 operations for this example. In realistic programs, there are many large super-block loops, after applying the instruction placement and branch expansion optimizations. Applying loop unrolling on these large super-block loops often expands the loop sizes to many tens of operations.

Section 7.2 shows how to eliminate anti-dependencies and output-dependencies in the unrolled loop. Therefore, the execution of several iterations can be interleaved.

7.1.5 Loop peeling

In realistic C programs, some loops iterate very few times on the average. Because loop unrolling may introduce some overhead cost, and the loop structure may impose constraints on register renaming, we propose loop peeling as an alternative approach. For example, if the following loop is iterated about 4 times on the average, we can peel off 4 iterations.

```
La: xxxx
    goto La if .true.;
Lb:
```

is transformed to

```
    xxxx
    goto Lb if .false.;
    xxxx
    goto Lb if .false.;
    xxxx
    goto Lb if .false.;
    xxxx
```

```
    goto La if .true.;
Lb:
    .....
La: xxxx
    goto La if .true.;
    goto Lb;
```

The first four iterations of the loop become sequential code, and can be combined with the original loop preheader basic block into a large super-block. The original loop body has been moved to the end of the function and is rarely executed. This optimization achieves an effect similar to totally unrolling the loop body.

7.1.6 Limiting code expansion

Code expansion can degrade the performance of the instruction cache and incur a high cost to maintain executable code in disk memories. We have employed three strategies in the IMPACT-I C compiler to control the amount of code expansion. The first strategy is to prevent the program from becoming X times bigger than its original size, where X is a fixed number, e.g., 1.5. The second strategy is to prevent the program from becoming larger than a fixed maximum size. Code optimizations that employ these two limits are formulated as follows.

```
Input = program G, maximum limit LIMIT, maximum expansion factor SCALE.
1) Identify all optimization opportunities, {P[i], i=1..n}.
2) orig_size = size(G);
3) i = 0;
4) while ((size(G)<LIMIT) and (size(G)<orig_size*SCALE) and (i<=n)) do
    apply P[i]; i=i+1;
```

We always start from the most profitable optimization and repeatedly apply optimizations until all optimizations have been applied or until the code size is larger than the limit. We have formulated function inline expansion and other code expansion optimizations on this framework.

The third strategy is applicable when we know the worst-case code expansion ratio due to an optimization. The procedure can be formulated as follows:

```
Input = program G, maximum expansion factor SCALE,
    maximum expansion factor K due to code optimization.
1) Partition G to nonoverlapping regions, {R[i], i=1..n},
    that form a complete cover of G.
2) orig_size = size(G);
   max_expansion = (SCALE-1)*orig_size;
3) i = 0; H = {};
4) while (((size(H)*K)<max_expansion) and (i<=n)) do
    H = H + {R[i]}; i=i+1;
5) apply optimization on every region in H;
```

Because the number of branch slots that are allocated for a predict-taken branch is fixed, we have applied the above procedure to decide where to allocate branch slots.

7.2 Reducing the Length of a Critical Path

Expanding the sizes of super-blocks by code copying is effective. However, the reduction of the schedule length is small. Careful analysis of the machine code has pinpointed anti-dependencies, output-dependencies, and memory-dependencies as the primary targets of more code optimizations to reduce the lengths of critical paths.

In the following subsections, we describe several code transformation techniques that have been implemented in the IMPACT-I C compiler. Instead of describing many code patterns that we have observed that benefit from these optimizations, we create an artificial example that is simple enough to convey the basic ideas. The real implementation is substantially more involved and covers many more cases.

7.2.1 Induction variable expansion

Because we enlarge super-blocks by code copying, register anti-dependencies and output-dependencies become explicit. This can be most easily shown by an example.

```
    r2 = 1;
    r3 = _x + 120;
La: r4 = memory[r3];
    r2 = r2 + r4;
```

```
    r3 = r3 - 4;
    goto La if (r3 > _x);
Lb:
```

can be unrolled into

```
    r2 = 1;                 /* accumulator */
    r3 = _x + 120;
La: r4 = memory[r3];        /* first iteration */
    r2 = r2 + r4;
    r3 = r3 - 4;
    goto Lb if (r3 <= _x);
    r4 = memory[r3];        /* second iteration */
    r2 = r2 + r4;
    r3 = r3 - 4;
    goto Lb if (r3 <= _x);
    r4 = memory[r3];        /* third iteration */
    r2 = r2 + r4;
    r3 = r3 - 4;
    goto Lb if (r3 <= _x);
    r4 = memory[r3];        /* fourth iteration */
    r2 = r2 + r4;
    r3 = r3 - 4;
    goto La if (r3 > _x);
Lb:
```

The entire unrolled loop forms a super-block. Without further code optimization, we schedule this loop and find out that the schedule is very conservative due to anti-dependencies and output-dependencies. An obvious problem is with the loop induction variable $r3$, which is incremented and used in each iteration.

We have implemented a special code optimization, *induction variable expansion*, that generates the value of the loop induction variable for each iteration as soon as possible. The previous example is transformed into

```
    r2 = 1;                  /* accumulator */
    r3 = _x + 120;
La: r10 = r3 - 4; r11 = r3 - 8; r12 = r3 - 12; r13 = r3 - 16;
    r4 = memory[r3];         /* first iteration */
    r2 = r2 + r4; r3 = r10;
    goto Lb if (r10 <= _x);
    r4 = memory[r10];        /* second iteration */
```

```
    r2 = r2 + r4; r3 = r11;
    goto Lb if (r11 <= _x);
    r4 = memory[r11];       /* third iteration */
    r2 = r2 + r4; r3 = r12;
    goto Lb if (r12 <= _x);
    r4 = memory[r12];       /* fourth iteration */
    r2 = r2 + r4; r3 = r13;
    goto La if (r13 > _x);
Lb:
```

If the loop induction variable is not used after leaving the loop, it is eliminated from the loop body by dead code elimination.

```
    r2 = 1;                 /* accumulator */
    r3 = _x + 120;
La: r10 = r3 - 4; r11 = r3 - 8; r12 = r3 - 12; r13 = r3 - 16;
    r4 = memory[r3];        /* first iteration */
    r2 = r2 + r4;
    goto Lb if (r10 <= _x);
    r4 = memory[r10];       /* second iteration */
    r2 = r2 + r4;
    goto Lb if (r11 <= _x);
    r4 = memory[r11];       /* third iteration */
    r2 = r2 + r4;
    goto Lb if (r12 <= _x);
    r4 = memory[r12];       /* fourth iteration */
    r2 = r2 + r4;
    r3 = r13;
    goto La if (r13 > _x);
Lb:
```

The induction variable expansion optimization eliminates anti-dependencies and output-dependencies due to the loop variable, and effectively enables the upward code percolation of some computations from a later iteration to a previous iteration.

7.2.2 Register renaming

Very little code motion is possible in the previous example after induction variable expansion. However, a closer look identifies the problem with writing the result of the memory loads into the same register $r4$. The memory load operations of later iterations

cannot be moved up because of anti-dependencies and output-dependencies. A useful technique is to rename registers. Then, the loop becomes,

```
    r2 = 1;                  /* accumulator */
    r3 = _x + 120;
La: r10 = r3 - 4; r11 = r3 - 8; r12 = r3 - 12; r13 = r3 - 16;
    r4 = memory[r3];   r20 = memory[r10];
    r30 = memory[r11]; r40 = memory[r12];
    r2 = r2 + r4;
    goto Lb if (r10 <= _x);
    r2 = r2 + r20;
    goto Lb if (r11 <= _x);
    r2 = r2 + r30;
    goto Lb if (r12 <= _x);
    r2 = r2 + r40;
    r3 = r13;
    goto La if (r13 > _x);
Lb:
```

The first part of the loop body becomes very parallel. Four addition operations can be issued in the first cycle, and four memory load operations can be issued in the second cycle. For a high-issue-rate machine, we have implemented a more aggressive register renaming scheme that introduces additional move operations. The above example becomes

```
    r2 = 1;                  /* accumulator */
    r3 = _x + 120;
La: r10 = r3 - 4; r11 = r3 - 8; r12 = r3 - 12; r13 = r3 - 16;
    r4 = memory[r3];   r20 = memory[r10];
    r30 = memory[r11]; r40 = memory[r12];
    r2 = r2 + r4;
    r100 = r2 + r20;
    r101 = r100 + r30;
    r102 = r102 + r40;
    goto Lb if (r10 <= _x); r2 = r100;
    goto Lb if (r11 <= _x); r2 = r101;
    goto Lb if (r12 <= _x); r2 = r102;
    r3 = r13; goto La if (r13 > _x);
Lb:
```

Because the IMPACT architecture permits multiple branch operations to be issued per cycle, and several operations can write to the same register per instruction, the last part of the loop contains 8 independent operations that can be issued at the same time.

The only remaining sequential section comprises the four addition operations in the middle of the loop. Traditional tree height reduction optimization can further eliminate two cycles from the schedule.

The ($r2 = r100$) and ($r2 = r101$) operations can be moved out of the loop by creating two new basic blocks that bridge the loop and the targets. Because the profile-based instruction placement optimization produces a layout that minimizes off-trace cost, the number of additional register move operations that this optimization introduces is small.

7.2.3 Global variable migration

Most optimizing compilers map scalar local variables that may not be accessed through pointers to registers. However, few compilers map global scalar variables and fields to registers. Because a memory load operation requires two cycles to produce a result, the memory access can increase the length of the critical path. For example, to increment a global memory variable, the first operation is to load the original value from the memory into a register, the second operation is to increment the register by one, and the third operation is to write the value of the register back to the memory. The three operations require four cycles of execution.

We have implemented global variable migration in the IMPACT-I C compiler. The algorithm can be informally stated as follows:

```
Input: a program G.
1) Identify all loops {L[i], i=1..n} in G.
2) for each loop L[i] begin
     if (all loads and stores to a memory location can be identified) then
       reg = a new (virtual) register;  /* before register assignment */
       for every entry path to L[i]
         load the memory value into reg;
       if (reg is ever modified in L[i]) then
         for every exit path from L[i]
           store the reg value back to the memory;
```

```
        Eliminate all loads and stores to that memory location in L[i];
        Replace references to that memory location by reg in L[i];
      end
    end
  end
```

7.2.4 Operation combining

We have described *operation combining* and *operation folding* in Chapter 6. Some (essential) flow-dependent operations can be executed concurrently after applying the two optimizations. An example of operation combining is shown in the following code segment:

```
r1 = r2 + 5;                    r1 = sp + 5;
r3 = memory[sp + r1];    ->     r3 = memory[r2 + r1];
```

The first operation of the transformed code uses two source operations whose values are determined early in the function. Therefore, the schedule may be reduced by one cycle, if the value of $r2$ is produced by an operation on a critical path.

An example of operation folding is shown in the following code segment:

```
r1 = r1 + 1;
goto L if (r1 > 100);  ->    r1 = r1 + 1;  goto L if (r1 > 99);
```

Because the two operations are packed into the same instruction, the comparison operation uses the old value of $r1$. This code pattern frequently occurs in inner loops.

7.2.5 Post-increment computation

Some loop induction variables cannot be eliminated from the loop body. Because an induction variable increment operation uses and writes the same register, it is in some way anti-dependent and flow-dependent with other operations that use the register. A simple optimization is to selectively transform pre-increment to post-increment style. For example,

```
r1 = r1 + 4;
r2 = memory[r1];
```

can be transformed into

```
r2 = memory[r1 + 4];
r1 = r1 + 4;
```

This transformation converts a flow-dependence to an anti-dependence. In the IMPACT architecture, the transformed code may be executed as one instruction, while the original code requires two instructions.

7.2.6 Memory disambiguation

Unlike data dependencies on registers, memory dependencies are difficult to resolve because pointer analysis is difficult for the C programming language. Our memory disambiguation tool can distinguish local variables from global variables, different local variables, different global variables, and different structure fields. However, our memory disambiguation tool is not powerful enough to disambiguate two pointer accesses, at the time of this writing.

CHAPTER 8

EXPERIMENTS

We have implemented many powerful code improving techniques, including function inline expansion, instruction placement, loop unrolling, memory disambiguation, register renaming, branch prediction, and an integrated register allocation and code scheduling algorithm, which are tailored to multiple-operation-issue processors.

The degree of freedom to move operations across branch operations depends greatly on the underlying compiler and hardware support. We have identified and implemented three static code scheduling models: restricted code percolation, general code percolation, and speculative execution.

We have identified the IMPACT architectural framework of multiple-operation-issue processors that is supported by our current compiler technology. Within this framework, the instruction set, the microarchitecture, and the code scheduling model can be specified in a technology file. Within this framework, code scheduling is done entirely at the time of compilation. The compiler packs operations into long instruction words. The underlying processor microarchitecture issues operations to the execution hardware in the order in which these operations are fetched. In this chapter, we present experimental data that show the effectiveness of using an aggressive static code scheduling model and a simple in-order execution hardware, as in the IMPACT framework. We also compare this performance with that achieved by using out-of-order hardware under the restricted code percolation model. The experimental data show that the IMPACT framework is simple and yet powerful.

The experimental data in this chapter are derived from some important nonnumerical programs with realistic input data.

8.1 Summary of the Compiler Support

Code improving techniques for generating efficient sequential code in the IMPACT-I C compiler can be categorized into two groups: machine-independent optimizations and machine-dependent optimizations. Machine-independent optimizations include classical local and global code optimizations [Aho 86], function inline expansion [Hwu 89c], instruction placement optimization [Chang 88], [Hwu 89a], loop unrolling, intelligent generation of *switch* statements [Chang 89c], and jump optimization. Machine-dependent optimizations include profile-based branch prediction, constant preloading, graph-coloring-based register allocation [Chaitin 82], [Chow 84], and code scheduling. A profiler has been integrated into the IMPACT-I C compiler. The decision making components of the code improving techniques use profile information, in addition to static loop analysis. When hardware resources are scarce, the profile information helps to identify the most frequently executed program sections and the most frequently accessed variables.

8.1.1 Code efficiency

It is important to measure the performance of multiple-operation-issue architectures using highly optimized code, because a naive compiler may produce redundant operations that show deceptive parallelism.

We compare the code quality that is produced by the IMPACT-I C compiler to that with the MIPS C compiler on a DEC 3100 workstation. Table 8.1 shows the benchmark programs that are used in this chapter. The *name* column shows the names of the benchmark programs. The *size* column shows the number of lines of C code in each benchmark program. The *description* column briefly describes each benchmark program.

Using the execution times of the executable programs that are generated by the IMPACT-I C compiler (-O5, beta release 0.1) as the basis, we show the code quality

that is achieved by the MIPS C compiler (-O4, release 2.1) and the GNU C compiler (-O, release 1.37.1). Table 8.2 compares the execution times of executable programs that are generated by the IMPACT-I C compiler, the MIPS C compiler, and the GNU C compiler on a DEC3100 machine. The *global* column shows the speedups of benchmark programs that are achieved by global code optimizations (-O5) over itself, and, therefore, the numbers are all ones. The *local* column shows the (negative) speedups of benchmark programs that are achieved by turning off global code optimizations. Note that global code optimizations improve program performance only by a small amount over local code optimizations. The $mips - O4$ column shows the speedups of benchmark programs that is achieved by the MIPS C compiler over the IMPACT-I C compiler with global code optimizations. The $gnu - O$ column shows the speedups of benchmark programs that are achieved by the GNU C compiler over the IMPACT-I C compiler with global code optimizations.

Note that the -O5 option in the IMPACT-I C compiler does not include profile-based classic code optimizations. Further performance improvement due to these profile-based code optimizations is reported in [Chang 91b].

8.1.2 Code generation for multiple-operation-issue machine

A code generator for a parameterized multiple-operation-issue architecture has been implemented. The code generator performs profile-based branch prediction to support squashing branch [McFarling 86], [Chang 89b]. The IMPACT-I C compiler performs several code transformations that enlarge the scope of static scheduling, including function inline expansion, instruction placement, loop unrolling, loop peeling, and branch expansion. The compiler also performs several code transformations that reduce the lengths of critical paths, including induction variable expansion, register renaming, global variable register allocation, operation combining, operation folding, and memory disambiguation.

Prepass code scheduling is performed prior to register allocation to reduce the effect of artificial data dependencies that are introduced by register assignment [Hwu 88b], [Goodman 88]. Postpass code scheduling is performed after register allocation.

The code scheduling algorithm consists of the following steps: 1) Form traces from basic blocks that are likely to be executed as a sequence. 2) Form a large super-block from each trace of basic blocks by code duplication. A super-block has a unique entry point and one or more exit points. 3) Construct a dependence graph for each super-block. 4) Improve the dependence graph by removing dependence arcs that can be resolved at compile time. 5) Compute live-variable information. For each branch path, live-variable information tells us what variables must not be destroyed when that branch path is taken. 6) Schedule the refined dependence graph according to machine constraints.

Our code scheduling algorithm is a variant of the trace-scheduling algorithm [Fisher 81]. Forming super-blocks eliminates the bookkeeping complexity due to upward code motion. Our code scheduler moves code both upward and downward across branch operations. Moving operations from above a branch operation to below is always safe. On the other hand, moving operations from below a branch to above is not always safe. There are two major restrictions on upward code motion.

(1) The moved operation must not destroy some value that is needed when the branch operation is taken.

(2) The moved operation must not cause an exception or trap that may terminate the program execution.

For example, it is not safe to move a division operation above a branch because of the possibility of dividing by zero. It is not safe to move a memory load operation above a branch because of the possibility of memory access violation. We have implemented a code scheduling algorithm that observes the above two restrictions. We refer to this algorithm as *restricted code percolation.*

It is possible to free the code scheduler from the second restriction if the architecture defines that the division operation and the memory load operation do not cause exceptions. Instead of trapping on divide by zero or illegal memory access, a magic value is returned. Page faults can be handled as usual. We refer to this code scheduling model as *general code percolation.* Most commercial processors already have a set of nontrapping

unsigned arithmetic operations. The hardware support for the general code percolation model is the addition of a set of nontrapping memory load operation opcodes, which can be provided with low cost and in an upward compatible way from existing architectures.

With aggressive hardware support, the first restriction can also be removed. Smith, Lam, and Horowitz have described such a scheme [Smith 90]. This scheme squashes operations if the branch direction is incorrectly predicted. We have implemented a scheduling method in which operations can be freely moved above N branch operations in the same super-block, where N is a design parameter. We refer to this scheduling model as *speculative execution.* In the next section, we show the relative performance of the three static code scheduling models. N is set to 32 for the speculative execution model.

8.1.3 Available parallelism

In Chapter 7, we described several code optimizations that enlarge super-blocks and reduce the lengths of critical paths. Figures 8.1 and 8.2 clearly show that these code optimizations are very effective. We assume that all operations take unit time (1 cycle) and that there are no limitations on the numbers of function units. We further assume perfect branch prediction. Figure 8.1 shows the average number of operations that are executed per cycle for each benchmark program, when the issue bandwidth is limited to at most 4 operations per cycle. Figure 8.2 shows the average number of operations that are executed per cycle for each benchmark program, when the issue bandwidth is limited to at most 8 operations per cycle.

8.2 The Effect of Static Code Scheduling

In this section, we evaluate the performance of the IMPACT processor architecture that has been described in Chapter 3. We apply the code optimization techniques that have been described in Chapters 4, 5, 6, and 7 to each benchmark program. Starting from very efficient sequential code, the IMPACT-I C compiler generates code for multiple-instruction-issue architectures. The experimental data in this section clearly indicate

that multiple-instruction-issue processors outperform single-instruction-issue processors by large margins.

8.2.1 Methodology

A machine description file has been written to describe the instruction set, the microarchitecture, and the code scheduling model of each processor architecture under study. The machine description file is used to guide the IMPACT-I C compiler to optimize each benchmark program for each processor architecture. We have chosen an instruction set that is a super-set of the MIPS instruction set to establish a strong single-instruction-issue base architecture. The microarchitectures use in-order execution and have deterministic operation latencies. Each processor includes a 64-entry integer register bank and a 32-entry floating-point register bank. The architecture uses a squashing branch scheme and profile-based branch prediction. One branch slot (one instruction) is automatically allocated for each instruction that contains a predict-taken branch operation. Using a profiler, we measure the execution count of every operation and collect branch statistics. From the profile information, we can derive the best and the worst case execution times of each super-block, assuming an ideal cache. The worst case is due to long operation latencies that protrude from one super-block to another super-block. The measurement data indicate that the difference between the best-case and the worst-case execution times is always negligible. In the following discussion, we consistently use the worst case execution time measure.

The experiment produces a total of $(X * Y)$ numbers, where X is the number of processor configurations under study, and Y is the number of benchmark programs. Let $cycle(i, j)$ denote a function that returns the number of cycles to execute the benchmark program j on the machine i. Let $cycle(1, j)$ denote a function that returns the number of cycles to execute the benchmark program j on the base architecture. We define the $speedup(i)$ function as the harmonic mean of $(cycle(1, .)/cycle(i, .))$ over all benchmarks. The harmonic mean is used instead of the arithmetic mean to report results conservatively.

8.2.2 Base architecture

The base architecture is a single-cycle-issue processor that uses the general code percolation model. All function units are pipelined. The *base* column of Table 8.3 shows the operation latencies.

Considering one slot penalty for each branch, the base architecture has achieved an execution rate of better than 0.95 operation per cycle for the benchmark programs.

8.2.3 Restricted code percolation

Figure 8.3 shows the speedups of twelve machines that use restricted code percolation over the base architecture. Each bar in Figure 8.3 is labeled with XrY, where X is the number of operations in an instruction word, and Y is the memory load latency. Except for the memory load latency, operation latencies are the same as that of the base architecture. There are no restrictions on the numbers of function units. Every operation code can be executed from any one operation slot of an instruction.

When the memory load latency is 2 cycles, the two-issue machine with restricted code percolation achieves about a 1.4 speedup over the one-issue machine with general code percolation. When the memory load latency is 2 cycles, the four-issue machine with restricted code percolation achieves about a 1.7 speedup over the one-issue machine with general code percolation.

8.2.4 General code percolation

Figure 8.4 shows the speedups of twelve machines that use general code percolation over the base architecture. Each bar in Figure 8.4 is labeled with XgY, where X is the number of operations in an instruction word, and Y is the memory load latency. Note that the $1g2$ machine is the base architecture. Except for the memory load latency, operation latencies are the same as that of the base architecture. There are no restrictions on the numbers of function units. Every operation code can be executed from any one operation slot of an instruction.

When the memory load latency is 2 cycles, the two-issue machine achieves about a 1.64 speedup over the one-issue machine with general code percolation. When the memory load latency is 2 cycles, the four-issue machine achieves about a 2.06 speedup over the one-issue machine with general code percolation. The improvement from restricted code percolation to general code percolation is substantial for high-issue-rate architectures.

8.2.5 Speculative execution

Figure 8.5 shows the speedup of twelve machines that use speculative execution over the base architecture. Each bar in Figure 8.5 is labeled with XsY, where X is the number of operations in an instruction word, and Y is the memory load latency. Except for the memory load latency, operation latencies are the same as that of the base architecture. There are no restrictions on the numbers of function units. Every operation code can be executed from any one operation slot of an instruction.

When the memory load latency is 2 cycles, the two-issue machine with speculative execution achieves about a 1.65 speedup over the one-issue machine with general code percolation. When the memory load latency is 2 cycles, the four-issue machine with speculative execution achieves about a 2.08 speedup over the one-issue machine with general code percolation. Although speculative execution consistently performs better than general code percolation, the improvement is not significant.

8.2.6 The effect of limiting function unit resources

For the general code percolation model, we measure the effect of limiting function unit resources. Each bar in Figures 8.6 and 8.7 is labeled $XgY.Z$, where X is the number of operations in an instruction, Y is the memory load latency, and Z is the limited function unit resource. For ($Z = 1br$), at most one branch operation can be packed into an instruction. For ($Z = 1st$), at most one memory store operation can be packed into an instruction. For ($Z = 1st.1ld$), at most one memory load or store operation can be packed into an instruction. For ($Z = 1st.1ld.1br$), at most one branch operation can

be packed into an instruction, and at most one memory load or store operation can be packed into an instruction. Figure 8.6 presents the result for single-cycle memory load latency, and Figure 8.7 presents the result for two-cycle memory load latency.

The experimental data indicate that the ability to execute multiple branch and memory load operations is important for high-issue-rate architectures. On the other hand, limiting each instruction to contain at most one memory store operation degrades performance only slightly. Therefore, the data cache interface should support multiple concurrent read ports and a single write port for high-issue-rate architectures.

8.2.7 The effect of changing the memory load latency

Figure 8.8 shows the effect of changing the memory load latency for the general code percolation model. Each bar in Figure 8.8 is labeled with XgY, where X is the number of operations in an instruction, and Y is the memory load latency.

For high-issue-rate architectures, increasing the memory load latency dramatically reduces the instruction-level parallelism, because memory load operations often appear on critical paths.

8.2.8 The effect of increasing branch slots

Figures 8.9 and 8.10 show the effect of increasing the number of branch slots. Each bar is labeled $XgY.brZ$, where X is the number of operations in an instruction, Y is the memory load latency, and Z is the number of branch slots for a predict-taken branch. Figure 8.9 shows the result for single-cycle memory load latency, and Figure 8.10 shows the result for two-cycle memory load latency.

Code optimizations in the IMPACT-I C compiler do not change the program control flow. Therefore, the number of branch misses is unchanged for different machine configurations. For high-issue-rate architectures in which the execution times are shorter, the branch penalty becomes much greater.

8.3 The Effect of Dynamic Code Scheduling

Existing processor architectures use the restricted code percolation model, because illegal memory accesses can cause exceptions. The scheduling result under the restricted code percolation model represents what can be achieved by extending existing processor architectures to multiple-instruction-issue architectures. The topic of this section is to study the effect of static code scheduling and dynamic code scheduling on the restricted code percolation model.

We compare the performance of three design styles for various instruction fetch rates. The first design style is to apply static code scheduling, under the restricted code percolation model, for in-order execution architectures. This can be accomplished by replicating the datapath of a pipelined processor to support higher instruction fetch bandwidth. The second style is to complement the first style by using out-of-order execution hardware. Static code scheduling is still under the restricted code percolation model. The third style is to extend the capability of static code scheduling by using the general code percolation model, and uses in-order execution hardware. Because in-order execution hardware is much simpler than out-of-order execution hardware, if the second and the third design styles achieve comparable performance, the third design style is preferred.

8.3.1 Methodology

We have instrumented a code generator to insert extra code into the user program to generate instruction traces. Each element of an instruction trace is an instruction word of the IMPACT processor architecture. Each instruction may contain one or more operations, depending on the instruction fetch bandwidth parameter. For each memory and branch operation, we record the memory address and the branch direction in order to simulate branch logic and cache accesses.

The experimental procedure consists of the following steps: (1) Select a typical input for each benchmark. (2) Compile the benchmark for a selected machine configuration. (3)

Generate an instruction trace for the benchmark. (4) Analyze the instruction trace on the fly. (5) Repeat step (2) to step (4) for each benchmark and each machine configuration.

The same set of benchmarks as in the previous section are used. The inputs that we have chosen for each benchmark program are realistic. The simulator consumes entire traces. Some of the instruction traces contain more than 20 million instructions.

The trace analyzer simulates a simple dynamic code scheduling model that has an infinite number of reservation stations for each function unit. Hardware register renaming is supported for both the static code scheduling and the dynamic code scheduling models. The control unit fetches an instruction (N operations) per cycle, except when a mispredicted branch operation has recently been encountered and caused the control unit to refill its pipeline. After an instruction has been decoded, those operations that have not obtained all source operands are placed in the reservation stations; otherwise, operations are directly submitted to the function units. An operation is moved from a reservation station to a function unit as soon as its source operands have been obtained.

A branch operation that has been decoded but has not yet computed the condition code is called a *pending branch.* Tha trace analyzer allows instructions to execute ahead of an infinite number of pending branches, simulating unlimited branch lookahead. The simulation result is an upper bound on the performance of dynamic code scheduling.

Memory load operations are allowed to bypass (the order in the cache access queue) other memory store operations when the memory addresses do not overlap. A large write buffer is simulated to allow the accumulation of memory store operations.

8.3.2 Base architecture

The base architecture is a single-operation-issue machine that uses in-order execution hardware and restricted code percolation. All function units are pipelined. The *latency* column of Table 8.3 shows the operation latencies. One branch slot is allocated for each predicted-taken branch operation.

8.3.3 Ideal cache

Figure 8.11 shows the speedups of the three design styles over the base architecture, with an ideal data cache. Each bar represents the harmonic mean of the speedup of a machine configuration over all benchmark programs. Each bar is labeled XYZ, where X is the number of operations that can be packed into an instruction word, Y is r or g (r for restricted percolation, g for general percolation), and Z is i or o (i for in-order execution, o for out-of-order execution).

8.3.4 Realistic cache

Figure 8.12 shows the speedup of the three design styles over the base architecture with an 8K data cache, while Figure 8.13 shows the speedup with an 16K data cache. Each bar represents the harmonic mean of the speedup of a machine configuration over all benchmark programs. Each bar is labeled XYZ, where X is the number of operations that can be packed into an instruction word, Y is r or g (r for restricted percolation, g for general percolation), and Z is i or o (i for in-order execution, o for out-of-order execution).

8.3.5 Analysis

Although the summary data indicate that the performances of an in-order execution under the general code percolation model and that of an out-of-order execution under the restricted code percolation model are comparable, individual execution data show that each approach performs better for a different subset of the benchmark programs.

We have identified two major reasons why dynamic scheduling occasionally performs better than static scheduling.

First, code transformation techniques that enlarge the scope of static code scheduling cannot be applied across hashing jump operations. A hashing jump operation is an unconditional branch operation with a data-dependent branch target. Because the compiler cannot determine the branch target statically, branch target expansion cannot be

applied, and loop unrolling cannot be applied for a loop that ends with a hashing jump. Unfortunately, the most important loop of the *cccp* program ends with a hashing jump operation that implements a *switch* statement in C.

Second, memory disambiguation at compile time is limited, while perfect memory disambiguation can be achieved at run time. Many important loops contain memory load and store operations through pointers. After loop unrolling, the computation of different iterations cannot be interleaved because of memory dependencies that cannot be resolved by the code scheduler. With dynamic code scheduling, the memory load operations of a loop iteration do bypass the memory store operations of the previous loop iteration.

When the above two problems do not exist, the general code percolation model can interleave several loop iterations. For example, under the general code percolation model, the inner loop

```
La: r1 = memory[r0];
    r2 = r2 + r1;
    r0 = r0 + 1;
    goto La if (r0<=r3);
Lb:
```

can be transformed to a more efficient inner loop body,

```
La: r1=memory[r0];  r10=memory[r0+1];  r20=memory[r0+2]; r30=memory[r0+3];
    r2 = r2 + r1;  r0 = r0 + 1; goto Lb if (r0>=r3);
    r2 = r2 + r10; r0 = r0 + 1; goto Lb if (r0>=r3);
    r2 = r2 + r20; r0 = r0 + 1; goto Lb if (r0>=r3);
    r2 = r2 + r30; r0 = r0 + 1; goto La if (r0<r3);
Lb:
```

The memory load operations from several iterations are executed as soon as possible. On the other hand, the memory load operation cannot be moved across a previous branch operation in the restricted code percolation model. Even though the out-of-execution hardware can execute the memory load operation of each iteration as soon as it is fetched, the execution time of each iteration has already increased by one cycle because the two cycle load latency is not covered by independent operations.

The experimental data indicate that static scheduling and dynamic scheduling have their own merits and limitations. Therefore, to achieve the highest level of performance, using out-of-order execution hardware under the general code percolation model may be the best approach. An interesting future research work is to evaluate the performance and the cost-effectiveness of this approach.

8.4 The Importance of a Prepass Code Scheduling

In Section 6.3, we described the problem with implementing register allocation and code scheduling as two separate phases. If register allocation is applied before code scheduling, then many artificial data dependencies are introduced by the binding of several virtual registers to one physical register. In a previous study of small numeric kernels [Hwu 88b], we have shown that the artificial data dependencies that are introduced by register allocation can degrade the benchmark performance by an average of about 30% for a processor that can issue two operations per cycle.

Figures 8.14 and 8.15 demonstrate the importance of adding a prepass code scheduling for the set of nonnumeric benchmark programs in Table 8.1. In Figures 8.14 and 8.15, lines whose labels are suffixed by *.np* indicate the speedups of benchmark programs (over restricted code percolation, issue 1 operation per cycle), when prepass code scheduling is disabled. The other lines show the speedups of benchmark programs, when prepass code scheduling is enabled. We can clearly see that prepass code scheduling is very important.

Table 8.1 Benchmarks.

name	*size*	*description*
cccp	4787	GNU C preprocessor
cmp	141	compare files
compress	1514	compress files
eqn	2569	typeset mathematical formulas for troff
eqntott	3461	Boolean minimization
espresso	6722	Boolean minimization
grep	464	string search
lex	3316	lexical analysis program generator
qsort	250	quick sort
tbl	2817	format tables for troff
wc	120	word count
yacc	2303	parsing program generator

Table 8.2 Speedup on MIPS-R2000 processor.

benchmark	*global*	*local*	*mips -O4*	*gnu -O*
cccp	1.0	0.95	0.93	0.92
cmp	1.0	0.95	0.95	0.95
compress	1.0	0.95	0.98	0.94
eqn	1.0	0.88	0.87	0.87
eqntott	1.0	0.62	0.96	0.75
espresso	1.0	0.91	0.98	0.87
grep	1.0	0.87	0.97	0.82
lex	1.0	0.97	0.99	0.97
qsort	1.0	0.94	1.00	0.94
tbl	1.0	0.94	0.98	0.93
wc	1.0	0.97	0.96	0.87
yacc	1.0	0.87	1.00	0.91

Table 8.3 Operation latencies.

fn	*base*	*SPARC*	*i860*
integer alu	1	1	1
barrel shifter	1	1	1
integer mul	3	47	11
integer div	25	?	59
load	2	2	2
store	-	-	-
FP alu	3	10	3
FP conv	3	10	4
FP mul	4	12	5
FP div	25	64	38

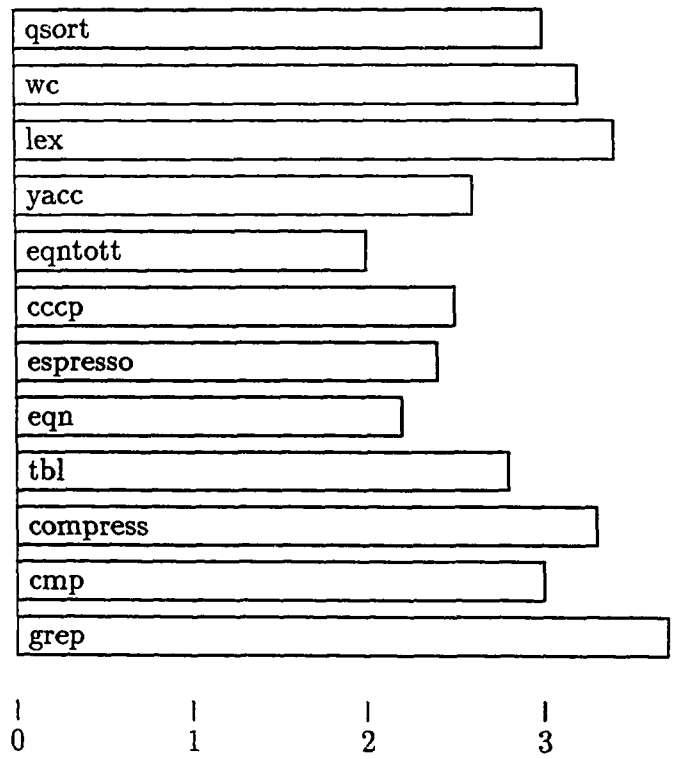


Figure 8.1 Operations per cycle (issue at most 4).

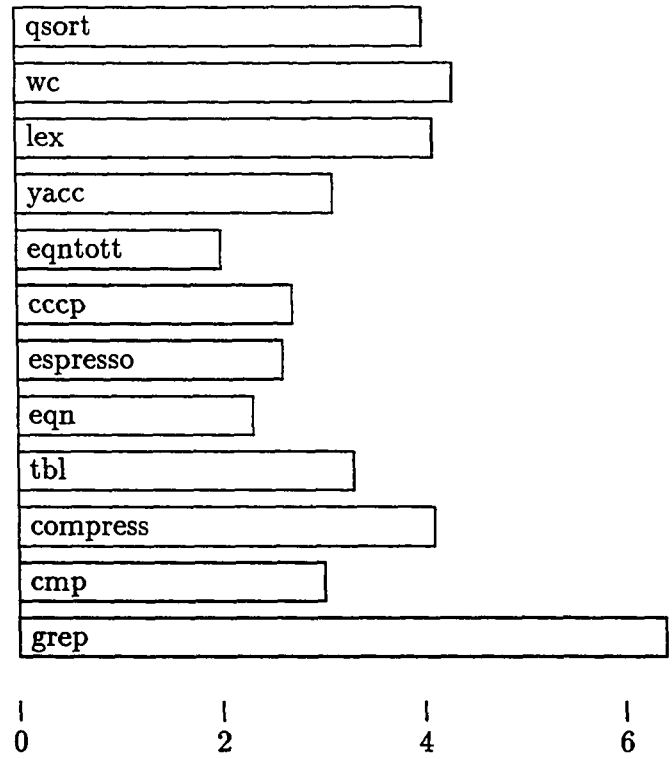


Figure 8.2 Operations per cycle (issue at most 8).

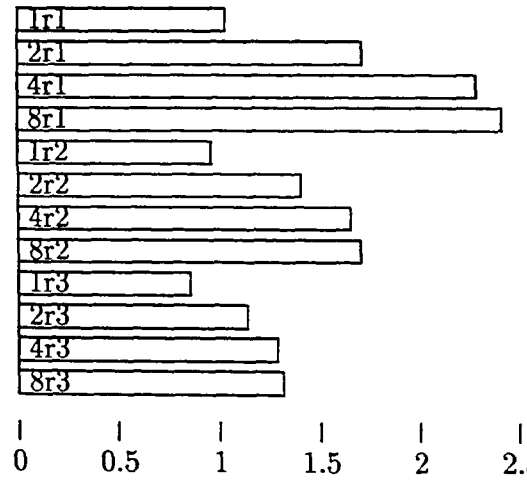


Figure 8.3 Restricted code percolation.

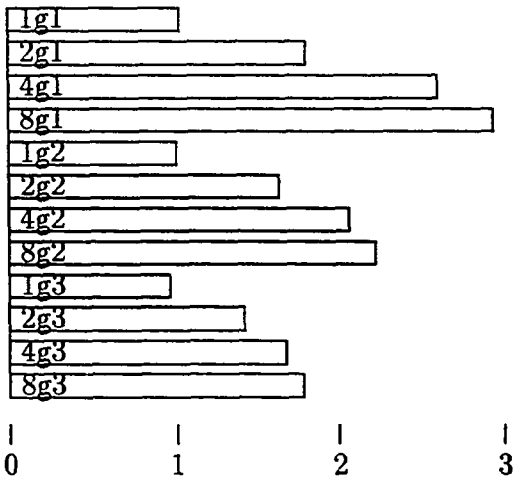


Figure 8.4 General code percolation.

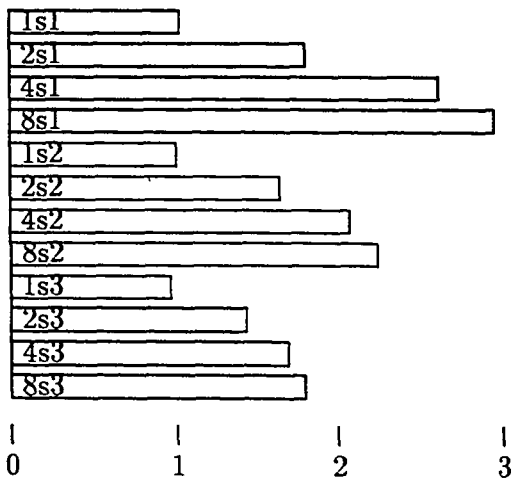


Figure 8.5 Speculative execution.

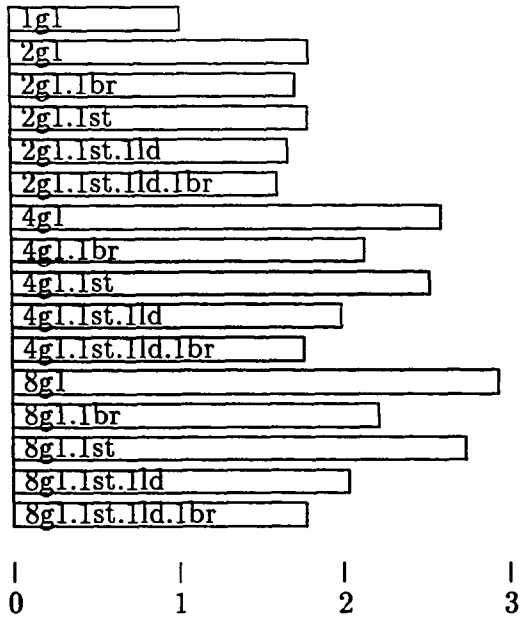


Figure 8.6 Limited function resource, load delay 1.

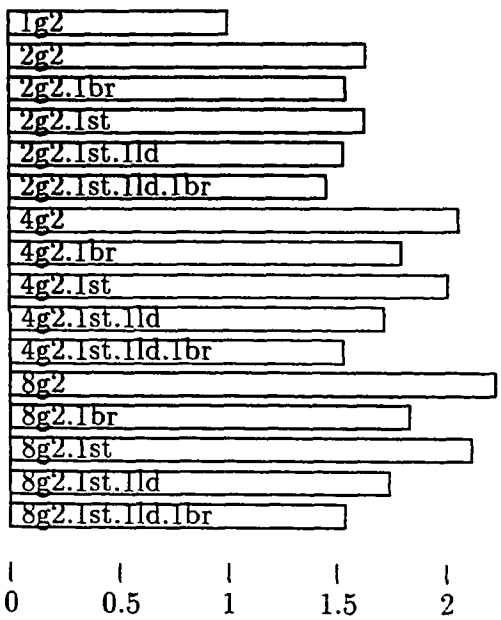


Figure 8.7 Limited function resource, load delay 2.

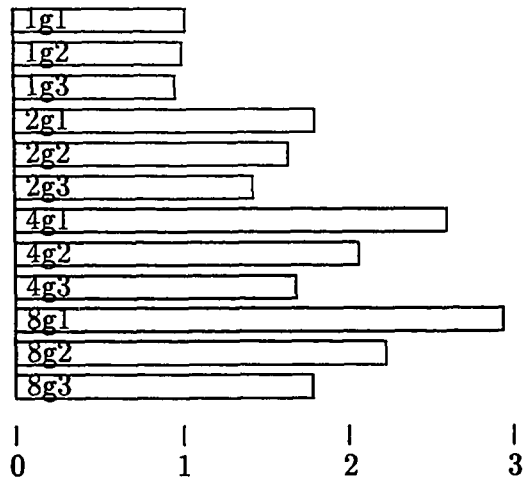


Figure 8.8 Different memory operation latencies.

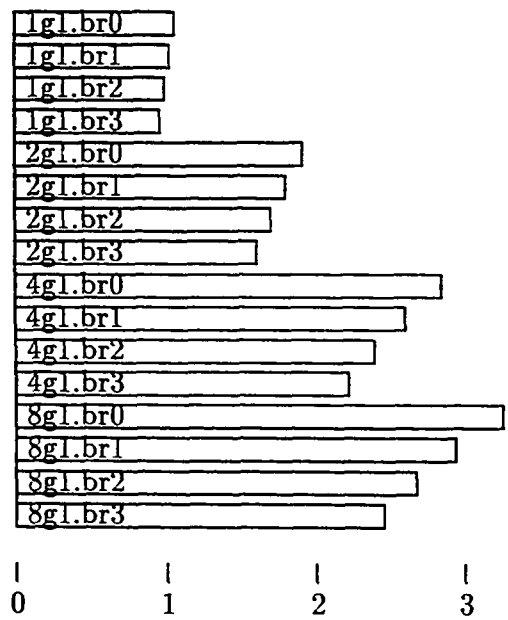


Figure 8.9 Adding branch slots, load delay 1.

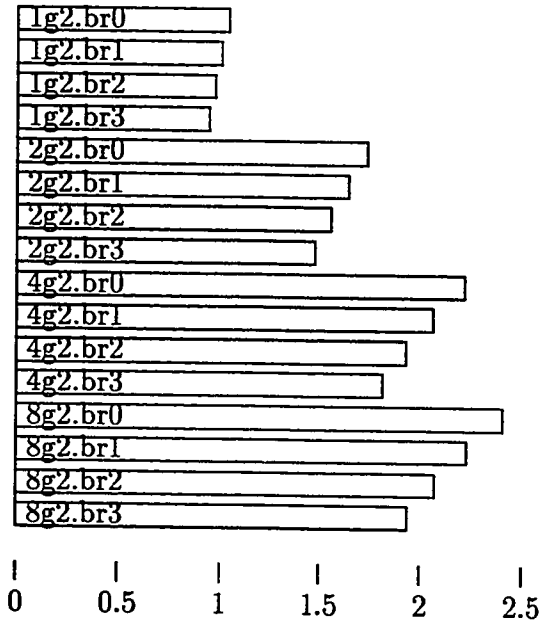


Figure 8.10 Adding branch slots, load delay 2.

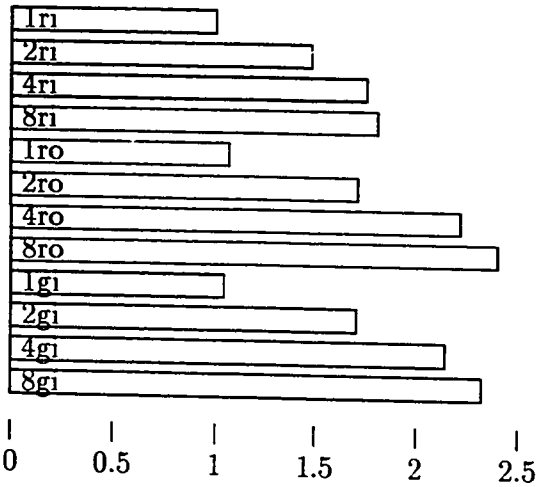


Figure 8.11 Execution rate (ideal cache).

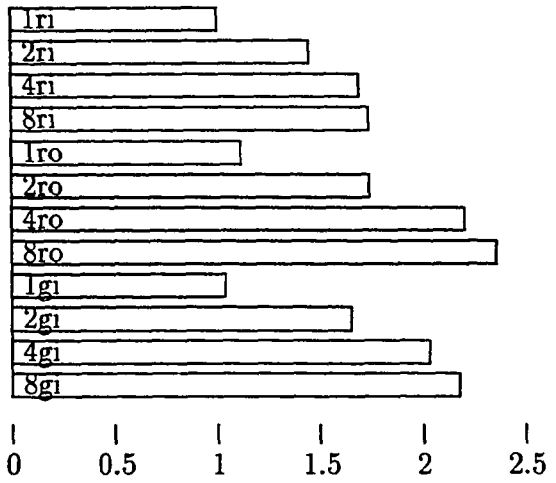


Figure 8.12 Execution rate (8K cache).

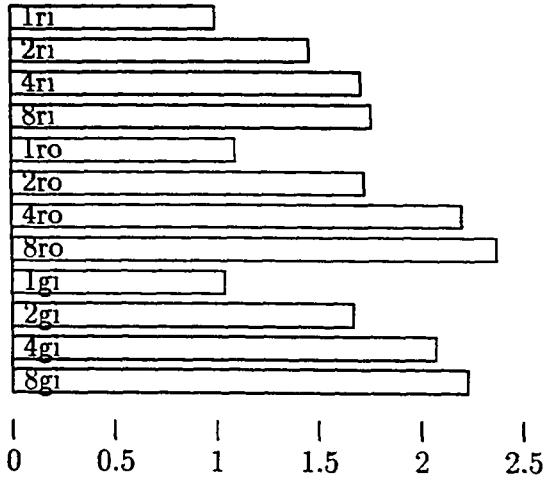


Figure 8.13 Execution rate (16K cache).

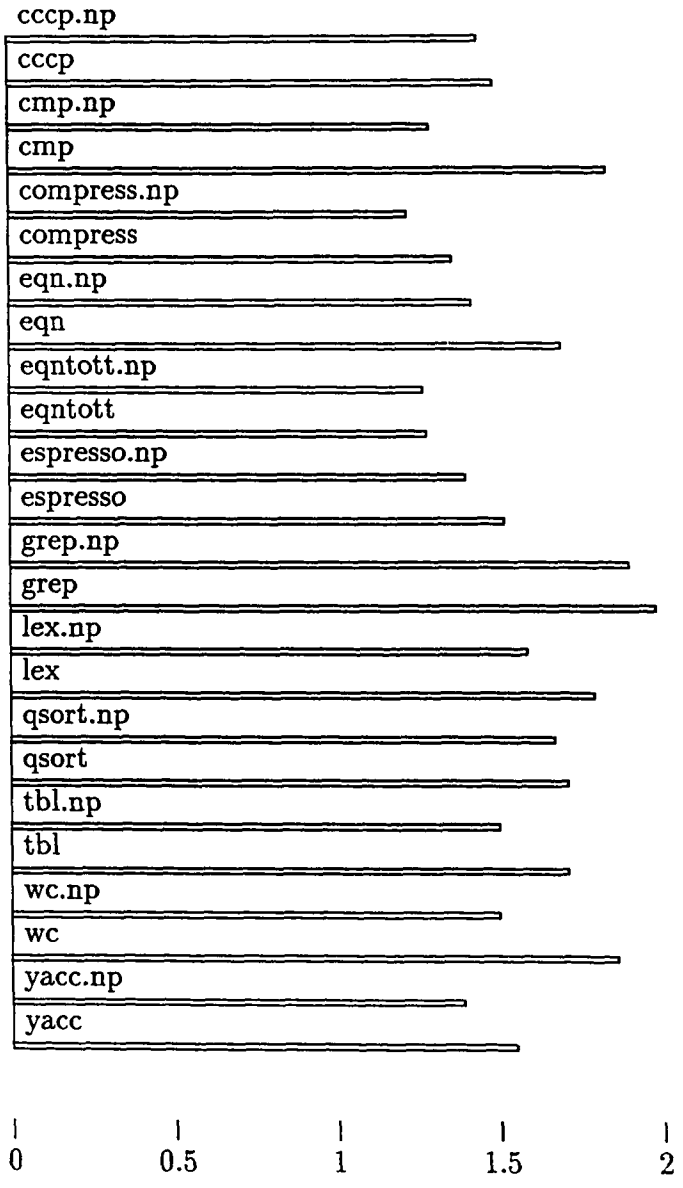


Figure 8.14 Speedup (issue at most 2 operations per cycle)

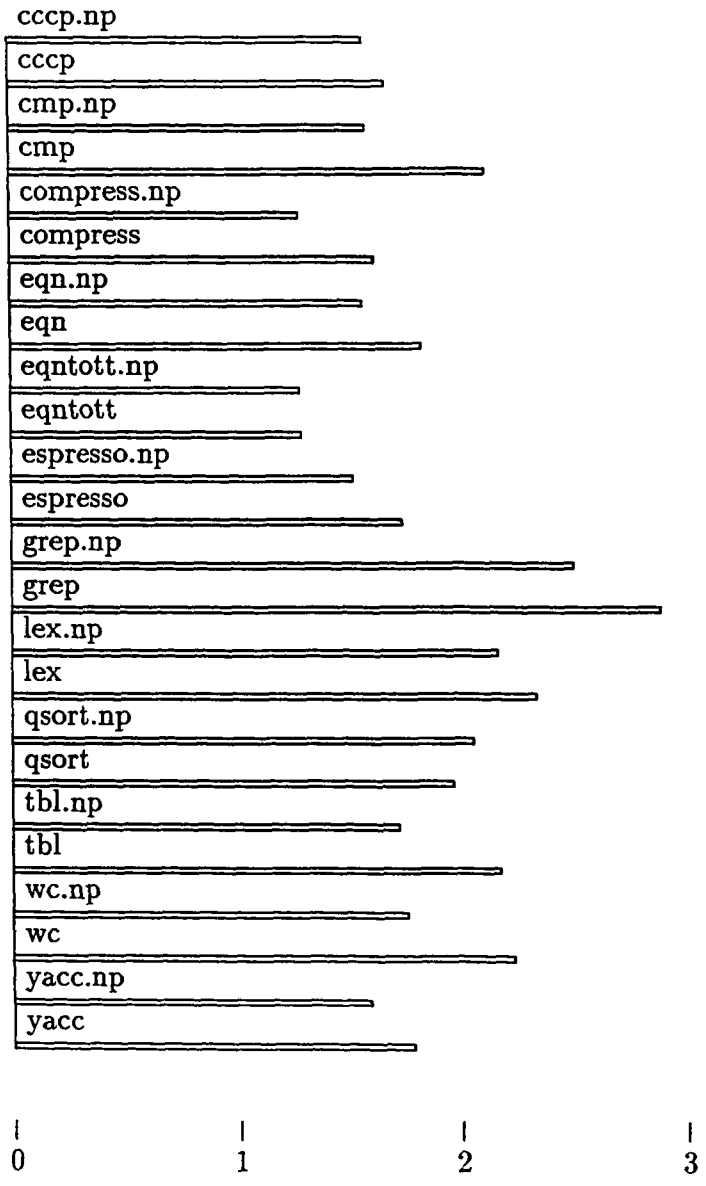


Figure 8.15 Speedup (issue at most 4 operations per cycle)

CHAPTER 9

INLINE TARGET INSERTION

In Chapter 8, the experimental data show that it is important to execute multiple branch operations per cycle in a multiple-operation-issue machine. In this chapter, we develop a squashing branch that allows branch operations to be fetched from branch slots. Not only can branch operations be executed in parallel, they can also be pipelined in Inline Target Insertion. This chapter is an extension to two previous papers on Inline Target Insertion [Chang 89b], [Hwu 90].

9.1 Introduction

The instruction sequencing mechanism of a processor determines the instructions to be fetched from the memory system for execution. In the absence of branch instructions, the instruction sequencing mechanism keeps requesting the next instructions in the linear memory space. In this sequential mode, it is easy to maintain a steady supply of instructions for execution. Branch instructions, however, disrupt the sequential mode of instruction sequencing. Without special hardware and/or software support, branches can significantly reduce the performance of pipelined processors by breaking the steady supply of instructions to the pipeline [Kogge 81].

Many hardware methods for handling branches in pipelined processors have been studied [Smith 81], [Lee 84], [DeRosa 88], [McFarling 86], [Hsu 86], [Ditzel 87]. An important class of hardware methods, called Branch Target Buffers (or Branch Target Caches), uses buffering and extra logic to detect branches at an early stage of the pipeline, predict the

branch direction, fetch instructions according to the prediction, and nullify the instructions fetched due to an incorrect prediction [Lee 84]. Branch Target Buffers have been adopted by many commercial processors [Lee 84], [Horst 90]. The performance of such hardware methods is determined by their ability to detect the branches early and to predict the branch directions accurately. High branch prediction accuracy, about an 85-90% hit ratio, has been reported for hardware methods [Smith 81], [Lee 84], [McFarling 86]. Another advantage of using Branch Target Buffers is that they do not require recompilation or binary translation of existing code. However, the hardware methods suffer from the disadvantage of requiring a large amount of fast hardware to be effective [Lee 84], [Hwu 89b]. Their effectiveness is also sensitive to the frequency of context switching [Lee 84].

Compiler-assisted methods have also been proposed to handle branches in pipelined processors. Table 9.1 lists three such methods. Delayed Branching has been a popular method to absorb branch delay in microsequencers of microprogrammed microengines. This technique has also been adopted by many recent processor architectures including IBM 801 [Radin 82], Stanford MIPS [Hennessy 81], Berkeley RISC [Patterson 82], HP Spectrum [Birnbaum 86], SUN SPARC [Sparc 87], MIPS R2000 [Kane 87], Motorola 88000 [Melear 89], and AMD 29000 [Amd]. In this approach, instruction slots immediately after a branch are reserved as the *delay slots* for that branch. The number of delay slots has to be large enough to cover the delay for evaluating the branch direction. During compile-time, the delay slots following a branch are filled with instructions that are independent of the branch direction, if the data and control dependencies allow such code movement [Gross 82]. Regardless of the branch direction, these instructions in the delay slots are always executed. McFarling and Hennessy reported that the first delay slot can be successfully filled by the compiler for approximately 70% of the branches, and the second delay slot can be filled only 25% of the time [McFarling 86]. It is clear that delayed branching is not effective for processors requiring more than one slot.

Another compiler-assisted method, called Delayed Branches with Squashing, has been adopted by some recent processors to complement delayed branching [McFarling 86],

[Chow 87], [Melear 89], [Intel 89]. That is, the method is used when the compiler cannot completely fill the delay slots for delayed branching. In this scheme, the number of slots after each branch still has to be large enough to cover the branch delay. However, instead of moving independent instructions into branch delay slots, the compiler can fill the slots with the predicted successors of the branch. If the actual branch direction differs from the prediction, the instructions in the branch slots are scratched (squashed or nullified) from the pipeline.

On the least expensive side, the hardware predicts all conditional branches to be either always taken (as in Stanford MIPS-X [Chow 87]) or always not-taken (as in Motorola 88000 [Melear 89]). Predicting all the instructions to be taken achieves about a 65% accuracy whereas predicting not-taken is at about 35% [Smith 81], [Lee 84], [Emer 84]. Predicting all the branches to be either taken or not-taken limits the performance of delayed branches with squashing. Furthermore, filling the branch slots for predicted-taken branches requires code copying in general. Predicting all branches to be taken can result in a large amount of code expansion.

McFarling and Hennessy proposed Profiled Delayed Branches with Squashing. In this scheme, an execution profiler is used to collect the dynamic execution behavior of programs such as the preferred direction of each branch [McFarling 86]. The profile information is then used by a compile-time code restructurer to predict the branch direction and to fill the branch slots according to the prediction. To allow each branch to be predicted differently, an additional bit to indicate the predicted direction is required in the branch opcode in general [Intel 89]. Through this bit, the compiler can convey the prediction decision to the hardware. McFarling and Hennessy also suggested methods for avoiding adding a prediction bit to the branch opcode. Using pipelines with one and two branch slots, McFarling and Hennessy showed that the method can offer comparable performance with hardware methods at a much lower hardware cost. They suggested that the stability of execution profile information in compile-time code restructuring should be further evaluated.

This chapter examines the extension of McFarling and Hennessy's idea to processors employing deep pipelining and multiple-instruction-issue. These techniques increase the number of slots for each branch. As a result, four issues arise. First, there are only 3 to 5 instructions between branches in the static program (see Section 9.4.2) . To fill a large number of slots (on the order of ten), one must be able to insert branches into branch slots. Questions arise regarding the correct execution of branches in branch slots. Second, the state information about all branch instructions in the instruction pipeline becomes large. Brute force implementations of return from interrupts and exceptions can involve saving/restoring a large amount of state information of the instruction sequencing mechanism. Third, the code expansion due to code restructuring can be very large. It is important to control such code expansion without sacrificing performance. Fourth, the time penalty for refilling the instruction fetch pipeline due to each incorrectly predicted branch is large. It is very important to show extensive empirical results on the performance and stability of using profile information in compile-time code restructuring. The first three issues were not addressed by McFarling and Hennessy [McFarling 86]. The second issue was not addressed by previous studies of hardware support for precise interrupt [Hwu 87], [Smith 85a].

To address these issues, we have specified a compiler and pipeline implementation method for Delayed Branches with Squashing. We refer to this method as Inline Target Insertion to reflect the fact that the compiler restructures the code by inserting predicted successors of branches into their sequential locations. Based on the specification, we show that the method exhibits desirable properties such as simple compiler and hardware implementation, clean interrupt/exception return, moderate code expansion, and high instruction sequencing efficiency. We also provide a proof that Inline Target Insertion is correct. Our correctness proof of filling branch slots with branch instructions is also applicable to a previously proposed hardware scheme [Pleszkun 87].

9.2 Background and Motivation

9.2.1 Branch instructions

Branch instructions reflect the decisions made in the program algorithm. Figure 9.1(a) shows a C program segment which finds the largest element of an array. There are two major decisions in the algorithm. One decides whether all the elements have been inspected, and the other decides whether the current element is larger than all the other ones inspected so far.

With the register allocation/assignment assumption in Figure 9.1(b), a machine language program can be generated as given in Figure 9.2. There are three branches in the machine language program. Instruction D ensures that the looping condition is checked before the first iteration. Instruction I checks if the loop should iterate any more. Instruction F determines if the current array element is larger than all of the others visited so far.

The simplified view of the machine language program in Figure 9.2 highlights the effect of branches. Each arc corresponds to a branch in which the head of an arc is the *target instruction.* The percentage on each arc indicates the probability for the corresponding branch to occur in execution. The percentages can be derived by program analysis and/or execution profiling. If the percentage on an arc is greater than 50%, it corresponds to a *likely branch.* Otherwise, it corresponds to an *unlikely branch.*

The instructions shown in Figure 9.2(a) are *static instructions.* These are the instructions generated by the compilers and machine language programmers. During program execution, each static instruction can be executed multiple times due to loops. Each time a static instruction is executed, it generates a *dynamic instruction.* A dynamic branch instruction which redirects the instruction fetch is called a *taken branch.*

9.2.2 Instruction sequencing for pipelined processors

The latency of decoding and executing branch instructions complicates instruction sequencing in pipelined processors. A simple hardware example suffices to illustrate the problem of instruction sequencing for pipelined processors. The processor shown in Figure 9.3 is divided into four stages: instruction fetch (IF), instruction decode (ID), instruction execution (EX), and result write-back (WB). The instruction sequencing logic is implemented in the EX stage. The *sequencing pipeline* consists of the IF, ID, and EX stages of the processor pipeline. When a compare-and-branch[1] instruction is processed by the EX stage,[2] the instruction sequencing logic determines the next instruction to fetch from the memory system based on the comparison result.

The dynamic pipeline behavior is illustrated by the timing diagram in Figure 9.4. The vertical dimension gives the clock cycles and the horizontal dimension, the pipeline stages. For each cycle, the timing diagram indicates the pipeline stage in which each instruction can be found.

The pipeline fetches instructions sequentially from memory until a branch is encountered. In Figure 9.4, the instructions to be executed are $E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow E \rightarrow F$. However, the direction of branch I is not known until cycle 7. By this time instructions J and K have already entered the pipeline. Therefore, in cycle 8, instruction E enters the pipeline while J and K are scratched. The nonproductive cycles introduced by incorrectly fetching J and K reduce the throughput of the pipeline.

9.2.3 Deep pipelining and multiple-instruction-issue

The rate of instruction execution is equal to the clock frequency times the number of instructions executed per clock cycle. One way to improve the instruction execution rate is to increase the clock frequency. The pipeline stages with the longest delay (critical

[1]Although the compare-and-branch instructions are assumed in the example, the methods apply to condition code branches as well.

[2]Although unconditional branch instructions can redirect the instruction fetch at the ID stage, we ignore the optimization in this example for simplicity.

paths) limit the clock frequency. Therefore, subdividing these stages can potentially increase the clock frequency and improve the overall performance. This adds stages in the pipeline and creates a deeper pipeline. For example, if the instruction cache access and the instruction execution limit the clock frequency, subdividing these stages may improve the clock frequency. A timing diagram of the resultant pipeline is shown in Figure 9.5. Now, four instructions are scratched if a compare-and-branch redirects the instruction fetch. For example, $I_2 - I_5$ may be scratched if I_1 redirects the instruction fetch.

Another method to improve instruction execution rate is to increase the number of instructions executed per cycle. This is accomplished by fetching, decoding, and executing multiple instructions per cycle. This is often referred to as *multiple-instruction-issue*. The timing diagram of such a pipeline is shown in Figure 9.6. In this example, two instructions are fetched per cycle. When a compare-and-branch (I_1) reaches the EX stage, five (I_2, I_3, I_4, I_5, I_6) instructions may be scratched from the pipeline.[3]

As far as instruction sequencing is concerned, multiple-instruction-issue has the same effect as deep pipelining. They both result in an increased number of instructions which may be scratched when a branch redirects the instruction fetch.[4] Combining deep pipelining and multiple-instruction-issue will increase the number of instructions to be scratched to a relatively large number. For example, the TANDEM Cyclone processor requires 14 branch slots due to deep pipeline and multiple-instruction-issue [Horst 90].[5] The discussions in this chapter do not distinguish between deep pipelining and multiple-instruction-issue; they are based on the number of instructions to be scratched by branches.

---

[3]The number of instructions to be scratched from the pipeline depends on the instruction alignment. If I_2 rather than I_1 were a branch, four instructions (I_3, I_4, I_5, I_6) would be scratched.

[4]A difference between multiple-instruction-issue and deep pipelining is that multiple likely control transfer instructions could be issued in one cycle. Handling multiple likely control transfer instructions per cycle in a multiple-instruction-issue processor is not difficult in Inline Target Insertion. The details are not within the scope of this chapter.

[5]The processor currently employs an extension to the instruction cache which approximates the effect of a Branch Target Buffer to cope with the branch problem.

9.3 Inline Target Insertion

Inline Target Insertion consists of a compile-time code restructuring algorithm and a run-time pipelined instruction fetch algorithm. The compile-time code restructuring algorithm transforms a sequential program P_s into a parallel program P_p. Inline Target Insertion is correct if the instruction sequence generated by executing P_p on a pipelined instruction fetch unit is identical to that generated by executing P_s on a sequential instruction fetch unit. In this section, we first formally define the sequential instruction fetch algorithm. Then, we formally define the code restructuring algorithm and the pipelined instruction fetch algorithm of Inline Target Insertion. From the formal models of implementation, we will derive a proof of correctness.

9.3.1 Sequential instruction fetch

In a sequential instruction fetch unit, $I_s(t)$ is defined as the dynamic instruction during cycle t. The address of $I_s(t)$ will be referred to as $A_s(I_s(t))$. The target instruction of a branch instruction $I_s(t)$ will be referred to as $target(I_s(t))$. The next sequential instruction of a branch instruction $I_s(t)$ will be referred to as $fallthru(I_s(t))$. The sequential instruction fetch algorithm (SIF) is as follows:

Algorithm SIF begin

if ($I_s(t)$ is a taken branch) then

$A_s(I_s(t+1)) \leftarrow A_s(target(I_s(t)))$;

else

$A_s(I_s(t+1)) \leftarrow A_s(I_s(t)) + 1$; [6]

end

The *correct successors* of a dynamic instruction $I_s(t)$ are defined as the dynamic instructions to be executed after $I_s(t)$ as specified by SIF. The k^{th} correct successor of

[6] In the discussions, all address arithmetics are in terms of instruction words. For example, $address \leftarrow address + 1$ advances the *address* to the next instruction.

$I_s(t)$ will be denoted $CS(I_s(t), k)$. It should be noted that $CS(I_s(t), k) = I_s(t + k)$. For a sequential program, P_s, whose execution starts from instruction I_0, the instruction sequence is $(I_0, CS(I_0, 1), CS(I_0, 2), ..., CS(I_0, n))$, where $CS(I_0, n)$ is the first terminating instruction.

9.3.2 Compiler implementation

The compiler implementation of Inline Target Insertion involves compile-time branch prediction and code restructuring. Branch prediction marks each static branch as either likely or unlikely. The prediction is based on the estimated probability for the branch to redirect an instruction fetch at run time. The probability can be derived from program analysis and/or execution profiling. The prediction is encoded in the branch instructions.

The *predicted successors (PS)* of an instruction I are the instructions which tend to execute after I. The definition of predicted successors is complicated by the frequent occurrence of branches. Let $PS(I, k)$ refer to the k^{th} predicted successor of I. The predicted successors of an instruction can be defined recursively:

(1) If I is a likely branch, then $PS(I, 1)$ is $target(I)$. Otherwise, $PS(I, 1)$ is $fallthru(I)$.

(2) $(I_1 = PS(I, k)) \wedge (I_2 = PS(I_1, 1)) \rightarrow I_2 = PS(I, k + 1)$.

For example, one can identify the first five predicted successors of F in Figure 9.2 as shown below. Since F is a likely branch, its first predicted successor is its target instruction H. The second predicted successor of F is I, which is a likely branch itself. Thus, the third predicted successor of F is I's target instruction E.

$$
\begin{aligned}
 & & H &= PS(F, 1) \\
(H = PS(F, 1)) \wedge (I = PS(H, 1)) & \rightarrow & I &= PS(F, 2) \\
(I = PS(F, 2)) \wedge (E = PS(I, 1)) & \rightarrow & E &= PS(F, 3) \\
(E = PS(F, 3)) \wedge (F = PS(E, 1)) & \rightarrow & F &= PS(F, 4) \\
(F = PS(F, 4)) \wedge (H = PS(F, 1)) & \rightarrow & H &= PS(F, 5)
\end{aligned}
$$

The code restructing algorithm for Inline Target Insertion is shown below. It is also illustrated by Figure 9.7.

Algorithm $ITI(N)$ begin

(1) Open N *insertion slots* after every likely branch. [7]

(2) For each likely branch I, adjust its target label from the address of $PS(I,1)$ to (the address of $PS(I,1) + N$).

(3) For each likely branch I, copy its first N predicted successors ($PS(I,1)$,$PS(I,2)$, ...,$PS(I,N)$) into its slots.[8] If some of the inserted instructions are branches, make sure they branch to the same target after copying.[9]

end

The goal of ITI is to ensure that *all original instructions find their predicted successors in the next sequential locations.* This is achieved by inserting the predicted successors of likely branches into their next sequential locations.

We refer to the slots opened by the ITI Algorithm as *insertion slots* instead of more traditional terms such as *delay slots* or *squashing delay slots.* The insertion slots are associated only with likely branches. The instructions in the insertion slots are duplicate copies. All the others are original. This is different from the usual meaning of the terms *delay slots* and *squashing delay slots.* They often refer to sequential locations after both likely and unlikely branches, which can contain original as well as duplicate copies.

Figure 9.8 illustrates the application of $ITI(N = 2)$ to a part of the machine program in Figure 9.2. Step 1 opens two insertion slots for the likely branches F and I. Step 2

[7]It is possible to extend the proofs to a nonuniform number of slots in the same pipeline. The details are not in the scope of this chapter.

[8]This step can be performed iteratively. In the first iteration, the first predicted successors of all likely branches are determined and inserted. Each subsequent iteration inserts one more predicted successor for all the likely branches. It takes N iterations to insert all of the target instructions to their assigned slots.

[9]This is trivial if the code restructuring works on assembly code. In this case, the branch targets are specified as labels. The assembler automatically generates the correct branch offset for the inserted branches.

adjusts the branch labels so that F branches to $A(H)+2$ and I branches to $A(E)+2$. Step 3 copies the predicted successors of F (H and I) and I (E and F) into the insertion slots of F (H' and I') and I (E' and F'). Note that the offsets are adjusted so that I' and F' branch to the same target instructions as I and F. The reader is encouraged to apply $ITI(N=3)$ to the code for more insights into the algorithm.

With Inline Target Insertion, each instruction may be copied into multiple locations. Therefore, the same instruction may be fetched from one of the several locations. The *original address*, $A_o(I)$, of a dynamic instruction is the address of the original copy of I. The *fetch address*, $A_f(I)$, of a dynamic instruction I is the address from which I was fetched. In Figure 9.8, the original address of both I and I' is the address of I. The fetch addresses of I and I' are their individual addresses.

It should be noted that ITI moves $fallthru(I)$ of a likely branch I to $A_o(I)+N+1$, which is an original address.

9.3.3 Sequencing pipeline implementation

The sequencing pipeline is divided into $N+1$ stages. The sequencing pipeline processes all instructions in their fetch order. If any instruction is delayed due to a condition in the sequencing pipeline, e.g., instruction cache miss, all of the other instructions in the sequencing pipeline are delayed. This includes the instructions ahead of the one being delayed. The net effect is that the entire sequencing pipeline freezes. This ensures that the relative pipeline timing among instructions is accurately exposed to the compiler. It guarantees that when a likely branch redirects instruction fetch, all instructions in its insertion slots have entered the sequencing pipeline. Note that this restriction applies only to the instructions in the sequencing pipeline; the instructions in the execution pipelines, e.g., data memory access and floating point evaluation, can still proceed while the instruction sequencing pipeline freezes.

The definition of time in instruction sequencing separates the freeze cycles from the execution cycles. Freeze cycles do not affect the relative timing among instructions in the sequencing pipeline. Cycle t refers to the t^{th} cycle of program execution excluding

the freeze cycles. Instruction $I(k,t)$ is defined as the dynamic instruction at the k^{th} stage of the sequencing pipeline during cycle t; $I(1,t)$ is the tail and $I(N+1,t)$ is the front of the fetch pipeline. The implementation keeps an array of fetch addresses for all of the instructions in the sequencing pipeline. The fetch address for the instruction at stage i in cycle t will be referred to as $A_f(I(i,t))$.

A hardware function $REFILL$[10] is provided to reload the instruction fetch pipeline from any original address. $REFILL$ is called when there is a program startup, an incorrect branch prediction, or a return from interrupt/exception. It is easy to guarantee that the program startup address is an original address. We will show in the next subsection that the appropriate original address for a program to resume after incorrect branch prediction and interrupt/exception handling is always available.

$REFILL(pc)$ begin

$A_f(I(N+1,t+1)) \leftarrow pc$;

for $k = 1..N$ do $A_f(I(N-k+1,t+1)) \leftarrow pc+k$;

end

The pipelined instruction fetch algorithm (PIF) that is implemented in hardware is shown below. The sequencing pipeline fetches instructions sequentially by default. Each branch can redirect the instruction fetch and/or scratch the subsequent instructions when it reaches the end of the sequencing pipeline. If a branch redirects the instruction fetch, the next fetch address is the adjusted target address determined in Algorithm ITI. If the decision of a branch is incorrectly predicted, it scratches all of the subsequent instructions from the sequencing pipeline.

[10] $REFILL$ is excluded from the accounting of time when proving the correctness of Inline Target Insertion. $REFILL$ may be physically implemented as loading an initial address into $A_f(I(1,t))$ and subsequently computing $A_f(I(1,t+k)) = A_f(I(1,t+k-1))+1$, for $k = 1...N$. $REFILL$ is included in the accounting of time when evaluating the performance of Inline Target Insertion (Section 9.4).

Algorithm $PIF(N)$ begin

if $(I(N+1,t)$ is not a branch) then

$A_f(I(1,t+1)) \leftarrow A_f(I(1,t)) + 1;$

for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t));$

else if $(I(N+1,t)$ is likely and is taken) then

$A_f(I(1,t+1)) \leftarrow A_o(target(I(N+1,t))) + N;$

for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t));$

else if $(I(N+1,t)$ is unlikely and is not taken) then

$A_f(I(1,t+1)) \leftarrow A_f(I(1,t)) + 1;$

for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t));$

else if $(I(N+1,t)$ is unlikely but is taken) then

$REFILL(A_o(target(I(N+1,t))));$

else if $(I(N+1,t)$ is likely but is not taken) then

$REFILL(A_f(I(1,t)) + 1);$

end

Figure 9.9(a) shows a timing diagram for executing the instruction sequence $(E \rightarrow F \rightarrow H \rightarrow I \rightarrow E)$ of the machine program in Figure 9.8(a). With Inline Target Insertion (Figure 9.8(e)), the instruction sequence becomes $(E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E')$. In this case, the branch decision for F is predicted correctly at compile time. When F reaches the EX stage in cycle 4, no instruction is scratched from the pipeline. Since F redirects the instruction fetch, the instruction to be fetched by the IF stage in cycle 5 is E' (the adjusted target of F) rather than the next sequential instruction G.

Figure 9.9(b) shows a similar timing diagram for executing the instruction sequence $(E \rightarrow F \rightarrow G)$. With Inline Target Insertion, the instruction fetch sequence becomes $(E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow G)$. In this case, the branch decision for F is predicted incorrectly at compile time. When F reaches the EX stage in cycle 4, instructions H'

and I' are scratched from the pipeline. Since F does not redirect the instruction fetch, the instruction fetch pipeline is refilled from the next sequential instruction G.

9.3.4 Correctness of implementation

Branches are the central issue of Inline Target Insertion. Without branches, the sequencing pipeline would simply fetch instructions sequentially. The instructions emerging from the sequencing pipeline would be the correct sequence. Therefore, the correctness proofs of the compiler and pipeline implementation will focus on the correct execution of branches. For pipelines with many slots, it is highly probable to have branches inserted into insertion slots (see Section 9.4.2). In the case where there are no branches in insertion slots, the correctness follows from the description of the ITI Algorithm. All branch instructions would be original and they would have their first N predicted successors in the next N sequential locations, whereas a branch instruction in an insertion slot cannot have all of its N predicted successors in the next N sequential locations. For example, in Figure 9.8(e), questions arise regarding the correct execution of F'. When F' redirects the instruction fetch, how do we know that the resulting instruction sequence is always equivalent to the correct sequence $F \rightarrow H \rightarrow I$...?

Definition 1 *Inline Target Insertion is correct if the instruction sequence that is generated by* (PIF, P_p) *is* $(I_0, CS(I_0,1), CS(I_0,2), ..., CS(I_0,n))$, *where* $CS(I_0,n)$ *is the first stop instruction.*

We shall prove that the instruction sequence that is issued by (PIF, P_p) is identical to that issued by (SIF, P_s). Unfortunately, it is difficult to compare the output of PIF and SIF on a step-by-step basis. We will first identify sufficient conditions for (PIF, P_p) to generate the same instruction sequence as (SIF, P_s), and then show that these conditions are guaranteed by Inline Target Insertion.

To help the reader to read the following lemmas and theorems, we list important notations in Table 9.2. We define two assertions on the state variables of the instruction fetch pipeline.

R(t): $I(i,t) = PS(I(N+1,t), N-i+1), i = 1...N.$

S(t): $A_f(I(1,t)) = A_o(I(N+1,t)) + N.$

Theorem 1 states that these two equality relations are sufficient to ensure the correctness of Inline Target Insertion.

Theorem 1 *If $R(t)$ and $S(t)$ are true for all t, then $I(N+1,t) = CS(I_0,t)$.*

Proof: The theorem can be proved by induction on t.

$P(t) : I(N+1,t) = CS(I_0,t)$.

Induction basis: *From the definition of $REFILL$, $I(N+1,0) = I_0$. $P(0)$ is true for $t = 0$.*

Induction step: *Assuming $P(t)$ is true, we show $P(t+1)$ is also true.*

Case 1: *$I(N+1,t)$ is not an incorrectly predicted branch.*

According to PIF, $I(N+1,t+1) = I(N,t)$. $R(t)$ implies that $I(N,t) = PS(I(N+1,t),1)$. For a correctly predicted instruction $I(N+1,t)$, $PS(I(N+1,t),1)$ is equal to $CS(I(N+1,t),1)$. Hence, $I(N+1,t+1) = I(N,t) = PS(I(N+1,t),1) = CS(I(N+1,t),1) = CS(I_0,t+1)$.

Case 2: *$I(N+1,t)$ is unlikely but is taken.*

PIF performs $REFILL(A_o(target(I(N+1,t))))$ at t. According to the definition of $REFILL$, $I(N+1,t+1)$ becomes $target(I(N+1,t))$ which is $CS(I(N+1,t),1)$. Hence, $I(N+1,t+1) = CS(I(N+1,t),1) = CS(I_0,t+1)$.

Case 3: *$I(N+1,t)$ is likely but is not taken.*

PIF performs $REFILL(A_f(I(1,t))+1)$ at t. According to the definition of $REFILL$ and $S(t)$, $A_f(I(N+1,t+1)) = A_f(I(1,t))+1 = A_o(I(N+1,t))+N+1$. Because $I(N+1,t)$ is a likely branch, ITI allocates N insertion slots after $A_o(I(N+1,t))$, and $fallthru(I(N+1,t))$ is at $A_o(I(N+1,t))+N+1$.[11] *Because $I(N+1,t)$ is*

[11] It should be noted that, if $I(N+1,t)$ is a likely branch, the original copy of $fallthru(I(N+1,t))$ is always at $A_o(I(N+1,t))+N+1$ according to ITI. Therefore, $A_o(I(N+1,t))+N+1$ is a legal argument for $REFILL$.

not taken, $CS(I(N+1,t),1)$ *is* $fallthru(I(N+1,t))$. *Hence,* $I(N+1,t+1) = fallthru(I(N+1,t)) = CS(I(N+1,t),1) = CS(I_0,t+1)$.

□

Theorem 1 shows that $R(t)$ and $S(t)$ are sufficient to ensure correct execution. Therefore, we formulate the next theorem as the ultimate correctness proof of Inline Target Insertion.

Theorem 2 *ITI and PIF ensure that* $R(t)$ *and* $S(t)$ *are true for all* t.

Theorem 2 has a standard induction proof. We start by proving that $R(0)$ and $S(0)$ are true. Then we show that, if $R(t)$ and $S(t)$ are true, $R(t+1)$ and $S(t+1)$ are also true. Because PIF and ITI are complex algorithms, we need to consider several cases in each step of the proof. Instead of presenting the proof as a whole, we will first present several lemmas, from which the proof of Theorem 2 naturally follows.

Lemma 1 *Let* I_{entry} *be an original instruction. If* $REFILL(A_o(I_{entry}))$ *is performed at time* t *so that* I_{entry} *is* $I(N+1,t+1)$ *then* $R(t+1)$ *and* $S(t+1)$ *are true.*

Proof:

ITI ensures that the original instructions find their N predicted successors in their next N sequential addresses. $R(t+1)$ *naturally follows the definition of REFILL.*

$A_f(I(1,t+1)) = A_f(I(N+1,t+1)) + N$ *is implied by the definition of REFILL. Because* $A_f(I(N+1,t+1)) = A_o(I(N+1,t+1))$, $A_f(I(1,t+1)) = A_o(I(N+1,t+1))+N$. *Therefore,* $S(t+1)$ *is also true.*

□

Lemma 1 shows that refilling the instruction fetch pipeline from an original address ensures that $R(t+1)$ and $S(t+1)$ are true. The instruction sequence pipeline is initialized by $REFILL(A_o(I_0))$, where I_0 is the entry point of a program. It follows from Lemma 1 that $R(0)$ and $S(0)$ are true.

We proceed to prove that, if $R(t)$ and $S(t)$ are true, $S(t+1)$ is also true. We first prove for the case when $I(N+1,t+1)$ is fetched from its original address, and then prove for the case when $I(N+1,t+1)$ is fetched from one of its duplicate addresses.

Lemma 2 *If $R(t)$ and $S(t)$ are true and $A_f(I(N+1,t+1)) = A_o(I(N+1,t+1))$, then $S(t+1)$ is also true.*

Proof:

Since $I(N+1,t+1)$ is fetched from its original address, $I(N+1,t)$ cannot be a likely branch. We need to consider only the following two cases.

Case 1: *$I(N+1,t)$ is not a branch or is an unlikely branch which is not taken.*

PIF performs $A_f(I(1,t+1)) = A_f(I(1,t)) + 1$ for this case.

Adding 1 to both sides of $S(t)$ results in $A_f(I(1,t)) + 1 = A_o(I(N+1,t)) + N + 1$. Because ITI allocates insertion slots only for likely branches and $I(N+1,t)$ is not a likely branch, the original addresses of $I(N+1,t)$ and $I(N+1,t+1)$ must be adjacent to each other. In other words, $A_o(I(N+1,t)) + 1 = A_o(I(N+1,t+1))$. Hence, $A_f(I(1,t+1)) = A_f(I(1,t)) + 1 = A_o(I(N+1,t)) + N + 1 = A_o(I(N+1,t+1)) + N$. Therefore, $S(t+1)$ is true.

Case 2: *$I(N+1,t)$ is an unlikely branch but is taken.*

PIF performs $REFILL(A_o(target(I(N+1,t))))$ at time t. The correctness of $S(t+1)$ follows from Lemma 1. Note that $A_o(target(I(N+1,t)))$ is an original (and therefore legal) address for REFILL.

□

The case in which $I(N+1,t+1)$ is fetched from an insertion slot is fairly difficult to prove. We will first prove an intermediate lemma.

Lemma 3 *If $A_f(I(N+1,t+1)) \neq A_o(I(N+1,t+1))$, then there must be a k that satisfies all of the following four conditions.*

(1) $0 \leq k \leq N-1$.

(2) $I(N+1, t-k)$ *is a likely branch.*

(3) There are no likely branches between $I(N+1, t-k+1)$ *and* $I(N+1, t)$ *inclusively.*

(4) There is no incorrectly predicted branch between $I(N+1, t-k)$ *and* $I(N+1, t)$ *inclusively.*

Proof:

Since $I(N+1, t+1)$ *is not fetched from its original address, it must be fetched from an insertion slot. Therefore, there must be at least one likely branch among the* N *instructions fetched before* $I(N+1, t+1)$*. The one that is fetched closest to* $I(N+1, t+1)$ *satisfies (1), (2), and (3).*

We can prove (4) by contradiction. Assume that there was an incorrectly predicted branch between $I(N+1, t-k)$ *and* $I(N+1, t)$ *inclusively. Then, a* $REFILL$ *was performed after* $(t-k-1)$ *at an original address. Because there was no likely branch between* $I(N+1, t-k+1)$ *and* $I(N+1, t)$ *inclusively,* $I(N+1, t+1)$ *must be fetched from its original address. This is a contradiction to the hypothesis of this Lemma:* $A_f(I(N+1, t+1)) \neq A_o(I(N+1, t+1))$.

□

Lemma 4 *If* $A_f(I(N+1, t+1)) \neq A_o(I(N+1, t+1))$ *and* $R(t)$ *and* $S(t)$ *are true, then* $S(t+1)$ *is also true.*

Proof:

We will use the k *found in Lemma 3.*

Case 1: $k = 0$.[12]

$I(N+1, t)$ *is a likely branch. In this case,* PIF *performs* $A_f(I(1, t+1)) = A_o(target(I(N+1, t))) + N$. $R(t)$ *implies that* $I(N, t) = PS(I(N+1, t), 1)$. *Because* PIF *performs* $A_f(I(N+1, t+1)) = A_f(I(N, t))$ *for this case,* $I(N+1, t+1) = PS(I(N+1, t), 1) = target(I(N+1, t))$ *and* $A_o(I(N+1, t+1)) = A_o(target(I(N+1, t)))$. *Therefore,* $A_f(I(1, t+1)) = A_o(target(I(N+1, t))) + N = A_o(I(N+1, t+1)) + N$.

[12]Case 1 could be included in Case 2 of the proof. We separate the two cases to make the proof more clear.

Case 2: $1 \leq k \leq N-1$.

(1) Because $I(N+1,t-k)$ was a likely branch, PIF performed $A_f(I(1,t-k+1)) = A_o(target(I(N+1,t-k))) + N$.

(2) Because $I(N+1,t-k)$ was a likely branch, $I(N,t-k) = target(I(N+1,t-k))$. Therefore, $A_o(I(N+1,t-k+1)) = A_o(I(N,t-k)) = A_o(target(I(N+1,t-k))$.

(3) Because there was no likely branch between $I(N+1,t-k+1)$ and $I(N+1,t)$ inclusively, $A_f(I(1,t+1)) = A_f(I(1,t-k+1)) + k$.

(4) From (1), (2) and (3), $A_f(I(1,t+1)) = A_o(I(N+1,t-k+1)) + N + k$.

(5) Because there was no likely branch between $I(N+1,t-k+1)$ and $I(N+1,t)$ inclusively, $A_o(I(N+1,t-k+1)) + k = A_o(I(N+1,t+1))$.

(6) From (4) and (5), $A_f(I(1,t+1)) = A_o(I(N+1,t+1)) + N$.

□

Lemmas 2 and 4 together ensure that, if $S(i)$ and $R(i)$ are true for $0 \leq i \leq t$, $S(t+1)$ is also true. We proceed to show that $R(t+1)$ is also true.

Lemma 5 *If $R(t)$, $S(t)$, and $S(t+1)$ are true, then $R(t+1)$ is also true.*

Proof:

Case 1: *$I(N+1,t)$ is an incorrectly predicted branch.*

For this case, PIF performs a REFILL. Lemma 1 ensures that $I(i,t+1) = PS(I(N+1,t+1),N-i+1), i = 1...N$ after a REFILL.

It remains to be shown that the argument to REFILL is an original address. If $I(N+1,t)$ is an unlikely branch, the argument to REFILL is $A_o(target(I(N+1,t)))$ which is an original address.

If $I(N+1,t)$ is a likely branch, the argument to REFILL is $A_f(I(1,t)) + 1$. According to Lemmas 2 and 4, $A_f(I(1,t)) + 1 = A_o(I(N+1,t)) + N + 1$. Because $I(N+1,t)$ is a likely branch, ITI ensures that $A_o(I(N+1,t)) + N + 1 = (A_o(fallthru(I(N+1,t))))$.

Case 2: *$I(N+1,t)$ is not an incorrectly predicted branch.*

(1) From Lemmas 2 and 4, $A_f(I(1,t+1)) = A_o(I(N+1,t+1)) + N$.

(2) According to ITI, an original instruction can find its predicted successors in the next sequential instructions. Therefore, $I(1,t+1)$ must be $PS(I(N+1,t+1),N)$ to be placed in $A_o(I(N+1,t+1)) + N$.

(3) Because $I(N+1,t)$ is not an incorrectly predicted branch, PIF performs for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t))$. Therefore, $R(t)$ implies that $I(i,t+1) = PS(I(N+1,t+1),N-i+1)$ for $i = 2...N$.

(4) From (2) and (3), $R(t+1)$ is true.

□

Proof of Theorem 2 By induction on t. It follows from Lemma 1 that $R(0)$ and $S(0)$ are true. From Lemmas 2, 4, and 5, if $R(t)$ and $S(t)$ are true, $R(t+1)$ and $S(t+1)$ are also true.

□

9.3.5 Interrupt/exception return

The problem of interrupt/exception return arises when interrupts and exceptions occur to instructions in insertion slots. For example, assume that the execution of code in Figure 9.8(e) involves an instruction sequence, $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E' \rightarrow F'$. Branch F is correctly predicted to be taken. The question is, if H' caused a page fault, how much instruction sequencing information must be saved so that the process can resume properly after the page fault is handled? If one saved only the address of H', the information about F being taken is lost. Since H' is a not a branch, the hardware would assume that I' was to be executed after H'. Since I' is a likely branch and is taken, the hardware would incorrectly assume that G and H resided in the insertion slots of I'. The instruction execution sequence would become $H' \rightarrow I' \rightarrow G \rightarrow H \rightarrow ...$, which is incorrect.

The problem is that resuming execution from H' violated the restriction that an empty sequencing pipeline always starts fetching from an original instruction. The hardware does not have the information that H' was in the first branch slot of F and that F was taken before the page fault occurred. Because interrupts and exceptions can occur to instructions in all insertion slots of a branch and there can be many likely branches in the slots, the problem cannot be solved by simply remembering the branch decision for one previous branch.

A popular solution to this problem is to save all of the previous N fetch addresses plus the fetch address of the reentry instruction. During exception return, all of the $N+1$ fetch addresses will be used to reload their corresponding instructions to restore the instruction sequencing state to before the exception. The disadvantage of this solution is that it increases the number of states in the pipeline control logic and can therefore slow down the circuit. The problem becomes more severe for pipelines with a large number of slots.

In Inline Target Insertion, interrupt/exception return to an instruction I is correctly performed by $REFILL(A_o(I))$. The memory address $A_o(I(N+1,t))$ is always available in the form of $A_f(I(1,t)) - N$ (Theorem 2). One can record the original addresses when delivering an instruction to the execution units. This guarantees that the original addresses of all instructions active in the execution units are available. Therefore, when an interrupt/exception occurs to an instruction, the processor can save the original address of that instruction as the return address. Lemma 1 ensures that $R(t+1)$ and $S(t+1)$ are true after $REFILL$ from an original address.

Figure 9.10 shows the effect of an exception on the sequencing pipeline. Figure 9.10(a) shows the timing of a correct instruction sequence $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E' \rightarrow F'$ from Figure 9.8(e) without exception. Figure 9.10(b) shows the timing with an exception to H'. When H' reaches the end of the sequencing pipeline (EX stage) at t, its $A_o(H')$ is available in the form of $A_f(I(1,t)) - N = A_f(E') - 2$. This address will be maintained

by the hardware until H' finishes execution.[13] When an exception is detected, $A_o(H')$ is saved as the return address. During exception return, the sequencing pipeline resumes instruction fetch from H, the original copy of H'. Note that the instruction sequence produced is $H \rightarrow I \rightarrow E'$, which is equivalent to the one without exception.

Note that the original copies must be preserved to guarantee clean implementation of interrupt/exception return. In Figure 9.8(e), if normal control transfers always enter the section at E', there is an opportunity to remove E and F after Inline Target Insertion to reduce code size. However, this would prevent a clean interrupt/exception return if one occurs to E' or F'. Section 4.2 presents an alternative approach to reducing code expansion.

9.3.6 Extension to out-of-order execution

Inline Target Insertion can be extended to handle instruction sequencing for out-of-order execution machines [Tomasulo 67], [Weiss 84], [Acosta 86], [Hwu 87], [Hwu 88a], [Smith 89]. The major instruction sequencing problem for out-of-order execution machines is the indeterminate timing of computing branching conditions and target addresses. It is not feasible in general to design an efficient sequencing pipeline in which branches always have their conditions and target addresses at the end of the sequencing pipeline. To allow efficient out-of-order execution, the sequencing pipeline must allow the subsequent instructions to proceed whenever possible.

To make Inline Target Insertion and its correctness proofs applicable to out-of-order execution machines, the following changes should be made to the pipeline implementation.

(1) The sequencing pipeline is long enough to identify the target addresses for program-counter-relative branches and for those whose target addresses can be derived without interlocking.

[13]The real original address does not have to be calculated until an exception is detected. One can simply save $A_f(I(1,t))$ and calculate only $A_o(I(N+1,t))$ when an exception actually occurs. This avoids requiring an extra subtractor in the sequencing pipeline.

(2) When a branch reaches the end of the sequencing pipeline, the following may occur:

(a) The branch is likely and its target address is not available yet. In this case, the sequencing pipeline freezes until the interlock is resolved.

(b) The branch is unlikely and its target address is not yet available. In this case, the sequencing pipeline proceeds with the subsequent instructions. Extra hardware must be added to hold the target address when it becomes available to recover from incorrect branch prediction. The execution pipeline must also be able to cancel the effects of the subsequent instructions emerging from the sequencing pipeline for the same reason.

(c) The branch condition is not yet available. In this case, the sequencing pipeline proceeds with the subsequent instructions. Extra hardware must be added to hold the repair address to recover from incorrect branch prediction. The execution pipeline must be able to cancel the effects of the subsequent instructions emerging from the sequencing pipeline for the same reason.

If a branch is program-counter-relative, both the predicted and alternative addresses are available at the end of the sequencing pipeline. The only difference from the original sequencing pipeline model is that the condition might be derived later. Since the hardware secures the alternative address, the sequencing state can be properly recovered from incorrectly predicted branches. If the branch target address is derived from run-time data, the target address of a likely branch may be unavailable at the end of the sequencing pipeline. Freezing the sequencing pipeline in the above specification ensures that all theorems hold for this case. As for unlikely branches, the target address is the alternative address. The sequencing pipeline can proceed as long as the alternative address is secured when it becomes available. Therefore, all of the proofs above hold for out-of-order execution machines.

9.3.7 Issuing multiple branch operations per cycle

Inline Target Insertion can be extended to handle multiple branch operations per cycle. For each instruction which can contain up to M operations, a linear ordering is assumed among the M operations. All M operations of an instruction can be branches. All branch operations, except one, must be unlikely (no branch slots). In other words, there are two situations: (1) all branch operations are unlikely, and (2) one branch operation is likely and all other branch operations are unlikely. Each branch slot is an instruction. Branch slots are allocated after an instruction, when there is a likely branch operation in the instruction.

We need to define the semantics of an instruction with multiple branch operations. Let $oper(i), i = 1...M$ denote the operations of the instruction at the end of the fetch pipeline ($I(N+1,t)$).

(1) next-pc = $A_f(I(1,t))$ + 1;

(2) for ($i = 1..M$) do

if (*oper*(i) is not a branch operation)
allow *oper*(i) to proceed in the instruction pipeline;

if (*oper*(i) is unlikely and is not taken)
allow *oper*(i) to proceed in the instruction pipeline;

if (*oper*(i) is unlikely but is taken)
allow *oper*(i) to proceed in the instruction pipeline;
squash *oper*(k), $k = i+1..M$;
squash all later instructions in the fetch pipeline;
next-pc = the target address of *oper*(i);

if (*oper*(i) is likely and is taken)
allow *oper*(i) to proceed in the instruction pipeline;
squash *oper*(k), $k = i+1..M$;
next-pc = the target address of *oper*(i) + N;

if ($oper(i)$ is likely but is not taken)

allow $oper(i)$ to proceed in the instruction pipeline;

squash all later instructions in the fetch pipeline;

(3) $A_f(I(1,t))$ = next-pc;

If an instruction does not contain a likely branch operation, then we consider the instruction as a unlikely branch instruction. If an instruction contains a likely branch operation, then we consider the instruction as a likely branch instruction. In the instruction level, the theorems that we have proven in this chapter remain valid.

9.4 Experiments

The code expansion cost and instruction sequencing efficiency of Inline Target Insertion can be evaluated only empirically. This section reports experimental results based on a set of production quality software from UNIX[14] and CAD domains. The purpose is to show that Inline Target Insertion is an effective method for achieving high instruction sequencing efficiency for pipelined processors. All of the experiments are based on the an instruction set architecture which closely resembles MIPS R2000/3000 [Kane 87] with modifications to accommodate Inline Target Insertion. The IMPACT-I C Compiler, an optimizing C compiler developed for deep pipelining and multiple-instruction-issue at the University of Illinois, is used to generate code for all of the experiments.

9.4.1 The benchmarks

Table 9.3 presents the benchmarks chosen for this experiment. The *C lines* column describes the size of the benchmark programs in number of lines of C code (not counting comments). The *runs* column shows the number of inputs used to generate the profile databases and the performance measurement. The *input description* column briefly describes the inputs for the benchmarks. The inputs are realistic and representative of

[14]UNIX is a trademark of AT&T.

typical uses of the benchmarks. For example, the grammars for a C compiler and for a LISP interpreter are two of the ten realistic inputs for *bison* and *yacc*. Twenty files of several production-quality C programs, ranging from 100 to 3000 lines, are inputs to the *cccp* program. All of the twenty original benchmark inputs form the input to *espresso*. The experimental results are reported, based on the mean and sample deviations of all program and input combinations shown in Table 9.3. The use of many different real inputs to each program is intended to verify the stability of Inline Target Insertion using profile information. The IMPACT-I compiler automatically applies trace selection and placement, and removes unnecessary unconditional branches via code restructuring.

9.4.2 Code expansion

The problem of code expansion has to do with the frequent occurrence of branches in programs. Inserting target instructions for a branch adds N instructions to the static program.[15] In Figure 9.8, target insertion for F and I increases the size of the loop from 5 to 9 instructions. In general, if Q is the probability that a static instruction is a likely branch ($Q = 0.18$ among all the benchmarks), Inline Target Insertion can potentially increase the code size by $N * Q$ (1.80 for $Q = 0.18$ and $N = 10$). Because code expansion can significantly reduce the efficiency of hierarchical memory systems, the problem of code expansion must be addressed for pipelines with a large number of slots.

Table 9.4 shows the static control transfer characteristics of the benchmarks. The *static cond.* (*static uncond.*) column gives the percentages of conditional (unconditional) branches among all the static instructions in the programs. The numbers presented in Table 9.4 confirm that branches appear frequently in static programs. This shows the need for being able to insert branches in the insertion slots (see Section 9.3.4). The high percentage of branches suggests that code expansion must be carefully controlled for these benchmarks.

[15]One may argue that the originals of the inserted instructions may be deleted to save space if the flow of control allows. We have shown, however, that preserving the originals is crucial to the clean return from exceptions in insertion slots (see Section 9.3.5).

A simple method to control code expansion is to reduce the number of likely branches in static programs using a threshold method. A conditional branch that executes fewer times than a threshold value is automatically converted into an unlikely branch. An unconditional branch instruction that executes fewer times than a threshold value can also be converted into an unlikely branch whose branch condition is always satisfied. The method reduces the number of likely branches at the cost of some performance degradation. A similar idea has been implemented in the IBM Second Generation RISC Architecture [Bakoglu 89].

For example, if there are two likely branches A and B in the program, A is executed 100 times and it redirects the instruction fetch 95 times; B is executed 5 times and it redirects the instruction fetch 4 times. Marking A and B as likely branches achieves correct branch prediction 99 (95+4) times out of a total of 105 (100+5). The code size increases by $2 * N$. Since B is not executed nearly as frequently as A, one can mark B as an unlikely branch. In this case, the accuracy of branch prediction is reduced to be 96 (95+1) times out of 105. The code size increases only by N. Therefore, a large saving in code expansion could be achieved at the cost of a small loss in performance.

The idea is that all static likely branches cause the same amount of code expansion but their execution frequency may vary widely. Therefore, by reversing the prediction for the infrequently executed likely branches reduces code expansion at the cost of a slight loss of prediction accuracy. This is confirmed by results shown in Table 9.5. The *threshold* column specifies the minimum dynamic execution count per run, below which likely branches are converted to unlikely branches. The $E[Q]$ column lists the mean percentage of likely branches among all instructions and the $SD[Q]$ column indicates the sample deviations. The code expansion for a pipeline with N slots is $N * E[Q]$. For ($N = 2$) with a threshold value of 100, one can expect a 2.2% increase in the static code size. Without code expansion control (threshold=0), the static code size increase would be 36.2% for the same sequencing pipeline. For an 11-stage sequencing pipeline ($N = 10$) with a threshold value of 100, one can expect about an 11% increase in the static code size. Without code expansion control (threshold=0), the static code size increase would

be 181% for the same sequencing pipeline. Note that the results are based on control intensive programs. The code expansion cost should be much lower for programs with simple control structures such as scientific applications.

9.4.3 Instruction sequencing efficiency

The problem of instruction sequencing efficiency is concerned with the total number of dynamic instructions scratched from the pipeline due to all dynamic branches. Since all insertion slots are inserted with predicted successors, the cost of instruction sequencing is a function of only N and the branch prediction accuracy. The key issue is whether the accuracy of compile-time branch prediction is high enough to ensure that the instruction sequencing efficiency remains high for large values of N.

Evaluating the instruction sequencing efficiency with Inline Target Insertion is straightforward. One can profile the program to find the frequency for the dynamic instances of each branch to go in one of the possible directions. Once a branch is predicted to go in one direction, the frequency for the branch to go in other directions contributes to the frequency of incorrect prediction. Note that only the correct dynamic instructions reach the end of the sequencing pipeline in which branches are executed. Therefore, the frequency of executing incorrectly predicted branches is not affected by Inline Target Insertion.

In Figure 9.11(a), the execution frequencies of F and I are both 100, and E and F redirect the instruction fetch 80 and 99 times, respectively. By marking F and I as likely branches, we predict them correctly for 179 times out of 200. That is, 21 dynamic branches will be incorrectly predicted. Since each incorrectly predicted dynamic branch creates N nonproductive cycles in the sequencing pipeline, we know that the instruction frequencing cost is $21*N$. Note that this number is not changed by Inline Target Insertion. Figure 9.11(b) shows the code generated by ITI(2). Although we do not know exactly how many times F and F' were executed, respectively, we know that their total execution count is 100. We also know that the total number of incorrect predictions for F and F'

is 20. Therefore, the instruction sequencing cost of Figure 9.11(b) can be derived from the count of incorrect predictions in Figure 9.11(a) multiplied by N.

Let P denote the probability that any dynamic instruction is incorrectly predicted. Note that this probability is calculated for all dynamic instructions, including both branches and nonbranches. The average instruction sequencing cost can be estimated by the following equation:

$$relative\ sequencing\ cost\ per\ instruction = 1 + P * N \tag{9.1}$$

If the peak sequencing rate is $1/K$ cycles per instruction, the actual rate would be $(1 + P * N)/K$ cycles per instruction.[16]

Table 9.4 highlights the dynamic branch behavior of the benchmarks. The *dynamic cond.* (*dynamic uncond.*) column gives the percentage of conditional (unconditional) branches among all the dynamic instructions in the measurement. The dynamic percentages of branches confirm that branch handling is critical to the performance of processors with a large number of branch slots. For example, 20% of the dynamic instructions of *bison* are branches. The P value for this program is the branch prediction miss ratio times 20%. Assume that the sequencing pipeline has a peak sequencing rate of one cycle per instruction ($K = 1$) and it has three slots ($N = 3$). The required prediction accuracy to achieve a sequencing rate of 1.1 cycles per instruction can be calculated as follows:

$$1.1 >= 1 + (1 - accuracy) * 0.2 * 3 \tag{9.2}$$

The prediction accuracy must be at least 83.3%.

Table 9.6 provides the mean and sample deviations of P for a spectrum of thresholds averaged over all benchmarks. Increasing the threshold effectively converts more branches into unlikely branches. With $N = 2$, the relative sequencing cost per instruction is 1.036 per instruction for threshold equals zero (no optimization). For a sequencing pipeline whose peak sequencing rate is one instruction per cycle, this means a sustained rate of

[16]This formula provides a measure of the efficiency of instruction sequencing. It does not take external events such as instruction misses into account. Since such external events freeze the sequencing pipeline, one can simply add the extra freeze cycles into the formula to derive the actual instruction fetch rate.

1.036 cycles per instruction. For a sequencing pipeline which sequences k instructions per cycle, this translates into $1.036/k$ (0.518 for $k = 2$) cycles per instruction. When the threshold is set to 100, the relative sequencing cost per instruction is 1.04. With $N = 10$, the relative sequencing cost per instruction is 1.18 for threshold equals zero (no optimization). When the threshold is set to 100, the sequencing cost per instruction becomes 1.20. Comparing Tables 9.5 and 9.6, it is obvious that converting infrequently executed branches into unlikely branches reduces the code expansion at little cost of instruction sequencing efficiency.

9.5 Conclusions

We have defined Inline Target Insertion, a cost-effective instruction sequencing method extended from the work of McFarling and Hennessy [McFarling 86]. The compiler and pipeline implementation offers two important features. First, branches can be freely inserted into branch slots. The instruction sequencing efficiency is limited solely by the accuracy of the compile-time branch prediction. Second, the execution can return from an interruption/exception to a program with one program counter. There is no need to reload other sequencing pipeline state information. These two features make Inline Target Insertion a superior alternative (better performance and less software/hardware complexity) to the conventional delayed branching mechanisms.

Inline Target Insertion has been implemented in the IMPACT-I C Compiler to verify the compiler implementation complexity. The software implementation is simple and straightforward. A code expansion control method is also proposed and included in the IMPACT-I C Compiler implementation. The code expansion and instruction sequencing efficiency of Inline Target Insertion have been measured for UNIX and CAD programs. The experiments involve the execution of more than a billion instructions. The size of programs, variety of programs, and variety of inputs to each program are significantly larger than those used in the previous experiments.

The overall compile-time branch prediction accuracy is about 92% for the benchmarks in this study. For a pipeline which requires 10 branch slots and fetches two instructions per cycle, this translates into an effective instruction fetch rate of 0.6 cycles per instruction (see Section 9.4.3). To achieve the performance level reported in this chapter, the instruction format must give the compiler complete freedom to predict the direction of each static branch. While this can be easily achieved in a new instruction set architecture, it could also be incorporated into an existing architecture as an upward compatible feature.

It is straightforward to compare the performance of Inline Target Insertion and that of Branch Target Buffers. For the same pipeline, the performance of both are determined by the branch prediction accuracy. Hwu, Conte and Chang [Hwu 89b] performed a direct comparison between Inline Target Insertion and Branch Target Buffers based on a similar set of benchmarks. The conclusion was that, without context switches, Branch Target Buffers achieved an instruction sequencing efficiency slightly lower than Inline Target Insertion. Context switches could significantly enlarge the difference [Lee 84]. All in all, Branch Target Buffers have the advantages of binary compatibility with existing architectures and no code expansion. Inline Target Insertion has the advantage of not requiring extra hardware buffers, better performance, and performance insensitive to context switching.

The results in this chapter do not suggest that Inline Target Insertion is always superior to Branch Target Buffering. Rather, the contribution is to show that Inline Target Insertion is a cost-effective alternative to Branch Target Buffer. The performance is not a major concern. Both achieve very good performance for deep pipelining and multiple-instruction-issue. The compiler support of Inline Target Insertion is simple enough not to be a major concern either. This has been proven in the IMPACT-I C Compiler implementation. If the cost of fast hardware buffers and context switching are not major concerns but binary code compatibility and code size are, then Branch Target Buffer should be used. Otherwise, Inline Target Insertion should be employed for its better performance characteristics and lower hardware cost.

Table 9.1 A summary of delayed branching mechanisms.

Scheme	Hardware features	Compiler features
Delayed branches	None	Fill slots with independent code
Delayed branches with squashing	Uniform prediction and squashing	Fill slots with independent code or instructions from the predicted path
Profiled delayed branches with squashing	Prediction bit and squashing	Execution profiling Fill slots with instructions from the predicted path

Table 9.2 A summary of important definitions used in the proofs.

$N+1$	The number of stages in the instruction sequencing pipeline
$I(k,t)$	The dynamic instruction occupying the k^{th} pipeline stage at cycle t
$A_f(I)$	The fetch address of dynamic instruction I
$A_o(I)$	The original address of dynamic instruction I
$PS(I,k)$	The k^{th} predicted successor of I
$CS(I,k)$	The k^{th} correct successor of dynamic instruction I
$R(t)$	$I(i,t) = PS(I(N+1,t), N-i+1),\ i = 1...N$
$S(t)$	$A_f(I(1,t)) = A_o(I(N+1,t)) + N$

Table 9.3 Benchmarks.

name	*C lines*	*runs*	*input description*
bison	6913	10	grammar for a C compiler, etc.
cccp	4660	20	C programs (100-3000 lines)
cmp	371	16	similar/dissimilar text files
compress	1941	20	same as cccp
eqn	4167	20	papers with .EQ options
espresso	11545	20	original espresso benchmarks
grep	1302	20	exercised various options
lex	3251	4	lexers for C, Lisp, awk, and pic
make	7043	20	makefiles for cccp, compress, etc.
tar	3186	14	save/extract files
tbl	4497	20	papers with .TS options
tee	1063	18	text files (100-3000 lines)
wc	345	20	same as cccp
yacc	3333	10	grammar for a C compiler, etc.

Table 9.4 Static and dynamic characteristics.

benchmark	static *cond.*	static *uncond.*	dynamic *cond.*	dynamic *uncond.*
bison	0.12	0.17	0.19	0.01
cccp	0.10	0.11	0.17	0.04
cmp	0.09	0.15	0.16	0.04
compress	0.09	0.14	0.11	0.01
eqn	0.08	0.12	0.21	0.02
espresso	0.09	0.12	0.13	0.02
grep	0.15	0.19	0.30	0.05
lex	0.15	0.16	0.30	0.01
make	0.12	0.14	0.18	0.01
tar	0.10	0.17	0.12	0.00
tbl	0.18	0.20	0.21	0.05
tee	0.09	0.15	0.29	0.07
wc	0.07	0.10	0.22	0.02
yacc	0.14	0.15	0.23	0.01

Table 9.5 Percentage of likely branches among all static instructions.

threshold	*E[Q]*	*SD[Q]*
0	18.1%	3.7%
1	4.8%	2.1%
10	2.1%	1.6%
20	1.8%	1.5%
40	1.5%	1.3%
60	1.3%	1.2%
80	1.2%	1.1%
100	1.1%	1.0%
200	0.9%	0.8%
400	0.6%	0.6%
600	0.5%	0.5%

Table 9.6 Probability of prediction miss among all dynamic instructions.

threshold	*E[P]*	*SD[P]*
0	0.018	0.010
1	0.018	0.010
10	0.019	0.010
20	0.019	0.010
40	0.020	0.010
60	0.020	0.010
80	0.020	0.010
100	0.020	0.010
200	0.023	0.010
400	0.023	0.010
600	0.025	0.011

(a)

```
MaxElement = 0;
for (i = 0; i < IMax; i++) {
    if (Array[i] > MaxElement) MaxElement = Array[i];
} ...
```

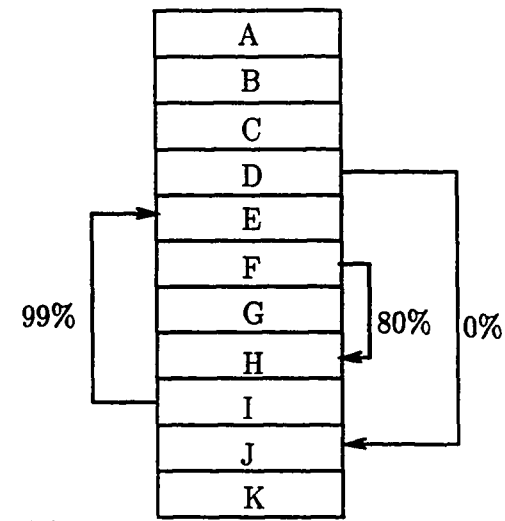
(b)

r1 ← i
r2 ← temporary for Array[i]
r3 ← IMax
r4 ← MaxElement

Figure 9.1 (a) An example C program for finding the largest element in Array. (b) The register assignment.

A: r4 ← 0
B: r1 ← 0
C: r3 ← IMax
D: if (r1 ≥ r3) goto J [unlikely]
E: r2 ← Array(r1)
F: if (r2 ≤ r4) goto H [likely]
G: r4 ← r2
H: r1 ← r1 + 1
I: if (r1 < r3) goto E [likely]
J: MaxElement ← r4
K: ...

(a)



(b)

Figure 9.2 (a) A machine language program generated from the C program shown in Figure 9.1. (b) A simplified view of the machine language program.

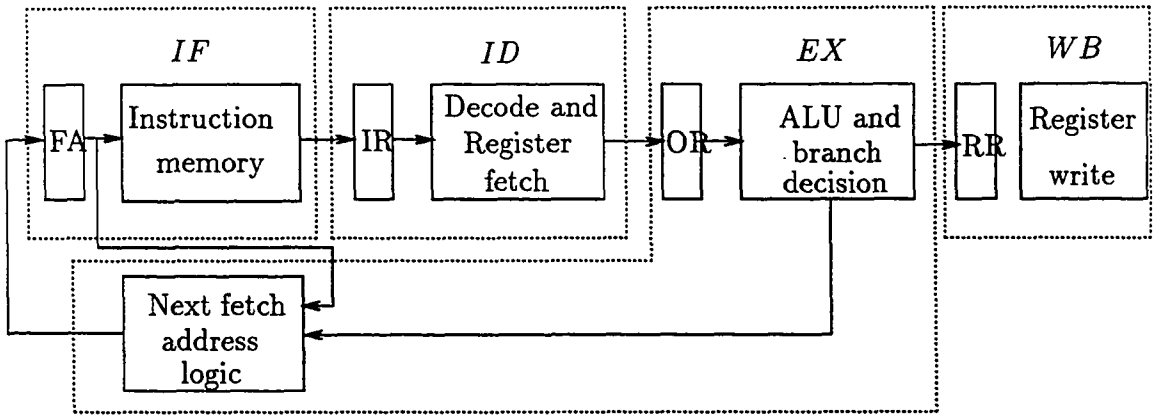


Figure 9.3 A block diagram and a simplified view of a pipelined processor.

	IF	*ID*	*EX*	*WB*
1	*E*			
2	*F*	*E*		
3	*G*	*F*	*E*	
4	*H*	*G*	*F*	*E*
5	*I*	*H*	*G*	*F*
6	*J*	*I*	*H*	*G*
7	*K*	*J*	*I*	*H*
8	*E*			*I*
9	*F*	*E*		

Figure 9.4 A timing diagram of the pipelined processor in Figure 9.3 executing the sequence of instructions $E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow E \rightarrow F$ of Figure 9.2. Instructions J and K are scratched from the pipeline because I is taken.

	IF_1	IF_2	ID	EX_1	EX_2	WB
1	I_1					
2	I_2	I_1				
3	I_3	I_2	I_1			
4	I_4	I_3	I_2	I_1		
5	I_5	I_4	I_3	I_2	I_1	
6	I_6					I_1

Figure 9.5 A timing diagram of a pipelined processor which results from further dividing the IF and EX stages of the processor in Figure 9.3.

	IF	ID	EX	WB
1	I_2,I_1			
2	I_4,I_3	I_2,I_1		
3	I_6,I_5	I_4,I_3	I_2,I_1	
4	I_8,I_7			I_1

Figure 9.6 A timing diagram of the pipelined processor which processes two instructions in parallel.

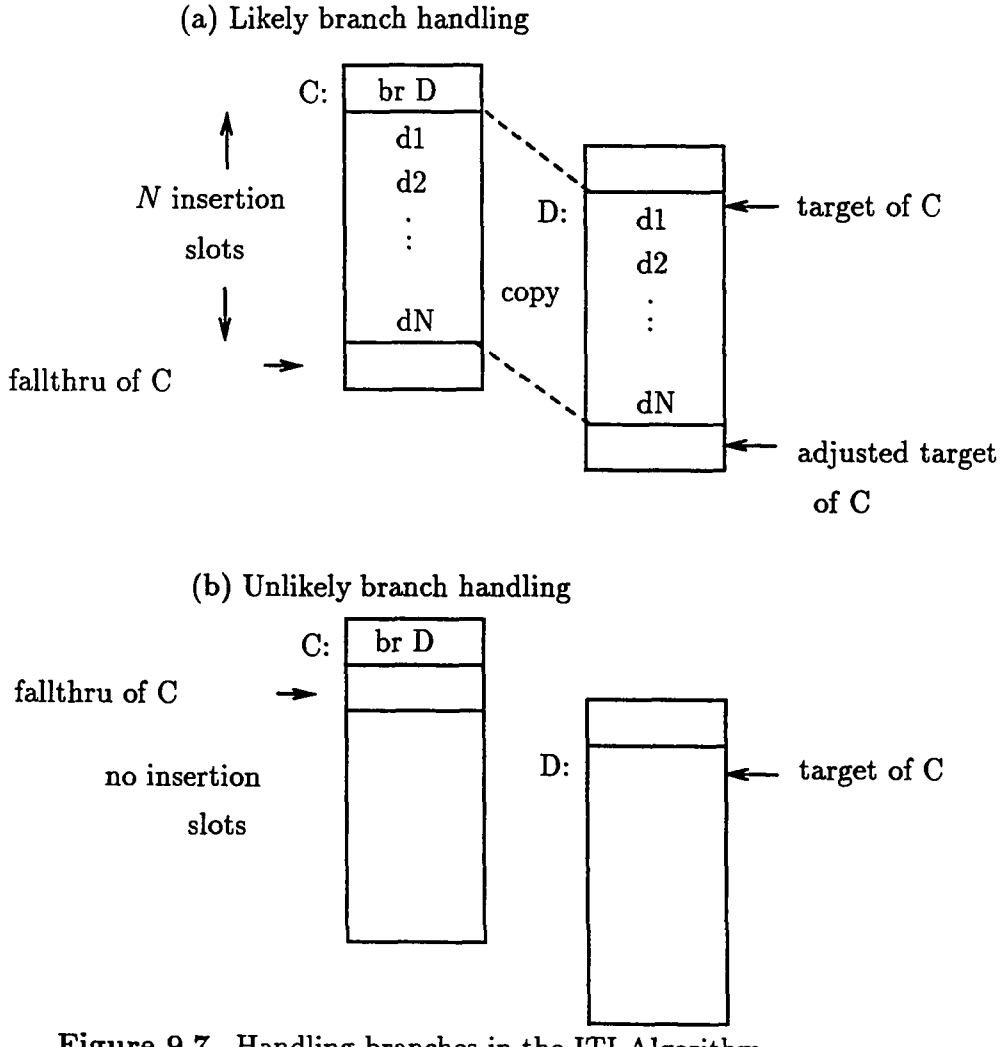


Figure 9.7 Handling branches in the ITI Algorithm.

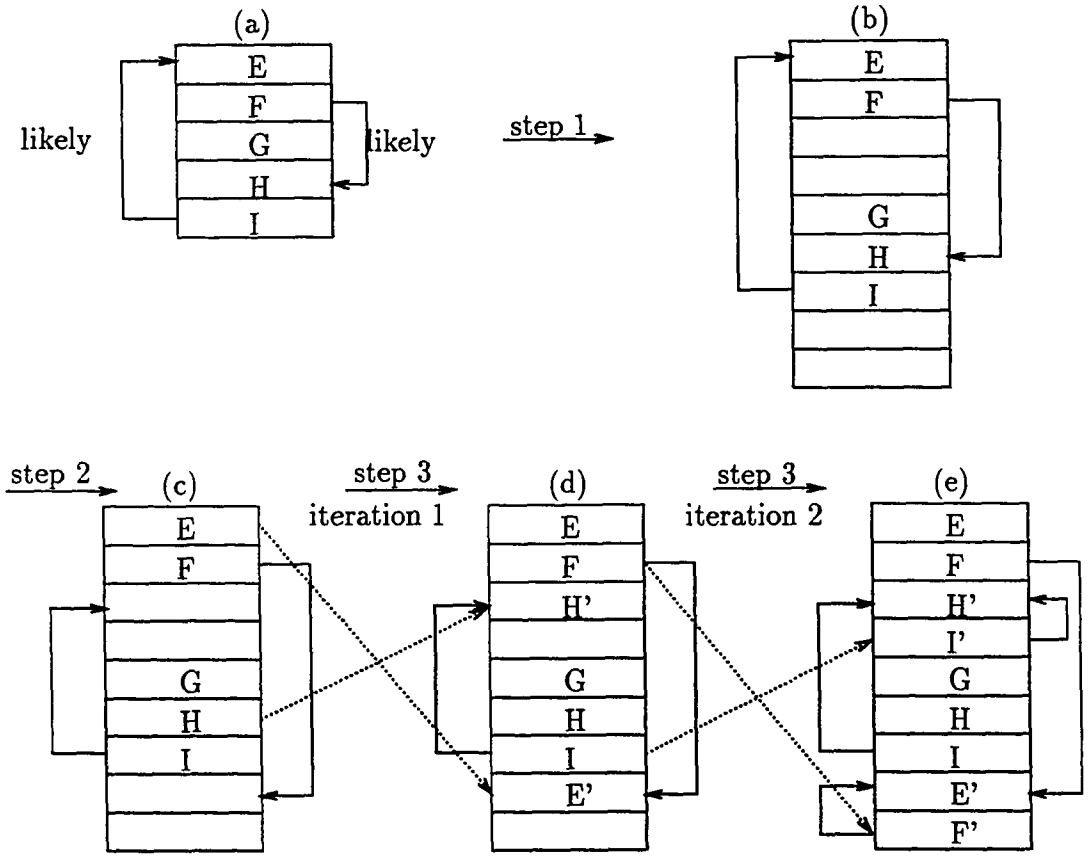


Figure 9.8 A running example of Inline Target Insertion.

(a)

	IF	*ID*	*EX*	*WB*
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	E'	I'	H'	F

(b)

	IF	*ID*	*EX*	*WB*
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
$REFILL(A_o(G))$				
5	I	H	G	F

Figure 9.9 (a) Timing diagram of a pipelined processor executing the sequence, $E \rightarrow F \rightarrow H'$ ... of instructions in Figure 9.8(e). (b) A similar timing diagram for the sequence, $E \rightarrow F \rightarrow G$ ... .

(a)

	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	E'	I'	H'	F
6	F'	E'	I'	H'

(b)

	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	E'	I'	H'	F
$REFILL(A_o(H))$				
6	E'	I	H	
7	F'	E'	I	H

Figure 9.10 (a) Timing diagram of a pipelined processor executing the sequence $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E'$ of instructions in Figure 9.8(e). (b) Timing diagram of a pipelined processor executing the sequence $E \rightarrow F \rightarrow H' \rightarrow I \rightarrow E$ of instructions in Figure 9.8(e) because of an interrupt at I'.

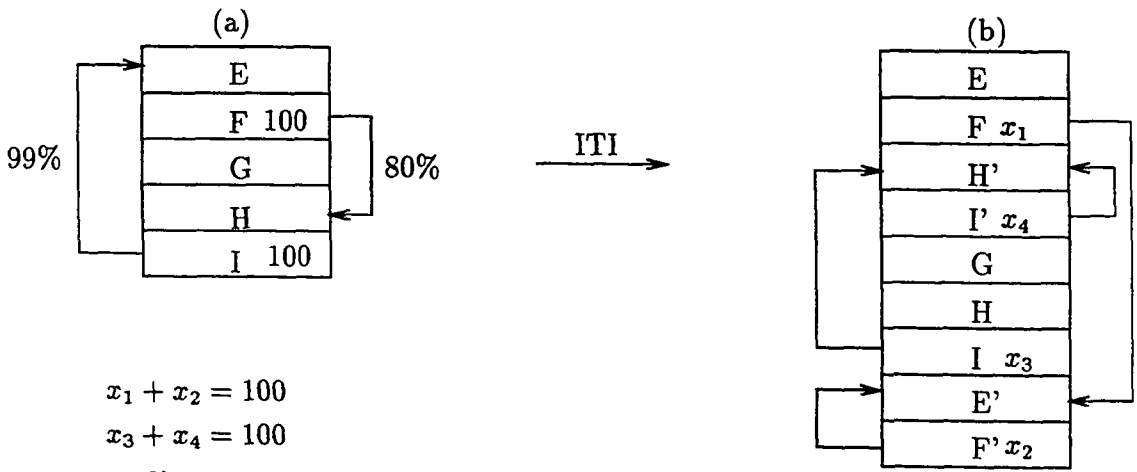


Figure 9.11 Evaluating the efficiency of instruction sequencing.

CHAPTER 10

CONCLUSIONS

10.1 Summary

For a set of realistic C programs, we have shown that an optimizing compiler can restructure the programs and identify instruction-level parallelism, which is then mapped onto the parallel microarchitectures to reduce the execution cycle count. This dissertation has shown that multiple-instruction-issue processors substantially outperform single-instruction-issue processors. For a four-operation-issue processor, programs run more than twice as fast as possible on the best single-operation-issue processor (assuming they have the same machine cycle time).

In the course of this research, we have developed the IMPACT-I C compiler. The IMPACT-I C compiler has an open architecture that allows quick changes. Additional code optimizations can be easily integrated and tested in the IMPACT-I C compiler. The IMPACT-I C compiler uses two levels of intermediate forms. Source code transformation techniques such as function inline expansion have been implemented in the high-level intermediate form, Hcode. Traditional code optimizations have been implemented in the low-level intermediate form, Lcode.

Automatic profiling capabilities have been added to the IMPACT-I C compiler. The decision components of code optimizations have access to the profile (run-time) information as well as to static loop analysis. When the resources, e.g., registers, function units, are scarce, the profile information helps the compiler to allocate resources to the most frequently executed program regions and to the most frequently accessed variables. We

have implemented profiling in three different levels: the preprocessor level, the Hcode level, and the Lcode level. All three have been effective.

It is important to evaluate the performance of multiple-instruction-issue architectures using highly optimized code for two reasons. First, a naive compiler can produce redundant computations that show deceptive parallelism. Second, we report the speedup over the most efficient sequential code. To generate efficient code, we have implemented a large set of code optimizations. The control components of these optimizations use the profile information. The code quality that is currently produced by the IMPACT-I C compiler for the DECstation is comparable to that of one of the best commercial C compilers.

We have proposed and implemented a large set of code transformation techniques that enlarge the scope of static code scheduling. We have also implemented a large set of code transformation techniques that reduce the lengths of critical paths. These code optimizations have exposed the instruction-level parallelisms of the benchmark programs to the code scheduler.

We have identified the importance of several new code optimizations. First, the instruction placement optimization reduces the number of taken branches, increases instruction cache sequential locality, and produces longer super-blocks. Second, the loop peeling optimization expands the scope of static code scheduling for infrequently iterated loops. Third, the branch target expansion optimization reduces the number of taken branches and enlarges the sizes of super-blocks. Fourth, the induction variable expansion and the register renaming optimizations allow unrolled loop iterations to be merged. Fifth, the integrated register allocation and code scheduling scheme reduces penalties due to artificial data dependencies that are introduced by register allocation.

We have identified three levels of static code scheduling models: restricted code percolation, general code percolation, and speculative execution. We have implemented code scheduling algorithms for each of the three models. For many multiple-instruction-issue architectures, we have shown that the general code percolation model is the most cost-effective model among the three.

Using the IMPACT-I C compiler, we have evaluated the performance of many multiple-instruction-issue architectures. We have evaluated the effect of limiting some function unit resources for different instruction issue bandwidths. The experimental data indicate that, for high issue rate architectures, the ability to execute multiple branch and memory load operations is important. We have also evaluated the effect of varying memory load operation latency. The experimental data show that increasing the memory load latency severely degrades the performance of high issue rate architectures. We have compared the effectiveness of static code scheduling and dynamic code scheduling to improve the performance of existing processor architectures. The experimental data indicate that static code scheduling and dynamic code scheduling have their own merits and limitations. The best approach might be to employ both static and dynamic code scheduling.

We have defined the IMPACT architectural framework of multiple-instruction-issue processors. Using a simple in-order execution microarchitecture, we have achieved high performance using compile-time code optimizations. We have developed the inline target insertion technique that allows multiple branch operations to be issued per cycle and branch operations to be fetched from branch slots.

We have released the first beta test version of the IMPACT-I C compiler to NCR in February 1991. We plan to release the IMPACT/AMD29K C compiler in April 1991. We also plan to release the second beta test version of the IMPACT-I C compiler in May 1991.

Companies that design and manufacture microprocessors are welcome to adopt the IMPACT architectural framework. Processor designs under the IMPACT framework are fully supported by the IMPACT-I C compiler technology.

10.2 Future Directions

The original contributors of the IMPACT-I C compiler are extending the IMPACT-I compiler technology for shared-memory multiprocessing. The first target machine is an Alliant FX/2800 multiprocessor, which uses i860 microprocessors as its node processors.

Nancy Warter, who wrote the i860 code generator, is in the process of implementing a compiler based on the IMPACT-I C compiler components for an extended C programming language, which has *doall* and *doacross* loop constructs. She will study how to partition a parallel program, e.g., nested loops, to tasks that are executed on superscalar microprocessors. Scott Mahlke, who wrote a large portion of the code optimizer, will design, implement and evaluate code optimizations for shared-memory multiprocessing. William Chen, who wrote the MIPS R2000 code generator, will study the interrelationships between code optimizations and shared-memory multiprocessor architectures.

Several new members of the IMPACT group are improving existing components and implementing new components for the IMPACT-I C compiler. John Holm is fine-tuning the SPARC code generator, which was written by Roland Ouellette. Rick Hank is constructing a new register allocator. Dan Lavery is studying the interrelationship between global graph-coloring register allocation and trace scheduling. Grant Haab is constructing a memory dependence analyzer with array subscript analysis. Roger Bringmann is fine-tuning his AMD29K code generator and will study how to generate code for embedded applications.

The author and several people in the IMPACT group are using the IMPACT-I C compiler to conduct several experiments. First, we are comparing the performance of superpipelining architectures to that of multiple-instruction-issue architectures. The benchmark programs include nonnumeric C application programs and some numeric FORTRAN application programs, e.g., SPEC FORTRAN benchmarks and Perfect Club benchmarks. These FORTRAN programs are converted to C using the GNU F2C program. Second, we are evaluating the effect of code optimizations on instruction-level parallelisms and on processor architectures, e.g., instruction cache. Third, we are designing more code optimizations for multiple-instruction-issue architectures (Chapter 7). We will study the interrelationships between these code optimizations and specific hardware features of multiple-instruction-issue architectures.

In the future, the author is interested in adding more frontends to the IMPACT-I compiler framework and extending the IMPACT-I C compiler to an object-oriented

compiler, perhaps C++, for distributed parallel processing. With frontends for other programming languages, such as LISP, Ada and Prolog, the author can show whether the multiple-instruction-issue code optimizations that have been developed for C are effective for other programming languages, and whether multiple-instruction-issue processors achieve substantial speedups of LISP, Ada or Prolog programs.

REFERENCES

[Acosta 86] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors," *IEEE Transactions on Computers,* vol. C-35, no. 9, pp. 815-828, September 1986.

[Agerwala 76] T. Agerwala, "Microprogram optimization: a survey," *IEEE Transactions on Computers,* vol. C-25, no. 10, October 1976.

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.

[Allen 74] F. E. Allen, "Interprocedural data flow analysis," *Proceedings of the IFIP Congress,* 1974.

[Allen 76] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Journal of ACM,* vol. 19, no. 3, March 1976.

[Allen 88] R. Allen and S. Johnson, "Compiling C for vectorization, parallelism, and inline expansion," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* June 1988.

[Alpert 88] D. B. Alpert and M. J. Flynn, "Performance trade-offs for microprocessor cache memories," *IEEE MICRO,* August 1988.

[Amd] Advanced Micro Devices, "Am29000 streamlined instruction processor, advance information," Publication Number 09075, Rev. A, Amendment /0, Sunnyvale, California.

[Arya 85] S. Arya, "An optimal instruction-scheduling model for a class of vector processors," *IEEE Transactions on Computers,* vol. C-34, no. 11, November 1985.

[Auslander 82] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," *Proceedings of the SIGPLAN Symposium on Compiler Construction,* June 1982.

[Bakoglu 89] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye, "The IBM RISC system/6000 processor: hardware overview," *IBM Journal of Research and Development,* vol. 34, no. 1, pp. 12-22, January 1990.

[Barth 78] J. M. Barth, "A practical interprocedural data flow analysis algorithm," *Journal of ACM,* vol. 21, no. 9, September 1978.

[Birnbaum 86] J. S. Birnbaum and W. S. Worley, "Beyond RISC: high precision architecture," *Spring COMPCON,* 1986.

[Breternitz 88] M. Breternitz Jr. and J. P. Shen, "Organization of array data for concurrent memory access," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture,* November 1988.

[Bruno 80] J. Bruno, J. W. Jones, III, and K. So, "Deterministic scheduling with pipelined processors," *IEEE Transactions on Computers,* vol. C-29, no. 4, April 1980.

[Chaitin 82] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM SIGPLAN Notice '82,* vol. 17, no. 6, June 1982.

[Chang 88] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures,* pp. 21-29, November 1988.

[Chang 89a] P. P. Chang, "Aggressive code improving techniques based on control flow analysis," M.S. Thesis, University of Illinois, Champaign-Urbana, (CSG-105), UILU-ENG-89-2228, September 1989.

[Chang 89b] P. P. Chang and W. W. Hwu, "Forward semantic: a compiler-assisted instruction fetch method for heavily pipelined processors," *Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture,* August 1989.

[Chang 89c] P. P. Chang and W. W. Hwu, "Control flow optimization for supercomputer scalar processing," *Proceedings of the 1989 International Conference on Supercomputing,* June 1989.

[Chang 91a] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: an architectural framework for multiple-instruction-issue processors," to appear in the *Proceedings of the 18th International Symposium on Computer Architecture,* May 1991.

[Chang 91b] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," Center for Reliable and High-performance Computing Report, University of Illinois, CRHC-91, April 1991.

[Chow 84] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," *Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions,* June 1984.

[Chow 87] P. Chow and M. Horowitz, "Architecture tradeoffs in the design of MIPS-X," *Proceedings of the 14th Annual International Symposium on Computer Architecture,* June 1987.

[Chow 88] F. C. Chow, "Minimizing register usage penalty at procedure calls," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* June 1988.

[Clark 87] D. W. Clark, "Pipelining and performance in the VAX 8800 processor," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* October 1987.

[Coffman 76] E. G. Coffman, *Computer and Job-Shop Scheduling Theory.* New York: Wiley, 1976.

[Cohn 89] R. Cohn, T. Gross, M. Lam, and P. S. Tseng, "Architecture and compiler tradeoffs for a long instruction word microprocessors," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

[Colwell 87] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1987.

[Dasgupta 80] S. Dasgupta, "Some aspects of high-level microprogramming," *Computing Surveys,* vol. 12, no. 3, September 1980.

[Davidson 86] J. W. Davidson, "A retargetable instruction reorganizer," *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction,* June 1986.

[Davidson 81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some experiments in local microcode compaction for horizontal machines," *IEEE Transactions on Computers,* vol. C-30, no. 7, July 1981.

[DeRosa 85] J. Derosa, R. Glackemeyer, and T. Knight, "Design and implementation of the VAX 8600 pipeline," *IEEE Computer,* May 1985.

[DeRosa 88] J. A. DeRosa and H. M. Levy, "An evaluation of branch architectures," *Proceedings of the 15th International Symposium on Computer Architecture,* May 1988.

[Ditzel 87] D. R. Ditzel and H. R. McLellan, "Branch folding in the CRISP microprocessor: reducing branch delay to zero," *Proceedings of the 14th Annual International Symposium on Computer Architecture,* pp. 2-9, June 1987.

[Dongarra 79.2] J.J. Dongarra and A.R. Jinds, "Unrolling loops in Fortran," *Software Practice and Experience,* vol. 9, no. 3, pp. 219-226, March 1979.

[Eickenmeyer 88] R. J. Eickenmeyer and J. H. Patel, "Performance evaluation of on-chip register and cache organizations," *Proceedings of the 15th International Symposium on Computer Architecture,* May 1988.

[Eisenbeis 88] C. Eisenbeis, "Optimization of horizontal microcode generation for loop structures," *Proceedings of the 1988 International Conference on Supercomputing,* July 1988.

[Ellis 86] J. R. Ellis, *Bulldog: a Compiler for VLIW Architectures.* Cambridge, Massachusetts: The MIT Press, 1986.

[Emer 84] J. Emer and D. Clark, "A characterization of processor performance in the VAX-11/780," *Proceedings of the 11th Annual Symposium on Computer Architecture,* June 1984.

[Ferrari 83] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

[Fisher 81] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers,* vol. C-30, no. 7, July 1981.

[Fisher 83] J. A. Fisher, "VLIW architectures and the ELI-512," *Proceedings of the 10th Annual Symposium on Computer Architecture,* June 1983.

[Flynn 85] M. J. Flynn, J. D. Johnson, and S. P. Wakefield, "On instruction sets and their formats," *IEEE Transactions on Computers,* vol. C-34, no. 3, March 1985.

[Foster 72] C. C. Foster and E. M. Riseman, "Percolation of code to enhance parallel dispatching and execution," *IEEE Transactions on Computers,* vol. C-21, pp. 1411-1415, December 1972.

[Gannon 88] D. Gannon, "Strategies for cache and local memory management by global program transformation," *Journal of Parallel and Distributed Computing,* vol. 5, 1988.

[Garey 79] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness.* New York: W.H. Freeman and Company, 1979.

[Gibbons 86] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction,* June 1986.

[Golumbic 90] M.C. Golumbic and V. Rainish, "Instruction scheduling beyond basic blocks," *IBM Journal of Research and Development,* vol. 34, no. 1, pp. 93-97, January 1990.

[Gonzalez 77] M. J. Gonzalez, "Deterministic processor scheduling," *Computing Surveys,* vol. 9, no. 3, September 1977.

[Goodman 88] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," *Proceedings of the 1988 International Conference on Supercomputing,* St. Malo, July 1988.

[Granski 87] M. Granski, I. Koren, and G. M. Silberman, "The effect of operation scheduling on the performance of a data flow computer," *IEEE Transactions on Computers,* vol. C-36, no. 9, September 1987.

[Gross 82] T. R. Gross and J. L. Hennessy, "Optimizing delayed branches," *Proceedings of the 15th Microprogramming Workshop,* pp. 114-120, October 1982.

[Gross 86] T. Gross and M. S. Lam, "Compilation for a high-performance systolic array," *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction,* June 1986.

[Hartley 88] S. J. Hartley, "Compile-Time program restructuring in multiprogrammed virtual memory systems," *IEEE Transactions on Software Engineering,* vol. 14, no. 11, November 1988.

[Hecht 75] M. S. Hecht and J. D. Ullman, "A simple algorithm for global data flow analysis problems," *SIAM Journal of Computing,* vol. 4, no. 4, December 1975.

[Hennessy 81] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: a VLSI processor architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations,* October 1981.

[Hennessy 82] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "Hardware/software tradeoffs for increased performance," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* March 1982.

[Hennessy 83] J. L. Hennessy and T. Gross, "Postpass code optimization of pipelined constraints," *ACM Transactions on Programming Languages and Systems,* vol. 5, pp. 422-448, ACM, July 1983.

[Hill 85] M. D. Hill and A. J. Smith, "Experimental evaluation of on-chip cache memories," *Proceedings of the 11th Annual Symposium on Computer Architecture,* June 1985.

[Hill 88] M. D. Hill, "A case for direct-mapped caches," *IEEE Computer,* December 1988.

[Horst 90] R.W. Horst, R.L. Harris, and R.L. Jardine, "Multiple instruction issue in the nonstop cyclone processor," *Proceedings of the International Symposium on Computer Architecture,* May 1990.

[Howland 87] M. A. Howland, R. A. Mueller, and P. H. Sweany, "Trace scheduling optimization in a retargetable microcode compiler," *Proceedings of the 20th International Microprogramming Workshop,* December 1987.

[Hsu 86] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," *Proceedings of the 13th International Symposium on Computer Architecture,* June 1986.

[Huson 82] C. A. Huson, "An in-line subroutine expander for parafrase," M.S. Thesis, University of Illinois, Champaign-Urbana, 1982.

[Hwu 86] W. W. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," *Proceedings of the 13th International Symposium on Computer Architecture,* pp. 297-306, June 1986.

[Hwu 87] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high performance out-of-order execution machines," *IEEE Transactions on Computers,* December 1987.

[Hwu 88a] W. W. Hwu, "Exploiting concurrency to achieve high performance in a single-chip microarchitecture," Ph.D. Dissertation, Computer Science Division Report, no. UCB/CSD 88/398, University of California, Berkeley, January 1988.

[Hwu 88b] W. W. Hwu and P. P. Chang, "Exploiting parallel microprocessor microarchitectures with a compiler code generator," *Proceedings of the 15th International Symposium on Computer Architecture,* May 1988.

[Hwu 89a] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," *Proceedings of the 16th International Symposium on Computer Architecture,* June 1989.

[Hwu 89b] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," *Proceedings of the 16th International Symposium on Computer Architecture,* May 1989.

[Hwu 89c] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic C programs," *Proceedings, ACM SIGPLAN'89 Conference on Programming Language Design and Implementation,* June 1989.

[Hwu 90] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," Center for Reliable and High-performance Computing Report, University of Illinois, Urbana-Champaign, 1990.

[IBM 90] IBM, *Special Issue on IBM RISC System/6000 Processor, IBM Journal of Research and Development,* vol. 34, no. 1, January 1990.

[Intel 89] Intel, "i860(TM) 64-Bit microprocessor," Order Number 240296-002, Santa Clara, California, April 1989.

[Isoda 83] S. Isoda, Y. Kobayashi, and T. Ishida, "Global compaction of horizontal microprograms based on the Generalized Data Dependency Graph," *IEEE Transactions on Computers,* vol. C-32, no. 10, October 1983.

[Jouppi 89a] N. P. Jouppi, J. Bertoni, and D. W. Wall, "A unified vector/scalar floating-point architecture," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

[Jouppi 89b] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

[Kane 87] G. Kane, *MIPS R2000 RISC Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1987.

[Karp 66] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: determinacy, termination, queuing," *SIAM Applied Math.,* pp. 1390-1411, November 1966.

[Kleir 71] R. L. Kleir, and C. V. Ramamoorthy, "Optimization strategies for microprograms," *IEEE Transactions on Computers,* vol. C-20, no. 7, July 1971.

[Kogge 81] P. M. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[Kuck 72] D. 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,* vol. C-21, pp. 1293-1310, December 1972.

[Kuck 81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependency graphs and compiler optimizations," *Proceedings of the 8th ACM Symposium on Principles of Programming Languages,* January 1981.

[Lam 88] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* June 1988.

[Landskov 80] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local microcode compaction techniques," *Computing Surveys,* vol. 12, no. 3, September 1980.

[Lawrie 75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers,* vol. C-24, no. 12, December 1975.

[Lee 84] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer,* January 1984.

[Li 88] Z. Li and P-C Yew, "Efficient interprocedural analysis for program parallelization and restructuring," *Proceedings of the ACM/SIGPLAN PPEALS,* 1988.

[Linn 83] J. L. Linn, "SRDAG compaction: a generalization of trace scheduling to increase the use of global context information," *Proceedings of the 16th Microprogramming Workshop,* October 1983.

[Mahlke 91] S. A. Mahlke, N. J. Warter, W. Y. Chen, P. P. Chang, and W. W. Hwu, "The effect of compiler optimizations on available parallelism in scalar programs," to appear in *Proceedings of the 1991 International Conference on Parallel Processing,* August 1991.

[Matick 84] R. E. Matick and D. T. Ling, "Architecture implications in the design of microprocessors," *IBM Systems Journal,* vol. 23, no. 3, 1984.

[McFarling 86] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," *Proceedings of the 13th International Symposium on Computer Architecture,* pp. 396-403, June 1986.

[McFarling 89] S. McFarling, "Program optimization for instruction caches," *Proceedings of the third International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

[Melear 89] C. Melear, "The design of the 88000 RISC family," *IEEE MICRO,* pp. 26-38, April 1989.

[Mitchell 88] C. L. Mitchell and M. J. Flynn, "A workbench for computer architects," *IEEE Design and Test of Computers,* Feburary 1988.

[Nicolau 84] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computer,* vol. C-33, no. 11, November 1984.

[Nicolau 85] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," *Proceedings of the International Conference on Parallel Processing,* pp. 614-618, August 1985.

[Patt 85] Y. N. Patt, W. W. Hwu, and M. C. Shebanow, "HPS, a new microarchitecture: rationale and introduction," *Proceedings of the 18th International Microprogramming Workshop,* pp. 103-108, December 1985.

[Patterson 82] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer,* pp. 8-21, September 1982.

[Pleszkun 87] A. R. Pleszkun, J. R. Goodman, W.-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter, "WISQ: a restartable architecture using queues," *Proceedings of the 14th International Symposium on Computer Architecture,* pp. 290-299, June 1987.

[Pleszkun 88a] A. R. Pleszkun and G. S. Sohi, "The performance potential of multiple functional unit processors," *Proceedings of the 15th International Symposium on Computer Architecture,* May 1988.

[Pleszkun 88b] A. R. Pleszkun and G. S. Sohi, "Multiple instruction issue and single-chip processors," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture,* November 1988.

[Przybylski 88] S. Przybylski, M. Horowitz, and J. L. Hennessy, "Performance tradeoffs in cache design," *Proceedings of the 15th International Symposium on Computer Architecture,* May 1988.

[Radin 82] G. Radin, "The 801 minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 39-47, March 1982.

[Ramamoorthy 74] C. V. Ramamoorthy, and M. Tsuchiya, "A high-level language for horizontal microprogramming," *IEEE Transactions on Computers,* vol. C-23, no. 8, August 1974.

[Rauscher 80] T. G. Rauscher, and P. M. Adams, "Microprogramming: a tutorial and survey of recent developments," *IEEE Transactions on Computers,* vol. C-29, no. 1, January 1980.

[Richardson 89] S. Richardson and M. Ganapathi, "Code optimization across procedures," *IEEE Computer,* February 1989.

[Russell 78] R. M. Russell, "The Cray-1 computer system," *Communications ACM,* vol. 21, no. 1, pp. 63-72, January 1978.

[Sahni 84] S. Sahni, "Scheduling multipipeline and multiprocessor computers," *IEEE Transactions on Computers,* vol. C-33, no. 7, July 1984.

[Scheifler 77] R. W. Scheifler, "An analysis of inline substitution for a structured programming language," *Communications of the ACM,* vol. 20, no. 9, September 1977.

[Sherburne 83] R. Sherburne, M. Katevenis, D. Patterson, and C. Sequin, "Local memory in RISCs," *Proceedings of the International Conference on Computer Design,* October 1983.

[Smith 81] J. E. Smith, "A study of branch prediction strategies," *Proceedings of the 8th International Symposium on Computer Architecture,* pp. 135-148, June 1981.

[Smith 82] A. J. Smith, "Cache memories," *Computing Surveys,* vol. 14, no. 3, ACM, September 1982.

[Smith 85a] J. E. Smith, and Pleszkun, "Implementation of precise interrupts in pipelined processors," *Proceedings of the 11th Annual Symposium on Computer Architectures,* June 1985.

[Smith 85b] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Transactions on Computers,* vol. C-34, no. 3, March 1985.

[Smith 85c] A. J. Smith, "Cache evaluation and the impact of workload choice," *Proceedings of the 12th International Symposium on Computer Architecture,* June 1985.

[Smith 87] A. J. Smith, "Line (block) size choice for CPU cache memories," *IEEE Transactions on Computers,* vol. C-36, no. 9, September 1987.

[Smith 89] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

[Smith 90] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," *Proceedings of the 17th International Symposium on Computer Architecture,* June 1990.

[Sohi 87] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high performance, interruptable pipelined processors," *Proceedings of the 14th Annual Symposium on Computer Architecture,* June 1987.

[Sohi 89] G. S. Sohi and S. Vajapeyam, "Tradeoffs in instruction format design for horizontal architectures," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989.

[Sparc 87] SUN Microsystems, *The SPARC™ Architecture Manual,* Part no. 800-1399-07, Revision 50, Mountain View, California: SUN, August 1987.

[Stallman 88] R. M. Stallman, *Internals of GNU CC*, 1988 (distributed with the GNU CC software).

[Su 84] B. Su, S. Ding, and L. Jin, "An improvement of trace scheduling for global microcode compaction," *Proceedings of the 17th Microprogramming Workshop,* November 1984.

[Thornton 70] J. E. Thornton, *Design of a Computer: The Control Data 6600.* Glenview, IL: Scott, Foresman and Co., 1970.

[Tarjan 83] R. E. Tarjan, *Data Structures and Network Algorithms.* Philadelphia, PA: SIAM, 1983.

[Tjaden 70] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Transactions on Computers,* vol. C-19, no. 10, pp. 889-895, October 1970.

[Tokoro 81] M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of microprograms," *IEEE Transactions on Computers,* vol. C-30, no. 7, July 1981.

[Tomasulo 67] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development,* vol. 11, pp. 25-33, January 1967.

[Tsuchiya 76] M. Tsuchiya, and M. J. Gonzalez, "Toward optimization of horizontal microprograms," *IEEE Transactions on Computers,* vol. C-25, no. 10, October 1976.

[Wall 86] D. W. Wall, "Global register allocation at link time," *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction,* June 1986.

[Wall 88] D. W. Wall, "Register windows vs. register allocation," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* June 1988.

[Warren 90] H. S. Warren, Jr., "Instruction scheduling for the IBM RISC system/6000 processor," *IBM Journal of Research and Development,* vol. 34, no. 1, pp. 85-92, January 1990.

[Weiss 84] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Transactions on Computers,* vol. C-33, no. 11, pp. 1013-1022, November 1984.

[Weiss 87] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1987.

APPENDIX A

MACHINE DESCRIPTION LANGUAGE

The IMPACT-I C compiler is a technology-file-driven compiler that is intended to generate code for multiple target machines. Machine dependent code optimizations such as constant preloading, register allocation, and code scheduling require some knowledge about the target machine. We have developed a simple language for conveying the machine dependent information to the IMPACT-I C compiler.

A.1 Basic Data Types

All integer values are expressed in two's complement number representation, and all floating-point values are expressed in IEEE floating-point representation. Characters follow the ASCII definition. Unsigned character is generally sufficient; however, signed character is also provided. Short integers are provided because some memory mapped I/O devices are short-integer addressable. To support all of the above data types, a large number of memory operations are provided. Memory operations support the following data types:

1) unsigned character (1 byte),

2) signed character (1 byte),

3) unsigned short (2 bytes),

4) signed short (2 bytes),

5) unsigned integer (4 bytes),

6) signed integer (4 bytes),

7) single-precision float (4 bytes), and

8) double-precision float (8 bytes).

When characters and short integers are loaded into registers, the values are automatically zero-extended or sign-extended. Therefore, the number of computation data types is restricted to the following data types:

1) unsigned integer,

2) signed integer,

3) single-precision float, and

4) double-precision float.

A.2 Register Resource

There can be several distinct register sets. Within each register set, registers can be used individually, or neighboring registers can be used as register-pairs. Our model assumes that the total register resource is an array of basic words. Then, we define how several abstract register sets are mapped to the total register resource. Each of these abstract register files represents a view. For example, we can model two disjoint register files as follows:

```
register-file = array [0..63] of 32-bit words;
scalar-register-file = array [0..31] of 32-bit words;
float-register-file = array [0..31] of 32-bit words;
double-register-file = array [0..15] of 64-bit words;
for (i=0..31) scalar-register-file[i] = register-file[i];
for (i=0..31) float-register-file[i] = register-file[i+32];
for (i=0..15) double-register-file[i] = register-file[(i*2)+32..(i*2)+33];
```

We have implemented a prioritized graph-coloring method to map from an infinite number of virtual registers to this physical register model. In our machine description language, the register set organization shown above is as follows:

```
(define_register_type sp1 (INT) 32 0 1)
(define_register_type fp1 (INT FLOAT) 32 32 1)
(define_register_type fp2 (DOUBLE) 16 32 2)
```

A.3 Operation Code

Each operation code is described by a triple *(name, type, delay)*. The *name* field uniquely identifies the operation code. The *type* field specifies whether this operation code represents an arithmetic operation, a control operation, a memory load operation, a memory store operation, a synchronization operation, or a combination of them. The *type* information has a special meaning to the code scheduler. For example, no code motion across a synchronization operation is allowed. The *delay* field specifies the suggested dependence distance from this operation to a next use of the operation's result. For most integer operations, the delay is one. The actual delay may vary at run time. Any additional delay is enforced by the hardware interlock logic.

```
(opcode no_op  0  (arith) void 0)
(opcode add    1  (arith) i    1)
(opcode mul    2  (arith) i    4)
(opcode jump   3  (cnt)   void 0)
(opcode ld_c   4  (load)  i    2)
(opcode st_i   5  (store) void 0)
(opcode fetch_and_add 6 (sync load store arith) i 2)
```

A.4 Operand Addressing Mode

Load/store architecture is very suitable for instruction pipelining. Assuming load/store architecture, we provide very simple operand types: integer, floating-point, label, and register operands. Five predefined functions allow us to define various operand modes.

```
(define_int_operand_type       INT1    -128  +128)
(define_float_operand_type     FLOAT1  0.0   0.0)
(define_double_operand_type    DOUBLE1 0.0   0.0)
(define_label_operand_type     LABEL1  offset)
(define_label_operand_type     LABEL2  direct)
(define_register_operand_type SR1      sp1)
(define_register_operand_type FR1      fp1)
(define_register_operand_type FR2      fp2)
```

One can specify an arbitrary range for an integer and a floating-point type. In the example above, the range of a short integer literal is between -128 and 128, and the floating-point constant literal can be only zero. Several constant integer operand modes may be specified. Constant preloading means loading constant values that do not fit into the constant literal fields in registers. An optimization is to place the preload operations in loop headers or in the function prologue section. An operand field can be one or more of the above types. For example, the first source operand of an *add* operation can be either an INT1 or a SR1.

```
(define_operand_mode DEST (SR1 FR1 FR2))
(define_operand_mode SRC (
    SR1 FR1 FR2 INT1 FLOAT1 DOUBLE1 LABEL1 LABEL2
))
```

A.5 Operation Model

An operation type is a triple *(opc, src, dest)*, where *opc* is an operation code, *src* is a list of source operands, and *dest* is a list of destination operands.

```
(define_operation_type add (add (DEST) (SRC SRC)))
(define_operation_type ld_c (ld_c (DEST) (SRC SRC)))
(define_operation_type st_i (st_i () (SRC SRC SRC)))
```

In the example above, *add* operation has one destination operand and two source operands. The opcode definition and operand modes are also specified.

A.6 Function Unit Model

A function unit is a set of operation types. For example, integer ALU can perform *mov*, *add*, *sub*, *mul*, and many other integer operations.

```
(define_operation_group ALU
    mov  add  sub  mul  div  eq   ne   gt   ge   lt   le
    lsl  lsr  asr  or   and  xor)
(define_operation_group LOAD
    ld_uc  ld_c  ld_uc2  ld_uc  ld_i   ld_f  ld_f2)
(define_operation_group STORE
    st_c  st_c2  st_i  st_f  st_f2)
```

A.7 Instruction Set Model

An instruction may contain one or more operations. Therefore, an instruction type is an ordered set of function units. For example, the instruction set of a machine that issues one operation per cycle may be specified as follows:

```
(define_instruction_type T1 (ALU))
```

```
(define_instruction_type T2 (FALU))
(define_instruction_type T3 (CNT))
(define_instruction_type T4 (SYNC))
(define_instruction_type T5 (LOAD))
(define_instruction_type T6 (STORE))
```

The instruction set of a machine that issues two operations per cycle is specified below.

```
(define_instruction_type T1 (ALU ALU))
(define_instruction_type T2 (ALU FALU))
(define_instruction_type T3 (ALU CNT))
(define_instruction_type T4 (LOAD CNT))
.......
```

The first line specifies that two integer ALU operations can be packed into one instruction. The second line specifies that one integer ALU and one floating-point ALU operation can be packed into one instruction. There is an implicit lexical ordering between operations in the same instruction word. For example, if the hardware allows two operations in the same instruction to write the same register, the operation in later lexical order should make the last write to that register.

APPENDIX B

EXAMPLES OF HCODE AND LCODE

Hcode and Lcode documentations are too long to be included in a dissertation. They are available as internal reports. In the following sections, some Hcode and Lcode files are provided to give the reader a general feeling about the two levels of intermediate forms. These Hcode and Lcode files are automatically generated from the source C program.

B.1 C Source Code

```
#define DIM 1200
typedef int T;

T x[DIM], y[DIM], z[DIM];

main() {
    int i;
    for (i=0; i<DIM; i++)
x[i] = y[i] + z[i];
    exit(0);
}
```

B.2 Hcode

Hcode Profile File:

```
(count 1)
(fn 0 0.000000e+00
(cg 0 (1 1.000000e+00))
)
(fn 1 1.000000e+00
(bb 0 1.000000e+00 (1 1.000000e+00))
(bb 1 1.000000e+00 (2 1.000000e+00))
(bb 2 1.200000e+03 (2 1.199000e+03) (3 1.000000e+00))
(bb 3 1.000000e+00)
)
```

Hcode with Profile Information:

```
(GVAR x ((GLOBAL)(INT)((A (signed 1200)))))
(GVAR y ((GLOBAL)(INT)((A (signed 1200)))))
(GVAR z ((GLOBAL)(INT)((A (signed 1200)))))
(BEGIN_FN main)
 (PROFILE 1 1.000000)
 (RETURN_TYPE ((GLOBAL)(INT)()))
 (LVAR i___1 ((AUTO)(INT)()))
 (FN_PRAGMA "optimize.trace")
 (FN_PRAGMA "profiled.1")
 (FN_PRAGMA "flatten")
 (ENTRY 1)
 (BB 1 (PROFILE 1.000000 (2 1 1.000000))
  (BB_PRAGMA "trace.1")
(assign (var i___1) (signed 0))
```

```
(IF (lt (var i___1) (signed 1200)) (THEN 2) (ELSE 3)) )
 (BB 2 (PROFILE 1200.000000 (2 1 1199.000000) (3 0 1.000000))
  (BB_PRAGMA "trace.2")
(assign (index (var x) (var i___1))
            (add (index (var y) (var i___1)) (index (var z) (var i___1))))
(postinc (var i___1))
(IF (lt (var i___1) (signed 1200)) (THEN 2) (ELSE 3)) )
 (BB 3 (PROFILE 1.000000)
  (BB_PRAGMA "trace.3")
(call (var exit) (signed 0) (EXPR_PRAGMA "cs.1"))
(RETURN) )
(END_FN main)
```

B.3 Lcode

Lcode after Global Code Optimizations:

```
(ms text)
(global _main)
(function _main 1.000000)
  (cb 1 1.000000 (flow 0 2 1.000000))
    (op 0 define ((mac $return_type i))(()()()))
    (op 1 define ((mac $local i))((i 0)()()))
    (op 2 define ((mac $param i))((i 16)()()))
    (op 3 prologue (())(()()()))
  (cb 2 1.000000 (flow 0 3 1.000000))
    (op 4 mov ((r 12 i))((i 0)()()))
    (op 5 mov ((r 41 i))((i 4)()()))
    (op 6 mov ((r 42 i))((i 8)()()))
    (op 7 mov ((r 43 i))((i 12)()()))
```

```
  (op 8 mov ((r 44 i))((i 16)()()))
(cb 3 240.000000 (flow 1 3 239.000000) (flow 0 4 1.000000))
  (op 9 ld_i ((r 25 i))((l _y)(r 12 i)()))
  (op 10 ld_i ((r 26 i))((l _z)(r 12 i)()))
  (op 11 add ((r 27 i))((r 25 i)(r 26 i)()))
  (op 12 st_i (())((l _x)(r 12 i)(r 27 i)))
  (op 13 ld_i ((r 29 i))((l _y)(r 41 i)()))
  (op 14 ld_i ((r 30 i))((l _z)(r 41 i)()))
  (op 15 add ((r 31 i))((r 29 i)(r 30 i)()))
  (op 16 st_i (())((l _x)(r 41 i)(r 31 i)))
  (op 17 ld_i ((r 33 i))((l _y)(r 42 i)()))
  (op 18 ld_i ((r 34 i))((l _z)(r 42 i)()))
  (op 19 add ((r 35 i))((r 33 i)(r 34 i)()))
  (op 20 st_i (())((l _x)(r 42 i)(r 35 i)))
  (op 21 ld_i ((r 37 i))((l _y)(r 43 i)()))
  (op 22 ld_i ((r 38 i))((l _z)(r 43 i)()))
  (op 23 add ((r 39 i))((r 37 i)(r 38 i)()))
  (op 24 st_i (())((l _x)(r 43 i)(r 39 i)))
  (op 25 ld_i ((r 3 i))((l _y)(r 44 i)()))
  (op 26 ld_i ((r 5 i))((l _z)(r 44 i)()))
  (op 27 add ((r 6 i))((r 3 i)(r 5 i)()))
  (op 28 st_i (())((l _x)(r 44 i)(r 6 i)))
  (op 29 add_u ((r 12 i))((r 12 i)(i 20)()))
  (op 30 add_u ((r 44 i))((r 44 i)(i 20)()))
  (op 31 add_u ((r 43 i))((r 43 i)(i 20)()))
  (op 32 add_u ((r 42 i))((r 42 i)(i 20)()))
  (op 33 add_u ((r 41 i))((r 41 i)(i 20)()))
  (op 34 bne_fs (())((r 12 i)(i 4800)(cb 3)))
(cb 4 1.000000 (flow 1 5 1.000000))
```

```
    (op 35 mov ((mac $P0 i))((i 0)()()))
    (op 36 jsr (())((l _exit)()()))
  (cb 5 1.000000)
    (op 37 epilogue (())(()()()))
    (op 38 rts (())(()()()))
(end _main)
```

Lcode after Multiple-Instruction-Issue Code Optimizations:

```
(ms text)
(global _main)
(function _main 1.000000)
  (cb 1 1.000000 (flow 0 2 1.000000))
    (op 0 define ((mac $return_type i))(()()()))
    (op 1 define ((mac $local i))((i 0)()()))
    (op 2 define ((mac $param i))((i 16)()()))
    (op 3 prologue (())(()()()))
  (cb 2 1.000000 (flow 0 3 1.000000))
    (op 4 mov ((r 3 i))((i 0)()()))
    (op 5 mov ((r 16 i))((i 4)()()))
    (op 6 mov ((r 17 i))((i 8)()()))
    (op 7 mov ((r 18 i))((i 12)()()))
    (op 8 mov ((r 19 i))((i 16)()()))
  (cb 3 240.000000 (flow 1 3 239.000000) (flow 0 4 1.000000))
    (op 9 ld_i ((r 4 i))((r 3 i)(l _y)()))
    (op 10 ld_i ((r 5 i))((r 3 i)(l _z)()))
    (op 11 add ((r 6 i))((r 4 i)(r 5 i)()))
    (op 12 st_i (())((r 3 i)(l _x)(r 6 i)))
    (op 13 ld_i ((r 7 i))((r 16 i)(l _y)()))
    (op 14 ld_i ((r 8 i))((r 16 i)(l _z)()))
```

```
    (op 15 add ((r 9 i))((r 7 i)(r 8 i)()))
    (op 16 st_i (())((r 16 i)(l _x)(r 9 i)))
    (op 17 ld_i ((r 10 i))((r 17 i)(l _y)()))
    (op 18 ld_i ((r 11 i))((r 17 i)(l _z)()))
    (op 19 add ((r 12 i))((r 10 i)(r 11 i)()))
    (op 20 st_i (())((r 17 i)(l _x)(r 12 i)))
    (op 21 ld_i ((r 13 i))((r 18 i)(l _y)()))
    (op 22 ld_i ((r 14 i))((r 18 i)(l _z)()))
    (op 23 add ((r 15 i))((r 13 i)(r 14 i)()))
    (op 24 st_i (())((r 18 i)(l _x)(r 15 i)))
    (op 25 ld_i ((r 0 i))((r 19 i)(l _y)()))
    (op 26 ld_i ((r 1 i))((r 19 i)(l _z)()))
    (op 27 add ((r 2 i))((r 0 i)(r 1 i)()))
    (op 28 st_i (())((r 19 i)(l _x)(r 2 i)))
    (op 29 add_u ((r 3 i))((r 3 i)(i 20)()))
    (op 30 add_u ((r 19 i))((r 19 i)(i 20)()))
    (op 31 add_u ((r 18 i))((r 18 i)(i 20)()))
    (op 32 add_u ((r 17 i))((r 17 i)(i 20)()))
    (op 33 add_u ((r 16 i))((r 16 i)(i 20)()))
    (op 34 bne_fs (())((r 3 i)(i 4800)(cb 3)))
  (cb 4 1.000000 (flow 1 5 1.000000))
    (op 35 mov ((mac $P0 i))((i 0)()()))
    (op 36 jsr (())((l _exit)()()))
  (cb 5 1.000000)
    (op 37 epilogue (())(()()()))
    (op 38 rts (())(()()()))
(end _main)
```

Lcode after Code Scheduling: We schedule the above Lcode function for a multiple-instruction-issue machine that can issue up to 16 operations per cycle. Operations whose (in) attributes are identical belong to the same instruction.

```
(ms text)
(global _main)
(function _main 1.000000)
  (cb 1 1.000000 (flow 0 2 1.000000))
    (op 39 define ((mac $swap i))((i 8)()())(in 0))
    (op 0 define ((mac $return_type i))(()()())(in 0))
    (op 1 define ((mac $local i))((i 0)()())(in 0))
    (op 2 define ((mac $param i))((i 16)()())(in 0))
    (op 3 prologue (())(()()())(in 1))
    (op 43 add ((mac $SP i))((mac $SP i)(i -24))(fn 0)(in 2))
    (op 41 st_i (())((mac $SP i)(i 20)(mac $FP i))(fn 0)(in 2))
    (op 42 st_i (())((mac $SP i)(i 16)(r 169 i))(fn 0)(in 2))
    (op 40 add ((mac $FP i))((mac $SP i)(i 24))(fn 0)(in 2))
  (cb 2 1.000000 (flow 0 3 1.000000))
    (op 4 mov ((r 0 i))((i 0)()())(in 3))
    (op 5 mov ((r 4 i))((i 4)()())(in 3))
    (op 6 mov ((r 3 i))((i 8)()())(in 3))
    (op 7 mov ((r 2 i))((i 12)()())(in 3))
    (op 8 mov ((r 1 i))((i 16)()())(in 3))
  (cb 3 240.000000 (flow 1 3 239.000000) (flow 0 4 1.000000))

    (op 10 ld_i ((r 8 i))((l _z)(r 0 i))())(in 4))
    (op 14 ld_i ((r 9 i))((l _z)(r 4 i))())(in 4))
    (op 18 ld_i ((r 10 i))((l _z)(r 3 i))())(in 4))
    (op 22 ld_i ((r 6 i))((l _z)(r 2 i))())(in 4))
    (op 9 ld_i ((r 15 i))((l _y)(r 0 i))())(in 4))
```

```
 (op 13 ld_i ((r 14 i))((l _y)(r 4 i)())(in 4))
 (op 17 ld_i ((r 13 i))((l _y)(r 3 i)())(in 4))
 (op 21 ld_i ((r 12 i))((l _y)(r 2 i)())(in 4))
 (op 25 ld_i ((r 11 i))((l _y)(r 1 i)())(in 4))
 (op 26 ld_i ((r 5 i))((l _z)(r 1 i)())(in 4))

 (op 11 add ((r 7 i))((r 15 i)(r 8 i)())(in 5))
 (op 15 add ((r 8 i))((r 14 i)(r 9 i)())(in 5))
 (op 19 add ((r 9 i))((r 13 i)(r 10 i)())(in 5))
 (op 23 add ((r 10 i))((r 12 i)(r 6 i)())(in 5))
 (op 27 add ((r 6 i))((r 11 i)(r 5 i)())(in 5))

 (op 12 st_i ()())((l _x)(r 0 i)(r 7 i))(in 6))
 (op 16 st_i ()())((l _x)(r 4 i)(r 8 i))(in 6))
 (op 20 st_i ()())((l _x)(r 3 i)(r 9 i))(in 6))
 (op 24 st_i ()())((l _x)(r 2 i)(r 10 i))(in 6))
 (op 28 st_i ()())((l _x)(r 1 i)(r 6 i))(in 6))
 (op 29 add_u ((r 0 i))((r 0 i)(i 20)())(in 6))
 (op 30 add_u ((r 1 i))((r 1 i)(i 20)())(in 6))
 (op 31 add_u ((r 2 i))((r 2 i)(i 20)())(in 6))
 (op 32 add_u ((r 3 i))((r 3 i)(i 20)())(in 6))
 (op 33 add_u ((r 4 i))((r 4 i)(i 20)())(in 6))
 (op 34 bne_fs ()())((r 0 i)(i 4800)(cb 3))(in 6))

(cb 4 1.000000 (flow 1 5 1.000000))
  (op 35 mov ((mac $P0 i))((i 0)()())(in 7))
  (op 36 jsr ()())((l _exit)()())(in 7))
(cb 5 1.000000)
  (op 44 ld_i ((mac $FP i))((mac $SP i)(i 20)())(fn 0)(in 8))
```

```
    (op 45 ld_i ((r 169 i))((mac $SP i)(i 16)())(fn 0)(in 8))
    (op 46 add ((mac $SP i))((mac $SP i)(i 24)())(fn 0)(in 8))
    (op 37 epilogue (())(()()()())(in 9))
    (op 38 rts (())(()()()())(in 10))
(end _main)
```

Scheduling Result:

```
(CONTROL)
( br_t_t 239.0000000000 _main )
( br_t_n 1.0000000000 _main )
( br_n_t 0.0000000000 _main )
( br_n_n 0.0000000000 _main )
( jump_t_t 0.0000000000 _main )
( jump_n_t 0.0000000000 _main )
( jump_rg_t_t 0.0000000000 _main )
( jump_rg_n_t 0.0000000000 _main )
( jsr_t_t 0.0000000000 _main )
( jsr_n_t 1.0000000000 _main )
( rts_t_t 0.0000000000 _main )
( rts_n_t 1.0000000000 _main )
(TIME)
( true_oper_count 6261.0000000000 _main )
( oper_issue_count 6261.0000000000 _main )
( best_cycle_count 968.0000000000 _main )
( worst_cycle_count 968.0000000000 _main )
```

VITA

Po-hua Chang was born in Taipei, Taiwan, on August 16, 1966. He recieved his B.A. degree in computer science from the University of California, Berkeley, and his M.S. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign. From 1987 to 1990, Po-hua was a research assistant in the Coordinated Science Laboratory. Since 1991, he has been a research associate in the Center for Reliable and High-performance Computing. His research interests include optimizing compilers, code synthesis tools, and computer architectures.