September 1989

UILU-ENG-89-2228
CSG-105

COORDINATED SCIENCE LABORATORY
College of Engineering

AGGRESSIVE CODE IMPROVING TECHNIQUES BASED ON CONTROL FLOW ANALYSIS

Po-Hua Chang

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-89-2228 (CSG-105)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 1800 G Street Washington, DC 20552
8a. NAME OF FUNDING/SPONSORING ORGANIZATION same as 7a 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF MIT 88-09478
8c. ADDRESS (City, State, and ZIP Code) same as 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Agressive Code Improving Techniques Based On Control Flow Analysis

12. PERSONAL AUTHOR(S)
Chang, Po-Hua

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	September 1989	108

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	control flow, program control, IMPACT-I C compiler, trace selection, global code compaction

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The IMPACT-I C compiler obtains, analyzes, and applies program control flow information to guide various code improving techniques. To obtain program control flow information, a system independent profiler has been integrated into the compiler front end. The control flow information obtained is converted into a data structure called a weighted control graph. Function inline expansion, trace based optimizations, software branch prediction technique, and instruction memory layout optimization can be applied to the weighted control graph. Function inline expansion drastically reduces the number of function calls in the program execution. The software branch prediction technique greatly reduces the cost of branch instructions for highly pipelined processors. Trace selection heuristics group basic blocks which tend to execute in a sequence into a trace. Conventional global code compaction techniques can be applied on traces. Finally, we show that instruction placement can lead to better instruction cache performance.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

AGGRESSIVE CODE IMPROVING TECHNIQUES BASED ON CONTROL FLOW ANALYSIS

BY

PO-HUA CHANG

B.A., University of California, Berkeley, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1989

Urbana, Illinois

ABSTRACT

The IMPACT-I C compiler obtains, analyzes, and applies program control flow information to guide various code improving techniques. To obtain program control flow information, a system independent profiler has been integrated into the compiler front end. The control flow information obtained is converted into a data structure called a weighted control graph. Function inline expansion, trace based optimizations, software branch prediction technique, and instruction memory layout optimization can be applied to the weighted control graph. Function inline expansion drastically reduces the number of function calls in the program execution. The software branch prediction technique greatly reduces the cost of branch instructions for highly pipelined processors. Trace selection heuristics group basic blocks which tend to execute in a sequence into a trace. Conventional global code compaction techniques can be applied on traces. Finally, we show that instruction placement can lead to better instruction cache performance.

ACKNOWLEDGEMENTS

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. He was always willing to explore new ideas and to ask for more concrete experimental results to support our ideas. Sadun Anik, Thomas Conte, Nancy Walters and all my colleagues in the Computer Science Group gave me many invaluable suggestions and references to further my study.

I wish to thank my parents for many years of love and support.

TABLE OF CONTENTS

CHAPTER 1.

INTRODUCTION

The IMPACT-I (Illinois Microarchitecture Project using Advanced Compiler Technology) C compiler is a microarchitecture technology file driven optimizing compiler which is currently being developed at the University of Illinois. An automatic profiler has been integrated with the compiler front end to collect program run-time information, which characterizes the control flow behavior of the program to be optimally compiled by the IMPACT-I C compiler. Knowing the program control flow behavior, the compiler is able to perform a series of analyses and trace-based optimization procedures.

A large portion of the first draft of this thesis was written jointly by my thesis advisor and myself for conference presentations [Cha88, Hwu88, Hwu89, Hwu89_2, Hwu89_3]; therefore, plural subjective pronouns are used to indicate his direct contribution to this thesis. In this thesis, we describe how the IMPACT-I C compiler obtains, analyzes, and applies program control flow information. The major contributions of this thesis are (1) the implementation of an efficient portable profiler, (2) the definition of a simple program intermediate form to concisely describe program control flow behavior, (3) the implementation of a function inline expansion facility, (4) the applicability study of trace-based code improving techniques on large integer programs, and (5) the design and evaluation of several code improving techniques which increase processor performance.

1.1. Initial Motive

The demand for high-speed microprocessors continues to increase. Applications include high-performance workstations, application specific processors, and implementations of mini- and main-frame computers. The design

constraints of these microprocessors are different from those of the conventional technologies to that require new design approaches. One promising approach to designing high-speed microprocessors is to use a sophisticated compiler to identify and exploit the parallelism both in the programs and in the microarchitecture. The program and microarchitecture parallelism must be balanced to achieve a cost-effective design.

With advances in VLSI technology, microprocessor designers can provide more microarchitectural parallelism to increase performance. Providing multiple decoding logics, multiple execution units and multiple data distribution buses, one exploits the horizontal parallelism by issuing and executing multiple instructions per machine cycle [Hwu87, Ell84]. Providing pipelined execution units [Ram77, Kog81], one exploits the vertical parallelism by overlapping the issue, decode and execution phases of several microoperations. When the microarchitectural parallelism is large enough to cover all fine-grain program parallelism, the remaining chip space can be used to implement high-speed memories. Adding an instruction buffer [Lee84] allows instruction prefetch and maintains continuous input to the execution unit. Increasing the size of the register file [Goo88] allows more variables to be kept in high-speed registers. Adding an on-chip cache [Hil85, Alp88, Smi82] can reduce the average memory access time. Since the chip space is limited, the processor designer must carefully decide what should be implemented.

On the software side, program parallelisms are often limited. Data and control dependencies between microoperations often force sequential execution, regardless of the processor's ability to execute multiple operations concurrently. Various aggressive code improving techniques, including, but not limited to, loop unrolling, variable renaming, software pipelining, trace scheduling and function in-line expansion can potentially uncover hidden

program parallelism. These code improving techniques, however, must be used with extreme care because they can drastically expand the code space.

We do not yet understand all the design criteria for deriving a perfectly balanced hardware and software design. Independent problems in hardware and compiler implementation are already critically complex. These problems must all be solved individually before we can study the combined effect. Hence, a major goal of our project is to develop a suite of programs which will allow the processor designers to test various alternatives quickly. Among all the tools under development, the IMPACT-I C compiler is a technology file driven optimizing compiler which is aimed to generate good codes for many processor configurations. Coupled with program analysis tools, the IMPACT-I C compiler can accurately estimate the execution time of programs on the target microarchitecture. The performance estimation by the compiler can be a useful feedback to the processor designer.

1.2. Previous Experience

We constructed a prototype code generator [Hwu88] in 1987 which builds dependence graphs [Kuc81] to render the fine-grain program parallelism, and employs a modified list scheduling technique to schedule microoperations into fewer instructions. The scheduling function issues microoperations on the critical paths first, considers many plausible combinations of microoperations which can be combined into an instruction and selects the best instruction-template [Tok81] which will lead to a minimum number of instruction-templates and resource conflicts in the future. Local code compaction is applied to more important basic blocks first and therefore allows operations and undesirable resource constraints to migrate to lesser important basic blocks. In addition, a simple register allocation mechanism is used to migrate

important memory variables into registers. Various compiler techniques, including data flow analysis, have been tested. Furthermore, we have designed a microarchitecture description language capable of modeling the instruction set architecture, the pipeline timing, and other hardware features. The code generator understands the microarchitecture description language and attempts to generate optimized codes for the processor configuration specified in the technology file. The prototype code generator was powerful enough for us to measure the performance of many small benchmark programs on various microprocessor configurations.

In constructing the prototype code generator, we have gained invaluable experience in advanced microcode compilation techniques and have observed some deficiencies in our prototype code generator.

We have learned that parallel microarchitecture requires substantially more registers to hold the intermediate values when evaluating several data independent expressions concurrently. This suggests the inclusion of a large register file in the microarchitecture and also a good register allocation algorithm in the compiler. The register allocation algorithm must at all times conserve register usage.

We have also found problems with phase-coupled register allocation and code scheduling techniques, and implemented an *ad hoc* heuristic solution. Allocating registers before code scheduling may introduce artificial data dependencies which will inhibit code motion. Another problem with allocating registers before code scheduling is that data-flow analysis [Aho86] cannot be accurately computed without a stable instruction space. For example, if a variable is used by two instructions before it is overwritten, we do not know which of the two instructions is the last use of the variable before code scheduling. An alternative approach is to schedule microoperations before

register allocation. This tends to move microoperations to the earliest positions where they can be executed and sometimes use more registers than necessary. When register spill codes have to be introduced or delay time has to be inserted to free some registers, the conventional register allocation schemes can no longer guarantee software interlocking. Software interlocking [Hen83] is a compiler technique to remove hazards [Kog81] due to data and control dependencies among instructions [Kuc81]. Therefore, we implement another pass to reschedule all microoperations after preliminary scheduling and register allocation to support software interlocking.

The prototype code generator supported only local code compaction, which restricts code motion to within a basic block. This was not a problem when we measured the Livermore [McM84] and Linpack [Don79] benchmark programs because those benchmarks contain large basic block bodies after loop unrolling. However, most integer programs, such as editors, text formatting programs and compilers, do not appear to have large basic blocks. To exploit fine-grain program parallelism, the compiler must apply global code motion. A popular control mechanism for global code motion is the *trace scheduling* algorithm [Fis81].

Trace scheduling guides global code motion by favoring most frequently executed program paths. Similarly, in many other compiler techniques, such as register allocation by weighted graph coloring [Cha82, Cho84] and loop optimizations, performance can be achieved by speeding up the access/execution time of the most frequently used variables/instructions. While optimizing the most important parts of a program can potentially increase the execution time of lesser critical parts due to code motion, a positive gain is expected in the overall performance.

Then the natural question to ask is how the compiler knows what variables and which program paths are accessed and executed most frequently. To estimate the importance of variables and instructions at compile time, loop structures are first identified. Objects in loop structures are assumed to be more important.

We feel very uneasy about the large number of *if* statements in large C integer programs. The static weight estimation technique cannot confidently predict the branch probability of an *if* statement. Estimating the *if* condition to be true half of the time is overly conservative. If one profiles many C application programs, it is obvious that most *if* conditions tend to result in one direction. We have thus decided to integrate an automatic profiler with our compiler. With accurate program control flow information collected during program run-time, the compiler can more confidently select critical sections of a program to be compiled optimally.

1.3. Program Control Flow

What is program control flow? In a sequential program, there is only one program counter which walks through the instruction space until it reachs a termination point. In a simple non-pipelined machine, there is only one active instruction in the machine. Before the completion of the active instruction, the processor must decide which instruction to execute next. Selection of the next active instruction is based on the current program state which consists of variable values, machine state, and the current program position. For arithmetic and logic instructions, the instruction in the next immediate memory location (PC+1) is the natural next instruction. For conditional branch instructions, the processor selects, among a set of potential next instructions, one instruction that is specified by the branch condition. In pipelined and SIMD (Single

Instruction and Multiple Data Stream) processors, the memory address of the next instruction is computed in similar fashion. In sum, the way in which the program counter moves from one instruction to another in some controlled fashion is called program control flow.

How is program control flow information useful? Knowing which instructions will immediately be executed, the processor can anticipate future events. For example, the processor can prefetch instructions and variables which will soon be used, without disturbing the execution of the active instruction. The compiler can properly encode the future knowledge in the code it generates and propagate that information to the microarchitecture. Later chapters of this thesis explore many potential uses of the program flow information.

1.4. Organization of Thesis

The body of this thesis consists of six main chapters. In Chapter 2, we describe some major concepts and results developed from previous research. In Chapter 3, implementation issues of an efficient program profiling tool and a suitable data structure to represent program control behavior are presented. In Chapter 4, we describe the function inline expansion capability. In Chapter 5, we record the control flow behavior of some application programs and design hardware and software techniques to increase performance. In Chapter 6, we show that trace-based code-improving techniques are suitable for compiling large application programs. In Chapter 7, memory layout techniques based on program control flow information are introduced. Finally in Chapter 8, we offer some concluding remarks.

CHAPTER 2.

BACKGROUND

Previous researches in compiler design and computer architecture have resulted in many performance improving techniques. In this chapter, we will describe a few techniques which can be furthur improved given that the compiler knows the program control flow information. In later chapters, algorithms and experimental results are provided.

2.1. Code Optimization

An important concept appearing in several code improving techniques is to reduce the execution time of the most intensively executed parts of a program. The first step of these code improving techniques is to identify the most critical regions of a program. Knowing the critical regions, program transformations are applied to reduce the execution time of these critical regions.

Loop-invariant-code-removal [Aho86] creates a loop header and moves instructions whose input operands are invariant in the loop body to the loop header. The first step of this technique is to identify loops. Loops are considered more important than straight-line code because loop bodies are usually executed many times. Moving code out of a loop results in smaller loop body which requires less execution time. We will use a simple example to illustrate such transformation.

```
while (loop condition) {
    a = b + 3;
    c = c - (a + d);
}
```

After applying loop-invariant-code-removal, the above loop has an additional loop header.

```
a = b + 3;
temporary = a + d;
while (loop condition) {
        c = c - temporary;
}
```

Without run-time information, the compiler looks for sections of a flow graph which resemble loops [Aho86]. Loop analysis also tells the degree of loop nesting. The exact algorithm can be found in the reference and needs not be reiterated here.

The outer loop is assumed to be executed a fixed number of times, usually ten. Inner loops are assumed to be the most important, and are assumed to be executed (10**degrees_of_loop_nesting) times. Then the importance of instructions and variables appearing in loop bodies can be estimated to be (loop_weight*number_of_occurrence).

Many code improving techniques have been designed to optimize inner loops. Loop unrolling unrolls several iterations of an inner loop and applies variable renaming, code motion and code compaction techniques to increase the degree of fine-grain program parallelism and to reduce the number of instructions using microcode compaction techniques [Gra87, Tok81, Fis81]. Software pipelining combines operations from several iterations of an inner loop and packs them into as few instructions as possible. In vector machines, inner loops are usually converted to vector instructions.

Other code improving techniques also use the instruction and variable *weight* information. Register allocation techniques reduce memory accesses by migrating memory variables into registers. Since the number of high-speed

registers are limited, a priority function based on the access frequency of variables is used to decide what variables should be maintained in registers. When spill codes have to be introduced to free registers, the variable usage information is again used to minimize the spill cost. The priority functions are complex and are carefully justified in Chow's paper [Cho84].

More aggressive global code motion, such as global microcode compaction techniques, allow micro-operations to move across basic block boundaries. The goal is to reduce the execution time of the most critical program paths. For example, if we know that basic block bbA has an empty micro-operation field and basic block bbB is very likely to be reached from basic block bbA, we can possibly move a micro-operation from basic block bbB to fill the empty slot in basic block bbA. Potentially, a cycle can be saved when basic block bbB is immediately executed after executing basic block bbA.

The accuracy of the instruction and variable weight information directly affect the performance of the above code improving techniques. In addition, global code motion techniques also require the knowledge of program control transfer between basic blocks. Clearly, purely static (compile-time) weight estimation, such as loop analysis, no longer suffices. There are three potential dangers.

First, the number of loop iterations is not known at compile time. While there are several loops of the same nesting level, one loop may iterate many more times than all other loops. Since high-speed resources such as registers are severly limited, it is necessary to allocate them to only the most important program regions.

Second, *if* statements can not be predicted with high accuracy at compile time, and can cause a loop which is rarely reached to be considered important. This undesirable condition appears in the following example.

```
if (not likely to be true) {
    while (many iterations) {
        .....
    } /* end while */
} /* end if */
```

Third, when loop nesting information is not propagated across function call boundaries, the importance of a function which is called from a loop in some other function may be underestimated. To implement inter-procedural register allocation and code motion, it is important to know the relative importance of all functions.

```
m() {
    /* straight-line code */
    .....
}
n() {
    while (many iterations) {
        m();
        .....
    }
}
```

Run-time information is much more accurate than static prediction. It is already very common that optimizing compilers support several styles of program profiling [Bell79, Gra83]. The profile information allows the programmers to improve the algorithm and coding techniques employed in the most intensively executed program regions. In order to automate all code improving techniques, we have integrated a profiler with our IMPACT-I C compiler to supply accurate instruction and variable *weight* information to various code

improving techniques. The run-time information is nicely encoded into a simple data structure. We will describe the implementation of the profiler and the data structure in the next chapter.

2.2. Branch Prediction

C programs are decision intensive, as indicated by the large number of *if* and *switch* statements and loop constructs. The conditional branches are implied by two-way decisions, and switches are implied by multi-way decision in the C statements. Each switch statement can be implemented by either a hashing jump or a sequence of conditional branches. Each iteration of a loop requires a two-way decision which determines whether or not the loop should be reiterated.

These conditional branches have damaging effects on the performance of pipelined processors. One reason is because the outcome of the branch may not be available immediately. Another reason is that the branch target address requires a full word addition which may require a machine cycle after the branch offset is available.

Existing approaches to the branch problem all require some amount of special hardware. For example, loop buffers have been used in CDC-7600 [Cdc75] and CRAY-1 [Cra78], and delayed branch is popular among RISC machines [Pat81]. To prefetch and decode instructions beyond a conditional branch instruction, multiple instruction stream and branch prediction strategies have been designed [IBM78, Smi81].

Lee and Smith described several mechanisms to perform branch prediction [Lee84]. In their paper, prediction based on branch history has achieved a mean success rate of higher than 90 percent.

McFarling and Hennessy examined several static (compile-time) and dynamic (hardware-assisted) branch prediction and control mechanisms [McF86]. Software branch prediction based on the program execution profile is reported to predict slightly better than a 128-entry hardware predictor. To overcome the problem of finding safe instructions to fill the branch delay slots, controlled squashing of the instructions in the delay slots by special hardware is proposed. Squashing capability permits the first couple of instructions in the predicted path to be placed in the delay slots. When branches are incorrectly predicted, the instructions in the delay slots are squashed before they affect the machine state.

The IMPACT-I C compiler also performs static branch prediction based on program execution profiling. We are able to isolate the effect of control flow transfer due to conditional branches, jump instructions, function call and return, and multi-way branch instructions. We also employ a more aggressive mechanism to conditionally squash instructions in the delay slots. Branch instructions are allowed to be placed in delay slots and conditionally squashed.

2.3. Trace Based Optimization

The idea of improving the most critical regions of a program again appears in trace scheduling. The algorithm repeatedly selects the most important program path, which is called a *trace*, compacts the trace, and inserts patch codes to remove the ill effect of global code motion.

Trace scheduling was first proposed by Fisher [Fis81] as a systematic approach to global microcode compaction. Since improvements and implementations of code improving techniques based on trace selection techniques have been reported [Lin83, Su84, Eli84]. These techniques are the most useful for generating efficient codes for application programs which are too large and too

complicated for human micro-programmers. However, most of the experimental results reported on using trace selection to assist global code motion have been based on small benchmarks with simple control structures.

It is thus necessary to conduct an applicability study of trace-based optimization for large integer programs, such as editors and compilers. We will report in a later chapter the effectiveness of several trace selection functions based on profile information. Our result shows that trace-based code improving techniques can substantially improve program performance.

Software pipelining and loop unrolling techniques which have been reported apply only to simple inner loops. However, we will show that there are usually several conditional branch instructions in inner loops in large integer programs

2.4. Memory Layout

The performance of a high-speed processor depends greatly on how fast the memory system can supply instructions and data.

In array and VLIW (Very Long Instruction Word) 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 [Law75] published a data alignment technique which allows parallel and conflict-free access to various slices of data for an array processor. Ellis [Eli84] discussed several memory-bank disambiguation methods, which distribute memory accesses evenly to each of the memory banks.

Although bank disambiguation techniques may not be required for high-speed scalar processors, the concept of intelligently organizing and placing information into memory to achieve fast information retrieval must be treasured and applied.

Given the execution profile information, the IMPACT-I C compiler can predict which basic blocks and variables will be executed and accessed in the future. It is then possible to group and place instructions and variables carefully in the memory space in such a way that more sequential and spatial localities are preserved. Better sequential locality suggests the use of a larger cache block size and a more aggressive prefetch algorithm. Instructions and variables can also be placed to reduce the number of cache conflicts.

The memory layout algorithms and their implications are presented in a later chapter.

CHAPTER 3.

PROFILE TOOL AND WEIGHTED CONTROL GRAPH

A system independent execution profiler [Cha88] has been implemented and integrated into the IMPACT-I C compiler frontend which is also implemented by the author. To profile a C program, the IMPACT-I 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 concurrently.

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 tool will be ported to various systems, the IMPACT-I compiler and profiler interface must also be completely system independent.

The IMPACT-I profiler is system independent for four reasons. First, the IMPACT-I profiler itself can execute on different systems. Second, the program with profiling probes can execute on different systems (even in parallel). Third, 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. Fourth, the profile information accumulated on an existing system can be used to guide the architecture design and code optimization for a nonexisting system.

In this chapter, we describe the implementation of the IMPACT-I profiler and its profile process.

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 *weigthted control graph* defined below is such a structure via which the profile information can be presented to the compiler.

A *control graph* is a directed graph where every node is a basic block and every arc is a branch path between two basic blocks. There is an arc emanating from node A to node B if and only if the final branch instruction in basic block A can potentially cause a control transfer to basic block B. The *node weight* is the average execution count of the corresponding basic block in a typical run of the program. The *arc weight* is the average number of times the corresponding branch path is taken in a typical run of the program. A *weighted control graph* is a control graph in which all 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 is an *incoming arc* of node B. From the opposite perspective, 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 furthur 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%).

3.2. Construction of a Weighted Control Graph

There are 8 major steps to generate profile information.

(1) The C source program is converted into a control graph.

(2) Constant folding and dead code removal are applied to the control graph to eliminate unreachable blocks. Then, jump optimizations are applied to merge basic blocks which are connected by unconditional branch instructions. The results of these optimizations are a modified control graph which has fewer and larger basic blocks.

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

(4) The control graph is converted 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 data to accumulate profile information in a database.

(7) The compiler constructs an identical control graph by repeating steps (1)-(3). 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 function is applied to verify that all weights have been gathered and assigned consistently.

3.2.1. Constant folding and dead code removal

Constant folding is a compiler technique which converts an expression whose source operands can be computed during compile time into a scalar value. For example, (x=12-4) can be reduced to (x=8).

If the result of a constant expression affects a control operator, the compiler may be able to identify unreachable code.

```
#define DEBUG 1
if (DEBUG) {
    /* reached only in DEBUG mode */
    fprintf(stderr, "> state = %d0, state);
}
```

In debugging and testing new programs, programmers usually insert print statements to monitor the program state. In this case, the compiler knows that the print statement will always be executed and can remove the redundant *if* statement. On the other hand, when it is no longer necessary to check the program state, DEBUG is set to zero and the print statement will never be reached. In that case, both the *if* statement and the print statement can be removed.

Similarly, loops whose loop control can be determined statically may be removed if the loop is never entered.

In our intermediate representation, each function has an ENTRY basic block. The following algorithm has been implemented to remove dead code one function at a time.

```
algorithm remove_unreachable_block(F)
  /*** mark all nodes unreachable ***/
  for (all nodes Ni of F) do
    Ni.visited = false;
  /*** mark all nodes which can be reached ***/
  push(ENTRY node of F);
  current = pop();
  while (current <> 0) do begin
```

```
        current.visited = true;
        for (all outgoing arcs Aj of current) do begin
            D = destination of Aj;
            if (D.visited = false) then
                push(D);
        end for
        current = pop();
    end while
    /*** remove all unreachable nodes ***/
    for (all nodes Nk of F) do
        if (Nk.visited = false) then
            remove Nk;
end algorithm
```

3.2.2. Loop generation

To reduce the number of branch instructions in loops, all loop structures are converted to *do while* loops in C.

```
for (A; B; C)
    D;
```

can be translated directly as

```
    A;
L0 :
    if (! B) goto L2;
L1 :
    D;
    C;
    goto L0;
```

```
        L2 :
```

or alternatively as a *do while* loop.

```
        A;
        if (B) {
            do {
                D;
                C;
            } while (B);
        }
```

becomes,

```
            A;
            if (! B) goto L1;
        L0 :
            D;
            C;
            if (B) goto L0;
        L1 :
```

In the above example, an unconditional jump instruction is eliminated. Similarly, *while* loops can be transformed to the *do while* form by duplicating the loop test.

3.2.3. Jump optimization

We will create some superficial code segments to demonstrate the functionalities of several jump optimizations which we have implemented in the IMPACT-I C compiler. The first type of jump optimization is to create by merging basic blocks which are connected by an unconditional jump instruction.

```
L0 :
    A;
    branch to L1;  /* no other branch result in L1 */
L1 :
    B;
    if (C) goto L2;
```

In the code shown above, the basic blocks L0 and L1 are connected by an unconditional jump, and this jump instruction is the only way to enter L1. In this case, the two basic blocks can be merged.

```
L0 :
    A;
    B;
    if (C) goto L2;
```

The second type of jump optimization is to remove basic blocks which contain only a single jump instruction.

```
L0 :
    A;
    if (B) goto L2;
L1 :
    goto L3;
L2 :
    goto L4;
L3 :
```

can be easily converted into

```
L0 :
    A;
    if (B) goto L4;
```

```
L3 :
```

Another jump optimization requires more code duplication.

```
L0 :
        A;
        goto L2;
L1 :
        B;
L2 :
        C;
        if (D) goto L4;
L3 :
```

can be transformed to

```
L0 :
        A;
        C;
        if (D) goto L4;
        goto L3;
L1 :
        B;
        C;
        if (D) goto L4;
L3 :
```

In the above example, if the chance of reaching L0 is much higher than reaching L1, a branch instruction is reduced. Also, classical local microcode compaction techniques work much better with larger basic blocks. Due to the

code expansion problem, this type of code optimization can be activated only by a special compiler option which specifies an upper limit on the size of a basic block that can be duplicated and absorbed.

We have implemented all three types of jump optimizations illustrated above and a couple others which have been found less effective because of their rare occurrences.

3.2.4. Probe insertion

After jump optimization, probes are placed at various places of the control graph. First, the compiler assigns each basic block in the program a unique identifier. For each basic block, the compiler inserts a probe to detect basic block execution count and the transition count. In order 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 data structure, which we call *tag_stack*, is provided to store and recover the *last_tag* value across function invocations. 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 to replace setjmp() with a probe which marks the top of the *tag_stack* and replace 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.

3.2.5. High-level language as an intermediate form

There are two major requirements to our profile tools. First, the same profile information can be used with a different physical code generation to evaluate architecture designs and implementations. Second, the profile information is available to the code optimizers to improve the quality of the code. To fulfill the two requirements, the part of the compiler which inserts profiling probes must have access to the program structures before the optimization and the physical code generation are performed.

The internal data structure and the symbol table of the IMPACT-I C compiler capture completely all information in the source programs. After semantic analysis, the compiler can generate two different intermediate forms. To generate profile codes, the C programming language is used as the intermediate

language. Alternatively, IL (intermediate format) code can be generated as the input to the code optimizer. Variable renaming is applied before intermediate code generation, and all variables have unique names in the intermediate code format.

The advantage of using C programming language as an alternative intermediate language is that the profile code can be compiled by different C compilers on various systems. Another advantage which we have not realized in the early design stage of the compiler is that we can test the internal representation and therefore the IMPACT-I compiler front end.

Each basic block is assigned a unique label. Transitions between basic blocks are implemented using *goto* statements of C.

```
while (a>10) {
        b = c-a;
        a--;
}
```

is translated to

```
        if (a>10) goto L0;
        goto L1;
L0 :
        b = c-a;
        a--;
        if (a>10) goto L0;
        goto L1;
L1 :
```

Since only labels and goto statements are used, one can choose other symbolic languages, such as BASIC, as the intermediate form.

3.2.6. Input data

The profile code can be compiled and installed in a public system. In our case, we have an university research environment where most jobs are CPU intensive CAD programs, text editing and formatting programs, and program compilations. Inputs from various users in the selected computer environment can be profiled and averaged. Inputs come from various people and represent the general system usage. Unlike benchmarking, the inputs used are from real applications, and the entire program execution time is monitored.

3.2.7. Profile data representation

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

```
struct _link {
        int destination;
        double weight;
        struct _link *next;
};
struct _node {
        double weight;
        struct _link *outgoing_arcs;
} NodeTable[MAX_NUMBER_OF_NODES];
```

The *destination* field of the _link structure specifies the unique node identification number of the destination block. The *weight* field of the _link structure is the number of times the arc has been taken. The *next* field of the _link 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 _link 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 _link structures are allocated statically by declaring two large arrays which are appended to the user program that is being monitored. For all programs which we have profiled so far, including programs which consist of more than ten thousand lines of C code, five thousand nodes and links are sufficient.

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

3.2.8. Profile database maintainance

The number of profile runs is also stored in the database file. Each run of the program generates a new set of node and arc weights. The profiler adjusts the profile data according to 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.

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.

3.2.9. Reconstruction of flow graph

The IMPACT-I profiler and the IMPACT-I C compiler share the same front end. 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 IL (intermediate language) format. The control graph can be furthur optimized, as long as the node and arc weights are also modified consistently.

3.2.10. Node and arc weight assignment

The names of 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 link. The second argument specifies the destination of a control link.

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

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

```
algorithm weight_assignment(P)
  for (all nodes Ni of P) do begin
    Ni.weight = NodeWeight(Ni.id);
    for (all outgoing arcs Aj of Ni) do begin
      D = destination of Aj;
      Aj.weight = ArcWeight(Ni.id, D.id);
    end for
  end for
end algorithm
```

3.2.11. 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 graphs of large integer programs usually consist 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 non-unique basic block id assignment or inconsistent basic block id assignment due to source code change.

3.3. Separate Compilation

Seperate compilation can not be done when it is necessary to assign each 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 according to a particular order 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 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 and how the probes are inserted and how the profile information is mapped to the source code. The compiler absorbs all the complexity and fully automates the compilation process.

3.4. Portability Issue

The IMPACT-I profiler is system independent. The IMPACT-I compiler front end and the profiler are written in C and can be ported to any system which has a working C compiler. The program with profile probes is also in C and can execute on different systems. And the profile data does not contain any system dependent parameters and can be applied on any system. The first advantage of our portable profiler is that the profiling process can be distributed over a large number of systems. The second obvious advantage is that the accumulated profile information can be used by the IMPACT-I system on all machines. Hence, the profiling process needs to be executed only once.

CHAPTER 4.

FUNCTION INLINE EXPANSION

Structured programming techniques encourage the use of functions. As a result, realistic C programs often execute a large number of function calls. Unfortunately, function calls cause performance problems by hindering compiler optimizations across function boundaries. Examples of compiler optimizations hindered by function calls include register allocation, code scheduling, common subexpression elimination, and constant propagation. The decreased effectiveness of these optimization techniques increases memory accesses, decreases pipeline efficiency, and increases redundant computation.

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 [Pat81]. Another example is the CRISP processor that uses stack buffers to capture the memory accesses to local variables so that the register allocation crossing function calls can be simulated in hardware [Dit87]. The problems with these hardware approaches are that they tend to consume a significant amount of hardware, stretch the processor cycle time, and provide little assistance for enlarging the scope of compiler code optimization.

Inline expansion has been employed to reduce the function calls in several compilers. Inline expansion replaces the function call statements with the function bodies to eliminate function calls. This technique trades off the static code size for reduced function call frequency [Hus82, Cho84, All88, Sta88, Str87, Geh84]. In the GNU C compiler, the programmers can use the keyword **inline** as a hint to the compiler for inline expanding function calls. In the MIPS C compiler, the compiler examines the code structure (e.g., loops) to

choose the function calls for inline expansion. In these compilers, little run-time information has been used to assist inline expansion.

4.1. Our Implementation

There are four major implementation issues regarding inline expansion: increase in the static code size, increase in the control stack size, unavailable function bodies, and conflicts of identifiers. If not carefully addressed, the first two issues could reduce the effectiveness of the memory hierarchy to such a degree that system performance decreases due to inline expansion. The IMPACT-I C Compiler addresses these two issues by using a heuristic which takes into account the estimated execution count, the estimated static code size, the estimated stack frame size, and the potential of recursion for each function call.

The estimated execution count for each function call is derived by profiling the program with a set of representative inputs. Instead of requiring the programmer to supply hints to the compiler, this approach requires the programmer to supply representative inputs to the program. Therefore, this approach is more suitable for speeding up the execution of 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 inline expander.

Inline expansion is performed before other code optimization techniques so that these other techniques can benefit from inline expansion. Since code optimizations change code size, the function inline expander can only base its decision on estimated code sizes. The inline expander uses the number of intermediate instructions in a function as the estimated code size of that function. Note that inline expansion itself also changes code sizes of functions.

Therefore, the code size of each function body is updated as function calls are expanded.

Since inline expansion is done before register allocation and code scheduling, the stack frame size for each function is not known to the function expander. The inline expander uses the declarations of parameters and local variables to estimate the stack frame size for each function call. The major concern here is to avoid introducing any function body with large stack frames into recursive call paths. For example, a recursive function m(x) is defined as follows.

```
m(x) { return (x?m(x-1)+m(x-2)+n(x):1); }
```

Assume n(x) has a large local data structure, as follows.

```
n(x) { int y[100000]; ..... }
```

If m(x) tends to be called with a large x value, expanding n(x) will dramatically increase the call stack size. Therefore, the IMPACT-I C Compiler examines the call graph to detect all recursive call paths so that situations as illustrated above can be avoided. Since inline expansion itself changes the parameter and local variable declarations for functions, the size of the stack frame for each function is updated as function calls are expanded.

The IMPACT-I C Compiler performs interprocedural analysis to identify all the function calls with function bodies that are unavailable for inline expansion. These functions are usually either written in assembly language or kept unavailable due to proprietary considerations. Even for standard library functions, properly locating the function bodies is a major implementation task.

A final note on the implementation is that each object (function, variable, etc.) in the program is assigned a unique identifier. Each identifier used by the programmer is renamed to ensure that the expanded function body does not

contain identifiers conflicting with the existing ones. This simplifies the management of the symbol table during inline expansion. See Appendix-A for details of the IMPACT-I inline expansion algorithm.

4.2. Experiment

Table 4.1 summarizes several important characteristics of our benchmarks. The *C lines* column shows the static code size of the C benchmark programs measured in the number of program lines. The *runs* column gives the number of different inputs used in the experiment. The *size* column shows the number of bytes required to store the profile information. The average storage requirement is about six (6) bytes per one line of C code. The *description* column describes the benchmark programs.

Note that we use the dynamic counts of intermediate instructions rather than those of machine instructions to keep the data general. The experiment involves measurements based on more than three billion intermediate instructions worth of program execution. The benchmark programs exhibit very different code sizes, control structures, and applications. The inputs to each program have been chosen to cover a wide spectrum of possible input patterns.

Table 4.2 presents the result of inline expansion. The *code inc* column gives the percentage of increase in static code size due to inline expansion. The *call dec* column gives the percentage of dynamic function calls eliminated by the inline expansion. The *ILs per call* column gives the average number of dynamic intermediate instructions executed between dynamic function calls after inline expansion. The *CTs per call* column gives the average number of dynamic control transfers executed between dynamic function calls after the inline expansion.

Table 4.1 : Benchmark Set

name	line	run	size(B)	description
cccp	4660	20	20411	GNU C preprocessor
cmp	371	16	1098	Compare text files
compress	1941	20	5086	File compression
eqn	4167	20	15860	Typeset mathematics
espresso	11545	20	49805	Logic minimization
grep	1302	20	2955	String search
lex	3251	4	26722	Lexical analyzer generator
make	7043	20	24258	Maintain files
tar	3186	14	10065	Create tape archives
tee	1063	18	1846	Replicate output
wc	345	20	1056	Word count
yacc	3333	8	29677	Parsing program generator

Note that the inline expansion mechanism eliminates large percentages of dynamic function calls for function call intensive programs. For programs with less frequent function calls to begin with, 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.

All in all, the inline expansion is very effective in that function calls only account for about 1% of the control transfers after inline expansion (see the *CTs per call* column). In terms of frequency, on the average there are hundreds of intermediate instructions executed between dynamic function calls.

Table 4.2 : Inline Expansion Result

name	code inc	call dec	DIs per call	CTs per call
cccp	17%	55%	506	95
cmp	3%	49%	265	58
compress	4%	91%	2324	368
eqn	22%	81%	197	58
espresso	22%	68%	576	90
grep	31%	99%	11214	4071
lex	23%	77%	7807	2880
make	22%	56%	362	77
tar	16%	43%	983	127
tee	0%	0%	15	6
wc	0%	0%	18310	5146
yacc	24%	80%	1205	303
AVG	15.3%	58.3%	3647	1107
SD	10.7%	32.0%	5808	1832

Therefore, function calls become unimportant in the hardware design considerations. Also, large scopes for compiler optimizations can be expected for the critical parts of the programs. The price, on the average, is a 20% increase in static code size.

CHAPTER 5.

BRANCH HANDLING

Conditional branch instructions pose serious problems to the processor pipeline design. Without special hardware capability, the instruction issue logic can not deliver instructions to the first stage of the processor pipeline before the direction of conditional branch instructions can be confirmed. Therefore, conditional branch instructions can potentially introduce no_ops to the pipeline.

Many special hardware mechanisms have been proposed to overcome the conditional branch problem. Some machines permit instruction prefetch from both the fall-through path and the branch target path while evaluating the branch condition code. Multiway instruction prefetch mechanism allows instructions to enter the pipeline as soon as the branch condition is available. However, multiway instruction prefetch requires higher memory bandwidth and has an exponential cost when branch instructions are extremely frequent. The exponential cost is due to prefetching new branch instructions which thereby introduce more potential branch paths.

Each conditional branch instruction introduces two branch paths : the fall-through path and the taken path. The address of the next instruction in the fall-through path is simply a natural increment of the address of the branch instruction and can be computed while fetching the branch instruction. On the other hand, the address of the first instruction of the taken path is specified by a branch offset. The branch offset is often specified in the constant literal field of the branch instruction and is available only when the branch instruction is fetched. When the branch offset is available, a full integer addition (PC'=PC+offset) is required to generate the target address. In a pipelined processor design where a short cycle time is desirable, the target address

computaion may require an additional cycle. Also, one or more machine cycles is required to access the instruction memory. To remove the target address and the memory overhead, delayed branch slots and instruction buffers are introduced.

Delayed branch has been used in several RISC implementations. [Pat81, Pat82] Delayed branch requires that the first few instructions, which are called the delay slots immediately following the branch instructions be executed regardless of the outcome of the branch instruction. In a sense, the branch instruction is moved up. Therefore, the branch target address evaluation and subsequent memory access can overlap with the execution of the instructions in the delay slots. The major problem with this scheme is that finding instructions to fill the delay slots is difficult due to severe data flow dependence.

To remove the memory overhead, instruction buffers and branch target buffers have been proposed and implemented [Rus78] When the first couple of instructions of the branch taken path can be found in the instruction buffer, the overhead of instruction memory access can be removed. The instruction buffer is extremely useful to capture small inner loops.

Due to the complexity of the decoding unit, unlike multiway prefetch, multiway decoding is not feasible. At best, the processor can decide to decode one of several potential branch paths. Since instruction decoding does not change machine state (neglecting complex addressing modes), predictive decoding can be easily done and undone, while evaluating the branch condition code. If the branch eventually results in the branch path, which is anticipatively decoded, the processor pipeline remains full. If the branch results in some other path, the instructions which have been decoded should be nullified. Nullification of an instruction can be done by simply toggling a single *valid* bit

and requiring only a small amount of hardware. To increase the probability of selecting the correct branch path, several branch prediction schemes have been proposed. [Smi81] Branch prediction can be done by the compiler at compile time or by special hardware at run time, or by a combination of the two. Hardware branch prediction is expansive and can become the critical path of the processor design. Software branch prediction lacks run-time information and can predict less accurately. In the following subsections, we will show that the profile information can be used to enhance software branch prediction. The result is comparable to hardware branch prediction schemes.

5.1. Special Hardware and Compiler Support

The IMPACT microarchitecture employs a combination of hardware and software techniques to alleviate the conditional branch problem. On the software side, the profile information enhances the software branch prediction strategy. On the hardware side, forward semantics and instruction squashing are supported.

5.1.1. Branch prediction

A branch is said *taken* if it brings the control flow away from the fall-through path. It has been reported by several researchers that more than 65% of all branches are taken. Thus the simplest software branch prediction method is to predict that all branches are taken. The reason for high taken branch frequency is that loops usually iterate several times and most *if* statements are evaluated to be false. For each iteration of a loop, a branch is taken to bring the control flow back to the loop header. When an *if* statement is evaluated to be false, a branch is taken to lead the control flow to the *else* part of the *if* statement.

It should be noticed that this simple static branch prediction method is compiler dependent and has to be re-evaluated when there is a change in the code generation strategy. If the compiler generates the *else* parts immediately after the conditional branch instructions of *if* statements, the taken branch frequency will decrease.

The IMPACT-I C compiler uses the profile information to lay out the instruction space to reduce the frequency of taken branch instructions. We favor the fall-through paths because it is desirable to execute as many consecutive instructions as possible. The result is a better cache sequential locality, which permits larger cache block size.

The average branching probability of a particular branch direction is arc_weight(E)/node_weight(N), where E is an outgoing arc of N. Among all outgoing arcs, the one most likely to be taken is predicted to be taken. If the predicted taken arc leads to the fall-through path, the branch is predicted to be not taken. On the contrary, if the predicted taken arc changes the control flow, the branch is predicted to be taken.

The prediction information must be encoded in the conditional branch instructions. One bit per each conditional branch instruction is the least cost for implementing software branch prediction. Addtional instruction decoding and prefetching logic must also be provided to extract and to apply the encoded branch information.

5.1.2. Forward semantic

Since control flow change can not be done instantaneously due to the address computation and the instruction memory access overhead, one or more cycles after each taken branch can be wasted. A simple solution is to execute a

specified number of instructions immediately following the branch instruction as if those instructions should be executed before the branch instruction. This strategy is called *delayed branch* and has been widely used in pipelined processor designs. The biggest problem with the delayed branch is that it is usually not possible to find instructions to fill all delay slots. It has been reported that there is only a 30% chance that the second branch delay slot can be filled.

An alternative method is to implement *forward semantic* with *instruction squashing*. Unlike delayed branch strategy which moves instructions from above the branch instructions to fill the delay slots, forward semantic moves the first few instructions from the most likely to be taken path to fill the delay slots. The advantage is that the code motion is no longer constrained by data flow dependence and all delay slots can be filled. When the branch direction agrees with the predicted direction, all processor cycles are fully utilized. However, when the branch direction disagrees with the predicted direction, the effect of the instructions which have been moved to the delay slots must be undone before the control flow can backtrack to the correct path. The cost of forward semantic strategy is therefore a function of the branch predictability. More accurate branch prediction strategy produces better forward semantic actions.

5.1.3. Instruction squash

To undo the effect of the instructions in the delay slots upon a branch prediction miss, special hardware is required to nullify the instructions and to recover the previous contents of memory components which have been affected by the instructions in the delay slots. Since the first few stages of processor pipelines do not affect the processor state, *instruction squash* can be easily implemented by masking the valid bits of the instructions in the first N stages

of the processor pipelines when there are N delay slots. The computation results and exception signals corresponding to the squashed instructions are ignored.

5.1.4. Delayed branch versus forward semantics

The major difference between the delayed branch model and the forward semantics model is the use of delay slots. Delayed branch moves instructions originally before the branch instruction to fill the delay slots, when data dependence relations are not violated. Forward semantics first select the path which is more likely to be reached, and then move the first few instructions of that path to fill the delay slots.

For the delayed branch model, the average cost of a branch instruction is $(1 + N - M)$, where N is the number of delay slots for a branch instruction and M is the average number of delay slots which can be successfully filled. Because of data dependence constraint, it is difficult to fill more than one delay slot. Therefore, the delayed branch is not suitable for very deeply pipelined design, which requires many cycles to evaluate the branch condition code and/or the branch target address.

For the forward semantics model, the average cost of a branch instruction is $(1 + (1-P)*N)$, where N is the number of instructions needed to be squashed upon a branch prediction miss and P is the probability of correct branch prediction. It is assumed that branch instructions can be moved into the delay slots, and thus all delay slots can be filled. The forward semantic model is very suitable for pipelined processor design because the effect of increasing N (as the pipeline becomes deeper), on the performance is scaled by $(1-P)$. For example, given that we can predict 90% of all conditional branch instructions correctly and 100% of all unconditional branch instructions correctly, the miss

ratio is less than 10%. Therefore, the cost of a branch instruction increases by less than 0.1 as N increases by 1.

5.2. Performance Measure of Forward Semantics

There are three types of branch instructions in the IMPACT model: *cond_br* (conditional branch instruction with constant branch offset field), *uncond_br* (unconditional branch instruction with constant branch offset field), and *jump* (unconditional branch instruction with data dependent branch target address).

The cost of a *cond_br* instruction is $(1 + (1-P)*N)$, where N is the number of instructions needed to be squashed upon a branch prediction miss and P is the probability of correct branch prediction. For example, if P is 90% and N is 2 cycles, the cost of a *cond_br* instruction is 1.2 cycles.

The cost of an *uncond_br* instruction is 1, because P is 100%. Since all delay slots can be filled, no cycle is wasted.

jump instruction is mostly used to implement a distanced function call, function return, and hashing jump (multi-way jump). Since the branch target address is data dependent, we do not know how to fill the delay slots at compile time. However, for function calls and returns, it is always possible to place instructions required in the calling and returning sequence, for example, the register saving and stack operations in the delay slots. The greatest loss is in generating hashing jumps, for which the delay slots cannot be filled. Since a hashing jump is equivalent to a long sequence of simple *cond_br* instructions, the cost is tolerable. Also, one is able to move instructions prior to the *jump* instruction to fill the delay slots, as in the delayed branch model. Since the prediction is always correct, the delay slots are never squashed, and thus moving instructions from above the *jump* instruction to fill the delay slots is

correct. This trickery, however, violates the fundamental definition of the forward semantic model, although it produces correct results. It may be desirable to define a different model which defines the correct usage of the delay slots rather than the direction of code motion. This new model is not in the scope of this thesis.

5.3. Code Expansion Due to Adding Delay Slots

Since basic blocks are often small, requiring a small number of delay slots per branch instruction leads to a rather large code size increase. Two important questions thus arise. First, can we reduce the number of delay slots? Second, can we avoid unnecessary code duplication?

The answer to the first question is yes. The forward semantic model allows us to postpone the evaluation of the condition code. The number of delay slots required is $N = (((X+Y) \bmod cycle\_time) - 1)$, where X is the instruction cache access time and Y is the time to add the branch offset with the PC to form the branch target address. It is assumed that the reader is familiar with the instruction fetch unit design. The reason for including the instruction cache access time in determining the number of delay slots is because the branch offset is specified as a constant literal field and is available only after the instruction is fetched from the cache. Additional time is required to add this offset with the PC (of the branch instruction) to generate the branch target address. If the instruction fetch and the addition can be done in the same cycle, as in a non-pipelined design, no delay slot is necessary. If instruction cache access and the 32-bit addition require one cycle each, one delay slot is required.

With branch prediction, it is not necessary to wait for the condition code evaluation. As soon as the branch target address is available, the program

control flow is transferred to a new path. The obvious advantage is that the number of delay slots is not dependent on the execution pipeline design. It is dependent only on the instruction cache access time and the delay of a 32-bit adder. This is true also for *uncond_br* instructions. For *jump* instruction, the 32-bit addition can be avoided and extra time is needed to transfer a register value to the prefetch program counter. Since register access occurs in the first stage of the execution pipeline, the delay is small.

The answer to the second question is also yes. When a small basic block is completely absorbed in the delay slot space of its predecessor basic block, there is no need to keep the absorbed basic block if it can not be reached from any other basic block. This advantage is especially clear when there is a sequence of two-way branch instructions. In our model, these branches can be tightly placed together.

5.4. Experiment

In this subsection, experimental results support our claim that system-independent profiling can contribute to the computer architecture design/implementation. The compiler uses the profile result to predict the direction of the branch during compile time.

5.4.1. Benchmark set

The benchmark set is listed in Table 5.1. The inputs to these execution instances are taken from a research computing environment. The *line* column indicates the number of non-empty lines in each program excluding comments. The *run* column is the number of different execution instances of the corresponding benchmarks used to derive the profiling information. It is worth mentioning that the profile information presented in the following

sections is based on the execution of a few billion intermediate language instructions, which would be extremely difficult to handle for benchmarking and trace driven simulation.

5.4.2. Static branch prediction with squashing

The distribution of various types of branch instructions is listed in Table 5.2. The *%cond_br* column of Table 5.2 indicates the total number of conditional branch instructions profiled as a percentage of all control transfer instructions. The *%uncond_br* column of Table 5.2 indicates the total number of unconditional branch instructions profiled as a percentage of all control transfer instructions. The *%switch* column of Table 5.2 indicates the total number of multiway branch instructions profiled as a percentage of all control transfer instructions.

If a branch is predicted to be taken at the compile time, the delay slots are filled with the first two instructions from the taken path. On the other hand, if a branch is predicted not to be taken at the compile time, the delay slots are filled with the first two instructions from the fall-through path. We assume that the delay slots can contain branch instructions. Therefore, each branch instruction correctly predicted takes one cycle to execute, and each incorrectly predicted branch takes (1+N) cycles to execute, where N is the number of instructions which are issued after the branch instruction and are to be squashed.

We evaluate the cost of executing branch instructions using compiler predictions. We require that our measurement be based on many different execution instances of large, frequently used benchmarks.

Table 5.2 : Percentage of Various Branch Types

name	%cond_br	%uncond_br	%switch
cccp	54.18	26.79	19.03
cmp	68.88	31.11	0.01
compress	68.08	31.92	0.00
eqn	60.21	38.97	0.82
espresso	77.76	21.47	0.78
grep	70.85	25.35	3.80
lex	71.86	28.12	0.02
make	81.24	18.48	0.27
tar	93.71	6.28	0.01
tee	66.28	33.72	0.01
wc	70.80	29.20	0.00
yacc	89.13	10.72	0.15

5.4.3. Conditional branches

We first examine the characteristics of the conditional branches corresponding to the two-way decisions in the C programs. These branches are due to *if* statements, the && || ?: expressions, and the loop control structures. Table 5.3 shows how well the compiler can predict these branches during compile time, based on the profile information. Column *TT* of Table 5.3 indicates the number of branches which are predicted to be taken and are actually taken, as a percentage of all conditional branches. Column *TN* of Table 5.3 indicates the number of branches which are predicted to be taken but are actually not taken, as a percentage of all conditional branches. Column *NT* of

Table 5.3 indicates the number of branches which are predicted not to be taken but are actually taken, as a percentage of all conditional branches. Column *NN* of Table 5.3 indicates the number of branches which are predicted not to be taken and are actually not taken.

Table 5.3 : Conditional Branch Statistics

name	TT	TN	NN	NT	hit_ratio
cccp	46.85	6.26	41.94	4.95	88.79
cmp	0.00	0.00	96.93	3.07	96.93
compress	18.35	2.82	67.06	11.77	85.41
eqn	14.06	3.55	78.66	3.73	92.72
espresso	26.50	6.30	58.00	9.20	84.50
grep	2.60	0.09	95.37	1.94	97.97
lex	48.03	0.58	50.18	1.20	98.21
make	47.48	3.38	46.70	2.45	94.18
tar	90.19	0.63	8.71	0.46	98.90
tee	24.70	12.31	62.73	0.26	87.43
wc	10.56	2.95	75.26	11.22	85.82
yacc	38.27	1.98	51.62	8.13	89.89

TT and *NN* columns correspond to correct prediction, and *TN* and *NT* columns correspond to incorrect branch prediction. The *hit ratio* column of Table 5.3 is the total correct branch prediction ratio and is simply a sum of *TT* and *NN* columns.

NN branches are desirable because they keep the instruction buffers and instruction caches effective.

5.4.4. Multiway branches

Table 5.4 shows the characteristics of the multiway decision implementation. The *%default* column indicates the percentage of the time the *default* case is reached for all switch statements. The *%hash* column indicates the percentage of all switch statements being implemented by hashing jumps. The *%prof* column indicates the percentage of all switch statements being implemented by branch sequences. The *total* column indicates the average number of cases per switch, except the default case. The *expected* column indicates the expected number of comparisons required to resolve a switch statement implemented as a branch sequence.

Each multiway decision (switch statement) can be implemented by either 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 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 cost is too high, a hashing jump will be used.

For some benchmarks, the *%default* percentage is high. 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.

Table 5.4 : Multiway Branch Statistics

name	%default	%hash	%prof	total	expected
cccp	92.36	53.52	46.48	3.40	3.12
cmp	0.00	0.00	100.00	3.00	1.00
compress	0.00	0.00	100.00	10.00	1.00
eqn	82.98	76.21	23.79	6.97	6.07
espresso	66.18	0.00	100.00	2.71	1.86
grep	0.01	0.00	100.00	12.00	1.50
lex	30.73	0.00	100.00	12.81	5.02
make	39.39	0.00	100.00	8.85	4.71
tar	0.00	0.00	100.00	6.31	1.21
tee	0.00	0.00	100.00	3.00	1.00
wc	0.00	0.00	100.00	3.00	1.60
yacc	46.91	0.00	100.00	6.21	4.73

CHAPTER 6.

TRACE SELECTION

Code optimization techniques such as register allocation, code compaction, variable renaming, common subexpression elimination, copy propagation, dead code removal, constant folding and strength reduction can perform significantly better by favoring the important execution paths while penalizing the unimportant ones. Trace selection techniques with profiling information identify the important execution paths in terms of frequently invoked sequences of basic blocks.

Trace selection was first proposed by Fisher [Fis81] as a systematic approach to global microcode compaction. Since then, improvements and implementations of optimizations based on trace selection techniques have been reported [Lin83, Su84, Ell84, How87]. These techniques are useful for generating efficient codes 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. The IMPACT-I C compiler has been stable enough for us to compile large C programs. This allows us to observe the performance of trace selection algorithms on large C application programs over many runs. For different trace selection algorithms, we report the distribution of control transfers categorized according to their potential impacts on the microcode optimizations.

6.1. Trace Scheduling

We refer readers who are unfamiliar with trace scheduling to the original paper by Fisher [Fis81]. 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 codes at the *split* and *rejoin* points to preserve correctness. The three functions are described in great detail in Ellis's dissertation [Ell84].

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, unless they can be absorbed by jump optimization, to be considered when forming later traces. 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 that 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 also 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 multi-cycle 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 compacting 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 are desirable and how all basic blocks should be partitioned to various traces. It would be grossly complicated for the trace selection function to deal with micro-architecture dependent factors such as degree of hardware parallelism. Disregarding the hardware

limitations, the trace selection functions try 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, either by 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.

Our IMPACT-I C compiler allows automatic profiling and provides accurate execution weights for all control graph nodes and arcs. The problem now 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 finer partitioned to five different types. Together with in-trace transition, there are a total of six transisiton types (T1-T6).

T1 connects the last node of a trace to the start node of a different trace.

T2 connects the last node of a trace to a middle node of a trace.

T3 connects a middle node of a trace to the start node of a trace.

T4 connects two middle nodes.

T5 connects two nodes within a trace.

T6 connects the last node of a trace to the

start node of the same trace.

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

Let %a, %b, %c, %d, %e and %f denote the percentage 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.

6.2. Trace Selection

6.2.1. General selection function

In his trace scheduling paper, Fisher presented the following trace selection algorithm with node weights as the selection criterion. Later, Ellis in his dissertation implemented the same general trace selection algorithm but used arc weights as the selection criterion.

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_sucessor_of(current);
  if (s==0) exit loop;
  add s to the trace;
  mark s visited;
  current = s;
/* 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;
/* compaction and bookkeep */
trace_compaction;
book_keep;
```

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

To ensure that loop headers become the leading nodes of traces, in growing trace forward and backward, crossing loop back-edges is prohibited.

6.2.2. Selection according to node weight

Node weight is the execution count of a basic block. This number can either be estimated statically by loop analysis, or dynamically profiled by an automatic profiler. In this thesis, all weights used in the trace selection functions are strictly derived from the average program profile accumulated over many runs.

```
best_successor_of(node)
    n = Of all immediate successors of node,
        n has the highest execution count;
    if (n is visited) return 0;
    return n;
best_predecessor_of(node)
    n = Of all immediate predecessors of node,
        n has the highest execution count;
    if (n is visited) return 0;
    return n;
```

6.2.3. Selection according to arc weight

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 is taken. Hence, arc weight is a better selection criterion than node weight.

```
best_successor_of(node)
    e = Of all edges leaving node, e has the
        highest execution count (highest probability);
    n = the destination of e;
    if (n is visited) return 0;
    return n;
best_predecessor_of(node)
    e = Of all edges entering node, e has the
        highest execution count (highest probability);
    n = the source of e;
    if (n is visited) return 0;
    return n;
```

6.2.4. Selection with minimum arc probability requirement

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 a node Na is reached through an arc Ab is [arc_weight(Ab) / node_weight(Na)]. In Section 6.3, we measure the performance of this selection heuristic with several MIN_PROB values.

```
best_successor_of(node)
    e = Of all edges leaving node, e has the
        highest execution count (highest probability);
    if (probability(e) <= MIN_PROB) return 0;
    n = the destination of e;
    if (n is visited) return 0;
    return n;
best_predecessor_of(node)
    e = Of all edges entering node, e has the
        highest execution count (highest probability);
    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)));
```

6.3. Experiments

6.3.1. Procedure

The compiler compiles and profiles the benchmark programs by inserting extra codes 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.

6.3.2. The benchmark

Several programs from different application domains are chosen mainly because of their popularity and substantial program size. Each of these programs is run several times with realistic inputs. We have also made a special effort to exercise nearly all program options.

In Table 6.1, the *name* column lists the program name, the *line* column shows the number of non-empty lines of C code after preprocessing in each of the benchmarks, and the *run* column indicates the number of runs under profiler monitoring.

6.3.3. Percentage of transaction types

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. The *terminal* column is the percentage of control flow transitions from the end of a trace to the beginning of another trace (T1 type). The *loop* column is the percentage of control flow transitions from the end of a trace to the beginning of the same trace (T6 type). The *desirable* column is the percentage of control flow transitions within traces (T5 type). The *undesirable* column is the sum of T2, T3 and T4 type transitions.

Table 6.2 corresponds to the selection according to node weight function. Table 6.3 corresponds to the selection according to arc weight function. Tables 6.4 through 6.7 demonstrate the effect of imposing additional minimum branch probability requirement.

Table 6.1 : Benchmark Set

name	line	run	size(B)	description
cccp	4660	20	20411	GNU C preprocessor
cmp	371	16	1098	Compare text files
compress	1941	20	5086	File compression
eqn	4167	20	15860	Typeset mathematics
espresso	11545	20	49805	Logic minimization
grep	1302	20	2955	String search
lex	3251	4	26722	Lexical analyzer generator
make	7043	20	24258	Maintain files
tar	3186	14	10065	Create tape archives
tee	1063	18	1846	Replicate output
wc	345	20	1056	Word count
yacc	3333	8	29677	Parsing program generator

Table 6.2 : Selection According to Node Weight

name	desirable	undesirable	loop	terminal	trace	loop
cccp	39.60%	17.51%	27.00%	15.88%	1.9	1.8
cmp	83.02%	4.23%	12.74%	0.02%	6.9	7.0
compress	58.90%	26.49%	10.64%	3.97	4.0	3.5
eqn	61.62%	28.73%	6.07%	3.58%	4.1	8.5
espresso	44.87%	20.12%	24.24%	10.76%	2.3	2.2
grep	54.50%	7.19%	1.89%	36.42%	2.7	4.4
lex	62.15%	2.95%	34.14%	0.76%	2.8	2.8
make	39.08%	7.69%	35.21%	18.03%	1.8	1.7
tar	11.20%	1.92%	85.16%	1.72%	1.1	1.1
tee	66.86%	16.59%	16.23%	0.32%	4.0	4.0
wc	57.29%	24.18%	7.48%	11.05%	3.2	6.0
yacc	47.57%	12.99%	29.85%	9.59%	2.2	2.4
AVG	52.22%	14.22%	24.22%	9.34%	3.1	3.8
SD	17.88%	9.45%	22.28%	10.53%	1.5	2.3

Table 6.3 : Selection According to Arc Weight

name	desirable	undesirable	loop	terminal	trace	loop
cccp	49.59%	11.85%	27.20%	11.36%	2.3	2.0
cmp	83.03%	4.23%	12.72%	0.02%	6.9	7.0
compress	73.95%	11.45%	10.65%	3.95%	5.8	3.5
eqn	86.95%	5.55%	5.87%	1.63%	10.2	10.6
espresso	55.71%	15.76%	23.22%	5.32%	3.0	2.8
grep	79.53%	3.86%	14.48%	2.13%	6.2	6.9
lex	63.27%	2.08%	34.32%	0.33%	2.8	2.8
make	45.83%	3.57%	36.27%	14.34%	1.9	1.7
tar	12.97%	0.55%	85.15%	1.33%	1.2	1.1
tee	67.10%	16.55%	16.23%	0.10%	4.0	4.0
wc	75.89%	9.03%	7.46%	7.62%	5.5	7.0
yacc	54.08%	9.91%	29.87%	6.14%	2.6	2.5
AVG	62.33%	7.97%	25.29%	4.52%	4.4	4.3
SD	20.57%	5.33%	21.49%	4.65%	2.6	2.9

Table 6.4 : Minimum Branch Probability = 60%

name	desirable	undesirable	loop	terminal	trace	loop
cccp	42.58%	4.41%	21.46%	31.56%	1.8	1.8
cmp	83.03%	4.23%	12.72%	0.02%	6.9	7.0
compress	68.93%	7.74%	10.64	12.68	3.8	3.5
eqn	84.99%	3.61%	5.97%	5.44%	7.6	9.8
espresso	52.87%	9.67%	20.79%	16.68%	2.4	2.3
grep	77.33%	1.85%	12.74%	8.08%	4.7	5.3
lex	63.22%	1.96%	34.24%	0.58%	2.8	2.8
make	44.89%	2.60%	36.14%	16.37%	1.9	1.7
tar	12.87%	0.44%	85.14%	1.56%	1.2	1.1
tee	75.06%	0.29%	8.17%	16.48%	4.0	4.0
wc	75.88%	9.02%	7.48%	7.61%	5.5	7.0
yacc	53.25%	8.05%	29.19%	9.51%	2.4	2.4
AVG	61.24%	4.49%	23.72%	10.55%	3.8	4.1
SD	21.01%	3.34%	21.98%	8.97%	2.1	2.7

Table 6.5 : Minimum Branch Probability = 70%

name	desirable	undesirable	loop	terminal	trace	loop
cccp	41.05%	3.74%	21.55%	33.68%	1.8	1.9
cmp	83.03%	4.23%	12.72%	0.02%	6.9	7.0
compress	61.85%	3.11%	5.00%	30.05%	2.8	2.6
eqn	81.69%	2.35%	5.48%	10.48%	5.8	9.5
espresso	45.64%	5.37%	18.24%	30.75%	1.9	2.1
grep	77.24%	1.80%	12.74%	8.22%	4.7	5.3
lex	63.19%	1.79%	34.14%	0.88%	2.8	2.8
make	43.99%	2.08%	35.70%	18.23%	1.8	1.7
tar	12.77%	0.38%	85.14%	1.71%	1.2	1.1
tee	75.00%	0.24%	8.17%	16.6%	4.0	4.0
wc	75.88%	9.02%	7.48%	7.61%	5.5	7.0
yacc	46.25%	4.62%	24.11%	25.02%	2.0	2.1
AVG	58.97%	3.23%	22.54%	15.27%	3.4	3.9
SD	21.36%	2.42%	22.31%	12.28%	1.9	2.7

Table 6.6 : Minimum Branch Probability = 80%

name	desirable	undesirable	loop	terminal	trace	loop
cccp	37.12%	2.35%	20.73%	39.80%	1.6	1.8
cmp	83.02%	4.22%	12.74%	0.02%	6.9	7.0
compress	55.89%	1.83%	5.00	37.29	2.3	2.6
eqn	79.05%	1.66%	5.23%	14.06%	4.9	10.0
espresso	42.16%	4.05%	17.98%	35.81	1.8	2.0
grep	76.76%	1.73%	12.74%	8.76%	4.5	5.3
lex	62.99%	1.37%	33.81%	1.83%	2.8	2.8
make	43.56%	1.93%	35.63%	18.88%	1.8	1.7
tar	12.64%	0.32%	85.14%	1.91%	1.1	1.1
tee	75.00%	0.24%	8.17%	16.6%	4.0	4.0
wc	47.19%	2.16%	0.00%	50.65%	2.0	0.0
yacc	44.65%	3.33%	23.31%	28.71%	1.9	2.1
AVG	55.00%	2.10%	21.71%	21.19%	3.0	3.4
SD	21.06%	1.26%	22.86%	16.98%	1.7	2.8

Table 6.7 : Minimum Branch Probability = 90%

name	desirable	undesirable	loop	terminal	trace	loop
cccp	31.5%	1%	19.4%	53.14%	1.5	1.6
cmp	73.62%	0.88%	0.00%	25.50%	3.9	0.0
compress	53.48%	1.13%	4.98%	40.40%	2.2	2.6
eqn	76.34%	1.13%	5.12%	17.42%	4.3	10.3
espresso	38.09%	1.65%	15.85%	44.40%	1.6	1.9
grep	76.45%	1.69%	12.74%	9.13%	4.5	5.3
lex	61.39%	0.77%	33.35%	4.48%	2.6	2.8
make	40.64%	1.24%	34.25%	23.87%	1.7	1.6
tar	12.53%	0.27%	85.14%	1.91%	1.1	1.1
tee	75.00%	0.24%	8.17%	16.60%	4.0	4.0
wc	39.05%	0.00%	0.00	60.95%	1.6	0.0
yacc	38.68%	1.6%	19.33%	40.40%	1.7	2.1
AVG	51.40%	0.97%	19.86%	28.18%	2.6	2.8
SD	21.15%	0.57%	23.51%	19.42%	1.3	2.8

6.3.4. Discussion of results

As we have expected, arc weight is a better selectia 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 percentages decline slightly. However, as the minimum requirement rises, fewer and smaller traces are formed, leading to a low percentage of in-trace transitions.

In any case, the in-trace transition (%e) is several times larger than the off-trace transitions (%b, %c, %d). 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.

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.

Of all 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.

An inner loop as seen by the IMPACT-I 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 branches in inner loops.

CHAPTER 7.

INSTRUCTION PLACEMENT

7.1. Introduction

The instruction memory hierarchy has received only moderate attention because conventional machines typically have a high microcycle count per instruction, and thus demand low instruction bandwidth. For instance, a VAX-11/780 takes 10.5 microcycles to execute every 3.8 bytes of instructions [Eme84]. An 8-byte instruction buffer that prefetches instructions during idle cache cycles provided 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 that require a much higher instruction bandwidth. For example, the VAX 8600 implementation requires 3.8 instruction bytes for every 6 microcycles. Futher reducing the number of microcycles per instruction will increase the instruction memory bandwidth requirement and suggests the need of better instruction hierarchy designs.

Many processor architectures have adopted instruction formats and semantics which allow the instruction units to be efficiently pipelined [Rus78, Hen81, Pat81, Pat82]. To simplify instruction decoding, these processor architectures specify fixed instruction formats which unfortunately prevent conventional encoding techniques from reducing the program size. To simplify instruction sequencing, these processors specify instructions whose functionality are close to the microinstructions of the microprogrammed processors and prevent the use of powerful opcodes to 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. The cost is an increase in the dynamic code size and, consequently, an increase of the instruction bandwidth requirement.

Compiler code improvement techniques often increase code size. Inline expansion reduces function call overhead at the cost of increased code size [Hwu89]. Loop unrolling increases code scheduling flexibility at the cost of increased code size [Ell84]. Trace scheduling extracts the program parallelism at the cost of increased code size [Fis81, Ell84]. These techniques rely on the instruction memory hierarchy to absorb the code expansion cost so that the program execution speed can be improved. This adds a 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 [Smi82, Hil85]. For example, the MIPS-X processor uses an 2048-byte, 8-way set-associative instruction cache with 8-byte blocks. This approach is limited by the fact that the cache cycle time increases as the size and set-associativity increase and the fact that only a limited amount of hardware is available [Eic88, Alp88, Mit88, Prz88]. To make it 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. However, previous research on the instruction cache design have failed to examine the importance of compiler instruction placement algorithms.

With advances in the compiler technology, an increasing number of microarchitecture design parameters have been exposed to the compiler. Examples of this trend include pipeline latency [Hen83, Rad82], parallel data path [Ell84, Fis81, Col87], and register-memory hierarchy. The advantage of exposing these microarchitecture details to the compiler is that the compiler

can generate codes to take advantage of the microarchitecture features (pipelining, parallel data paths, and fast registers) without expensive hardware schemes. We believe that the instruction memory hierarchy should also be exposed to the compiler for improving the system performance.

In this section, we present an instruction placement algorithm which improves the efficiency of caching in the instruction memory hierarchy. Based on dynamic profiling, this algorithm maximizes spatial locality and minimizes cache mapping conflicts of the instruction accesses. The instruction placement algorithm has been implemented in the IMPACT-I and produced instruction placement for realistic C programs. The instruction placement for each program is based on the execution of millions of instructions using typical input files.

The instruction cache performance for each program, after applying the instruction placement algorithm, is measured by trace driven simulation. We demonstrate that the instruction layout algorithm can efficiently exploit small (below 2048B), direct-mapped instruction caches with large (64B) blocks. Direct mapped caches with large blocks are desirable due to their low control overhead (tag store and hit detection logic). The effect of varying the cache design parameters (cache size, block size, block sectoring, partial loading) is presented.

7.2. Algorithm

The goal of the IMPACT-I C compiler instruction placement mechanism is to lay out the target program to maximize the spatial locality and to minimize the mapping conflict. To maximize the 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 will be used when that block is

brought in cache. To minimize the mapping conflict, functions with overlapping lifetimes are mapped into different blocks of the cache. The mechanism is implemented in five major steps: execution profiling, function inline expansion, trace selection, function layout, and global layout.

Step 1. Execution profiling. In our C compiler, a program is represented by a weighted call graph. A call graph is a directed graph where every node is a function and every arc is a function call. A weighted call graph is one in which all the nodes and arcs are marked with their execution frequencies.

Each node of the weighted call graph is represented by a weighted control graph. A control graph for a function is a directed graph where 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 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 very 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 body if possible. The goal is to transform all the important inter-function control transfers into intra-function control transfers. Inline expansion reduces the dynamic inter-function control transfers to a small percentage

(about 1%) of all control transfers and, consequently, provides two major advantages. First, the spatial locality is increased in that almost all control transfers are within individual functions. Second, removing function calls also reduces potential cache mapping conflicts among functions.

Step 3. Trace* selection. For each function, basic blocks which tend to execute in sequence are grouped into traces. The traces are the basic units of instruction placement to maximize spatial locality. A recent paper gave a detailed description and evaluation of the trace selection algorithm of the IMPACT-I C Compiler [Cha88]. Note that the inline expansion step provides large functions to enhance the size of the traces selected.

Step 4. Function layout. By carefully placing traces of each function in a sequential order, spatial locality can be further preserved. 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 non-zero execution count (profiled count) have all been placed. Traces with zero execution count (profiled count) are moved to the bottom of the function. This results in a smaller effective function body and allows more effective parts of functions to be packed into each page.

Step 5. Global layout. Each function is assumed to have two parts : effective and non-executed parts. 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 inter-function cache conflicts are further reduced (already reduced by inline expansion).

---

* The term *trace* here is used as in the *trace scheduling* for global microcode compaction. It is not used as in the *trace driven simulation*. In this section, if the term *trace* is used as in the *trace driven simulation*, it will appear as *dynamic trace* whenever an ambiguity may occur.

In order to evaluate the effectiveness of our code layout scheme, we randomly select one input for each benchmark to take the traces of dynamic instruction accesses. These dynamic traces include instruction accesses to both the user code and the library code; they do not include any access to the kernel code.

In summary, the IMPACT-I instruction placement is based on profile information and the performance evaluation presented in this paper is based on trace driven simulation.

Appendix C gives an outline of the IMPACT-I instruction placement algorithm.

7.3. Experimentation

Table 7.1 and Table 7.2 summarize several important characteristics of our benchmarks. The *C lines* column shows the static code size of the C benchmark programs measured in the number of program lines. The *runs* column gives the number of different inputs used in the profiling process. The *instructions* column gives the dynamic code size of the benchmark programs, measured in number of million dynamic instructions. The *control* column gives the dynamic count of control transfers other than function call/return executed during the profiling process, measured in number of million dynamic instructions. Both *instructions* and *control* are accumulated for all the runs. The *input description* describes the nature of the inputs used in the profiling process.

Table 7.1 : Benchmark Set

name	C lines	runs	input description
cccp	4660	20	C programs (100-3000 lines)
cmp	371	16	similar/dissimilar text files
compress	1941	20	same as cccp
grep	1302	20	exercised various options
lex	3251	4	lexers for C, Lisp, awk, and pic
make	7043	20	makefiles for cccp, compress, etc.
tee	1063	18	text files (100-3000 lines)
tar	3186	14	save/extract files
wc	345	20	same as cccp
yacc	3333	8	grammar for a C compiler, etc.

Table 7.2 : Profile Results

name	instructions	control
cccp	11.7M	2.2M
cmp	2.2M	0.5M
compress	19.6M	3.1M
grep	47.1M	17.1M
lex	3052.6M	1125.9M
make	152.6M	32.4M
tee	0.43M	0.17M
tar	11M	1.5M
wc	7.8M	2.2M
yacc	313.4M	78.7M

7.4. Basic Experiments

Table 7.3 offers the inline expansion results. The *code inc* column gives the percentage of increase in static code size due to inline expansion. The *call dec* column gives the percentage of dynamic function calls eliminated by the inline expansion. The *DI's per call* column gives the average number of dynamic instructions executed between dynamic function calls after inline expansion. The *CT's per call* column gives the average number of dynamic control transfers executed between dynamic function calls after inline expansion.

Table 7.3 : Inline Expansion Results

name	code inc	call dec	DI's per call	CT's per call
cccp	17%	55%	506	95
cmp	3%	49%	265	58
compress	4%	91%	2324	368
grep	31%	99%	11214	4071
lex	23%	77%	7807	2880
make	34%	59%	388	82
tee	0%	0%	15	6
tar	16%	43%	983	127
wc	0%	0%	18310	5146
yacc	24%	80%	1205	303

For most of the benchmark programs, the inline expansion mechanism successfully eliminates a large percentage of the dynamic function calls. After inline expansion, the frequency of function calls is much smaller than the frequency of intra-function control transitions (branches). It is also observed that hundreds of dynamic instructions are executed per function call. The obvious gain is that register save and restore costs across function boundaries

are greatly reduced. A more subtle advantage that directly affects the performance of our instruction placement algorithm is that the function inline expansion mechanism enlarges function bodies and reduces inter-function interactions. More sequential and spatial localities can be found in larger function bodies. Reducing inter-function interactions also removes potential cache mapping conflicts among interacting functions. The result is that most of the complexity in the global layout process can be shifted to the intra-function layout process (trace selection and placement) which is much simpler to implement. We have thus decided to implement a simple global layout process based on a variant of the depth-first-search algorithm.

Table 7.4 Trace Selection Results

name	neutral	undesirable	desirable	trace length
cccp	55.23%	3.74%	41.05%	1.8
cmp	12.74%	4.23%	83.03%	6.9
compress	35.04%	3.15%	61.85%	2.8
grep	20.96%	1.80%	77.24%	4.7
lex	35.02%	1.79%	63.19%	2.8
make	53.93%	2.08%	43.99%	1.8
tar	86.85%	0.38%	12.77%	1.2
tee	24.77%	0.24%	75.00%	4.0
wc	15.09%	9.02%	75.88%	5.5
yacc	49.13%	4.62%	46.25%	2.0

Table 7.4 presents the trace selection results. The *neutral* column gives the percentage of control transfers from the end of a trace to the start of a trace. The average percentage (about 39%) for this category suggests that a careful selection of a linear ordering of traces could significantly increase the spatial locality. The function layout step in the instruction placement algorithm has been devised to capture this spatial locality. The *undesirable*

column gives the percentage of control transfers which enter and/or exit traces at a non-terminal basic block. The *desirable* column gives the percentage of control transfers which go from a basic block to its successor in a trace. The small average percentage (about 3%) in the *undesirable* column and the large average percentage (about 58%) in the *desirable* column indicate that once the control is transferred into a trace, it is likely to remain through the end of that trace. This justifies our approach to use the traces as units of instruction placement. The *trace length* column gives the average number of basic blocks in each trace. On the average, each trace contains 3.4 basic blocks. Since each basic block in the IMPACT-I code contains about 4 machine instructions (4 bytes each), the unit of instruction placements contains about 54 bytes. Considering the spatial locality among traces, a reasonable prediction of a good instruction block size would be about 64 bytes.

We use trace-based analysis to evaluate the effectiveness of our code layout scheme. A trace is generated by feeding a randomly selected input (typical size) to each benchmark program. These dynamic traces include both user code and library code, but not kernel code.

Table 7.5 shows the instruction memory access characteristics of the benchmark programs and their corresponding dynamic traces. The *total static bytes* column gives the number of machine code bytes generated for each benchmark program. The *effective static bytes* column gives the number of machine code bytes which have a non-trivial execution count. The *dynamic accesses* column gives the number of dynamic instruction accesses recorded in each dynamic trace.

Table 7.5 : Static and Dynamic Code Sizes of Benchmarks

name	total static size	effective static size	dynamic access
cccp	51.6K	29.6K	1.5M
cmp	2.8K	2.0K	0.3M
compress	15.6K	8.8K	2.8M
grep	11.1K	4.5K	0.1M
lex	40.4K	29.7K	51.9M
make	55.0K	34.1K	1.8M
tar	25.8K	15.7K	0.2M
tee	6.5K	3.4K	0.1M
wc	3.1K	2.6K	1.1M
yacc	35.7K	27.0K	3.3M

The effective static program size ranges from 2K to 34K whereas the total static program size ranges from 2.8K to 55K. Since the IMPACT-I compiler places the effective and ineffective parts of the program into different pages, only the effective part needs to be accommodated in the main and cache memories. As a result, when a page is transferred from the secondary memory to the main memory, all the bytes of that page are likely to be used.

7.5. Caching Experiments

The primary goal of the IMPACT-I instruction placement mechanism is to improve the cache performance and to reduce the cost of instruction memory hierarchy. As for the instruction caches, the goal is to minimize the data storage size and the control overhead (set-associativity and tag storage) to obtain the desired cache hit ratio and memory traffic. Direct-mapped caches are used in all the measurements due to their minimal set-associativity

overheads. In the next two tables, the effectiveness of the instruction placement mechanism for minimizing the data storage and the tag storage are shown.

Table 7.6 and Table 7.7 show the effect of varying cache size for a fixed block size (64 bytes). The *miss* columns give the cache miss ratios. The *traffic* columns give the ratios of the number of main memory accesses over the number of dynamic instruction accesses (memory traffic ratio). Note that for the block size of 64 bytes, a 2K-byte instruction cache provides a low miss ratio (average 0.5%) with a reasonable memory traffic ratio (average 8%). As a result, less than 1% of instruction accesses need to wait for the data from an outside cache or the main memory. Also, the bus to the outside cache and the main memory is only loaded by 8% of the instruction access traffic. Even a small instruction cache of 512 bytes provides a reasonable miss ratio (average 1.4%) with a moderate memory traffic ratio (average 22%). Comparing the cache sizes to the static program sizes reveals that the instruction placement algorithm is successful in mapping the programs into small caches.

Table 7.8 and Table 7.9 show the effect of varying the block size for a fixed cache size of 2048 bytes. In general, the miss ratios decrease and the memory traffic ratios increase as the block size increases. The miss ratios decrease with the increase of the block size because each cache miss brings in more useful bytes for larger block sizes. The instruction placement algorithm maximizes this effect by placing the bytes which are accessed close in time in the same block. The traffic ratios increase with the increase of the block size because each cache miss also brings in more useless bytes for large block sizes. The instruction placement mechanism minimizes this effect also by placing in the same block the bytes which are accessed close in time.

Table 7.6 : The Effect of Varying Cache Size (16K-4K)

name	16K		8K		4K	
	miss	traffic	miss	traffic	miss	traffic
cccp	0.29%	4.65%	0.86%	13.79%	1.53%	24.40%
cmp	0.01%	0.15%	0.01%	0.15%	0.01%	0.15%
compress	0.00%	0.07%	0.00%	0.07%	0.00%	0.08%
grep	0.06%	0.88%	0.06%	0.88%	0.06%	0.91%
lex	0.00%	0.03%	0.01%	0.09%	0.01%	0.21%
make	0.07%	1.16%	0.32%	5.06%	0.69%	11.10%
tar	0.09%	1.43%	0.09%	1.51%	0.24%	3.88%
tee	0.06%	0.92%	0.06%	0.92%	0.06%	0.092
wc	0.00%	0.06%	0.00%	0.06%	0.00%	0.06%
yacc	0.02%	0.26%	0.02%	0.28%	0.23%	3.64%

Table 7.7 : The Effect of Varying Cache Size (2K-512byte)

name	2K		1K		0.5K	
	miss	traffic	miss	traffic	miss	traffic
cccp	2.70%	43.13%	3.52%	56.32%	4.24%	67.87%
cmp	0.01%	0.15%	0.01%	0.15%	0.01%	0.17%
compress	0.01%	0.08%	0.01%	0.09%	3.54%	56.63%
grep	0.06%	0.87%	0.07%	1.11%	0.60%	9.62%
lex	0.03%	0.48%	0.06%	0.93%	0.31%	4.96%
make	1.35%	21.59%	2.03%	32.46%	2.44%	39.02%
tar	0.27%	4.27%	0.42%	6.76%	0.61%	9.79%
tee	0.08%	1.2%	0.08%	1.28%	0.08%	1.33%
wc	0.00%	0.06%	0.00%	0.06%	0.00%	0.06%
yacc	0.49%	7.86%	1.17%	18.73%	1.99%	31.89%

Table 7.8 : The Effect of Varying the Block Size (16B-32B)

name	16B		32B	
	miss	traffic	miss	traffic
cccp	7.53%	30.10%	4.32%	34.58%
cmp	0.04%	0.15%	0.02%	0.15%
compress	0.02%	0.07%	0.01%	0.08%
grep	0.19%	0.76%	0.10%	0.82%
lex	0.08%	0.33%	0.05%	0.38%
make	4.24%	16.95%	2.40%	19.19%
tar	0.72%	2.90%	0.42%	3.32%
tee	0.25%	0.98%	0.13%	1.06%
wc	0.01%	0.06%	0.01%	0.06%
yacc	1.13%	4.53%	0.66%	5.25%

Table 7.9 : The Effect of Varying the Block Size (64B-128B)

name	64B		128B	
	miss	traffic	miss	traffic
cccp	2.70%	43.13%	2.10%	67.33%
cmp	0.01%	0.15%	0.01%	0.16%
compress	0.01%	0.08%	0.00%	0.09%
grep	0.06%	0.91%	0.03%	1.01%
lex	0.03%	0.48%	0.02%	0.69%
make	1.35%	21.59%	0.95%	30.39%
tar	0.27%	4.27%	0.20%	6.37%
tee	0.08%	1.20%	0.04%	1.41%
wc	0.00%	0.06%	0.00%	0.06%
yacc	0.49%	7.86%	0.52%	16.78%

For a fixed cache size, the larger the block size, the smaller the number of tags that are required to manage the cache. For a 2K-byte instruction cache, the 64-byte block size provides a low miss ratio (average 0.5%) and reasonable memory traffic ratio (average 8%). The configuration requires only 16 tags, successfully minimizing the control overhead.

Note that the memory traffic ratio is rather high for benchmarks *cccp* and *make*. Also, since the cache miss penalty increases with the block size, the effective cache access time may increase in spite of the decreased miss ratio. For some systems (especially multiprocessor systems), it is desirable to decrease the memory traffic ratio and the cache miss penalty at the cost of increasing the miss ratio.

We assume that the memory or secondary cache is interleaved and can deliver one data per cycle after the initial access delay. We also assume that the data for which the cache miss occurs are the first data delivered after the initial memory access delay. To furthur reduce the cache miss penalty, the processor resumes execution as soon as the accessed data come back from main memory. Subsequent instruction fetches after a cache miss, if sequential, can directly obtain the instructions from the memory bus as the cache block is being repaired. When a branch is taken before the block is completely filled, the CPU is stalled until the block is completely transferred.

For a 64-byte block size and a 4-byte memory bus, 16 cycles are required after the initial memory access to complete the block transfer. Due to the large transfer size, the CPU may be stalled for two reasons. First, our layout algorithm does not guarantee that the data for which the repair sequence is incurred is positioned at the beginning of the cache block. Second, the CPU is stalled while repairing the part of the cache block in front of the data for which the miss is incurred. The average number of stalled cycles caused by

each cache miss is about half of the block, assuming random access pattern. For a 64-byte block and a 4-byte memory bus, the CPU is stalled for about 8 cycles. Including the initial memory access cost, the effective cache access time may increase although the miss ratio is lower than, for example, the 32 byte block size configuration.

One approach to decreasing the memory traffic ratio and the cache miss penalty while increasing the miss ratio is to partition each block into sectors and only bring in the accessed sector upon cache miss. The memory traffic ratio is reduced because the number of memory accesses caused by each cache miss is reduced to the size of each sector (rather than the size of each block), and thus fewer unused entries are fetched. The miss ratio increases because the spatial locality is not fully exploited. Since the instructions placed into the same block are likely to be executed near each other in time, not bringing in the rest of a missing block can be expected to cause more cache misses.

The *sector* column in Table 7.10 presents the effect of dividing the 64B blocks into sectors of 8 bytes each for a 2048B cache. A comparison with the *64B* column in Table 7.9 shows that, for programs causing large memory traffic ratios, sectoring the blocks decreases the memory traffic ratio at the cost of increasing the miss ratio. The problem with this approach is that it increases the miss ratio to such a degree (e.g., cccp) that the average cache access time can actually increase.

An alternative scheme is to load only part of the missing block, from the accessed location to the end of that block or to a valid entry previously loaded in. The processor resumes execution as soon as the accessed location comes back from main memory.

The *partial* column in Table 7.11 presents the effect of loading only part of the missing block. The *avg.fetch* column shows the average transfer size (in

4-byte entities) for a cache miss. The *avg.exec* column indicates the average number of consecutive instructions (4-bytes each) used starting from a cache miss point to a taken branch or another cache miss. A comparison with the *64B* column shows that, for programs causing large memory traffic ratio, this approach can significantly reduce the memory traffic ratio at the cost of only slightly increasing the miss ratio. Note that for programs with extremely small miss ratio and memory traffic ratio, this scheme can actually increase both ratios. However, since the traffic ratios are so low for these programs, a slight increase does not have visible effect on the system performance.

The code generated by the IMPACT-I C compiler very closely matches the physical code of a fixed instruction format (32bits/instruction) RISC type processor. To show that our result is more general, we will repeat the 2K, 64B block, partial loading experiment after code scaling. We scale the code to 0.6, 0.8 and 1.2 of its original size. The scaling affects the size of all basic blocks uniformly. The instruction size is still assumed to be 4 bytes, and therefore, the effect of code scaling is shown as changes in the number of instructions in each basic block. For each basic block, the number of instructions is rounded to the nearest integer value. The resultant scaling factor on the effective program sizes is 0.5, 0.7 and 1.1 for uniform scaling factors of 0.6, 0.8 and 1.2 respectively, after rounding the basic block instruction counts to integers.

The result supports our claim that our compiler instruction layout optimization is generally applicable to many instruction sets and compilers with differing code improving ability. A richer instruction set may reduce the number of instructions to realize the intermediate form. But the experimental result seems to indicate that the cache performance is rather stable.

Table 7.10 : Sectored Cache

name	sector	
	miss	traffic
cccp	13.88%	27.76%
cmp	0.33%	0.65%
compress	0.47%	0.94%
grep	0.11%	0.21%
lex	0.18%	0.35%
make	8.82%	17.64%
tar	1.62%	3.25%
tee	1.31%	2.62%
wc	0.16%	0.33%
yacc	2.79%	5.57%

Table 7.11 : Partial Loading

name	partial			
	miss	traffic	avg.fetch	avg.exec
cccp	2.86%	33.78%	11.8	8.2
cmp	0.05%	0.66%	14.2	12.3
compress	0.07%	0.99%	13.9	12.0
grep	0.02%	0.24%	12.6	9.9
lex	0.04%	0.41%	11.1	7.8
make	1.52%	19.77%	13.0	10.1
tar	0.28%	3.55%	12.8	12.2
tee	0.21%	3.00%	14.0	9.9
wc	0.02%	0.33%	14.9	12.7
yacc	0.55%	7.13%	13.1	9.0

Table 7.12 : The Effect of Code Scaling (0.5 & 0.7)

name	0.5		0.7	
	miss	traffic	miss	traffic
cccp	2.60%	25.88%	3.02%	31.02%
cmp	0.06%	0.77%	0.05%	0.75%
compress	0.08%	1.05%	0.07%	1.00%
grep	0.03%	0.31%	0.02%	0.27%
lex	0.02%	0.21%	0.03%	0.32%
make	1.26%	13.75%	1.57%	18.22%
tar	0.32%	4.30%	0.27%	3.16%
tee	0.24%	2.97%	0.24%	2.99%
wc	0.02%	0.37%	0.02%	0.36%
yacc	0.65%	5.81%	0.64%	6.75%

Table 7.13 : The Effect of Code Scaling (1.0 & 1.1)

name	1.0		1.1	
	miss	traffic	miss	traffic
cccp	2.86%	33.78%	3.21%	36.73%
cmp	0.05%	0.66%	0.05%	0.70%
compress	0.07%	0.99%	0.07%	1.02%
grep	0.02%	0.24%	0.02%	0.25%
lex	0.04%	0.41%	0.04%	0.41%
make	1.52%	19.77%	1.78%	23.10%
tar	0.28%	3.55%	0.32%	4.09%
tee	0.21%	3.00%	0.23%	2.95%
wc	0.02%	0.34%	0.02%	0.36%
yacc	0.55%	7.13%	0.42%	4.68%

CHAPTER 8.

CONCLUSION

8.1. Automatic Profiler

We have shown that a totally portable profiler can be integrated into a C compiler. The compiler front end remains machine independent. The profiling process can be distributed over a network of different computers. The profile result can be used on all machines.

8.2. Function Inline Expansion

Using the profile result, the function inline expansion facility in our IMPACT-I C Compiler can remove a large percentage of function calls/returns with modest code expansion. The inline expansion results in large working space for global code optimizations and also better cache locality.

8.3. Conditional Branch Handling

Most conditional branch instructions are highly biased in the benchmark programs which we have profiled. Using the profile information, we have shown that software branch prediction and forward semantics together perform as well as expansive hardware branch prediction and instruction buffering mechanisms.

8.4. Trace Selection

We have tested several trace selection algorithms and concluded that trace selection is indeed effective for optimizing large scalar programs. The off-trace cost can be reduced by imposing higher minimum branch probability requirement.

8.5. Instruction Placement

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 lifetime into memory locations which do not contend with each other in cache.

Using trace-driven simulation, we have demonstrated that the instruction layout algorithm can efficiently exploit small, direct-mapped instruction caches with large blocks. High instruction cache performance is achieved due to low miss ratio, low memory traffic ratio, and fast hardware. The effect of varying the cache design parameters (cache size, block size, block sectoring, and partial loading) has been presented.

8.6. Future Work

We are continuing this research in several directions. First, we are expanding the benchmark set to include more than 30 UNIX and CAD programs. Second, we are studying other code improving techniques, such as loop unrolling and software pipelining. Third, more experiments on register allocation and code scheduling will be performed. Our final goal is to construct a powerful tool for designing a high performance computer system, based on the IMPACT optimizing compilers. The tool will assist computer microarchitects to achieve a cost-effective design which balances the complexity of both the compiler technology used and the hardware techniques used.

REFERENCES

[Aho86] A.V. Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1986.

[All88] R. Allen and S. Johnson, "Compiling C for Vectorization, Parallelism, and Inline Expansion," Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988.

[Alp88] D.B. Alpert and M.J. Flynn, "Performance Trade-offs for Microprocessor Cache Memories," IEEE MICRO, August 1988.

[Bell79] AT&T Bell Laboratories, UNIX Programmer's Manual, section 1, AT&T Bell Laboratories, Murray Hill, N.J., January 1979. [prof, gprof, tcov]

[Cdc75] Control Data 7600 Hardware Reference Manual 60367200, Control Data, Arden Hills, MN, 1975.

[Cha82] G.J. Chaitin, "Register Allocation & Spilling Via Graph Coloring," ACM SIGPLAN Notice, Vol. 17-6, June 1982.

[Cha88] P.P. Chang and W.W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode," Proceedings of the 21th Annual Workshop on Microprogramming and Microarchitectures, November 1988.

[Cho84] F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring," Proceedings of the ACM SIGPLAN Symposium on Compiler Constructions, June 1984.

[Col87] 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.

[Cra78] R.M. Russell, "The Cray-1 Computer System," Communications of the ACM, Vol. 21, No. 1, January 1978, pp. 63-72.

[Dit87] D.R. Ditzel, H.R. Mclellan, and A.D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor," Proceedings of the 14th Annual International Symposium on Computer Architecture, June 2-5, 1987.

[Don79] J.J. Dongarra, LINPACK User's Guide, the Society for Industrial and Applied Mathematics, 1979.

[Eic88] 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.

[Ell84] J.R. Ellis, Bulldog: A Compiler for VLIW Architectures, The MIT Press, 1985, Ph.D. Dissertation, Yale, 1984.

[Eme84] 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.

[Fis81] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, Vol. c-30, No. 7, July 1981.

[Geh84] Narain Gehani, Ada Concurrent Programming, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[Goo88] J.R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," Proceedings of the 1988 International Conference on Supercomputing, ACM, July 1988.

[Gra83] S.L. Graham, P.B. Kessler and M.K. McKusick, "An Execution Profiler for Modular Programs," Software Practice and Experience, Vol. 13, John Wiley & Sons, Ltd., New York, 1983.

[Gra87] Michael Granski, Israel Koren, and Gabriel 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.

[Hen81] 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.

[Hen83] J.L. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983.

[Hil85] M.D. Hill and A.J. Smith, "Experimental Evaluation of On-Chip Cache Memories," Proceedings of the 11th Annual Symposium on Computer Architecture, June 17-19, 1985.

[How87] Michael A. Howland, Robert A. Mueller and Philip H. Sweany, "Trace Scheduling Optimization in a Retargetable Microcode Compiler," Proceedings of the 20th International Microprogramming Workshop, December 1987.

[Hus82] C.A. Huson, An In-line Subroutine Expander for Parafrase, M.S. Thesis, University of Illinois at Urbana-Champaign, 1982.

[Hwu87] W.W. Hwu, HPSm: Exploiting Concurrency to Achieve High Performance in a Single-chip Microarchitecture, Ph.D. Dissertation, Computer Science Division, University of California, Berkeley, December 1987.

[Hwu88] Wen-mei W. Hwu and Pohua P. Chang, "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator," The 15th International Symposium on Computer Architecture Conference Proceedings, May 1988.

[Hwu89] W.W. Hwu and P.P. Chang, "Inline Function Expansion for Compiling C Programs," ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, June 1989.

[Hwu89_2] Wen-mei W. Hwu and Pohua P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," The 16th Annual International Symposium on Computer Architecture Conference, May 1989.

[Hwu89_3] Wen-mei W. Hwu, Thomas M. Conte, and Pohua P. Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," The 16th Annual International Symposium on Computer Architecture Conference, May 1989.

[IBM78] IBM Maintainance Library 3033 Processor Complex Theory of Operation/Diagrams Manual, Vol. 1-3, Jan. 1978, IBM, Poughkeepsie, N.Y.

[Kog81] P.M. Kogge, The Architecture of Pipelined Computers, McGraw-Hill, 1981.

[Kuc81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependency Graphs and Compiler Optimizations," Proceedings of 8th ACM Symposium on Principles of Programming Languages, January 1981.

[Law75] Duncan H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Transactions on Computers, Vol. c-24, No. 12, December 1975.

[Lee84] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, January 1984.

[Lin83] J.L. Linn, "SRDAG Compaction: A Generalization of Trace Scheduling to Increase the Use of Global Context Information," Proceedings of the 16th Microprogramming Workshop, Downingtown, PA, October 1983.

[McF86] S. McFarling and J.L. Hennessy, "Reducing the Cost of Branches," the 13th International Symposium on Computer Architecture Conference Proceedings, June 1986.

[McM84] F.H. McMahon, L.L.N.L. FORTRAN KERNELS: MFLOPS, Lawrence Livermore National Laboratories, 1984.

[Mit88] C.L. Mitchell and M.J. Flynn, "A Workbench for Computer Architects," IEEE Design and Test of Computers, February 1988.

[Pat81] D.A. Patterson and C.H. Sequin, "RISC-1: A Reduced Instruction Set VLSI Computer," Proceedings of the Eighth Symposium on Computer Architecture, May 1981.

[Pat82] D.A. Patterson and C.H. Sequin, "A VLSI RISC," IEEE Computer, September 1982.

[Prz88] S. Przybylski, M. Horowitz, and J.L. Hennessy, "Performance Tradeoffs in Cache Design," The 15th International Symposium on Computer Architecture Conference Proceedings, May 1988.

[Rad82] G. Radin, "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, March 1982.

[Ram77] C.V. Ramamoorthy and H.F. Li, "Pipelined Architecture," Computing Surveys, Vol. 9, No. 1, March 1977.

[Rus78] R.M. Russell, "The Cray-1 Computer System," Comm. ACM, Vol. 21, No. 1, January 1978.

[Smi81] J.E. Smith, "A Study of Branch Prediction Strategies," Proc. Eighth Symp. Computer Architecture, May 1981.

[Smi82] A.J. Smith, "Cache Memories," Computing Surveys, Vol. 14, No. 3, September 1982.

[Sta88] R.M. Stallman, Internals of GNU CC, Free Software Foundation, Inc., 1988.

[Str87] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley Publishing Company, July 1987.

[Su84] B. Su, S. Ding, and L. Jin, "An Improvement of Trace Scheduling for Global Microcode Compaction," Proceedings of the 17th Microprogramming Workshop, New Orleans, LA, November 1984.

[Tok81] M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of Microprograms," IEEE Transactions on Computers, Vol. c-30, No. 7, July 1981.

APPENDIX A.

INLINE EXPANSION ALGORITHM

1) For each function, apply local inline analysis;
2) Apply global inline analysis;
3) Perform global inline decision;
4) Perform inline expansion;

```
LocalInlineAnalysis() {
        1 : create a new node;
        2 : compute various attributes :
                (func_name, func_weight, param, func_id)
        3 : assign unique id to each expression;
        4 : compute (destination) = immediate subcalls;
                one entry per call site.
                callee = ### for call through pointers;
        5 : compute (param_size, local_size)
        6 : compute Pset(f);
        7 : compute (size) = number of IL instructions;
}

GlobalInlineAnalysis() {
        1 : replace the callee of all call sites to external functions with
                $$$;
        2 : compute the outgoing edges of $$$ and ###;
        3 : Linearization();
        4 : DetectRecursion();
}
```

```
Linearization() {
        sort all nodes in such a way that more important
        functions appear in front of the list; (profile weight)
}

DetectRecursion() {
        for all functions Fi do
                LCALL(Fi) = Fi.destination;     /* immediate successors */
        GCALL(Fi) = LCALL(Fi) + GCALL(Fk) for all
                successors Fk of Fi.
        if (Fi in GCALL(Fi)) then Fi is recursive;
}

InlineDecision() {
        for all call arcs do
                if (caller or callee is $$$ or ###)
                        mark the arc "not_expandable";
                if (arc.weight < MIN_INLINE_WEIGHT)
                        mark the arc "not_expandable";
                mark the arc "expandable";
        sort all "expandable" edges according to weight; (profile weight)
        new_size = original_size = ProgSize();
        according to the sorted order (most important first)
                if (Expandable(e) {
                        if (code expansion after e < MAX_RATIO) {
                                mark e "expand";
                                new_size = ProgSize();
                                NewLocalSize();
                        }
                }
}
```

```
Expandable(e(Fi, Fj)) {
    for a call from Fi to Fj;
    1) Fj must precede Fi in the linear list;
    2) if (Fi is recursive) Fj.param_size+Fj.nlocal_size < MAX_DATA_SIZE;
}

ProgSize() {                    /* estimate new program size */
    for all functions Fi do
        Fi.nsize = Fi.size;
    according to the linear list order
        Fi.nsize += Fk.nsize;  for every 'expand' call site
            in Fi, e=(Fi, Fk);
    prog_size = sum of all Fi.nsize;
}

NewLocalSize() {                /* estimate new local declaration size */
    for all functions Fi do
        Fi.nlocal_size = Fi.local_size;
    according to the linear list order
        Fi.nlocal_size += Fk.nlocal_size; for every 'expand' call
            site in Fi, e=(fi, Fk);
}

InlineExpansion() {
    according to the linear list order
    expand all 'expand' edges;
}
```

APPENDIX B.

SAMPLE PROFILE PROGRAM

Source program

```
main() {
   int i, c;
   for (i=0; i<10; i++)
      c = (i>3 && i<5)? 1: 0;
}
```

Profile program

```
int main() {
   int i_____1;
   int c_____1;
      ProfProbe0(0, "", "pfile");
      ProfProbe1(1);
lll_L1 :
      ProfProbe3(1);
      i_____1 = 0;
      if (i_____1<10) goto lll_L2;
      else goto lll_L3;
lll_L2 :
      ProfProbe3(2);
      c_____1 = (((i_____1>3) && ((ProfProbe4(4)), (i_____1<5))) ?
        ((ProfProbe4(5)),1) : ((ProfProbe4(6)),0));
      i_____1++;
      if (i_____1<10) goto lll_L2; else goto lll_L3;
lll_L3 :
      ProfProbe3(3);
      ProfProbe2();
      return;
}
```

APPENDIX C.

INSTRUCTION PLACEMENT ALGORITHM

```
MIN_PROB = 0.7;

Algorithm TraceSelection {
    /** select the best immediate successor of the basic block, bb **/
    best_successor(bb) {
        ln = the outgoing arc with the highest execution count.
        if (weight(ln)==0) return 0;
        if (weight(ln)/weight(bb) < MIN_PROB) return 0;
        if (weight(ln)/weight(destination(ln)) < MIN_PROB) return 0;
        if (destination(ln) has been selected) return 0;
        return ln;
    }
    /** select the best immediate predecessor of the basic block, bb **/
    best_predecessor(bb) {
        ln = the incoming arc with the highest execution count.
        if (weight(ln)==0) return 0;
        if (weight(ln)/weight(bb) < MIN_PROB) return 0;
        if (weight(ln)/weight(source(ln)) < MIN_PROB) return 0;
        if (source(ln) has been selected) return 0;
        return ln;
    }
    trace_select(F) {
        int trace_id = 0;
        if (weight(F)==0),{
            /** for non-executed functions, each basic
             ** block forms a trace.
             **/
            for (all BBi in F) {
                trace_id = trace_id + 1;
                BBi.trace_id = trace_id;
            }
            return;     /** exit function **/
```

```
    }
    /** for non-zero weight functions. **/
    sort all BBi in F according to weight(BBi);
    mark all BBi in F not-selected;
    while (there are not-selected BB) {
        trace_id = trace_id + 1;
        seed = the not-selected BB with the highest
            execution count;
        seed.trace_id = trace_id;
        /** grow the trace forward **/
        current = seed;
        for (;;) {
            ln = best_successor(current);
            if ((ln==0) or (destination(ln)==ENTRY)) {
                break;          /** exit for loop **/
            }
            s = destination(ln);
            s.trace_id = trace_id;
            current = s;
        }
        /** grow the trace backward **/
        current = seed;
        for (;;) {
            if (current==ENTRY) {
                break;          /** exit for loop **/
            }
            ln = best_predecessor(current);
            if (ln==0) {
                break;          /** exit for loop **/
            }
            s = source(ln);
            s.trace_id = trace_id;
            current = s;
        }
    }
}
```

```
Algorithm FunctionBodyLayout {
      mark all traces un-visited;
      function space = 0;
      current = ENTRY trace;
L1 :  while (current<>0) {
            mark current visited;
            place the trace into the function space;
            /** try to find a connection to a trace header.
             ** we consider only non-zero weight traces.
             **/
            best = best trace connected to the current trace's
                  tail. (terminal to terminal connection only)
            if (weight(best)<>0) {
                  current = best;
                  continue;        /* goto L1 */
            }
            /** if there is no sequential locality at all,
             ** we will start from the most important not-visited
             ** trace.
             **/
            best = the most important trace among not-selected traces;
            if (best==0) {
                  /** all traces have been processed. **/
                  break;                  /* goto L2 */
            } else {
                  current = best;
                  continue;        /* goto L1 */
            }
      }
L2 :
}

Algorithm GlobalLayout {
      * assume a call graph is available.
      find all call sites (Fi, Fj) == Fi calls Fj;
      weight(Fi, Fj) = sum of all calls from Fi to Fj;
```

```
        except when Fi==Fj, weight(X,X) = 0.
    for each function Fi,
        determine the size of its active region.
        determine the size of its non-active region.
    /** apply depth-first-search, mark every node **/
    Fi.visit = false for all Fi;
    from functions Fi on top of the call graph hierarchy (e.g. "main")
        if (Fi.visit==false)
            Visit(Fi);
    /** layout the function according to the depth-first order. **/
    according to DFS order, layout the effective region of all
    functions.
    according to the same DFS order, layout the non-active region of
    the functions.
}

Visit(F) {
    static int id=1;
    F.visit = true;
    F.id = id++;
    sort all subcalls from F by weight(F, Fj);
    /** from the most important to the least important call site. **/
    for all callees Fj in the sorted order
      if (Fj.visit==false)
        Visit(Fj);
}
```