THE IMPACT SC140 CODE GENERATOR

BY

CHRISTOPHER JOHN SHANNON

B.S.,University of Illinois at Urbana-Champaign, 2000

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, 2002

Urbana, Illinois

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for giving me the opportunity to do research in such an outstanding group. The undergraduate research that I conducted with IMPACT inspired me to stay for graduate school and allowed me to grow as an engineer. I would like to thank the many members of the IMPACT group, past and present, whose work built the framework from which this thesis was developed. Special thanks are due to John Sias, who became my main sounding board for both IMPACT and general computer architecture questions. Without his countless hours of advice and assistance, this work would not have been possible. I would also like to thank Matthew Merten for his early mentoring and continued support, especially with regards to the modulo scheduler. Several other members of the research group have provided many valuable insights into my work, particularly Ron Barnes with regard to the machine description file and Erik Nystrom with regard to the front end of the compiler.

In addition, I would like to thank all the people at StarCore, especially Roger Bringmann. The compiler group has provided me with all the software and manuals needed to develop this code generator. They continue to provide me answers to my many questions

about the intricacies of the SC140. The summer internship I spent there was invaluable and became the basis for this work.

Finally, I would like to thank my family and friends for their continued support throughout this process. Most importantly, I would like to thank my fiancée, Amy, for her boundless love and compassion. She has continued to encourage me in all my endeavors, despite our distance apart. This work could not have been completed without her devotion and understanding.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

The performance characteristics of embedded processors can differ greatly from their general-purpose counterparts. Embedded architectures are usually designed to run computationally intensive applications efficiently while minimizing code size and power consumption. Many of the actual chips need to meet certain area or power specifications, and almost all need to be manufactured at very low cost. Since these processors generally run a small set of specific applications, the architecture can easily be customized for these programs. Also, applications are often handwritten in the target machine's assembly code, instead of a high-level language, to achieve the best performance possible. General-purpose processors, however, need to efficiently run a very diverse set of applications, almost always written in a high-level language. In order to maximize the performance across all these applications, execution is parallelized as much as possible.

As embedded applications have become more complex and diverse, embedded processor cores today incorporate some features traditionally found only in general-purpose machines. The generation of embedded processors currently on the market is focused on wide-issue design and increased compilability. Part of this trend is driven by the fact that many companies do not have the resources required to hand-code all of their applications, so they instead compile from a high-level language. This places the burden of achieving the desired level of performance on a compiler, instead of a programmer.

This thesis documents the adaptation of an advanced, general-purpose, instruction-level parallelism (ILP) uncovering compiler to generate code for an embedded digital signal processing (DSP) core. The IMPACT compiler [1] offers many advanced ILP features in a research-inspired infrastructure. The StarCore SC140 is a wide-issue DSP core capable of capitalizing on large amounts of parallelism as exposed by the compiler. Creating a code generator that targets the SC140 presented many unique challenges due to the core's design.

This code generator was then used to quantify the effectiveness of several ILP-inspired optimizations in the DSP domain. A suite of optimizations was run on several benchmarks to determine their effectiveness. The amount of unrolling that was performed on loops and its side effects were analyzed. Also, the benefit of modulo scheduling was quantified, with direct comparisons to unrolled code.

The code generator was also used to explore some architectural changes to the SC140. An analysis of the usefulness of an increased number of predicate registers was performed. Also, experiments were run with various configurations of resources and grouping restrictions.

Experiments run on the current SC140 configuration were run through a simulator and produced instruction accurate results. Performance of benchmarks on enhanced hardware configurations was estimated statically by the compiler using basic block profile weights and accurate instruction latencies.

This thesis is organized as follows. Chapter 2 describes the SC140 core architecture, focusing on features relevant to code generation. Chapter 3 gives a comprehensive

description of the IMPACT compilation framework, providing examples of changes required by the code generator. Chapter 4 provides details of the code generation module itself. Chapter 5 describes several aspects of the SC140 architecture that could not be represented in the IMPACT machine description and proposes potential solutions to deal with these problems. Chapter 6 details the performance trade-offs that were explored using the code generator. The experimental results are found in Chapter 7. Some future directions and unimplemented features are discussed in Chapter 8. Finally, Chapter 9 contains some concluding remarks.

CHAPTER 2

THE STARCORE SC140

The StarCore SC140 DSP core, pictured in Figure 2.1, is the first implementation of the StarCore SC100 architecture family. The target markets for this chip range from wireless handsets and base stations to network and data communication. This chip is designed to support many different system-on-a-chip configurations, potentially including more than one core. The SC140 can interface with an on-chip memory subsystem at full core frequency.

The SC140 architecture differentiates itself in the DSP market by offering a high level of code compilability. The SC140 provides a rich, orthogonal 16-bit instruction set with major portions devoted to control code. This, along with a single data space and byte addressability, enables a compiler to generate efficient code for the chip. This means that DSP applications and kernels can easily be developed in the C programming language. The SC140 also allows up to six instructions, occupying up to eight instruction words, to be executed each cycle using a variable-length execution set (VLES) model. Since each VLES is defined statically by the compiler, similar to a very long instruction word (VLIW) model, this programming model fits well with an ILP compiler.

In the following sections, the SC140's core architecture and instruction grouping mechanism will be introduced. A more detailed explanation of the SC140 can be found in the core reference manual [2].

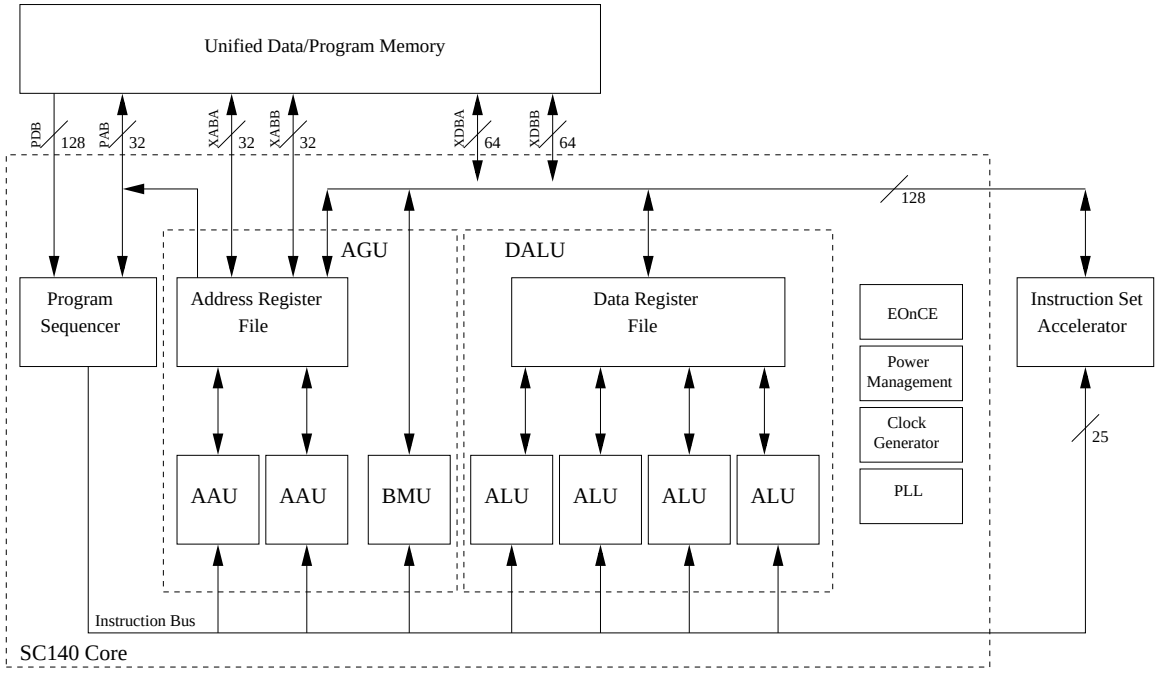


Figure 2.1: SC140 block diagram.

2.1 Core Architecture

The SC140 core provides several unique architectural features that need to be introduced here. Both integer and fractional data types are supported in the hardware in order to give the programmer flexability in choosing a style of code development. Fractional data types are activated by programmer-inserted directives, either directly in the assembly or in C code intrinsics. The SC140 separates computation into two distinct units, each unit containing its own register file and several computational units.

The first is called the data arithmetic and logic unit (DALU) and performs arithmetic and logical operations on data operands. The DALU is explained further in Section 2.1.1.

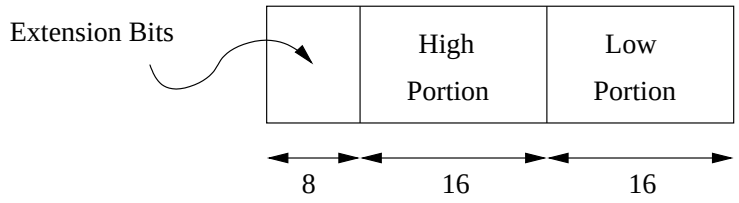


Figure 2.2: Data register layout.

The second is called the address generation unit (AGU) and performs address calculations. The AGU is explained further in Section 2.1.2. Both register files can contain integer data, and equivalent operations can be performed in both units. This puts a greater emphasis on the compiler's instruction selection algorithm as the location of the source operands can determine the actual instruction to be used. Specific examples of this will be shown in Chapter 4.

2.1.1 DALU

The DALU contains a register file of sixteen 40-bit data registers and four parallel arithmetic logic units (ALUs). Each ALU contains a multiply-accumulate (MAC) unit and a bit-field unit (BFU). All computation in the DALU occurs on 40-bit data, in the form shown in Figure 2.2. The extension bits in each data register are mainly used to prevent register overflow. When a register overflows its maximum value, an overflow bit is set on the machine. Special instructions can then limit the register value to its maximum value when it is written to memory. When this limiting occurs, only the stored value will change, with the register contents remaining intact. Accesses to the data registers can be in widths of 8, 16, 32, or 40 bits, depending on the instruction.

The MAC unit is used as the main arithmetic processing unit of the SC140 and contains both a multiplier and an adder. The multiplier executes 16-bit fractional or integer multiplication between two's complement signed, unsigned, or mixed operands. The 32-bit product is right justified, sign extended, and may optionally be added to a data register. The MAC unit can also be used to perform rounding, saturation, comparisons, and shifting.

The existence of these MAC units shows the emphasis placed on computationally intensive DSP algorithms. MAC instructions are essential to achieving good performance in DSP programs, but are not found directly in the C programming language. Without programmer-inserted calls, it is the compiler's responsibility to find instruction sequences that can be converted to MAC instructions.

The BFU is the logic part of the ALU. It contains a 40-bit parallel bidirectional shifter, a mask generation unit, and a logic unit. The BFU is used for shifts, rotates, logical operations, and sign/zero extensions.

2.1.2 AGU

The AGU contains a register file of sixteen 32-bit address registers, two address arithmetic units (AAUs), and a bit mask unit (BMU). Each AAU has two 32-bit full adders that are used for both address and general computation. Explicit instructions are used to perform arithmetic operations on the address registers. In addition, the AGU generates change-of-flow program addresses and updates the stack pointers as needed.

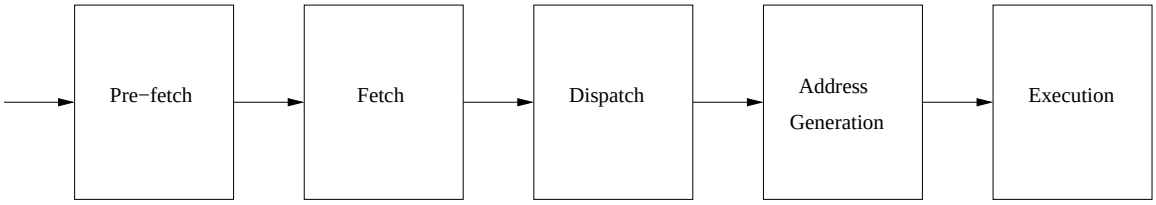


Figure 2.3: Instruction pipeline stages.

The most important function of the AGU is performing moves to and from memory. These instructions can each access 8, 16, 32, or 64 bits of memory and operate on signed integer, unsigned integer, or fractional data.

2.2 Instruction Pipeline

The SC140 employs a five-stage pipeline, shown in Figure 2.3. The first three stages, pre-fetch, fetch, and dispatch, occur in the program sequencer unit. The address generation and execution stages occur in the AGU and DALU, respectively.

An important aspect of the instruction pipeline is that AGU and DALU instructions are executed in different stages. This is necessary to achieve a single cycle memory access. The address of a memory access is first computed in the AGU. This address can be in any number of forms, including a stack offset, a register value, a constant, or some combination of registers and offsets. Once this address is calculated, the actual interface to the memory hierarchy will occur in the execution stage. An overview of each pipeline stage's responsibilities is shown in Table 2.1.

Another important aspect of the instruction pipeline is that the program address calculation for change-of-flow instructions occurs in the address generation stage. This

Table 2.1: Overview of pipeline stages.

Pipeline Stage	Description
Pre-fetch	- Generate addresses for program fetch - Update fetch counter (FC)
Fetch	- Read fetch set from memory
Dispatch	- Dispatch instructions - Decode AGU instructions
Address Generation	- Decode DALU instructions - Generate addresses for data load and store operations - Perform address calculations: normal and change-of-flow - Perform AGU arithmetic instructions - Update AGU registers
Execution	- Read source operands to DALU - Read source register for memory store operations - Perform data calculations (multiply and add) - Write DALU results to destination registers - Write destination register for memory load operations

fact directly determines the branch penalty that occurs on the SC140. Since there is no branch prediction hardware in the pipeline, instructions after a branch are always fetched from the fall-through path of the instruction stream. When a branch is not taken, the pipeline does not need to be flushed as valid instructions are ready to be dispatched. When a branch is taken, however, all previous stages in the pipeline contain invalid instructions. This results in a four-cycle delay for all taken branches. The target address of an unconditional jump or jump-to-subroutine (JSR) instruction can be determined immediately after the instruction is decoded, resulting in a three-cycle penalty.

This pipeline relationship between the AGU and DALU imposes some instruction sequencing restrictions on the compiler. For example, an AGU instruction cannot directly follow a memory load that produces one of its source operands. This is because the second

instruction would need its source available at the same time that the memory is being accessed.

2.3 Hardware Looping

An essential element in the performance of a DSP algorithm is efficient loop execution. To achieve this performance, the SC140 has a fully optimized looping mechanism that allows up to four levels of loop nesting. This mechanism uses a set of specialized registers to determine at run-time the iteration count of a loop. Assembly language programming directives provide the mechanism with the loop's boundaries. Using this information, the loop-back branch and its four-cycle latency are completely eliminated.

Loops on the SC140 are classified in two ways, short and long loops, and are marked differently in the assembly code. A short loop has a body containing no more than two execution sets, with each set being a VLES. These sets are stored in a buffer during execution of the loop, eliminating the need to fetch them in each iteration. A long loop has a body of three or more execution sets. For these loops, the iteration count is checked while the third-to-last set is being executing. This way the beginning of the loop can be fetched without incurring any timing penalty when the loop-back should be taken.

2.4 Conditional Execution

The SC140 provides for conditional execution of its entire instruction set. This is achieved through a single bit that resides in a status register. This flag is referred to as the true bit (T-bit). Several instructions implicitly modify the T-bit. For example,

the compare-for-equal (CMPEQ) instruction compares its two source operands. If their values are equal, it will set the T-bit to one. If they are not equal, the T-bit is set to zero.

Complementing these comparison instructions are several operations that implicitly read from the T-bit. These instructions execute only if the T-bit contains a certain value. For example, the JF (jump if false) instruction only jumps to a target location if the T-bit is "false," or zero.

There are also a handful of special program-control identifiers that can cause any given instruction to execute conditionally. These identifiers are IFT, IFF, and IFA. When one these identifiers precedes an instruction, the T-bit is checked to determine if execution should actually occur. If the condition is not met, the instruction becomes a no-operation (NOP). A given instruction will only execute if the T-bit is 1 for IFT and 0 for IFF. An IFA instruction will execute unconditionally. The existence of these specialized identifiers is essential for doing predication and the importance of these identifiers will be further explored in Section 2.5.2.

2.5 Instruction Grouping

In order to utilize all of the SC140's resources in every clock cycle, the compiler must parallelize the program effectively using the VLES structure. The SC140 can execute up to six instructions per VLES, with four in the DALU and two in the AGU.

The actual instruction grouping is provided through the use of prefix words. These words will be further introduced in Section 2.5.1. A VLES can also be broken up into

subgroups of instructions. These subgroups are an important concept when dealing with conditional execution. An introduction to subgrouping is provided in Section 2.5.2.

2.5.1 Prefix words

There are two ways that grouping information can be encoded in the SC140: serial and prefix grouping. In either of these formats, the dispatch unit can easily determine which instructions are to issued in each cycle.

The serial grouping method uses the two most significant (MS) bits in the instruction in order to encode the execution set boundary. A value of 00 in the two MS bits of an instruction word indicates that this word is to be grouped along with the next instruction word. Any other value in the two MS bits indicates that this is the last instruction in the execution set. Two bits are used here because the number of instructions that need to use these prefix bits for valid data exceed the number of instructions that do not. For this reason, three-fourths of the encoding space contains instructions that indicate an end of grouping. If only one bit was used instead, only half of the encoding space could be used for these instructions.

The drawback of this grouping method is that only a subset of the SC140 instructions can use this serial encoding. If there is no room for these extra two bits in an instruction's encoding, then this instruction must mark the end of a VLES. Thus, the combination of instructions used in a VLES can severely limit its length.

The prefix grouping method uses a one- or two-word prefix placed at the beginning of an execution set to convey all of the grouping information. Since these prefixes are either 16 or 32 bits long, other useful information is encoded here. In addition to the

number of instructions in the execution set, the prefix also contains hardware looping and conditional execution information.

Another piece of information contained in the prefix words is the encodings for the high register banks. On the SC140, only the bottom eight registers from both the DALU and AGU are available to all instructions without penalty. If any of the upper eight registers are used, a two-word prefix is needed to encode the additional bits.

The drawback of this grouping method is that the prefix words take up part of the fetch bandwidth. Thus, when using a two-word prefix, only six more words can be fetched to occupy the VLES. This becomes a limiting factor when multiple-word instructions are being used. Also, since the prefix contains a large amount of information, for any given VLES several bits will likely not be utilized. For example, if a high register is accessed in a VLES, a two-word prefix will be used regardless of the presence of hardware looping and conditional execution.

2.5.2 Subgrouping

The SC140 also allows for the conditional execution of a subgroup of instructions in an execution set. A subgroup represents the instructions within an execution set that are predicated on the same condition. Only two subgroups can be specified in a given VLES, and they are specified with combinations of the IFT, IFF, and IFA identifiers. Due to this restriction on the number of subgroups, a VLES cannot contain all three identifiers simultaneously. Oftentimes in predicated regions, though, it may be beneficial to use all three identifiers in a single clock cycle. This restriction will cause the compiler to break up the execution sets to include only two of the subgroups.

Table 2.2: Condtional execution options for subgroups.

Assembly Syntax	**Meaning**
[IFA] inst [insts]	Unconditional execution of the set
IFT inst [insts] IFF inst [insts]	Execution of subgroup1 if T-bit == 1 Execution of subgroup2 if T-bit == 0
IFT inst [insts]	Execution of the whole set if T-bit == 1
IFF inst [insts]	Execution of the whole set if T-bit == 0
IFT inst [insts] IFA inst [insts]	Execution of subgroup1 if T-bit == 1 Always execute subgroup2
IFF inst [insts] IFA inst [insts]	Execution of subgroup2 if T-bit == 0 Always execute subgroup2

In an execution set using two subgroups, two basic restrictions must be followed. First, each subgroup can contain up to two DALU instructions and 1 AGU instruction. Second, if the IFA subgroup is used, it must be the second subgroup in the set. A summary of legal conditional execution options for subgroups is found in Table 2.2.

2.6 Compilability

Several features of the SC140 architecture lend themselves particularly to efficient compilability. The VLES structure, although a variant of the VLIW structure, is a well-understood programming technique. The VLES allows for the compiler to easily determine an efficient schedule by understanding the resources and grouping options available. Since the VLES can allow up to six instructions to be executed in parallel, an ILP compiler does well to uncover the parallelism needed to utilize the entire width of the machine. The main factor deterring parallelism on this machine is the number

of general-purpose registers available. In practice, the fact that only 32 registers are provided on the SC140 limits the amount of parallelism that can be uncovered.

While the lack of branch prediction can be a deterrent to performance, the use of simple predication alleviates some of this problem. Although a larger number of predicates would help in hyperblock formation, the T-bit does allow for simple optimizations to be performed. While subgrouping is an efficient means of incorporating predication into the architecture, the restrictions placed on the number and order of these subgroups detract from the ability of the scheduler to find sufficient parallelism in many cases.

CHAPTER 3

THE IMPACT C COMPILER

The IMPACT compiler is a retargetable optimizing C compiler developed at the University of Illinois and is designed to serve two important purposes. First, IMPACT is an easily retargetable compiler that provides a large number of basic and ILP-based optimizations. Second, IMPACT provides an extensible framework for the easy implementation of new optimizations and transformations.

In order to facilitate the evaluation of a diverse set of compilation techniques, the IMPACT compiler is designed to be very modular. The compiler is essentially broken up into several stages, with distinct input and output files. These intermediate files are interfaced with IMPACT's modules by a series of shell scripts. A block diagram of the entire compilation process is shown in Figure 3.1.

At a high level, the IMPACT compiler is divided primarily into two sections, distinguished by the form of intermediate representation (IR) used in each. The level of IR closest to the source code is called *Pcode* and is used exclusively in the front end of the compiler. The lower level of IR is referred to as *Lcode* and is used exclusively in the back end of the compiler.

IMPACT's compilation phases will be further discussed in the following sections. Section 3.1 discusses the front end and Pcode transformation phases of the compiler. Section 3.2 discusses the Lcode optimization and code generation phases of the compiler.

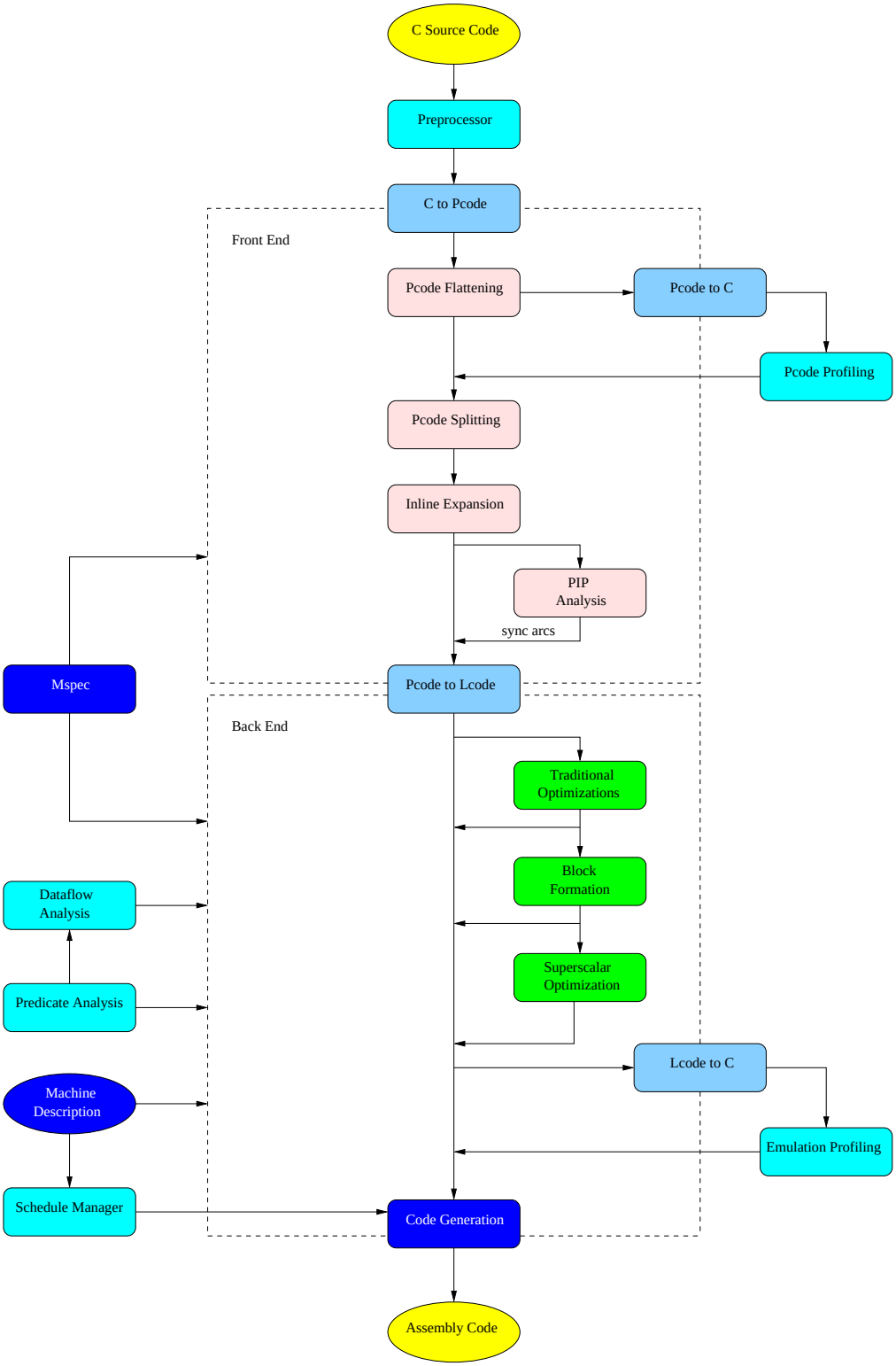


Figure 3.1: Block diagram of the IMPACT C compiler.

Section 3.3 describes machine-specific pieces of the compiler that are used to customize IMPACT based on the target architecture.

3.1 The Front End

The Pcode IR includes source-level information that allows it to perform high-level transformations easily. Pcode is semantically equivalent to parsed C code. The internal data structures of Pcode exist as an abstract syntax tree. The external file representation of the Pcode tree is a series of atoms and parentheses, printed in prefix notation, as is required to maintain the modularity of the compiler.

The following sections better describe each module in the front end of IMPACT.

3.1.1 Pcode generation

The front end of the IMPACT compiler begins by converting preprocessed C code into the Pcode intermediate language. This is accomplished through the Edison Design Group (EDG) front end. This is a third-party tool that reads in C/C++ code and converts it into its own IR. IMPACT then uses this IR to print out Pcode.

The first transformation applied to the Pcode is flattening. The Pcode must be flattened in order to allow inlining to occur later in the front end. For example, C code has a notion of implicit temporaries and allows the nesting of a function call within an expression, as in "`i = foo1() + foo2();`". Correct programming semantics would not be preserved if either of the functions in this statement were directly inlined. The inlined

code would only make sense if the function calls were split up onto different lines. Thus, the code would need to first be flattened and look like the following:

```
temp1 = foo1();

temp2 = foo2();

i = temp1 + temp2;
```

After this transformation, all calls in the Pcode exist in a statement either alone, or on the right-hand side of an assignment.

3.1.2 Pcode profiling

Once the Pcode has been flattened, the profiling process occurs. The Pcode is profiled by first converting it to equivalent C code. Annotated in this C code is a series of probes that will be used to gather execution information. This C code is then compiled on the target platform and executed. For the SC140, this implies that the C code needs to be compiled with the StarCore C Compiler and run on the SC140 simulator. When the program is executed, in addition to the normal outputs, the code will produce several files with profile information. Each of these files are then read in and merged with the Pcode to annotate the profile information.

In order for the process to work correctly, certain assumptions made by the StarCore compiler need to be met. In order for the compiler to properly set up the parameters to its function calls, all external definitions must contain all of the proper parameter types. For IMPACT to meet this restriction, the EDG module had to be changed from Kerrighan and Richie (K&R) C mode to the American National Standards Institute (ANSI) mode.

This required considerable external effort given the way IMPACT uses EDG to generate Pcode.

Once this profiling information is collected, it becomes vital for several of the modules in both the front and back end of the compiler.

3.1.3 Inline expansion

The first IMPACT optimization that relies on this profile information is function inlining [3]. To simplify the inlining process, each function in the program is split into its own file. In order to accomplish this, all static variables and structure definitions are renamed according to their original file location. This name mangling produces unique names that are visible to the entire program.

Inlining is a very important optimization that greatly benefits performance. There is a trade-off, however, in that it can significantly increase the code size. To maximize the benefits of inlining, only frequently executed function calls and callees with small body size should be inlined. Inlining a rarely called function will not improve performance significantly, and inlining a large function may greatly increase the code size, potentially leading to instruction cache conflicts or capacity misses.

3.1.4 Interprocedural analysis

The next module that is run on the Pcode is called Pcode interprocedural (PIP) analysis [4]. This process determines what dependencies exist with regard to global variables across function boundaries. Without having interprocedural information, multiple

extraneous dependencies would be drawn in the Pcode. This analysis also performs intraprocedural pointer disambiguation, performs dependence analysis, gathers alias and side effect information, and identifies targets of indirect function calls. This information is then merged back into the Pcode and is used by subsequent stages of the compilation. One current limitation of this analysis is the lack of array dependence analysis. Without this analysis, all accesses to an array are assumed to conflict. A finer-grained analysis of these relationships exists in IMPACT, but is not compatible with the current EDG front end.

3.1.5 Pcode to Lcode conversion

The final module in the front end of the compiler converts the Pcode into Lcode. This process essentially "lowers" the Pcode into simple instructions without any residual nesting. This lowering is analogous to the flattening performed earlier in the compilation. A mapping is then applied from the lowest form of Pcode to Lcode. This conversion process makes several calls to machine-specific functions, located in the Mspec module. This module is used throughout the entire compilation process, and is described in Section 3.3.1. Using information gathered from the Mspec, the Pcode is converted to equivalent Lcode.

An important machine-dependent restriction on the SC140 that is met in the front end of IMPACT is in setting up the parameters to a variable argument (vararg) function call. The SC140 convention for most calls is that the first two arguments to the call are passed through registers (either integer or address), while any other arguments are passed on the stack. When the call is to a vararg function, however, the rules are slightly

different. The last fixed argument and any variable arguments must be passed on the stack. If there are multiple fixed arguments, the other ones are passed according to the general parameter passing rules.

For example, the function `printf(char *, ...)` would need all of its arguments passed on the stack. The function `fprintf(FILE *, char *, ...)`, however, would have its first argument passed through address register r0, and the rest of the arguments on the stack.

This is a unique problem in that it is very unusual that the caller needs to know that the callee is a vararg function. The ramification of this for IMPACT is that additional information now needs to be passed to the Mspec routine that sets up the function parameters. This problem is also related to the K&R C mode problem in that the IMPACT EDG front end loses any vararg information for external function declarations. This information is preserved when EDG is in ANSI mode.

3.2 The Back End

Lcode is essentially a machine-independent instruction set implemented as a generalized register transfer language similar to most load/store architecture instruction sets. This is meant both to facilitate a straight-forward conversion from Lcode to machine code (Mcode), and to allow for optimizations to be performed in a traditional way. The optimizations that are performed on the Lcode are described in the following sections. The back end completes the entire compilation process, concluding with code generation.

After Pcode to Lcode conversion, the Lcode will only contain register types that are supported in the machine's hardware. For most other code generators, this means integer, float, and possibly predicate types. The SC140 is unique in that there are not any floating point registers, and that there are address registers.

The SC140 handles floating point operations by using software library functions. The parameters and return values for these functions remain in an IEEE standard format, but are kept in data registers. To properly handle this, floating point registers are propogated all the way to the code generator, where they are treated the same as integer registers. Any floating point constants existing in the Lcode are cast into integer operands in the code generator. This integer value is simply the bits used to encode the floating point number in the IEEE standard format.

To accomodate the address registers, IMPACT registers of type pointer are propagated to the back end of the compiler. This requires several changes to back end modules to accommodate these registers. Most notibly, the register allocator needs to be aware of a new register type and to handle the multiple register files correctly.

3.2.1 Lcode traditional optimizations

The Lcode that is produced from the PtoL phase is very unoptimized and is strictly generated for correctness. This code is first run through a phase of traditional optimizations to streamline the code. These transformations are all machine-independent in that they are performed regardless of the target architecture.

This phase includes performing several optimzations, including constant propogation, copy propogation, common subexpression elimination, redundant load/store elimination,

strength reduction, dead code removal, invariant code removal, global variable migration, induction variable elimination, predicate promotion, branch target expansion and branch elimination [5]. After these transformations are applied, the code can be thought of as being in an efficient, optimized form. This code can then be used as a baseline to determine the effectiveness of IMPACT's more advanced optimizations.

3.2.2 Hyperblock and superblock formation

Once the Lcode is in an optimized state, block formation is perfomed to group basic blocks together to form hyperblocks [6] and superblocks [7]. Hyperblock regions are carefully chosen based on a path's dependence height and profile weight. Regions that are not candidates for hyperblock formation may, however, be easily converted into superblocks. Due to the possible need for tail duplication in these formation algorithms, there is usually an increase in the code size.

The IMPACT hyperblock formation algorithm works by first forming the innermost hammock regions. It then looks to merge hammocks together into larger and larger hammock regions. The path selector then looks at the regions for nonideal sections caused by uneven dependence heights or infrequent paths. It will then optimize out those regions and tail duplicate the code where necessary. The resulting hyperblocks are generally large and can contain a chain of predicates that combine to form the control flow.

Since the SC140 only has essentially one predicate register, represented by the T-bit, much more restrictive hyperblock formation needs to take place. A new parameter was needed for this situation so that only simple hammock regions would be formed. The

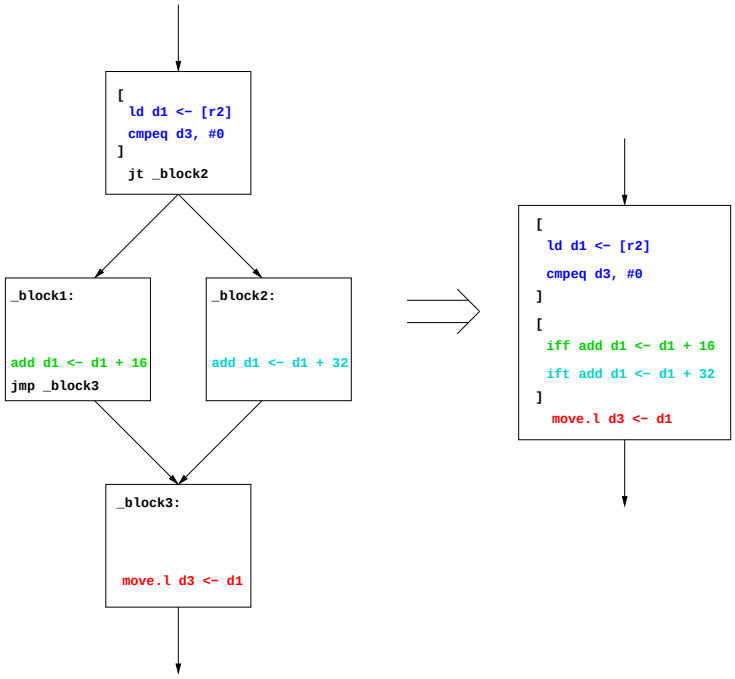


Figure 3.2: Hammock identification and code generation.

path selector can still optimize out undesirable sections of the hammock, but it will not receive hammocks containing more than four basic blocks. A simple example of the regions being formed is shown in Figure 3.2.

In this example, a simple hammock region is shown with data register d1 being updated differently based on the path taken. This hammock has a static size of seven instructions and a cycle count of 7 for either path. Given that both block1 and block2 have identical execution heights, and assuming that both paths have a significant profile weight, the hyperblock to the right will be generated by IMPACT. This hyperblock has a static size of five instructions and a guaranteed execution time of three cycles. The performance of the code will significantly benefit in this case.

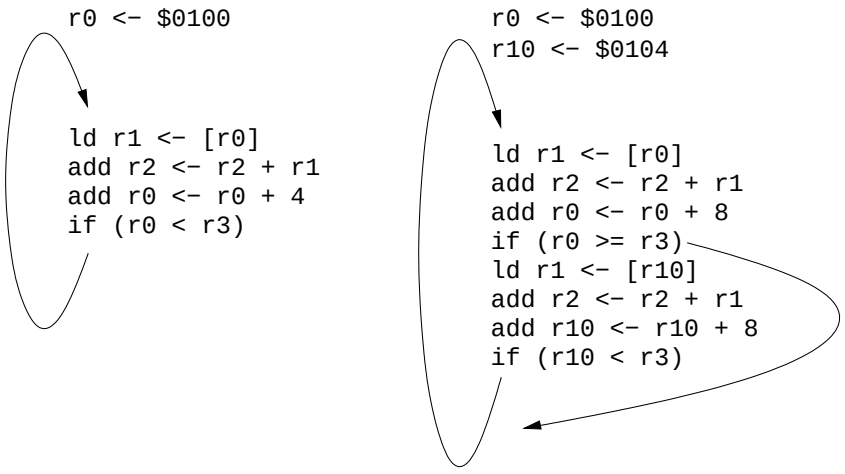
A further discussion of the tradeoffs involved in having multiple predicate registers is found in Section 6.2.

3.2.3 Superscalar optimizations

Exposing parallelism to the hardware is critical in achieving high performance on multiple issue machines. In order to uncover large amounts of ILP, IMPACT performs several specialized optimizations [8].

There are eight main optimizations that are performed in the scope of a single module, Lsuperscalar. The first two (loop unrolling and register renaming), are widely used and understood. The next three (accumulator variable expansion, induction variable expansion, and search variable expansion), expose increasing levels of ILP on unrolled loops. The final three (operation combining, strength reduction, and tree height reduction), are used to increase the utilization of the processor's arithmetic hardware. After these optimizations are completed, the block former will look for new opportunities to form superblocks. These opportunities usually arise from the amount of extra code that has been added as a side effect of superscalar optimizations.

When studying the effect of these optimizations on the SC140, loop unrolling had the biggest impact on program performance. Loop unrolling is the process of expanding the body of a loop to contain more than one iteration. A simple example of a loop body is shown in Figure 3.3(a). When this loop is unrolled, each instruction is duplicated by the unroll factor. There is a side exit placed between iterations and the final instruction is the loop-back branch. A common optimization associated with loop unrolling is induction variable expansion. This serves to break some cross-iteration dependencies that



(a) Original loop.

(b) Loop after unrolling and induction expansion.

Figure 3.3: Loop unrolling example.

are present in the unrolled loop. Figure 3.3(b) shows the example loop unrolled and optimized. Speculation above the side exit can now be performed to expose additional ILP. Note that additional optimizations could be performed to uncover more parallelism, such as accumulator variable expansion in this example.

While loop unrolling can provide significant performance improvement under certain situations, there are substantial trade-offs that are specifically seen on the SC140. The most obvious problem with loop unrolling is the amount of code growth that occurs. This extra code can potentially cause several problems for the code generation process. For example, after the unrolled loop is optimized by the compiler, the number of registers live in the loop will likely have been increased. This can lead to the register allocator inserting register spill and fill code within the loop body, immediately sacrificing performance. This

extra code also contributes to an increase in the static size of the program, leading to increased I-cache pressure and potentially increased power consumption.

The benefits of loop unrolling are explored in Section 7.1. The main loops of the benchmarks will either be unrolled a variable number of times, or modulo scheduled.

3.2.4 Code generation

At any point in the back end of the compiler, the Lcode can be translated into machine-specific code. This process is known as code generation, and can potentially be done for any target architecture. Some previous code generators that were written for IMPACT include x86, SPARC, MIPS, and Itanium.

The StarCore SC140 code generator that was produced for this work is described in detail in Chapter 4.

3.3 IMPACT Customization

In almost all stages of compilation, some machine-specific information may be needed by the modules. This information can take many forms, from the alignment of structures to instruction latencies. This information is provided to the compiler in two important pieces, the machine specification (Mspec) and machine description (mdes). The Mspec provides the compiler with a unified interface to machine-specific information. This module is described in Section 3.3.1. The machine description file is a table of information that is used by the instruction scheduler to produce a correct and efficient code schedule. This table is detailed in Section 3.3.2.

3.3.1 Machine specification

The Mspec module is essentially a large group of functions which the compiler uses to customize several stages of compilation. The compiler will make a general function call to the Mspec, which will then call the appropriate machine-specific function. This set of calls is divided into two files, one that contains calls used in the front end and one that contains calls used in the back end of the compiler.

The front end of IMPACT needs to make calls into the Mspec to determine things like the size of C types, alignment of structures and unions, and function calling conventions. When the Lcode is finally generated, it will be in a form that closely resembles the target's assembly code. Any additional machine-specific changes to the code are performed in the code generator.

The back end of IMPACT makes calls into the Mspec to better understand the relationship between Lcode and the target's assembly code. Since the Lcode is being aggressively optimized in the back end, the compiler needs to know what the eventual Mcode will look like. For example, the back end of IMPACT makes a call to the Mspec to determine if a certain Lcode operation will eventually become a function call. On the SC140, any floating point operation will need to become a call due to the lack of floating point support. This knowledge limits the amount of code motion that can be performed around these Lcode instructions.

3.3.2 Machine description

The IMPACT machine description file is written in specialized language, called MD [9]. This language is designed so that it is easy to write, read, and modify complex machine descriptions. This file is then compiled into a low-level equivalent file that can be easily parsed by the compiler.

The mdes file is accessed based on a specific processor opcode. This opcode can contain several different combinations of operands. For each valid combination, a set of resources is associated with it. These resources correspond to actual hardware constraints, such as issue slots or functional units.

When a particular instruction is being scheduled, it needs to pass several checks in the mdes before it is deemed legal. First, the source and destination operands are checked against the opcode's list of valid combinations. For the particular combination that is used, its set of resources is gathered. If a particular resource is not available in a given cycle, this instruction will fail the check. When no more instructions can be scheduled in a given cycle, the VLES is closed and the scheduler begins on the next one.

The MD language is designed for completely orthogonal instruction sets and machines with regular bypass paths. Aspects of this design conflict with modeling some of the SC140's grouping restrictions. These issues will be discussed in Chapter 5.

CHAPTER 4

THE IMPACT SC140 CODE GENERATOR

The work described in this thesis is the development of an accurate, efficient code generator targeting the StarCore SC140. The StarCore SC140 has several unique features, as described earlier. There were several challenges in developing a code generator that could correctly handle these features. A high-level view of the IMPACT SC140 code generator is shown in Figure 4.1. The code generator is logically separated into three distinct phases. Each phase will be detailed in the following sections.

4.1 Phase 1

The first phase of the code generator is responsible for converting the incoming Lcode into machine-specific instructions, referred to as Mcode, through an annotation process. Mcode is semantically identical to Lcode, but represents code that maps directly to correct assembly for the target machine. This process is described in Section 4.1.1. The code is then optimized to discover situations where the machine's instruction set can be better utilized. This optimization process is described in Section 4.1.2.

4.1.1 Code annotation

Code annotation is the first step in the code generation process and is performed individually on each Lcode operation in the code. In the simplest case, a one-to-one

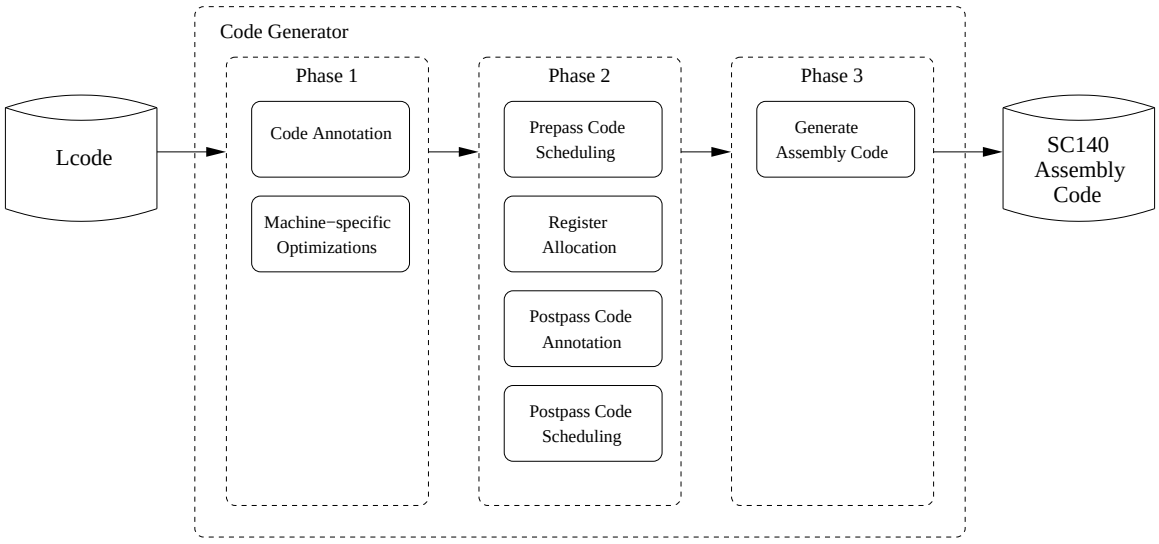


Figure 4.1: Block diagram of the IMPACT SC140 code generator.

correspondence exists between the incoming Lcode instruction and the available assembly instruction. When this occurs, a machine-specific opcode is simply added to the existing Lcode instruction.

The simple case only occurs when both the Lcode operation has an exact equivalent operation on the machine, and the operands are in the correct format. When either of these requirements are not met, an expansion to several Mcode instructions is usually needed. Examples of these two cases are shown in Figure 4.2.

In the first case, an integer divide instruction is not available in any form on the SC140. The functionality is provided through software and an explicit function call. First, each of the original source operands are moved into the appropriate parameter macros, `$P0` and `$P1`. The call to the 32-bit divide routine is then inserted as a jsr. The return value, in `$P0`, is then moved back into the original destination, completing the original divide instruction as intended.

```
    Lcode op                                  Mcode ops
1.  div (r 3 i) <- (r 1 i)(r 2 i)       1.  move.l (mac $P0 i) <- (r 1 i)
                                        2.  move.l (mac $P1 i) <- (r 2 i)
                                        3.  jsr __div32
                                        4.  move.l (r 3 i) <- (mac $P0 i)

1.  add (r 3 i) <- (r 1 i)(r 2 pnt)     1.  move.l (r 4 i) <- (r 2 pnt)
                                        2.  add (r 3 i) <- (r 1 i)(r 4 i)
```

Figure 4.2: Lcode to Mcode conversion.

In the second case, the source operands to the add are of different types, with one being of type integer and the other of type pointer. The SC140 has several variations of the add instruction, but none that takes operands from two different register files. While the add could be converted into either an integer add or a pointer add, the integer add will require fewer instructions due to the destination operand being of type integer. An explicit move instruction is needed to bring the pointer value into an integer register and then an integer add can be performed.

An explicit move instruction can also be required when an Lcode constant value is too large, or in the wrong source location. Much of the code in the annotation phase is there to check operands and convert them as needed. These annotation steps are performed in a greedy manner, and usually provides a good opportunity for future optimization to be performed.

4.1.2 Machine-specific optimizations

The last part of phase 1 is spent performing machine-specific optimizations. Since the annotation phase is done independently for each Lcode instruction, there could exist opportunities to collapse several instructions together. These situations are caused by having assembly instructions that are not representable in Lcode. For example, the SC140 has an ADDL1A instruction that will shift one of its source operands left once before adding it to another operand. Given the proper situation, using this opcode could replace two Lcode instructions. This optimization can only be performed if the intermediate value in the original Lcode is not used anywhere else in the code.

While all SC140 instructions are modeled in the machine description, not all of the instructions are currently utilized in the output assembly code. In general, this occurs in cases where the SC140 instruction does not have a corresponding Lcode instruction. In these cases, a specific transformation would be needed that looks for a particular situation. Other nonutilized SC140 instructions involve features of the architecture that are not currently supported by IMPACT, including hardware loops and fractional data. The better utilization of SC140 instructions is left for future development, with some ideas presented in Chapter 8.

4.2 Phase 2

The second phase of the code generator is reserved for instruction scheduling and register allocation. An initial pass of scheduling is first performed on the code, called prepass scheduling. This scheduling process is detailed in Section 4.2.1. At this point,

the code is using virtual registers for all of its operands. These registers are mapped to the actual machine's register file(s) in the register allocation module. This process is described in Section 4.2.2.

During register allocation, gaps in the schedule are introduced due to register spills and fills. This inserted code can have consequences to the performance and correctness of the code. Postpass annotation steps, which are needed to guarantee this correctness, are described in Section 4.2.3. Finally, a second pass of scheduling is performed to achieve the final schedule. This process is called postpass scheduling and is described in Section 4.2.4.

4.2.1 Prepass scheduling

Prepass scheduling performs code motion without regard to register constraints, as only virtual registers are used. This code motion can be done in several different ways. By default, an acyclic scheduler is used on the Lcode. A list scheduling algorithm is used to determine how each instruction can best be placed in the code. To do this, a priority is assigned to all instructions. Each instruction is then scheduled in order of its priority. An instruction is scheduled in the earliest clock cycle where all of its incoming dependencies are satisfied and the machine resources are available.

An alternate approach that can be used by the compiler is a variation of software pipelining [10], called modulo scheduling [11]. This scheduling technique will overlap successive iterations of a loop in order to uncover ILP. A new loop iteration can be initiated at a constant rate, called the initiation interval (II).

A high-level view of this process is shown in Figure 4.3. In this figure, each iteration of the loop is broken up into four stages, each taking II cycles to complete. The shaded

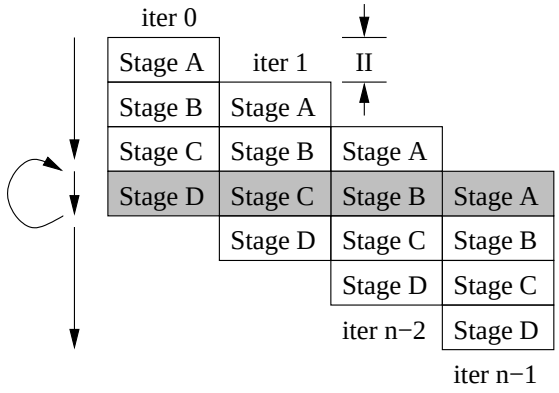


Figure 4.3: Execution of a modulo scheduled loop with four overlapped iterations.

region of code is called the *kernel*, and contains instructions from each iteration. This kernel is simply re-executed to advance each loop iteration by a stage. The code before the kernel is referred to as the *prologue*, while the code after the kernel is called the *epilogue*.

When modulo scheduling is performed on a loop, some register lifetimes may be longer than the initiation interval. When this occurs, consecutive iterations of the loop will end up conflicting with each other. In order to separate each iteration's registers, modulo variable expansion (MVE) [12] is employed. This technique will unroll the loop body and assign nonconflicting names to all register lifetimes in the unrolled loop. While MVE can generate efficient modulo scheduled loops, it has the negative effect of increasing the code size and adding register pressure. The negative effects of this register pressure can be seen in both the register allocator and postpass scheduler.

4.2.2 Register allocation

In order to properly reflect the machine resources, the register allocator is run on the code next. This process will calculate the live ranges of all registers and attempt to use as few registers as possible. Spill and fill code is inserted to save a register value when the number of live registers exceeds the number of registers available in the machine. To correctly handle this, the register allocator needs to be aware of the actual register files available on the machine. For the SC140, all registers of type integer were mapped to the DALU register file while registers of type pointer were mapped to the AGU register file. In order to properly evaluate the usefulness of multiple predicate registers, a predicate register file was also created for the SC140. This register file cannot be accessed during normal code generation, and was only used in the predication experiments.

When register allocating a modulo scheduled loop, spills and fills are avoided at almost any cost. This is because, in general, the loops are scheduled very compactly, without room for additional instructions. When these instructions are inserted into the schedule, performance almost always suffers. When generating modulo scheduled code for the SC140, these spills are almost inevitable given the size of the register files. The frequency with which these spills occur is shown in Section 7.1.

4.2.3 Postpass code annotation

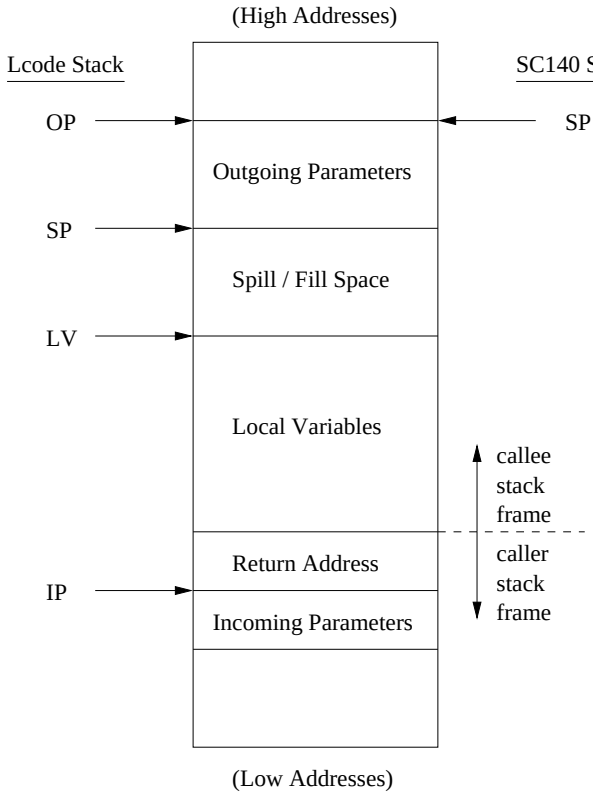
At this point in the Lcode, each stack reference is relative to only its own section (i.e., local variables, outgoing parameters, etc). Since the stack was adjusted to accommodate spill code during register allocation, all stack references in the program need to be updated

here. After the adjustments, all stack references are relative to a single stack pointer, SP. Figure 4.4 shows this process through an example.

The Lcode stack contains pointers to each of the main sections of the stack, with all references made relative to these pointers. Code generated for the SC140, however, can only contain references to one stack pointer, SP, which points to the top of the stack. Figure 4.4(a) shows the layout of both the Lcode and SC140 stack models. All accesses in the code are made through negative offsets to the pointers. Once the sizes of the stack regions is known, the conversion between the two can take place. The sizes used in this example are shown in Figure 4.4(b). By convention on the SC140, the return address space is always defined to be 8 bytes. For this example, the total size of the callee's stack is 112 bytes, with the return address and input parameters belonging to the caller.

A portion of Lcode before the conversion is shown in Figure 4.4(c). This particular example starts by reading an incoming parameter and a local variable. These values are used as outgoing parameters for a function call. Register 3 is spilled and filled around the JSR. All references to the stack are accomplished through each section's own pointer. Figure 4.4(d) shows this same portion of Lcode after the conversion to the SC140 stack model. All references are now relative only to the SP pointer. An important aspect of this process is that all stack references are still accessing the same exact stack location as before.

Fixing up the program's stack references is not the only task performed in the postpass annotation phase. When the register allocator spills certain registers, certain programming assumptions may be violated. One such violation occurs when spilling the source



(a) Lcode and SC140 stack configurations.

outgoing parameter space = 16
spill / fill space = 32
local variable space = 64
return address space = 8
incoming parameter space = 8

(b) Example stack sizes.

```
ld_i [(r 5 i)][(mac $IP pnt)(i −4)]  # incoming parm
ld_i [(r 4 i)][(mac $LV pnt)(i −32)] # local variable
st_i [][(mac $OP pnt)(i −4)(r 4 i)]  # 1st outgoing parm
st_i [][(mac $OP pnt)(i −8)(r 5 i)]  # 2nd outgoing parm
st_i [][(mac $SP pnt)(i −4)(r 3 i)]  # register spill
jsr  [][(printf)]                    # function call
ld_i [(r 3 i)][(mac $SP pnt)(i −4)]  # register fill
```

(c) Example using Lcode stack.

```
ld_i [(r 5 i)][(mac $SP pnt)(i −124)] # incoming parm
ld_i [(r 4 i)][(mac $SP pnt)(i −80)]  # local variable
st_i [][(mac $SP pnt)(i −4)(r 4 i)]   # 1st outgoing parm
st_i [][(mac $SP pnt)(i −8)(r 5 i)]   # 2nd outgoing parm
st_i [][(mac $SP pnt)(i −20)(r 3 i)]  # register spill
jsr  [][(printf)]                     # function call
ld_i [(r 3 i)][(mac $SP pnt)(i −20)]  # register fill
```

(d) Example using SC140 stack.

Figure 4.4: Stack annotation example.

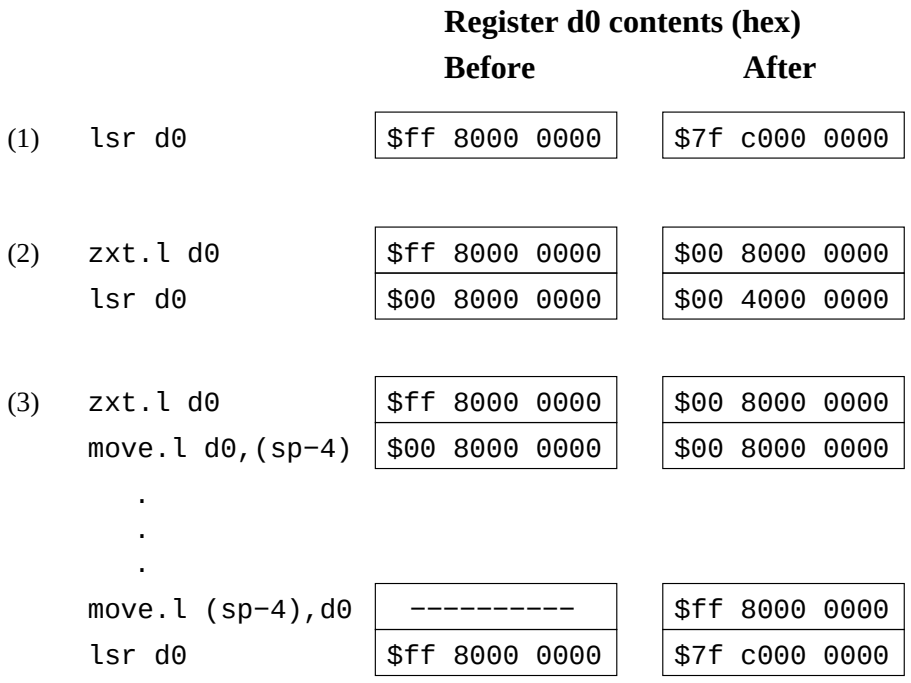


Figure 4.5: Postpass annotation example.

operand to a "logical shift right" instruction and is shown in Figure 4.5. This shift instruction occurs in the DALU and operates on all 40 bits of a source register. For correct 32-bit operation, a value of 0 must always be shifted into the 32nd bit, regardless of the value in the extension bits. This is because the IMPACT logical shift right instruction operates on 32 bits, and has no knowledge of the extra bits. If the extension bits contain other data, as in (1), an incorrect 32-bit result is produced. This "garbage" data that is in the extension bits of d0 before the shift can be produced by several situations in valid code. The most important detail, though, is that the value of the extension bits is undefined in normal 32-bit operation. In order to fix this problem, the extension bits must always be zero extended directly before the shift, shown in (2). In this case, the final result in d0 is what the compiler expects for a 32-bit operation.

When prepass scheduling occurs, the zero extension is free to move up in the code. While this may help the prepass schedule, it increases register d0's live range. When register allocation is performed, it is possible that d0 will have to be spilled in between the two instructions. This situation is shown in (3). The store to memory and the subsequent load are performed on 32-bit values. The load will initialize the extension bits by automatically sign extending them. In this example, the sign extension destroys the result of the zero extension and produces incorrect output.

The only way that this situation can be avoided in the code generator is to delay the creation of the zero extension instruction until after register allocation. Since the addition of this instruction does not create a new virtual register, it is perfectly legal to do. The code can still be scheduled during postpass, but this will guarantee that the intermediate register value is not spilled.

4.2.4 Postpass scheduling

At this point, the Mcode is in the proper form to be directly converted to the final assembly. Before this occurs, a final schedule is produced that includes all code that was created during this phase. For modulo scheduled loops, this step is not normally needed as the schedule from prepass is still valid. The only time that postpass is needed is when the register allocator inserts a spill in the loop.

After this step is complete, an attribute is attached to each instruction indicating its issue cycle, issue slot, and latency. This information is used by the final phase to properly complete the code generation.

4.3 Phase 3

The last phase of the code generator simply takes the optimized Mcode and prints out assembly. Great care has been taken throughout the code generator so that this process is fairly simple. No more transformations happen to the code, and every Mcode instruction corresponds to exactly one line of assembly.

One detail that needs to be handled in this process is the instruction grouping. The scheduler groups instructions together based on the information that it gathered from the mdes. Were the mdes able to model the target machine perfectly, the final schedule would always be correct, and the process would be trivial. However, several of the SC140 grouping rules could not be represented in the current mdes. Because of this, extra checking was required in this phase to ensure that correct code was produced.

In general, this checking code looks at a particular VLES (as specified by attributes), and checks for violations of these specific rules. When a rule is broken, the current VLES is split up to remove the violation. These situations, and their performance ramifications, are described in more detail in the next chapter.

CHAPTER 5

LIMITATIONS IN THE IMPACT MACHINE DESCRIPTION

Due to the nature of the VLES structure, many intricate hardware details need to be exposed to the compiler. For example, the SC140 does not provide hardware interlocks. Thus, instruction latencies and inter-VLES dependencies need to be dealt with at compile time. Since the compiler is performing all of the scheduling statically, there are several rules that it must follow in order to produce correct assembly code.

Usually, these hardware restrictions are reflected to the compiler by the mdes file. When the instruction scheduler forms a VLES, it looks at this mdes to determine what instructions can be legally scheduled in the same cycle. When this assembly code is then run through the assembler and simulator, no errors should be generated and the code should run correctly.

Many of the SC140 grouping restrictions were easily representable in the IMPACT mdes file. There were, however, several rules that cannot currently be represented in the file. The following sections present several examples of restrictions that were implemented by means other than the mdes. Potential extensions to the mdes structure will be discussed to lay the groundwork for future mdes development.

5.1 Subgroup Restrictions

VLES subgrouping is not easily handled in the current mdes structure. This is because of the way that subgroups are defined in the SC140. A subgroup exists within a VLES for each subset of instructions that is predicated on the same condition.

There are three main restrictions that the mdes would need to be aware of when scheduling. The first is that a VLES can contain a maximum of two subgroups. The second is that the IFA subgroup, if used, must be the last subgroup in the VLES. The third is that each subgroup can only have up to two DALU instructions and 1 AGU instruction.

In order for the mdes to fully understand the subgroup restrictions, it would need to be aware not only of the existence of a predicate, but also of the relative value of that predicate. In other words, the mdes would need to represent the fact that a particular instruction was grouped under the IFT, IFF, or IFA subgrouping.

The mdes can currently be used to distinguish between a predicated and an unpredicated instruction, but it is currently impossible to distinguish between two different predicate registers. The only information that the mdes has about instruction operands is their type (i.e., integer, float, etc.), but implementing these restrictions would require more than that. Since the current mdes cannot properly model this situation, a possible extension to the structure is now described.

If the mdes could determine the predicate type of an operand, this could be used to distinguish between different subgroups because of the way predicates are defined in the code generator. For each compare instruction on the SC140, two predicates are always

defined (the T-bit and its complement). Mirroring the way predication is done on the machine itself, the T-bit is always of type conditional-true and its complement of type conditional-false. Because of this, a direct mapping exists between a predicate and its type.

For any predicated Lcode instruction, the type of the predicate is not known. The type information only exists on the comparison that defines the predicate register. Once this information is available, a predicate of type conditional-true would reside in the IFT subgroup while a predicate of type conditional-false would correspond to the IFF subgroup.

Once the predicate type is associated with an instruction in the mdes, resources can easily be associated with them. A predicated instruction, whether conditional-true or false, will only have a limited set of resources available to it. Using this new information, all three restrictions described earlier could be modeled in the IMPACT mdes file.

A more general solution that would work when the predicate type does not imply the subgrouping will be introduced and explained in Section 5.3.

5.2 Subgroup Encoding

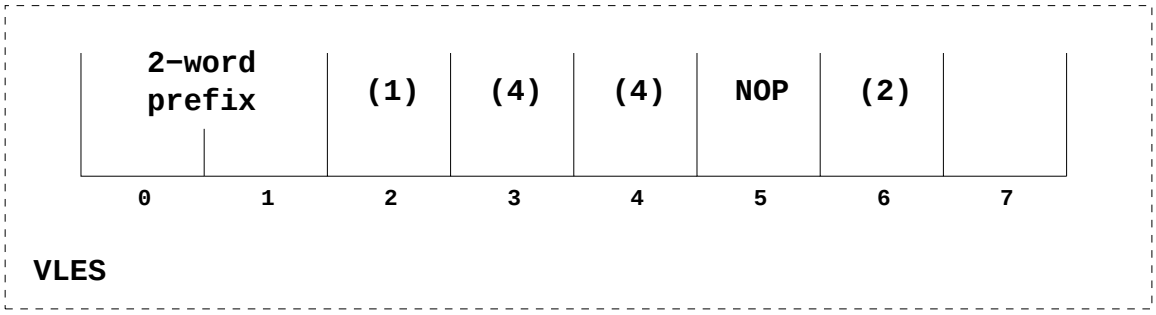
There is a subtle encoding issue that also needs to be addressed with regard to VLES subgrouping. A VLES can contain up to eight instruction words, occupying positions 0-7 within the execution set. When a VLES contains two subgroups, the instructions in subgroup 1 occupy the even positions, while subgroup 2 occupies the odd positions.

Under the assumption that a two-word prefix needs to be used, there will be three instruction words available for each subgroup. This may not initially seem like a problem in that each subgroup can only contain three instructions. A problem, though, arises when a multiple-word instruction is used in a subgroup. This can best be explained using the example provided in Figure 5.1. In this example, taken from actual code, the VLES initially formed by IMPACT is shown in part (a). There are two subgroups with four total instructions. The first three instructions will execute only if the T-bit is 0, while the last instruction will always execute. Upon initial inspection, the VLES looks legal since the first subgroup contains two DALU instructions and one AGU instruction, while the second subgroup only contains a single AGU instruction. The VLES uses a two-word prefix because of the conditional execution and the high register accesses. Including these two prefix words, the total number of instruction words is 7, which is less than the maximum value of 8.

The problem occurs when the VLES is actually encoded by the assembler. This encoding is shown in part (b) of the figure. The first two instruction words are occupied by the prefix. The first instruction from subgroup 1 (1) is placed in position 2. Then the first instruction from subgroup 2 (4) is placed in position 3. Since this instruction is two words long, the second word "overflows" and also occupies position 4. Since there are no more instructions from subgroup 2 to occupy position 5, a NOP is inserted. Then the next instruction from subgroup 1 (2) is placed in position 6. Instruction 3 is now waiting to be placed in the VLES, but there are no more even slots for it to occupy. Thus, this VLES was illegal.

```
                                                    instruction
                assembly code                       type   size
     [
(1)     iff  tfr     d2,d13                         DALU     1
(2)     iff  move.l  d4,r13                         AGU      1
(3)     iff  tfr     d11,d15                        DALU     1
(4)     ifa  move.l  (sp-312),r11                   AGU      2
     ]
```

(a) VLES assembly code.



(b) VLES encoding.

Figure 5.1: Restriction example.

In order for the mdes to properly recognize this case, it would have to understand that multiple-word instructions can use up the other subgroup's resources. This can be made possible after the mdes properly recognizes the subgroup boundaries. Once this is possible, multiple-word instructions can simply use up resources that were originally designated for the other subgroup.

5.3 Multiple Sources

Another SC140 grouping restriction is that a data register can be used as a source operand up to four times in a VLES. This includes any implicit uses of the operand, such as the accumulator source of a mac instruction.

IMPACT cannot currently handle this restriction in the mdes for the same reason that it cannot distinguish between subgroups. The mdes file carries no information about the actual register that is being used, only its type. For example, the mdes cannot tell a difference between `move.l r1 <- d2` and `move.l r7 <- d5`. It only knows that the source is a data register of type integer and that the destination is an address register of type pointer for each case.

Representing this restriction requires a more general solution than the one proposed for subgrouping. This is because the mdes now needs to look at the actual registers being used, with all registers being the same type. The mdes would need to associate a specific register with each of its operands. Each of these registers would then occupy a resource. When the resources are depleted, the grouping would be illegal. The following is a proposed extension to the mdes that could be used to model these specific situations.

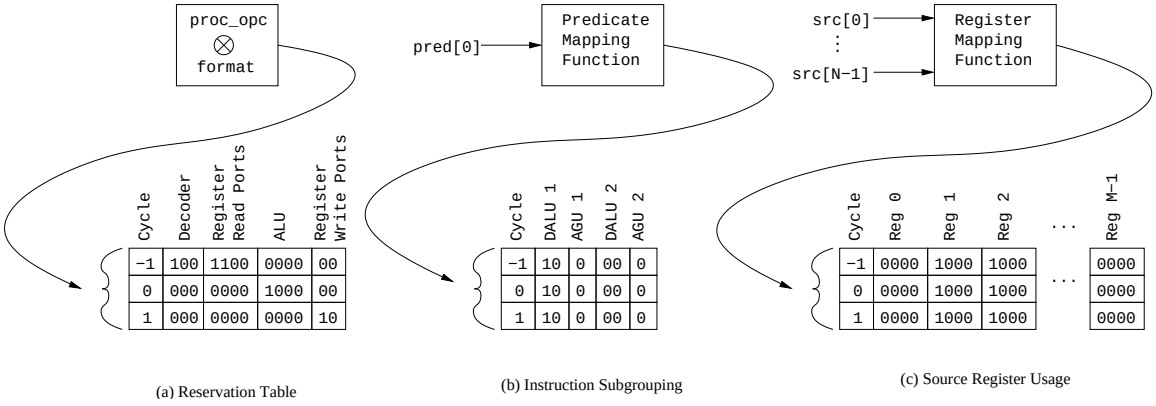


Figure 5.2: Diagram of mdes with additional resource tables.

To effectively model these SC140 restrictions, orthogonal resources must be added to the existing mdes structure. Figure 5.2(a) shows how the proc_opc and format of an instruction could index into the existing mdes reservation table. This idea can be extended by having the operands of each instruction index into new tables representing particular restraints.

Figure 5.2(b) shows a new table that is indexed based on the instruction's predicate register. A mapping function would be required to look at the predicate operand and to determine to which subgroup it would belong. For the SC140, this function would simply look at its predicate type, found at the register's definition. The indexed resource table would then account for each subgroup restriction. This particular example shows a table option for a DALU instruction. If the instruction is predicated, it can occupy either subgroup 1 or subgroup 2. If the instruction is not predicated, it is restricted to only occupy subgroup 2.

Figure 5.2(c) shows a new table that is indexed based on the instruction's source operands. This mapping function will allocate a new register resource to each unique source operand. In this way, the size of the table is constrained by the maximum number of source operands (N) and the issue width of the machine (W). The maximum size of the table would always be ($N * W$). Each register in the table has a resource corresponding to the number of times it can be used in a VLES. In this example, the current instruction uses Reg 1 and Reg 2 as source operands. There are four resources for each register, corresponding to the number of legal uses in the VLES.

These new tables would simply be concatenated onto the end of the normal resource bit vector used in the scheduler. While considerable effort would be required to implement this, it is a natural extension to the extisting structure. This work becomes a priority when looking at the performance lost due to these inefficiencies, discussed in Section 7.4.

CHAPTER 6

PERFORMANCE CONSIDERATIONS

A secondary goal of this work is to quantify the benefit of certain IMPACT optimizations and architecture extensions to the SC140. The target machines that IMPACT usually generates code for are general purpose processors running large applications. These architectures are usually quite robust, with large register files, on-chip caches, and branch prediction capabilities. Aggressive optimization has been proven to be beneficial for many of these processors. This work will attempt to determine the benefit and scalability of several common optimizations in the embedded domain, where resources are generally more constrained.

The first opportunity that will be examined is the benefit from modulo scheduling. The next opportunity to be examined is the possible benefit from increasing the number of available predicates. This increase would mean that more complex hyperblocks could be formed. Finally, any benefit from easing the issue restrictions will be quantified. The following sections introduce the concepts and motivation behind their analysis. Actual results and comparisons are shown in Chapter 7.

6.1 Modulo Scheduling

Modulo scheduling has been proven to be an effective scheduling technique on general purpose processors. An important trade-off when using modulo scheduling is that there is an increased amount of register pressure. This pressure comes from the need to create register copies across different iterations of the loop. While the SC140 does have 32 general purpose registers, the relative usage of the data and address registers is very important. For example, if only data registers are being used in a critical section of the loop, there are 16 registers from which to choose. The performance breakpoint occurs when the register allocator needs to add spill and fill code within the modulo scheduled loop. Performance will likely decrease when this occurs.

Experiments were run using modulo scheduling in comparison to loop unrolling optimizations. The number of overlapping iterations in a modulo scheduled loop is generally analogous to unrolling the loop by that same factor. There are, however, costs associated with the prologue and epilogue code that are also taken into account. The problem with loop unrolling is in determining how many times a loop should be unrolled. Because of this, different unrolling factors are used in the experiments.

6.2 Extending Predication

Possibly the most severe restriction that the SC140 places on IMPACT is in the number of predicate registers available. The hyperblock former is designed to exploit large predicated regions consisting of several frequently executing paths. When being

Table 6.1: Predicate definition truth table.

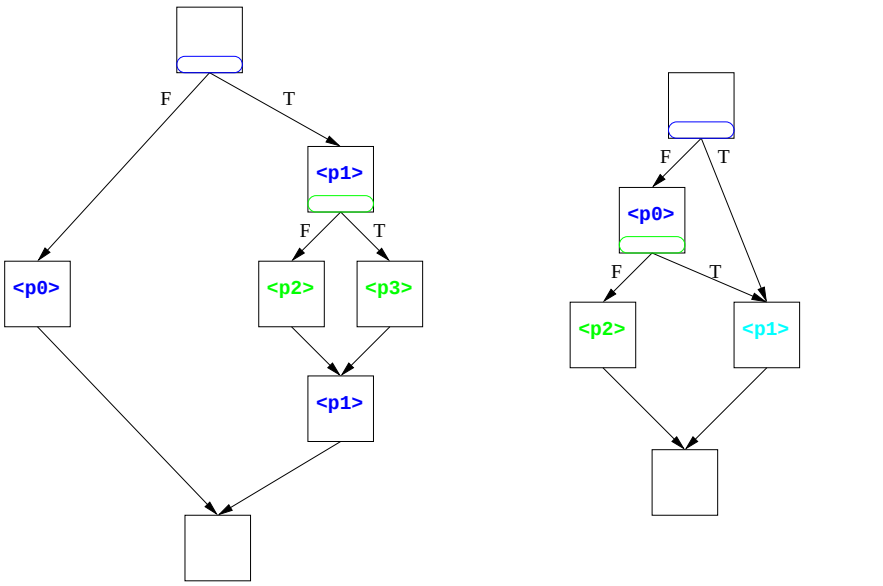
pSRC	Comp	UT	UF	CT	CF	OT	OF	AT	AF
0	0	0	0	-	-	-	-	-	-
0	1	0	0	-	-	-	-	-	-
1	0	0	1	0	1	-	1	0	-
1	1	1	0	1	0	1	-	-	0

restricted to only one predicate and simple hammocks, some important regions will not be formed, leaving potentially inefficient control code.

When multiple predicates are available, though, the architecture could potentially have to support multiple predicate types, including AND- and OR-types. The SC140 supports the conditional predicate model for defining predicates. A summary of common predicate types and their behavior is shown in Table 6.1. In the table, pSRC refers to the predicate value of the comparison instruction itself. Comp refers to the value of the condition being checked. UT/UF is unconditional-type, CT/CF is conditional-type, OT/OF is OR-type, and AT/AF is AND-type. Notice that when pSRC is 0, the predicates for the conditional-types remain unchanged.

The predication supported on the SC140 does not provide enough flexibility when dealing with real code examples. Figure 6.1 shows two control flow graphs that could be collapsed into a hyperblock given four available predicate registers.

In Figure 6.1(a), the top block will set p1 if a condition is true and p0 if it is false. Predicate p1 will then guard all of the instructions on the taken path. The first block on that path will set p3 if its condition is true and p2 if it is false. Since the comparison itself is guarded by p1, predicate p1 dominates the entire hammock. Thus, all control flow



(a) Example of a nested hammock requiring unconditional−type predicates.

(b) Example requiring OR−type predicates.

Figure 6.1: Control flow graphs of potential hyperblocks.

can be eliminated, and this can be represented in a single hyperblock with four predicate registers. This hyperblock requires the existence of only unconditional-type predicates.

In Figure 6.1(b), the top block will also set p0 and p1. The difference is that the first block on the fall-through path will set p1 as an OR-type predicate. This way the block guarded by p1 will execute if either path would have been taken. This entire graph can now be collapsed into a single hyperblock containing three predicate registers. This hyperblock requires the existence of both unconditional and OR-type predicates.

Attempting to quantify the benefit of multiple predicate registers on the SC140 cannot be done easily because the additional hardware is not visible to the simulator. To still gain performance numbers, while not using the simulator, a static analysis was used. This analysis relies on the attributes inserted by the instruction scheduler. These attributes

are attached to each instruction and give valuable grouping information, such as issue cycle. This information, combined with an accurate basic block profile, can give a total cycle count for a given program. These numbers are only missing the cycles spent in library code, but can still be meaningfully compared to one another.

Various papers in the literature explore the benefits of predication in embedded architectures. In most cases a form of partial predication is explored due to the extra encoding cost of a full predication model. One form of partial predication is the conditional move [13], which allows an instruction that will conditionally "commit" a speculative value based on a register's Boolean value. Another strategy is to use predicates as instruction prefixes [14]. A slot-specifc predication scheme [15] allowed for general if-conversion to be applied to the code while only specifying a single bit in each instruction.

6.3 Increasing the Register Files

Many of the optimizations that IMPACT uses to uncover large amounts of ILP will increase the number of virtual registers in use. For the SC140, this usually corresponds to the need for spill and fill code when register allocation occurs. It is important to determine just how much the size of the register files limits the benchmark performance. This will also be a way to determine if the integer or address registers are under more pressure from the generated code.

6.4 Issue Restrictions

There are several potential performance bottlenecks with regard to VLES grouping restrictions. IMPACT-generated code will see these problems as magnified because of the aforementioned deficiencies in the mdes. When a violation of the grouping rules is discovered and a VLES needs to be broken up, there is an immediate degradation of one cycle. It is interesting to attempt to determine the number of cycles actually lost to this problem to provide motivation for upgrading the mdes.

CHAPTER 7

RESULTS

Since the StarCore SC140 is a specialized, embedded processor, it is extremely important to choose benchmarks that represent typical telecommunication applications. There has recently been an effort in the embedded community to characterize embedded workloads, and to provide a way to compare the performance of embedded processors. Due to the diversity of embedded applications, these suites are usually broken into several sets, with each set relating to a different market.

The first suite used was developed by the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [16]. Included in this suite is a set of telecommunication benchmarks that closely represent the SC140's workload. The other suite that is used in this work was developed at the University of Michigan and is referred to as MiBench [17]. This suite also contains a set of benchmarks designed to characterize a telecommunications workload. Table 7.1 summarizes the benchmarks used in this work. The only benchmark that was not included from the two suites is FFT/IFFT from MiBench. This was excluded because the fast fourier transform algorithm was written using floating-point, which requires software calls on the SC140.

This section is divided into three sections. Section 7.1 looks at the performance tradeoffs between loop unrolling and modulo scheduling. Section 7.2 analyzes the benefits of an increased number of predicate registers. Section 7.3 shows the performance benefits

Table 7.1: Benchmark set.

Name	Suite	Description
adpcmdec	MiBench	Speech decompression using adaptive differential pulse code modulation
adpmcenc	MiBench	Speech compression using adaptive differential pulse code modulation
autcor00	EEMBC	Fixed point auto-correlation
conven00	EEMBC	Convolutional encoder
crc32	MiBench	32-bit cyclic redundancy check
fbital00	EEMBC	Fixed point bit allocation
fft00	EEMBC	Fixed point complex fast fourier transform
gsmdec	MiBench	Voice decoding using the Global Standard for Mobile communications
gsmenc	MiBench	Voice encoding using the Global Standard for Mobile communications
viterb00	EEMBC	Viterbi decoder

from increasing the integer and address register files. Section 7.4 looks at the number of cycles added due to scheduler-unaware grouping restrictions.

7.1 Unrolling and Modulo Scheduling

The first comparisons were made between loop unrolling and modulo scheduling. Figure 7.1 shows the benchmark results for three configurations. The first bar represents Lcode with loops unrolled up to two times. The second bar represents Lcode with loops unrolled up to six times (analogous to the machine width). The final bar represents modulo scheduled code. Any loops that were not modulo scheduled could be unrolled up to six times.

Each of these speedups is relative to optimized Lcode and represents code that was run through IMPACT's superscalar optimization suite. The benchmarks show an average

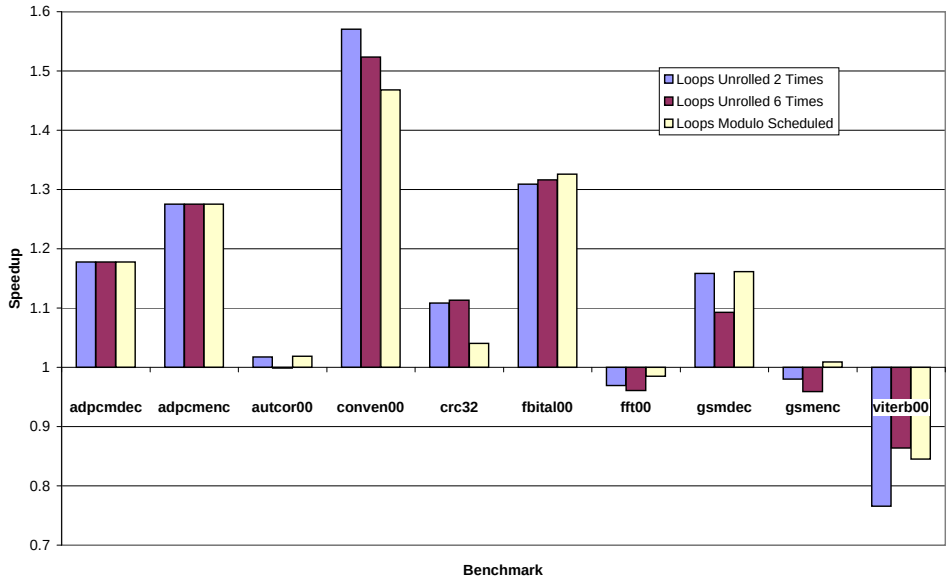


Figure 7.1: Speedups of superscalar-optimized code over classically-optimized code using various loop optimization techniques.

speedup across all configurations of 1.13, with only a 1% difference between configurations.

The best performing benchmark is conven00, which shows an average speedup of 1.51. The main hot-spot of this benchmark occurs in an inner loop which checks an array value. If that value is nonzero, the program will load two more array values, XOR them together, and store back that result. According to profile information, the value is nonzero about half of the time, and the loop will only execute three times before falling through. When run through the block former, this region becomes a frequently-executed hyperblock.

So, although this is a program hot spot, the loop actually has a low trip count for each instance. This accounts for the performance degradation when unrolling six times

since instructions from the final three iterations will be speculatively executed before the loop exit. For the modulo scheduled loop, the overhead costs involved, including the prologue and epilogue code, will dominate the execution since the kernel is only executed once in each instance of the loop.

The benchmark with the worst performance is viterb00, which shows an average slowdown of 17%. This is due to a combination of register spilling and inefficient grouping in the main computation loops. The register pressure will be explained in Section 7.3 and the amount of grouping inefficiency is shown in Section 7.4.

7.2 Available Predicate Registers

One seemingly large restriction on the SC140 is the existence of only one predicate, the T-bit. Since instructions can be predicated on the true and false conditions of the T-bit, it is analogous to having two predicates with each being set to opposite values on comparisons. Since the SC140 only has the T-bit, only simple hammocks can be formed by the hyperblock former. If the number of predicates in the architecture were increased, more complex regions could be combined together and optimized.

Figure 7.2 shows the speedups of superscalar-optimized code using additional predicates. The case of two predicates is close to the T-bit on the SC140, except that the predicates do not have to contain opposite values at all times. These results are gathered using profile weights and scheduling information. Thus, calls to library code are not accounted for in these numbers.

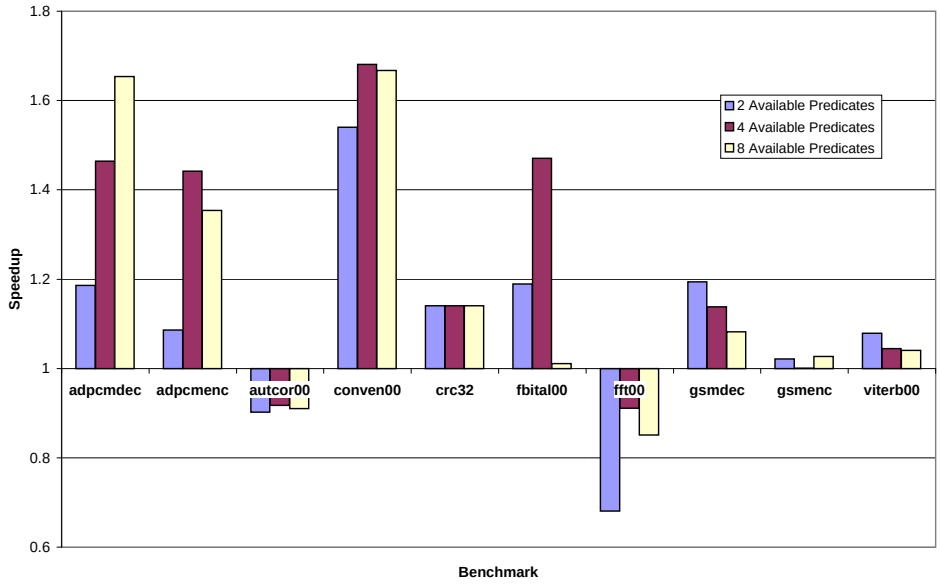


Figure 7.2: Speedups of superscalar-optimized code over classically-optimized code using additional predicate registers.

Most of the benchmarks studied benefit from an increased number of predicate registers, with average speedups of 10%, 22%, and 17% for 2, 4, and 8 predicates, respectively. It is interesting to note that performance decreases when going from 4 to 8 predicates. This is generally due to the hyperblock former creating regions that become too large for the SC140. The register pressure created in these regions causes the performance to suffer.

Slowdowns in the figure are also caused by large inefficient hyperblock regions being formed. The benchmarks that show slowdowns in this static analysis also show poor performance in Figure 7.1. These results, however, can differ from actual simulation results due to library code and the illegal groupings discussed in Section 7.4.

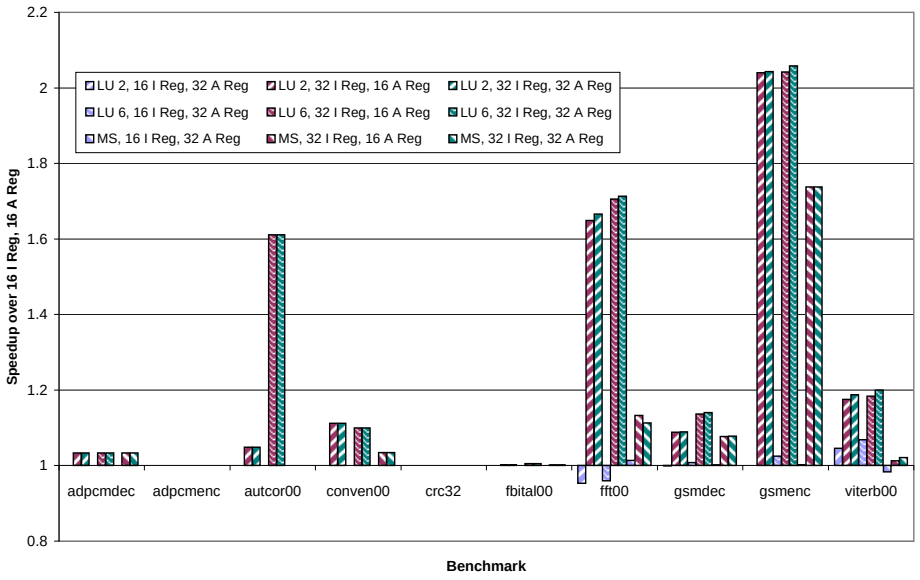


Figure 7.3: Speedups of superscalar-optimized code with an increased number of registers over the same code with 16 integer and 16 address registers.

7.3 Register Files

One of the biggest side effects of superscalar optimizations on the SC140 is the increased amount of register pressure. With only 32 general-purpose registers, split into 16 entry data and address sections, critical sections of code often contain more live ranges than available registers. In these cases, the register allocator is forced to add spills and fills to the code. This has a large impact on performance, especially in modulo scheduled loops.

To quantify the effect of this increased register pressure, experiments were run using the same optimization configurations, but with increased register files. Figure 7.3 shows the results of this for three combinations of register files: 16 integer and 32 address, 32 integer and 16 address, and 32 integer and 32 address.

Benchmarks with a significant amount of register pressure in critical loops show the biggest speedups in this experiment. Also, the integer registers are spilled much more often than address registers. The average speedup when increasing the number of integer registers from 16 to 32 is 20%, but only 0.2% for increasing the number of address registers by the same amount. Unrolling loops six times causes the most register pressure, as would be expected. Modulo scheduling a loop tends to cause less register pressure than unrolling that same loop twice. This is caused by a fundamental difference in scheduling techniques. The acyclic scheduler, used on unrolled code, uses a forward list scheduling technique, which tends to tends to push a lot of code to the beginning of the schedule. This will generally increase the live ranges of its variables. The modulo scheduler, however, uses reverse list scheduling which tends to move instructions down in the schedule. This will lead to shorter live ranges and less register pressure.

The benchmark that shows the most benefit from larger register files is gsmenc, with a maximum speedup of 2.07. The register pressure in this benchmark is almost completely on the integer registers, as shown in Figure 7.3. The average speedup for this benchmark by increasing the integer register file from 16 to 32 entries is 1.94. Increasing the number of address registers provides little improvement. This register pressure explains the benchmark's slowdown from superscalar optimizations shown in Figure 7.1.

An interesting observation is that some benchmark performance will decline when increasing the number of address registers. This occurs in cases where the integer registers are the only ones being spilled. Increasing the number of address registers does not eliminate any code, and simply causes the register allocator to produce a larger set of

live ranges. For a given subroutine call, more address registers are now live, causing an increased amount of spill and fill code to be inserted around the call.

7.4 Grouping Restrictions

As discussed in Chapter 5, there are several situations where a restriction that is required on the SC140 cannot be properly encoded in the IMPACT mdes. In these situations, the scheduler will produce an illegal VLES that will cause the assembler to punt. To avoid this, checking code in phase 3 of the code generator will detect these illegal groupings and splits the VLES to preserve correctness.

Figure 7.4 shows the percentage of benchmark cycles that are due to these restrictions being handled inefficiently. To determine these numbers, the scheduling attributes and profile weights are used to determine what the total cycle count should be. Each time a VLES needs to be broken up in phase 3, one cycle is added to the current control block's execution time. Using the profile weight, the additional cycles added in phase 3 are determined.

On average, 9% of the dynamic cycles in IMPACT-generated code is due to the scheduling inefficiencies. Unrolling the loops only twice fared slightly better than loops that were unrolled six times, due to the increased code size. The viterb00 benchmark showed the largest amount of inefficiency, with 19% of its cycles caused by the restrictions. This is the main reason that overall performance suffered for the benchmark with superscalar optimizations (see Figure 4.3). The slowdown associated with this particular

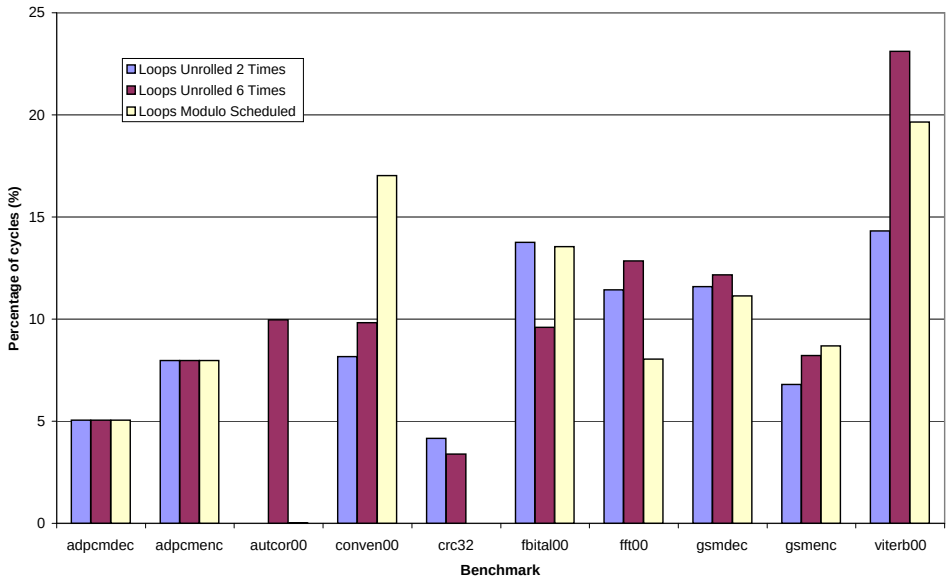


Figure 7.4: Percentage of statically determined cycles caused by breaking up restriction violations after scheduling.

problem, along with the previously discussed register pressure, completely accounts for the benchmark's average slowdown of 17%.

CHAPTER 8

FUTURE WORK

Although much was accomplished in creating this code generator, there are many nontrivial extensions that can still be made. These extensions are mostly needed to enhance performance since the primary goal of this work was functionality, which proved to be a valuable first step.

Some enhancements have been described earlier in this work, as in the additions to the mdes. The others will be briefly described in the following sections. Each of these extensions are current barriers to achieving performance comparable to the StarCore C compiler.

8.1 Hardware Loops

An extremely important aspect of any program is its performance when performing loops. This is especially true for DSP applications as they tend to have more computation time and less control flow. The SC140 supports efficient execution of loops through a fully optimized hardware looping mechanism.

The hardware supports these loops by maintaining four pairs of special registers. Each pair contains a loop start address register and a loop counter register. Specialized

assembly directives are provided to facilitate the easy setup of these loops. The hardware can handle up to four levels of nesting and arbitrary length loops.

In order to implement hardware loops in the code generator, loop boundaries would have to be easily identifiable in Lcode. While this only works for counted loops, the iteration count can be a dynamic value held in a register. To make this identification and conversion easier, these loops may be better identified and marked early in the compilation process. Since the PtoL conversion module marks the Lcode with some primitive loop identifiers now, this might be a good place for the analysis to happen.

8.2 Intrinsics

Another important feature that must be implemented properly to achieve performance is the use of intrinsics. Intrinsics are a way for the programmer to signal to the compiler that a certain hardware-supported instruction needs to be used. These specialized instructions are then converted to the proper Mcode instruction in the code generator.

These intrinsics need to be added to IMPACT on a case by case basis. This process would involve the hand-optimization of an application as well as the implementation in the compiler. There are, however, already several obvious opportunities for this to be used. For example, the SC140 has specialized instructions that are especially designed for the Viterbi decoder algorithm. It would likely prove too difficult to automatically identify situations where these instructions could be inserted directly into the existing Lcode. Instead, the original source could be annotated with intrinsic function calls that will later by replaced by the specialized instructions.

8.3 Machine-Specific Optimizations

Many other optimization opportunities will likely arise after careful analysis of the performance of specific applications. One likely area will be in finding opportunities to utilize more of the SC140 instruction set since there are many instructions that are not currently being used. These peephole optimizations may prove quite beneficial in certain situations.

Analysis of the register utilization may also prove valuable as an optimization opportunity. Since the registers are split up between data and address values, there may be better ways to take advantage of this fact. If the register usage is greatly slanted towards one over the other, as some results have shown, optimizations to convert some operations to the opposite side could be useful. Also, a minimization of the number of cross register file moves would also be helpful. Right now there are several of these move instructions, one for every time an operand value is in the wrong register file for a given operation.

CHAPTER 9

CONCLUSION

This thesis described a code generator developed for the IMPACT C compiler targeting the StarCore SC140. This code generator was designed to easily quantify the benefits of advanced ILP optimizations on an embedded DSP core.

The code generator itself provided several unique challenges due to architectural features found on the SC140. While many of these were successfully dealt with, some will require future changes to the IMPACT framework. For example, deficiencies in the current mdes were shown to cause a 9% increase in IMPACT-generated clock cycles. Any remaining features of the chip, including hardware loops and unused opcodes, can be easily incorporated into the code generator at a future time.

Results were generated for several realistic benchmarks using several compilation configurations. It was found that IMPACT's set of superscalar optimizations improved performance by about 13% on the SC140. A performance increase of 22% could be achieved by having four predicates available instead of one. Also, register pressure proved to be key to the performance of IMPACT's optimizations. A 20% improvement was gained by increasing the integer register file from 16 to 32 entries.

As the applications run on embedded processors become more complex and diverse, the architectures of these machines will begin to look like that of a general-purpose processor. With this general trend comes a greater emphasis on the efficiency of the

compiler's code generator. While there are some performance bottlenecks associated with applying superscalar optimizations to an embedded processor, each can easily be overcome with a modest amount of additional hardware.

REFERENCES

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

[2] StarCore, *SC140 DSP Core Reference Manual*, April 2001.

[3] B. C. Cheng, "A profile-driven automatic inliner for the IMPACT compiler," M.S. thesis, University of Illinois, Urbana, IL, 1997.

[4] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

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

[6] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[7] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[8] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, November 1992, pp. 808–817.

[9] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, "HMDES version 2.0 specification," IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-03, 1996.

[10] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, January 1993.

[11] B. R. Rau, "Iterative modulo scheduling," *International Journal of Parallel Processing*, vol. 24, pp. 3–64, February 1996.

[12] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code generation schema for modulo scheduled loops," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 158–169.

[13] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 138–150.

[14] D. A. Connors, J.-M. Puiatti, D. I. August, K. M. Crozier, and W. W. Hwu, "An architecture framework for introducing predicated execution into embedded microprocessors," in *Proceedings of the 5th Annual Euro-Par Conference*, August 1999, pp. 1301–1311.

[15] J. W. Sias, H. C. Hunter, and W. W. Hwu, "Enhancing loop buffering of media and telecommunications applications using low-overhead predication," in *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001, pp. 262–273.

[16] EDN Embedded Microprocessor Benchmark Consortium, March 2002, http://www.eembc.org.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 2001 International Workshop on Workload Characterization*, 2001, pp. 3–14.