COMPILER SUPPORT FOR SPARC ARCHITECTURE PROCESSORS

BY

ROLAND G. OUELLETTE

S.B., Massachusetts Institute of Technology, 1986

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, 1994

Urbana, Illinois

ABSTRACT

This work shows how a single compiler front-end and optimization suite may be used to generate high quality code specific processors. A C language front-end is used. An initial pass of the compiler is used to instrument the program in order to collect a trace of the dynamic behavior of programs into an execution profile used to guide later code optimization phases. Generic code optimization techniques are applied, then machine specific optimizations are performed. Code is then generated using a machine specific code generator, and then several more machine specific optimizations are performed. Results gathered in generating code for the Sparc architecture are presented in this work. In spite of an incomplete suite of optimizations, the output code is of comparable overall quality to that generated by the Sun compiler.

ACKNOWLEDGMENTS

My Mom and Dad's encouragement and support enabled me to achieve as much as I have.

I thank Professor Janek Patel with whom I spoke at great length; the conversations I had with him helped me to choose the University of Illinois.

I thank Professor Wen-Mei Hwu; the thesis work I have done was interesting and appropriately challenging. As a result, I was able to change careers and do exciting and challenging work on production quality compilers.

I thank all my friends in the Center for Reliable and High-performance Computing and in the Computer Science Department at the University. The time we spent working, sharing information and relaxing together allowed me to remain sane and fairly relaxed at a time that would have otherwise been extremely stressful. In particular, I thank John Coolidge, Johnny "Flame" Zweig, Heeren Pathak, Jeff Wilson, Rick, Don Meyer, Laura, Jostlin, Michael Chan and Jennifer Shannon.

I thank the many people I worked with on components of the IMPACT-I C compiler. At times I made extraordinary requests of them which they met. I worked closely with Po-hua Chang, Scott Mahlke, and William Chen. I also worked with Roger Bringmann and Nancy Warter. These people have spent many weekends and very late nights with me in the laboratory. The community effort was productive and enjoyable.

I thank all of the people at the Digital Equipment Corporation who sent me to attend graduate school. The Graduate Engineering Education Program (GEEP) is one of the best industrial education programs; I am very happy that I was accepted into the program. In particular, I thank Tim Stanley, Lee Peterson and Terry Ann Sarandrea, Masood Heydari, Jean Basmaji, Jack Rosen and Glenn Garvey.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 The Thesis

The premise of the IMPACT group's research into compilers is that it is possible to build compilers from a collection of components. With a suite of components and utility tools, research into new optimizations techniques and new computer architectures is facilitated. A new optimization can be written and tried with combinations of previously developed components. Further, retargeting the compiler requires only a new code generator, new machine and tuning parameters, and a few extra components. An easily retargeted compiler may be used to demonstrate new architectural features and/or new architectures. Using the same base compiler technology allows for more controlled optimization and architectural experimentation.

Effective architectural exploration requires a good compiler. Very misleading results are obtained if hand-assembled benchmarks are used, since they are small and their behavior may be unrepresentative of programs written in high level languages. If second-rate compiler technology is used to compile real code, misleading results may also be obtained. The machine code will contain disproportionately more call sequences and may use far fewer registers than a good compiler can use. Also since data for exiting machines are usually collected using a mature compiler technology, a fair comparison to a new architecture cannot be directly obtained. Finally, some architectural features require good compiler support to show their merits and faults; surprises occur late in the product life cycle and are difficult to address after an architecture is sold to customers.

To demonstrate that the existing IMPACT compiler could be retargeted easily, and to discover any problems in the process, I write a code generator for the Sparc architecture. The group's previous work targeting the MIPS architecture had been very positive. Code for MIPS ran faster than that generated by the compiler written by MIPS, which had a reputation as being one of the best production compilers available. We were interested in writing this code generator due to the wide availability of Sparc-based computers, especially in our laboratory, and the relatively poor machine code we had seen generated by the compilers available on that platform. To measure our success, the compiler was then tested using a group of benchmark programs. The results gathered were runtimes of those benchmark programs which were compared with the results when using the bundled Sparc C compiler supplied by Sun.

1.2 Organization of This Thesis

Chapter 1.2 provides an overview of the IMPACT C compiler. Chapter 2.5 describes optimizations performed for all target architectures. Chapter 3.7 contains descriptions of optimizations used for the Sparc architecture. Chapter 5 describes the code generator, the features implemented, and those left as future work. Performance measurements are given in Chapter 6. An analysis of the results, what was learned, and the problems encountered are given in the Conclusions, Chapter 7.

CHAPTER 2

OVERVIEW OF THE IMPACT-I C COMPILER

The IMPACT-I C compiler is written as a collection of programs. Each program performs a step in the translation of a previously debugged C program into an assembly language program optimized for a specific machine. The transformation occurs in stages, using two intermediate languages, H-code and L-code. Figure 2.1 is a schematic of the compiler and its stages. The next sections give an overview of the stages: profiling, H-code, L-code, optimizations on the intermediate code, and assembly code generation.

2.1 Profiling

One of the useful pieces of information about an instruction in a program is the frequency of its execution in the program; we define an instruction's *weight* as its frequency of execution. A series of instructions which are executed together is called a *basic block*; a basic block is always entered from the top and conceptually ends in a branch instruction. Since all of the instructions in a basic block have the same weight, it is only necessary to keep weights on a block by block basis. Branch instructions may have more than one following instruction; the instructions which follow a branch instruction are called *destinations*. The frequency with which one destination instruction is reached is called the *arc weight*.[1] The collection of all of the instruction and arc weights of a program is called the *execution profile* of the program, *profile information* or simply the *profile*.

[1]It is possible to derive all of the instruction weights from the arc weights, but not the other way around.

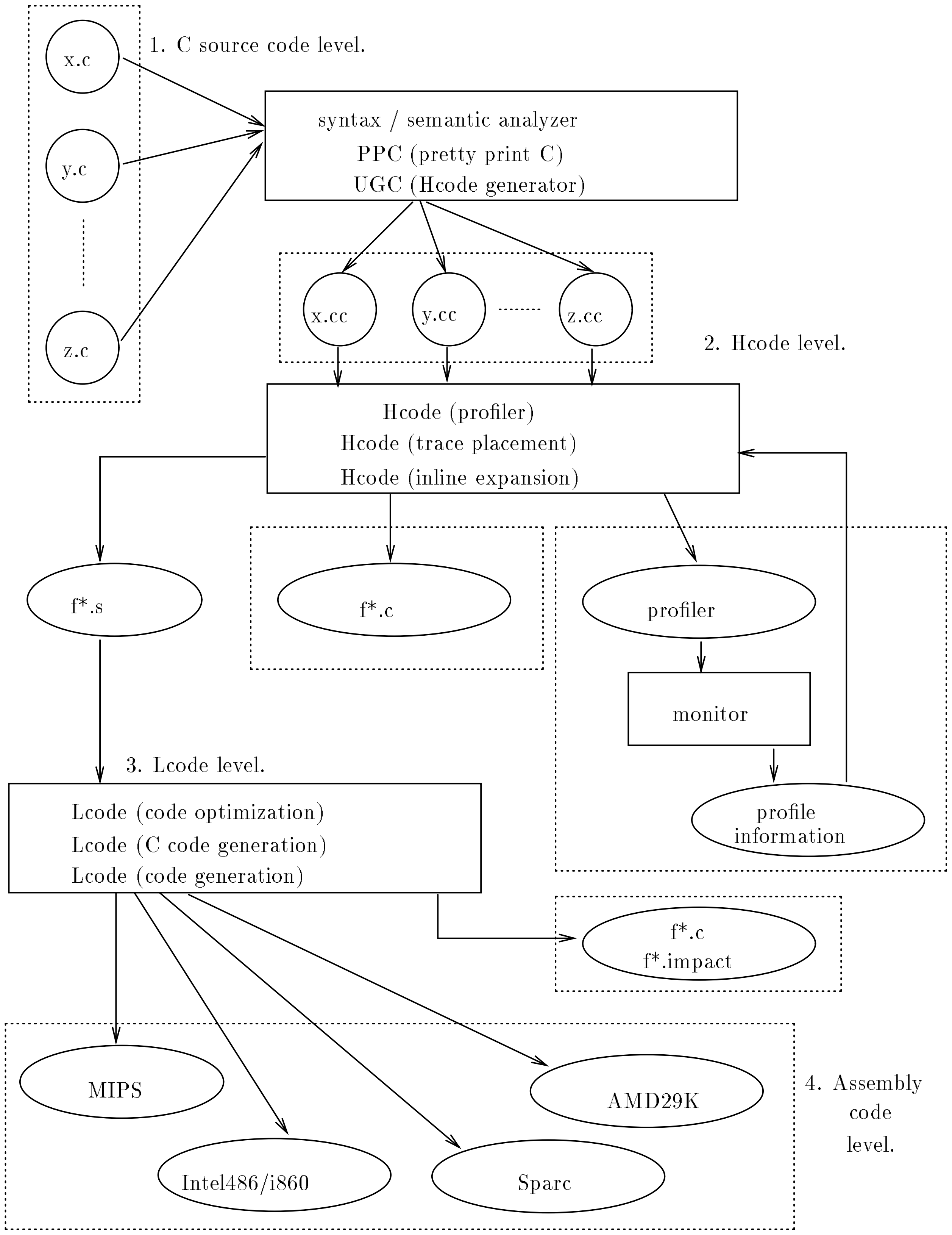

Figure 2.1 IMPACT-I Compiler Overview

2.1.1 Motivation

Some optimizations may always be performed with good results. The "traditional" optimization and peephole optimization are members of this group of techniques. Other optimizations incur some cost, such as code expansion, which must be weighed against their benefit. Procedure inlining, loop unrolling and trace expansion cause code expansion. The benefit of the transformation is seen in the shorter execution time of the transformed code.[2] Knowing how many times a code sequence is executed allows a compiler to make good choices in applying potentially costly code transformations.

Additionally, the profile information is useful for choosing the fastest dynamic code from two sequences of equal static cost. Some architectures include delayed branch instructions with the ability to conditionally annul the instruction in the delay slot. On machines with the ability to squash the delay instruction in either the taken or not taken direction, knowing the direction the branch usually goes at compile time allows the proper choice of branch instruction. Unequal execution time in branch-taken and the branch-not-taken path allows code reordering to arrange for the fastest dynamic instruction path in the average case.

2.1.2 Collection of profile information

The profile information is collected by the IMPACT-I C compiler as the first step in the compilation process. The C source code is transformed into the H-code, as described in the next section. Additional code is added to the H-code which will update a counter each time it is executed. The code is then transformed into instrumented C code which is then compiled with an ordinary C compiler. The instrumented program is then run on typical inputs, and profile data are collected. The results gathered for the program that are run on a number of input data sets are then merged into a single profile database. The merged data base is used in subsequent optimization phases.

[2] Excluding cache and memory effects of the larger code.

2.2 H-code Environment

H-code is a high level intermediate language and appears as a forest of semantically correct abstract syntax trees. It retains knowledge of the data and control structures of the original source code. After translation, the same memory references are made, and the same computations performed. The syntax of H-code, however, allows easier manipulation and transformation than is directly possible in C. The environment is used for inserting profiling probes, performing loop unrolling, procedure inlining, memory disambiguation, loop interchange, other control flow optimizations, H-code trace selection, and some traditional optimizations. These optimizations are explained in Chapter 3. Figure 2.2 is a schematic of the H-code environment.

2.3 L-code Environment

L-code is a low level intermediate language which resembles a generic machine language. The model of the machine includes an unbounded number of registers and the instructions shown in Appendix A.

Traditional compiler optimizations are performed on L-code. These include constant propagation, common subexpression elimination, dead code removal, loop induction variable removal and strength reduction, and L-code trace optimization. These optimizations are explained in Chapter 3. Afterwards, machine-dependent optimizations are performed. These include rearranging operands, propagating procedure call registers, instruction placement, constant preloading, and code scheduling. Figure 2.3 is a schematic of the L-code environment.

2.4 Code Generation

L-code is finally transformed into machine code. The work in this thesis involves using the Sparc architecture as the target. The work here includes mapping the L-code operations into instructions of the target machine, assigning registers used in L-code to those of the actual machine, adding code to spill and fill registers, adding code to save and restore registers, and performing some optimizations on the generated machine code. These are described in Chapter 5.

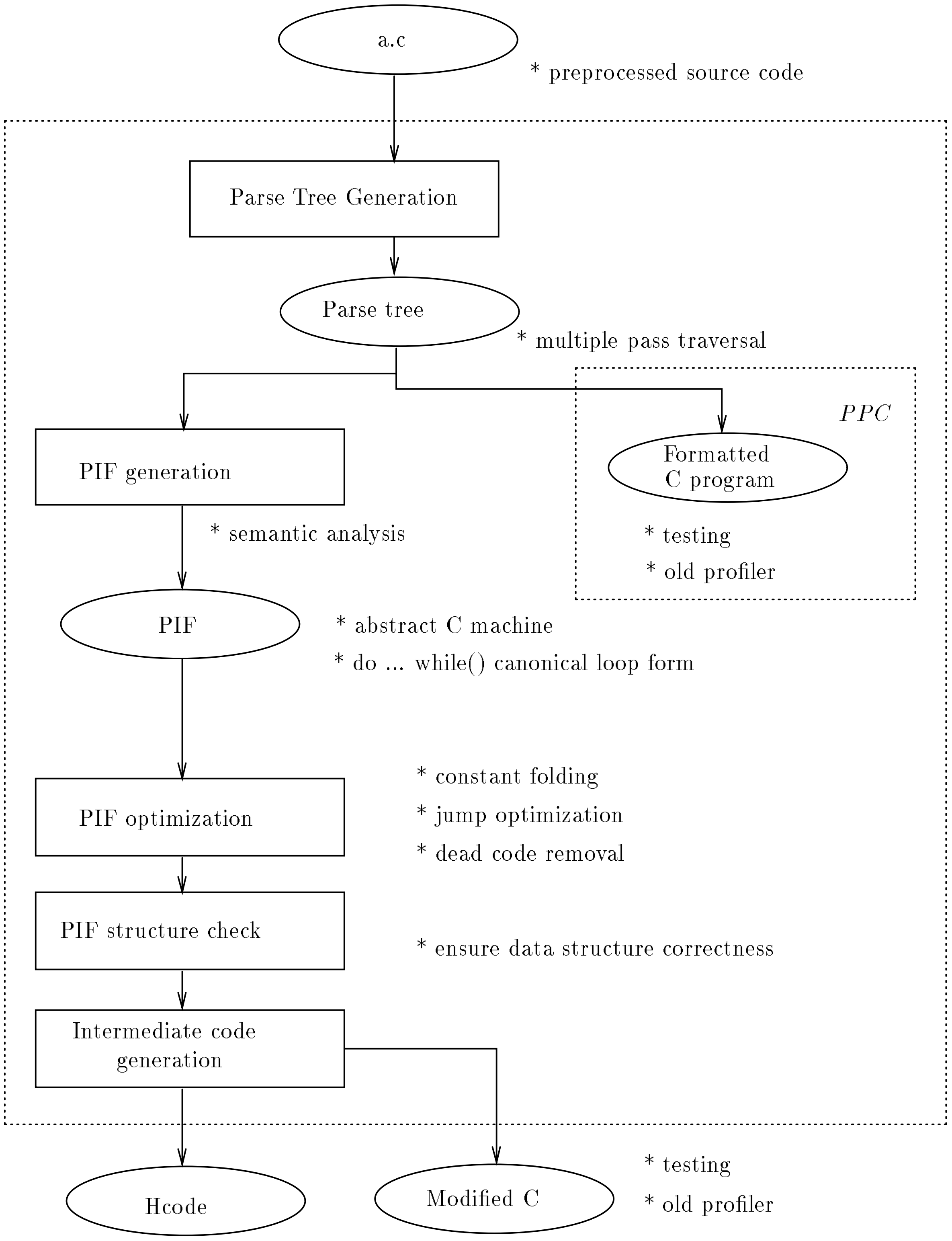

Figure 2.2 H-code Environment

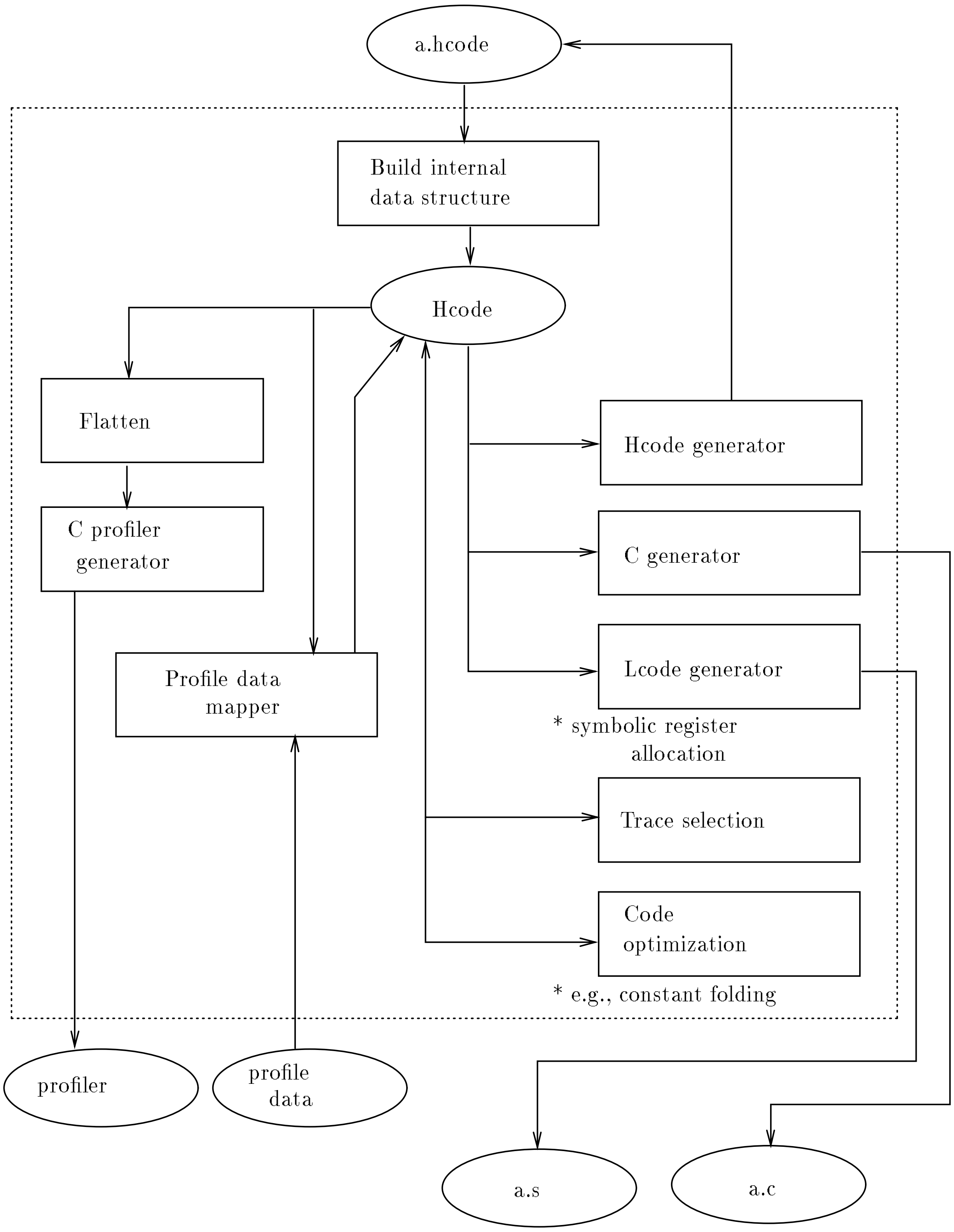

Figure 2.3 L-code Environment

2.5 Advantages of This Approach

The stand-alone programs and intermediate stages have enabled debugging of the compiler components. This is particularly useful in a research compiler which is usually in a state of flux. The fire-walls added and the facilitated inspection of intermediate results save development time. The use of a library of input and output functions for the intermediate code lowers the apparent cost of making each phase a program which stands on its own.

Conversion of data to and from printable representations makes the compilation process slower. Use of a binary file representation would speed compilations, but would be more complicated. A benefit of running in small self-contained stages is better use of memory.

The library I/O routines also resulted in a standard internal data representation. In the future, it might be possible to eliminate some or all of the stages, and to create a more efficient compiler, by simply using the standard data representations and translating between one representation and the next in memory.

The work here is to investigate the addition of a new back-end to the compiler. However, additional front-ends are also interesting. The source language does not necessarily have to be C. A C++ to H-code translator or a Fortran to H-translator is entirely possible.[3] Since other languages have differing semantics, additional H-code attributes might be necessary, but it seems likely that the same L-code could be used.

[3] The f2c Fortran to C source language translator may be used. However, since Fortran has more constraints, compiling it directly allows for further optimization.

CHAPTER 3

MACHINE-INDEPENDENT CODE OPTIMIZATIONS

This chapter contains descriptions of the machine-independent optimizations used in the IMPACT compiler. The optimizer phases transform either the H-code or L-code into equivalent, but faster versions of the same language. The H-code optimizations include function inline expansion, instruction placement and control flow optimization. The L-code optimizations include classical local and global optimizations and trace based optimizations.

3.1 Classical Optimizations

The traditional optimizations are generally classified as *global* and *local.* Both may be described by the substitution of a particular pattern of code with an equivalent, but faster sequence. The local group includes transformations which are applied within single basic blocks. Global optimizations are limited to a single function.

A list of the optimizations implemented by the L-code optimizer follows: constant propagation, copy propagation, memory copy propagation, operation combining, common subexpression elimination, dead code removal, constant folding, strength reduction, operation cancellation, code reordering, jump optimization, dead block elimination, loop invariant code removal, loop induction variable strength reduction, and loop induction variable elimination.

All of these optimizations are described and algorithms for their application are given in Aho Sethi and Ulman [1].

3.2 Inline Expansion

Use of procedural abstraction allows programmers to better divide a program into its constituent parts and reuse code. Unfortunately, a penalty is paid to execute a procedure call and return in compiled code. And further, the call site can inhibit many optimizations. Some processor architectures include features which are designed to lessen these costs, but incur a large cost in hardware instead.

Automatic procedure inlining is an effective alternative and has the added benefit of enabling further optimization. Procedures called with constant arguments are particularly good candidates for inline expansion. Often the constant argument is used to steer execution through the called routine, and so much of the inlined function becomes dead code which may be eliminated. Letting the register allocator work with large pieces of code causes better register assignments. Variable aliasing problems are reduced in some cases in which an inlined function has pointer type arguments. The need to save and restore registers around the call site is reduced. Procedure calls can inhibit loop optimizations such as loop unrolling. Function inlining within heavily weighted loops can be very beneficial.

There are certain hazards which have to be avoided when performing function inlining. The most obvious and important is code size expansion.[1] It is important to only inline functions at frequently executed call sites. Also, inlining creates new variables within a stack frame. If the inliner does not check, it is possible to create inefficiently large stack frames to hold all of those variables.

The order of function inlining is also important to consider. After the call graph has been developed, the IMPACT compiler chooses the sites to inline carefully. Considered are code size expansion, minimization of the number of dynamic calls, and the order in which to perform the inlining. The compile time cost of function inlining can vary greatly depending upon the ordering of the tasks. The IMPACT performs function inlining so as to keep the compile time acceptable short.

Certain call sites cannot be inlined: for instance library functions for which there is no source code. Functions called through pointer are difficult to inline as well. If a particular call through pointer is made rather frequently, the flow graph may be adjusted to add a test for equality and a direct call in the frequent case. The inliner can then do those cases in the usual way.

[1]Inlining a static function with a single caller results in a decrease in code size. The original function body may be deleted, because there are no longer any callers.

3.3 Instruction Placement

To achieve better performance from the memory subsystem, the IMPACT compiler transforms programs to achieve better locality and to reduce cache conflicts. The transformation is made in five steps. The first is collection of profile information. The second is function inlining, which improves spatial locality and reduces interfunction conflict. The next is trace selection, in which frequently executed sequences of basic blocks are duplicated and simplified to achieve better locality. The fourth is function layout, which reorders the traces of a program in decreasing order of weight. The last is global code layout which places functions which are executed close together in time close together in memory.

3.4 Control Flow Optimizations

Two control flow optimizations which the IMPACT compiler implements are described here.

Branches with multiple targets are generated by constructs like the C language `switch` statement. The optimizer sorts the target destinations by arc weight. In cases in which the execution time may be reduced by using simple tests and conditional branches, the computed jump is reduced to this form. Some cases may be best generated using a few tests for common cases and then a jump table. The IMPACT compiler does this. A jump table is used for the other cases.

Profile information enables optimization of two-way branches as well. Trace selection, by grouping frequently executed blocks into a sequence, reduces the number of taken branches. The profile information may be used to fill in a compile-time predicted direction bit on the instruction. Over ninety percent of dynamic branches were correctly predicted in the tests chosen [2].

3.5 Trace Copying Optimizations

To expand traces beyond just a few basic blocks, target basic blocks may be copied to the end of the trace of interest. The copied basic block will only be entered from the preceding basic block. The simpler control flow graph allows more aggressive optimization and scheduling. But to control code expansion, only frequently executed branches should be eliminated and their destinations copied inline.

3.6 Loop Unrolling

Using profile information, the IMPACT compiler can detect loops which are frequently executed. Loops that contain little computation which may be scheduled to execute while waiting for memory operations to finish may be improved by unrolling. The loop body is duplicated several times to create a new loop with adjusted copies of the controlling code. In addition to the scheduler benefit, the new code may be adjusted to contain fewer updates of the control variables. Loop unrolling has a cost in code expansion and also in register usage due to the additional temporary variables introduced. The unroller must be tuned to avoid creating loops that have more live variables than the target architecture has registers.

3.7 Loop Peeling

Loops which iterate a few times may benefit from an optimization such as loop unrolling called loop peeling. This optimization turns the first few iterations of a loop into straight-line code, leaving the original body of the loop intact. The sequential code may then be scheduled more aggressively.

CHAPTER 4

MACHINE-DEPENDENT CODE OPTIMIZATIONS

4.1 Instruction Selection

Most of the L-code instructions translate directly into a single machine code instruction. A few translate into several, and a few others are translated into run-time library calls. L-code can represent the change of machine state in a variety of ways, and knowledge of which instruction sequences are better than others may be exploited to improve the code.

For example, L-code contains a large number of branching operators, but some architectures only directly support a few. The machine specific optimization phase will adjust the computations and branches to avoid those which are not directly supported. See Section A.1 for a list of L-code branch operators.

Another example is integer multiplication and division on the original Sparc architecture. Run-time library support is used to support these operations. If one of the multiplicands or the divisor of one of these operations is known at compile time, replacing the call to the library function with a sequence of operations is better. These instructions may be further scheduled to reduce latency.

4.2 Operation Folding

Some of the operations in the Sparc architecture can perform two conceptual operations in one step. The load operations are such operations. They have two source arguments, a register argument and a literal value argument or two register arguments. The source arguments are added together to generate an effective memory address from which a value is loaded. A machine specific optimizer phase replaces pairs of additions and loads using a single source operand with a load using two source operands.

4.3 Guarded Operation

Some architectures include conditional move instructions. These instructions may be used to eliminate branch instructions. On architectures where these instructions are available, a machine specific optimization phase would have made the appropriate transformation. Sparc does not have conditional move instructions.

4.4 Constant Preloading

Some constants may be coded as part of a machine instruction. The integer computation instructions of the Sparc architecture can contain 13 bit literal values, and the %g0 register is always read as the constant zero. The Sparc architecture has instructions which may be used to generate integer constants. Pairs of these instructions may be used to develop thirty-two bit values, faster than a value may be loaded from memory. Some other architectures make it cheaper to load the constant from a table in memory.[1]

Floating point constants except zero must be loaded from memory on most architectures including Sparc. In order to avoid stalls, the values must be loaded or generated in advance of their use. If the procedure is small, and registers are available, constants may be generated or loaded as early as the beginning of the function. Alternatively, they may be scheduled just long enough before their use so that no stall is incurred. This works well unless the code is placed within a loop. For constants used within a loop, it is better to generate them in the loop preheader. The IMPACT compiler includes machine specific optimizations to load or generate constants at the right time.

[1]MIPS is an example. Alpha has sixty-four bit registers. Some sixty-four bit constants may be quickly generated using ldah, lda and shift instructions. Others are cheaper to load from memory.

4.5 Register Allocation

As a research instrument, the IMPACT compiler is used to generate code for hypothetical heavily pipelined VLIW and super-scalar machines. Conventional compilers perform register allocation prior to code scheduling, but this has the drawback of introducing data dependencies. The IMPACT compiler allocates registers after scheduling the L-Code. Constant preloading and aggressive scheduling lengthen the life time of the variables held in registers. Both constant preloading and code scheduling must be carefully tuned to avoid the need to spill registers. Once tuned, the IMPACT compiler rarely spilled registers with the Sparc back-end.

4.6 Code Scheduling

The IMPACT compiler uses the concept of a *super-block*. Frequently executed instruction traces are duplicated, creating a trace with no branches into the middle of the trace. This allows the scheduler to percolate *safe* operations above conditional branches within the trace. *Safe* operations are those which cannot cause an exception. For the Sparc architecture and C language, safe operations are all of the integer operations with the exception of division, modulo and remainder with unknown divisor. The initial implementation of the Sparc architecture had single-cycle integer instruction latency, so scheduling these was unimportant. Load instructions had two-cycle latency if done from cache. Loads are important to schedule, but since they may fault, they cannot be moved above conditional branches. The multiple instruction issue Sparc implementations which have become available since have longer effective instruction latencies and fairly stringent rules for issuing multiple instructions. Scheduling code for those machines is much more important.

CHAPTER 5

SPARC CODE GENERATION

The code generator is fairly simple. It runs several passes over the input code. Modifying it to run in fewer phases would be straightforward, but was unimportant to the research.

The first pass runs through all of the L-code tuples, assigning Sparc opcodes to the operations which can be done in one instruction. In that pass, those that require more than one machine instruction are expanded. This includes adding nops after branch instructions to fill their branch delay slots with an instruction; a later pass partially fills these slots with useful work. Literal operands which do not fit into the literal field of their computational instruction are expanded into a sequence, which generates the literal and then does the computation. Filling of the branch delay slot on `ret` is done after register allocation, when the register usage of the procedure is known.[1] Also, processing to reorder the the operands of some instructions is done in this step because certain operands can only be registers, and others may be registers or literals.

The next pass assigns registers using a register allocator utility. The register allocator used a graph-coloring algorithm based on one published by Chaitin [3]. The allocator maps the unbounded set used in the L-code representation onto the actual register set used by the machine. While writing the code generator, the register allocator was improved to make use of the argument registers as general registers after their end of life time and to spill the arguments to stack when that was beneficial. Variables not assigned to registers for their entire life time have spill and fill code generated around their use. Spill and fill code at procedure call sites is also generated.

[1]If the procedure is a leaf, does not need a stack frame and does not use too many registers, special prolog and epilog code sequences may be generated. That code omits the usual save and restore operations.

The next pass over the code expands the remaining operations. For some instructions it is less confusing in the previous stages if L-code operations not needing additional registers are not expanded until after the register allocation step.

The branch delay slot filler is very simple and only fills the slots in the following cases: If the instruction following the branch is a comparison instruction, that instruction is moved up into the delay slot. Otherwise, if the target of the branch is not a branch instruction, an instruction is copied from the target, the destination is adjusted downwards and the annul bit is set on the branch. If both fail, the nop is left in the delay slot. If the data flow information were still available, or if additional code had been written to regenerate it the delay slots could have filled from above in many cases. The results are better.[2]

The final pass then prints the assembler output to a file.

5.1 The Sun Optimizing Assembler

To deal with many of the code generator issues which arise, and so that the compiler need not parse and analyze the code within `asm` statements, many of the optimizations made by the bundled C compiler on the Sun OS 4.1.1 are made by the assembler. Examining the transformations made by the assembler was informative. We found that our suite of optimizations was missing some simple transformations described in Section 5.2.

The SUN assembler makes many of these transformations when paired with their C compiler. However, their assembler output contains many undocumented directives. Since I was unable to correctly insert these directives into my output assembly code, I was unable to determine how much these optimizations would have helped our code.

5.2 Need for Post Code Generation Optimization Phases

After all of the code is generated and set to emit, there are places in which the code may be improved. Most of these optimizations are in one way or another peephole optimizations.

[2] Also, instructions generating arguments could be moved into the delay slots of `jsr` instructions.

5.2.1 Conditional branches and condition codes

The most obviously missing optimization was using the condition code variants of the instructions to do branching. The L-code generated makes a Boolean register results, and uses them as the operands of a conditional branch instructions. Such code was carefully scheduled. A `subcc` instruction immediately before the branch was used to move that result into the condition code register. But this instruction may be removed by finding the instruction which generated the Boolean result and setting its CC bit. Unfortunately, that information was unavailable at code-generation time. An additional benefit would have been to free a register for other use by using the CC register to hold the temporary.

5.2.2 Scheduling register fill code

Scheduling the spill and fill code would have been very helpful. Spills and fills are done to the stack which usually results in cache hits. Nevertheless, scheduling the loads helps the uncommon cache miss case. The code generated would always stall in this case, since the very next instruction always used the result of the load.

Consecutive memory operations may be slow. Back to back stores can be made to have a usually low latency by use of a write buffer. But typical chip bus interface units can handle just one outstanding load at a time and stall on the next load. Exceptional bus interface units can handle a second outstanding load. Consecutive loads were occasionally generated due to fill code, but because the register allocator was rather good, this did not happen often. After the code is generated and L-code operations requiring multiple machine code instructions are expanded, more instructions are available for scheduling. It is then possible to improve the code by rescheduling the final code. Scheduling the loads used to fill registers is important, as is spacing the loads with computational instructions. Both would have helped to avoid pipeline stalls when cache misses occur.

5.2.3 Aligning loop tops

Better cache usage patterns can be obtained by aligning loop tops with the beginnings of cache lines. Adding the padding with nops must be done in a final scheduler or in a final post-pass when the exact machine instructions are known. Counterintuitively, these extra instructions help code to execute faster.

5.2.4 Scheduling for multiple instruction issue implementations

On chip implementations using multiple instruction issue logic, it is also important to schedule groups of instructions which may issue together. A final scheduler performs this task. All of the final scheduling tasks require data flow and register use information. That information may either be regenerated at this time, or be maintained from previous phases. Neither was done at the time, so it was not possible to accomplish even simple instruction scheduling.

CHAPTER 6

MEASUREMENTS

In order to test how well we fared in targeting the IMPACT compiler to the Sparc architecture, we selected a benchmark set of small and medium-sized programs. We wanted to include all of the C language programs from the SPEC benchmark suite. I was unable to test the gcc benchmark from the set, because I did not finish variable argument list support. We included the small toy benchmarks because we wanted to be able to hand inspect the results and see if there were any obvious deficiencies. The small benchmarks are also helpful when debugging new compiler code. The larger benchmarks are more representative of real user programs.

The cc compiler bundled with Sun OS 4.1.1 was used for comparison. The programs with the cc compiler used -O4 switch. Each program's elapsed execution was measured several times using the same input for both the cc compiled programs and the impcc compiled programs. The testing conditions were the same for all tests: a Sparc station SLC in multiuser mode, running X windows, with only the tester logged in. Although conditions were less than ideal, a fair comparison could be made. To help cope with the multiuser mode and varying random network traffic, the slowest run of the set was discarded and the rest were averaged. Table 6.1 contains the average runtimes.

All of the case in which the IMPACT compiler produced slower results can be explained by a final peephole optimization pass and final code scheduling pass made by the SUN cc compiler and not yet implemented in the IMPACT compiler. The code of the li benchmark case would have been helped by use of peephole optimization to remove the redundant subcc instructions from its many branches. The examples which show better results may be explained by the inlining and trace scheduling done by the IMPACT optimizer phases; eqntott, sort and espresso do particularly well despite the missing final optimization passes.

Table 6.1 Average Program Runtime (Seconds)

Benchmark	Sun cc	IMPACT
cccp	4.0	4.2
cmp	18.8	19.1
compress	36.3	38.0
eqn	20.4	20.4
eqntott	99.1	91.5
espresso	100.6	98.4
grep	12.8	12.5
lex	32.4	34.4
li	140.4	154.1
pic	103.8	113.4
sort	15.2	14.4
tbl	13.8	13.8
wc	12.3	12.2
yacc	4.9	5.0

CHAPTER 7

CONCLUSIONS

This work produced several useful results. The first task was to determine the actual size of the task. Six man-month's work was sufficient to produce an interesting code generator with which interesting research could be done. Production quality compilers take substantially more work to retarget.

When I began writing the code generator, we believed that little work beyond writing new code generators would be necessary to retarget the compiler. We knew that a few machine specific optimizations would be necessary and believed those could be written as an L-code to L-code transformation. We also thought that all machine specific scheduling could be done directly in L-code. We learned that the output of a naively written code generator is good, but may be further optimized with machine code to machine code transformations. See Section 5.2 for a list of additional optimization work to be done. This later led to M-code, a machine code level representation. Many of the machine code optimizations have since been implemented using M-code so that common optimizations may be shared among code generators.

Many problems with the Sparc architecture and the calling standard used were discovered. Use of a compiler with state of the art optimizations early in the design process exposes these problems early enough so that they may be fixed. The Sparc architecture was one of the first RISC architectures, and SUN has been extremely successful with it. But since its release, a great deal has been learned. The comparison with later architectures, such as the MIPS and Alpha architectures, highlight the defects of the Sparc architecture.

7.1 Problems with the Sparc Architecture

Listed below are the problem areas and plausible explanations of why the architecture included them.

7.1.1 Condition codes

Condition codes are a user visible bit of processor state that usually control conditional branch instructions. Previous architectures such as the VAX architecture set the condition codes as a side effect of most computational instructions. Subsequent conditional branch instructions would use the value of condition codes as an input.

The Sparc architecture improves upon this scheme by providing two versions of each computational instruction: one which sets the condition codes and one which leaves their state unchanged. This adds some flexibility in scheduling code, because the instruction which sets a condition code need not immediately precede its use in a branch instruction. This allows added parallelism on machines which can have multiple outstanding instructions. As noted in Section 5.2.1, the compiler did not take advantage of this flexibility, but it is a relatively straightforward thing to do. Modifications to the IMPACT compiler have since made data flow information available to the code generator in L-code. There is interest in adding this optimization to the Sparc code generator.

However, condition codes as an architectural feature make some architectural implementation details very difficult. The condition code register can become a heavily used bottleneck in a processor. That state is very difficult and expensive to implement in an architecture using multiple instruction issue.

A simple alternative is to use general processor registers for branching. Conditional branch instructions will then use a register operand in addition to the branch displacement. Using general registers requires an additional register read port and smaller branch displacements. An additional general register is also used to hold the temporary variable.

At the time the architecture was designed, use of condition codes may have been moderately advantageous, but they now greatly complicate multiple instruction issue implementations.

7.1.2 Branch delay slots

Use of a branch delay slot in an architecture can allow the processor architect to give the compiler writer the ability to explicitly add parallelism to machine code in a relatively simple way. While the next instruction to be executed is located, other work may be done. Published work shows that most of the time an instruction may be moved into a branch delay slot. The added compiler support necessary to fill the delay slots is not difficult to add if data flow information is preserved until code generation. To use the branch delay slot more frequently, an *annul* bit may be added to branch instructions. The bit causes the results of the instruction in the delay slot to be canceled if the branch instruction is not taken. Several commercially available architectures also use branch delay slots: Sparc, AMD29K, i860, and MIPS. Some, including Sparc, have annul bits in their branch instructions.

However, there is a problem with the way they scale. It is difficult to design a CPU implementation using multiple instruction issue and branch delay slots. If the branch instruction is located at the end of a block of instructions which issue together, it may be necessary to execute instructions from two different blocks. The instruction in the branch delay slot will be found at the next sequential location in memory, and the branch target will contain instructions which may also have to be executed. Instruction annulling of the branch delay slot further complicates the matter.

7.1.3 Register windows

The Sparc architecture has a feature called register windows. It implements a type of hardware stack of registers. Register windows are designed to reduce the cost of procedure call by saving and restoring groups of registers in a single instruction. The hypothesis is that typical code has many shallowly nested procedure calls. This is true of the Ackerman function benchmark compiled using a simple compiler, but is not true of programs in general, especially those compiled using modern optimizer technology including a well-tuned function inliner. Scientific Fortran codes in particular have few procedure calls.[1]

[1] The argument may be made that the functions of a program are spread across many files, and so cannot be inlined. This is not the case. The IMPACT compiler treated the whole program as a unit, and would inline functions across file boundaries. Digital Equipment Corporation's DEC C, DEC C++, DEC Fortran and perhaps other compilers can take as input many source files and create a single object, inlining functions where appropriate. Other vendors choose to do intermodule function inlining at link time, using an optimizing linker.

The architecture requires a designer to implement 120 registers, of which only 24 to 29 are available for use by the compiler writer. It had been thought that a compiler could seldom make use of as many as 24 registers, but better register allocator technology and longer instruction latency have shown that more registers are often useful. Processors using multiple instruction issue accentuate the need for additional registers. On such architectures, speculative instruction scheduling, further increases the demand for registers. It's unfortunate that more of the registers taking up space on the chip are not available to the compiler writer.[2]

The stack of register windows is conceptually unlimited, but is actually finite. When instructed to save or restore a register set, and one is not available, a trap is generated and run-time library code is executed to save or restore register sets on the stack. The save and restore routines contain many consecutive memory operations with little other work. The memory interface will limit performance. Fortunately, procedure calls may be made fairly rare.

Setjmp is an exceptional case for saving register windows. Because setjmp saves processor state, it is necessary for it to force the hidden register state to the stack and to save the current state into the jump buffer. This enables the run time library code to unwind the stack to the right place and correctly restart at the point of the setjmp. This makes setjmp an exceptionally slow operation.

7.2 Calling Standard Problems

The calling standard used in the SUN OS operating system on Sparc processors results in decreased performance. Changes to the calling standard may require users to recompile their programs with an updated compiler. Market forces make such changes difficult: simultaneous releases of new compilers would be needed, and independent software vendors might have to release new versions of their tools. Nevertheless, a calling standard is easier to modify than an instruction set architecture.

These are parts of the calling standard that could be improved.

[2]However each additional bit used to address the register file requires three bits in the instruction word. This may reduce the number of bits available within the instruction which may be used to hold literal values. Therefore, more instructions may be required to create some constant values. Loading constants from a global table may be cheaper on such machines.

7.2.1 Lack of null frame procedures

Small leaf routines which do not require more than seven registers and require no stack space may avoid using the `save` instruction in its call sequence. This avoids the possible penalty of a register window trap.

It would be advantageous to make the same optimization on small routines which call other functions but are not candidates for inlining or being inlined into. In compiling the routines of a module, it is best to first compile the routines which are deepest in the call graph. This allows their register usage to be known when their callers are compiled. Then if a routine only makes calls to functions whose register usage is known, and a temporary register is available for use as a saved register, the call frame of the procedure is unnecessary.[3] The return address may be moved to the register identified, computation and calls performed, and the return address restored.

In order to make debugger, run-time library and operating system code work with null stack frames, additional procedure descriptor information has to be generated. Such a change requires a calling standard modification.

7.2.2 Lack of preserved floating point registers

There are no floating point callee save registers. At every call site, all live floating point registers must be spilled. This makes floating point code run much slower than necessary.[4]

7.2.3 Passing floating point arguments

The calling standard requires that the first arguments passed in a function call be placed in integer registers. If the argument is a floating point value, an extra store to the stack and an extra load from the stack for each such argument must be done by the caller and the callee.

[3] Digital Equipment Corporation's DEC C, DEC C++ and DEC Fortran compilers implement a #pragma linkage directive. This linkage describes the register usage of a function. The compiler will take advantage of extra saved registers when calling these functions. Further, the compiler will enforce the register usage when compiling the function body.

[4] The #pragma linkage directive described above is especially helpful with numerical code which makes calls to math runtime library functions.

7.3 Cache Design of the Sparcstation I Family

Memory trace information and especially multiprocessor memory trace information are extremely valuable in designing new architectures and implementations of those architectures. Creating these traces is difficult and expensive and gives the holder a competitive advantage. The best traces are generated by large computer manufactures, who guard them closely as proprietary information. Seldom do vendors publish their traces.

The only traces widely available are those published by Eggers for the PDP-11, a machine whose address space is much smaller than current machines. It appears probable that these were used to design the original Sparc, like its predecessors at Berkeley, and because of their unrealisticly small use of memory led to under-designed cache and memory systems. The initial implementations were inexpensive, but did not perform especially well when running programs which had large data sets. Later machines used larger caches to alleviate this problem. Such implementation details may be changed easily, in contrast to calling standards and instruction set architectures.

APPENDIX A

L-CODE OPERATORS

This appendix lists the operators used in L-code. Each has up to three source operands and one destination operand.

A.1 L-Code Branch Operators

jump jump_fs jump_rg jump_rg_fs

beq bne bgt bge blt ble

beq_fs bne_fs bgt_fs bge_fs blt_fs ble_fs

bgt_u bge_u blt_u ble_u

bgt_u_fs bge_u_fs blt_u_fs ble_u_fs

beq_f bne_f bgt_f bge_f blt_f ble_f

beq_f_fs bne_f_fs bgt_f_fs bge_f_fs blt_f_fs ble_f_fs

beq_f2 bne_f2 bgt_f2 bge_f2 blt_f2 ble_f2

beq_f2_fs bne_f2_fs bgt_f2_fs bge_f2_fs blt_f2_fs ble_f2_fs

A.2 L-Code Computation Operators

no_op

mov mov_f mov_f2 gd_mov gd_mov_f gd_mov_f2

add add_u sub sub_u mul mul_u div div_u rem rem_u

abs

or and xor nor nand nxor or_not and_not

eq ne gt ge lt le gt_u ge_u lt_u le_u

lsl lsr asr

rev bit_pos

add_f sub_f mul_f div_f

abs_f

eq_f ne_f gt_f ge_f lt_f le_f

add_f2 sub_f2 mul_f2 div_f2

abs_f2

eq_f2 ne_f2 gt_f2 ge_f2 lt_f2 le_f2

f2_i i_f2 f_i i_f f2_f f_f2

A.3 L-Code Memory Operators

ld_uc ld_c ld_uc2 ld_c2 ld_i ld_f ld_f2

st_c st_c2 st_i st_f st_f2

fetch_and_add fetch_and_or fetch_and_and fetch_and_st fetch_and_cond_st

co_proc

expand

A.4 Other L-Code Operators

jsr rts prologue epilogue define alloc

APPENDIX B

THE SUN OPTIMIZING ASSEMBLER

The assembler supplied by SUN could perform a number of interesting optimizations. They are described below. They are simple post code generation peephole optimizations. Experiments with them showed how the IMPACT compiler needed to implement a similar set of optimizations.

The B switch allows the compiler to remove some useless instructions after unconditional branches. In some cases, many instructions were removed resulting in incorrect code.

The G switch rearranges some of the basic blocks and fills some delay slots differently.

The K switch enables a jump peephole optimization which transforms sequences with this pattern:

```
cond_br       label
nop
b             label1
delay_instr1
label:
b             label2
delay_instr2
```

into this more efficient sequence

```
other_cond_branch label2
delay_instr2
label:
b                 label1
delay_instr1
```

The L switch appears to be needed to enable other optimizations and works with short case statements.

The M switch enables a loop transformation.

The P switch transforms sequences matching this pattern:

```
Cond_br,a
instr1
instr1
```

into this more efficient sequence

```
Br
instr1
```

The Q switch reduces register aliasing and renames some to reduce register usage. Perhaps is done with an extended peephole optimization. It does not appear that registers were globally reallocated.

The R switch changes this pattern:

```
sethi    %hi(const), reg
or       reg, %lo(const), reg
instr    [reg], dest
```

into this equivalent but more efficient form

```
sethi    %hi(const), reg
instr    [reg+%lo(cont)], dest
```

Perhaps R caused many of the register assignments to change. Perhaps this enables a back-end register allocator.

The S switch enables filling of branch delay slots.

The T switch enables renaming of temporary registers, that is, registers which were defined once and used once. The optimization occurs when the source register used in computing the temporary is also unused.

The U switch seems to control further register renaming.

Some of the directives used by the assembler appeared to indicate the beginnings and ends of register lifetimes. It seems reasonable that not supplying this information caused the problems encountered in assembling IMPACT generated assembler with optimizations enabled.

APPENDIX C

REGISTER AND MEMORY USAGE OF RTL SUPPORT FUNCTIONS

The first Sparc processor implementation included no integer divide, remainder or multiply instructions. The machine instead supported a multiply step instruction which could multiply a 32 bit number by a 13 bit number. Code for multiplications, divisions and remainder operations is part of the run-time library and uses the multiply step instruction. A multiply takes about 45 instructions. Divisions and remainders take much longer. The Sparc architecture was later amended to contain an integer multiply operation.

The run-time library operations supporting multiply, divide and remainder were hand coded to be as efficient as possible. They performed all of their work in the six argument registers and used no stack space, so they were leaf routines. Additionally they access no memory.

This made them very special functions. Ordinary function calls inhibit optimizations, especially function calls within loops. A machine specific optimization for Sparc could exploit the special information about multiply function calls. Some adjustment of the register allocator would also be needed. The register allocator in use at the time of this work also used argument registers as temporary variables, but not across function calls. In contrast to ordinary function calls, the global registers are not changed by calls to these support functions. The register allocator could be improved to take advantage of this special information.

APPENDIX D

REFERENCES NOT CITED

The IMPACT compiler has been used by our group many times in published research. As a component in a large group project, the citations given below help to place the work done in this thesis into its proper context.

Reference [4] is a detailed description of the IMPACT compiler framework.

Reference [5] shows how use profile information may be used to make traditional code optimizations more effective.

Reference [6] is a technical report containing a more thorough treatment of material of [5].

Reference [7] describes control flow optimizations which the IMPACT compiler also used.

Reference [8] is a technical report showing the advantages of scheduling code prior to register allocation.

Reference [9] shows the advantages of scheduling superblocks especially on superpipelined superscalar processors.

Reference [10] shows the importance of function inlining in compiling C programs.

Reference [11] shows how instruction placement may be improved after function inlining has been performed.

Reference [12] is a later version of the report [11] published as an article.

Reference [13] describes some of the early work of Po-Hua Chang and Wen-Mei Hwu applying trace selection to large C programs. The trace scheduling technology was later incorporated into the IMPACT-I compiler.

Reference [14] shows how compiler technology may be used to improve performance by improving instruction cache use.

Reference [15] is the doctoral dissertation of Po-hua Chang who wrote much of the original code of the IMPACT-I compiler. That compiler generated code for a simulated VLIW architecture computer.

Reference [16] is the M.S. thesis of Scott Mahlke who wrote several of the optimizers used in the IMPACT compiler.

Reference [17] is the M.S. thesis William Chen wrote a code generator for the MIPS architecture for use with the IMPACT compiler.

Reference [18] is a technical report on the topic of code expanding optimizations and the added requirements to be met by instruction cache logic.

Reference [19] examines the performance problems of machines that use multiple instruction issue architectures, but have a limited number of registers.

Reference [20] uses the IMPACT compiler to compare results of static and dynamic code scheduling on processors using multiple instruction issue architectures.

Reference [21] examines the value of compiler assisted prefetch of data and its impact upon performance and memory interface design.

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[2] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[3] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, June 1982.

[4] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[5] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[6] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," Tech. Rep. CRHC-91-12, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, April 1991.

[7] P. P. Chang and W. W. Hwu, "Control flow optimization for supercomputer scalar processing," in *Proceedings of the 1989 International Conference on Supercomputing*, July 1989.

[8] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Tech. Rep. CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[9] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three superblock scheduling models for superscalar and superpipelined processors," Tech. Rep. CRHC-91-25, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, October 1991.

[10] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic C programs," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[11] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," Tech. Rep. CSG-123, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1990.

[12] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," *IEEE Transactions on Computers*, accepted for publication.

[13] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," in *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, November 1988.

[14] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 242–251, May 1989.

[15] P. Chang, *Compiler Support for Multiple Instruction Issue Architectures.* Ph. D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[16] S. A. Mahlke, N. J. Warter, W. Y. Chen, P. P. Chang, and W. W. Hwu, "The effect of compiler optimizations on available parallelism in scalar programs," in *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 142–145, August 1991.

[17] W. Y. Chen, "An optimizing compiler code generator: A platform for risc performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[18] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The effect of code expanding optimizations on instruction cache design," Tech. Rep. CRHC-91-17, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[19] S. A. Mahlke, W. Y. Chen, P. P. Chang, and W. W. Hwu, "Scalar program performance on multiple-instruction-issue processors with a limited number of registers," in *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, January 1992.

[20] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 25–33, November 1991.

[21] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 69–73, November 1991.