IA-64 CODE GENERATION

by

VIKRAM S. RAO

a thesis submitted to the Graduate Faculty of

North Carolina State University

in partial fulfillment of the

requirements for the Degree of

Master of Science

Electrical and Computer Engineering

RALEIGH

JUNE 2000

APPROVED BY:

CHAIR OF ADVISORY COMMITTEE

Abstract

Vikram Rao. IA-64 code generation. (Under the direction of Dr. Tom Conte).

This work presents an approach to code generation for a new 64-bit Explicitly Parallel Instruction Computing (EPIC) architecture from **Intel**, called IA-64. The major contribution of this work is the design of a machine independent optimizer, **munger**, that transforms code generated originally for a Very Long Instruction Word (VLIW) processor, called Tinker, to one that can run on the IA-64 architecture. The munger does this transformation by reading in a set of rules that specify a mapping from Tinker specific code to IA-64 specific code. The aim is to do this transformation outside the compiler back-end, thereby being able to take advantage of any optimizations that the back-end might perform on the code. This would also preclude rewriting the existing back-end significantly, to support the new architecture. The primary motivation for this approach was the fairly large similarity between the Tinker, and the IA-64 architectures. Besides, Tinker is an experimental VLIW architecture that supports a number of features to exploit instruction level parallelism (ILP) and can be easily extended to support new features. This makes the back-end for Tinker an ideal compiler to retarget for the IA-64 architecture, since it already performs most ILP optimizations that are supported on the IA-64.

Biographical sketch

Vikram Rao was born on February 15, 1976 in Tamil Nadu, India. He graduated with a 5-year integrated Master of Technology degree in Electrical Engineering, from the Indian Institute of Technology, Bombay, India, in August 1998. During his years as an undergraduate, he participated in the cooperative education program working with the Electronics and Radar Development Estabilishment (LRDE), Bharat Electroncs Ltd., and Silicon Automation Systems, all in Bangalore, India.

He then went on to the master's program at North Carolina State University working under the direction of Professor Thomas M. Conte. While working toward his master's degree, Vikram spent a summer interning with the compiler development group at BOPS Inc., in Raleigh, North Carolina. He plans to take up a job in the area of compilers and microarchitecture.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most compilers perform lexical, syntactic, and semantic analysis, code optimization, and code generation, in this order [1]. Code optimization includes classical optimizations such as common subexpression elimination, constant folding, code motion, and strength reduction [1]. The code generator of a compiler transforms programs from a compiler's intermediate representation into assembly language or binary machine code. Code generation can be divided into three phases: *Code selection* translates the operations in the intermediate representation to target machine instructions; *instruction scheduling* orders the instructions to make the best use of processor resources; *register allocation* replaces pseudo-registers used in the intermediate representation with real registers and spills excess pseudo-registers to memory.

Another important aspect of a back-end compiler is the ability to retarget it easily, *i.e.*, to generate code for a different processor. If the target processor has an architecture similar to the original processor, it is possible to retarget the compiler to generate code for the new processor, without making significant changes to the existing back-end. This could be achieved by making all machine dependent transformations outside the back-end and only making changes in the back-end needed to integrate it with the machine dependent part.

Machine dependent changes might include translating instructions, generated for the original architecture, to operations that will run on the new architecture or that have an equivalent operation in the new instruction set. The changes may also include merging one or more instructions in the original code stream into a single instruction. Another possible transformation is a change of the operand formats. Clearly these changes can be made outside the back-end compiler and information about these changes can be passed to the scheduler module within the back-end, through the use of a machine description file.

In the particular case of one-to-many mapping of instructions, VLIW/EPIC architectures are especially suitable, since some of the IR expansion may be hidden by effective global scheduling, *i.e*, the additional instructions generated by the mapping may be scheduled in empty slots in the VLIW multi-ops, thereby hiding their latencies. Even though instructions are added to the instruction stream the performance impact may be minor. The following section identifies some of the key ideas and the motivation for the work presented in this report.

1.1 Motivation

Retargetability is of importance in a back-end compiler since the number of machines is large and it is not feasible to design a new compiler for each one. This work attempts to retarget a back-end compiler designed for a VLIW processor called Tinker, to the IA-64 architecture. The VLIW computing paradigm has recently made a come back with the IA-64 architecture, which is a VLIW style architecture that is being designed currently by **Intel**. My work is largely motivated by the fact that the IA-64 architecture depends heavily on compiler support for most of its optimizations, and also due to the fact that there is a large similarity between the Tinker and the IA-64 architecture. Since the existing compiler back-end for Tinker implements a number of ILP optimizations such as speculation, predication, and modulo scheduling, which are also supported by the IA-64 architecture, it is an ideal candidate for retargetability.

The similarity in the architectures allows the retargeting to be done without making significant changes to the back-end. The existing back-end uses a fairly machine independent intermediate representation (IR) called **Rebel**. All ILP optimizations in the back-end, which are common to the two architectures, are performed on Rebel code. Since the munger uses Rebel as its input and output formats, it is independent of the back-end. The differences between the two architectures are communicated to the back-end through a machine description file.

The contribution of my thesis is an IA-64 code generator composed of (1) a machine description of the Itanium processor (an implementation of the IA-64 architecture) and (2) a rule-based peephole optimizer, called the **munger**, which translates Tinker specific operations to IA-64 compatible operations. These two modules have been integrated within the Tinker compiler to generate code for the IA-64 architecture. This work focused primarily on the ease of retargetability of the back-end compiler rather than performance of the resulting IA-64 compatible code.

1.2 Compiler phases

The work in this report leveraged the existing back-end compiler (called LEGO [14, 15]). The LEGO back-end compiler accepts intermediate code generated for a VLIW architecture, called **Tinker** [2].

The phases for converting C source code to IA-64 assembly code are shown in Fig. 1.1. Their correspondence to conventional compiler passes is described below,

1. The C source code is first fed through the **IMPACT** front-end. IMPACT is an optimizing ILP compiler that was developed at the University of Illinois. It generates an intermediate code called *Lcode*. This pass basically involves parsing the high-level language and performing classical optimizations on the code to generate an intermediate three-address code.

2. A non-optimizing pass, called **Elcor**, translates the Lcode into Rebel. This pass

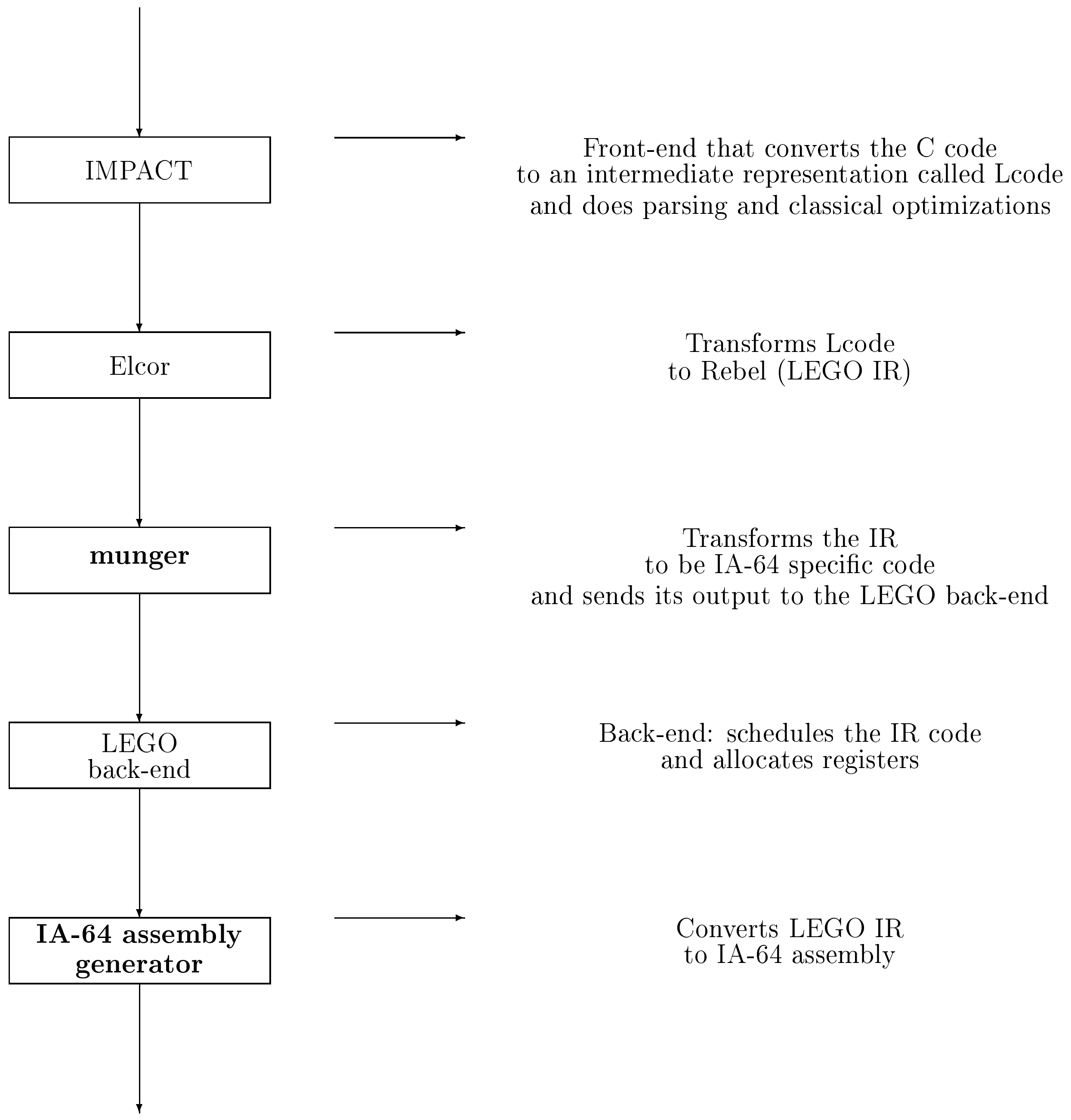
IMPACT
Front-end that converts the C code to an intermediate representation called Lcode and does parsing and classical optimizations
Elcor
Transforms Lcode to Rebel (LEGO IR)
munger
Transforms the IR to be IA-64 specific code and sends its output to the LEGO back-end
LEGO back-end
Back-end: schedules the IR code and allocates registers
IA-64 assembly generator
Converts LEGO IR to IA-64 assembly

Figure 1.1: The phases in the process of IA-64 code generation

has no analog in a conventional compiler and is present here simply because LEGO only accepts the IR that Elcor produces.

3. The Rebel code is transformed to an IA-64 specific Rebel code by passing it through the munger. Thus the IA-64 code generation portion is actually independent from the rest of the back-end compiler as previously mentioned. This pass is also non-existent in conventional compilers.

4. The munged code is then passed through the LEGO back-end compiler for scheduling and register allocation. The LEGO back-end may apply many different ILP optimizations such as global scheduling, speculation, predication and if-conversion, operation combining, and multi-way branching. The scheduled code may then be passed through an assembly code generator and eventually linked to form machine code. This pass is similar to the back-end of most conventional compilers and involves code scheduling and register allocation.

5. The scheduled Rebel code is passed through an IA-64 assembly code generator. This pass is usually a part of the back-end in conventional compilers and isn't a separate pass as shown here.

The remainder of the report is organized as follows. Firstly, a brief overview of existing code generation methods is described. This chapter describes three approaches to code generation and also talks about peephole optimization as an alternative to conventional code generation. Secondly, I briefly introduce the salient features of the IA-64 architecture. This chapter describes features of IA-64 which support various ILP optimizations and also shows in what ways the Tinker architecture is different. The munger is then described in detail. This chapter describes the algorithm used by the munger and also the syntax of the mapping rules. Finally some results and conclusions are presented. This chapter presents statistics about the performance of the munger, specifically the amount of expansion in the IR, that is caused by the mapping of single instructions into sequences of instructions. It also discusses the most commonly used mapping rules and their causes. The munged code has not been validated by running it on a real Itanium workstation or simulator as yet. That is part of the future work.

Chapter 2

Prior work

There has been a lot of work done in the area of compiler back-ends, especially on code generation and retargetable code generation. Since the sixties there have been primarily three directions that the research has taken: interpretive code generation, pattern-matched code generation and table-driven code generation [4].

1. **Interpretive code generation:** This approach generates code for a virtual machine and then attempts to expand the code to one for a "real" architecture. Code generation is "macro"-like. The machine description is hard coded into the code generation algorithm. Such code generators are usually hand coded and informal [4].

   There are two sub-classes of interpretive code generators:

(a) Hand-written interpreters that implement a one-to-one or one-to-many mapping between the virtual machine and the real-machine instructions. This class requires many code generator decisions to be made by the front end. Since all the mapping rules have to be written by hand, the author of the rules has to decide the optimal mapping.

(b) In the other class of code generators, translation of the abstract instruction stream to instructions for a real machine is done by interpreting the changes made to the abstract machine state and generating actual machine instructions which cause equivalent actions on the actual machine.

Interpretive code generators suffer from a number of limitations [4]. Firstly, the diversity in addressing modes, target machine data types, and instructions, makes it very hard to anticipate a variety of machine organizations in one virtual machine. Secondly, code generation languages are closely tied to a specific language or machine, so they cannot be considered fully portable. The description of the target machine is hard coded into the code generation algorithm. This forces the code generation algorithm to be changed when the machine description changes. Finally, the implementer has to write the mapping rules, which may result in sub-optimal mappings.

2. **Pattern-matched code generation:** This approach tries to avoid the retargetability problems of the interpretive code generation approach. It separates the machine description from the code generation algorithm. Depending on the choice of instruction patterns, pattern matching may be performed either by heuristic search or parsing.

   The idea is to encode the machine characteristics into a pattern tree. The code generator algorithm then traverses a parse tree from the source language and tries to find a suitable match for it in the pattern tree [4].

   Such an approach was taken by Aho *et al* [5], where they use a dynamic programming algorithm to derive optimal code sequences for register machines. This approach only considers arithmetic expressions though. A related work by Aho *et al* [6] describes a tree-manipulation language called *twig* which is used in conjunction with the dynamic algorithm described above [5], to build fast and efficient code generators. A *twig* program is a set of pattern-action rules with a syntax as shown below [6]:

   label:*pattern*[{*cost*}][={*action*}]

   Label is an identifier analogous to the left-hand nonterminal symbol of a production, *pattern* is a parenthesized prefix expression representing a tree, *cost* is C code executed by the pattern matcher when it finds a subtree matching the tree pattern, and *action* is also C code that is executed by the matcher when it is determined that this rule is part of the minimal cost cover.

Although pattern-matched code generation deals with the problem of separating the code generation process from the machine description, it suffers from a limitation. It is very hard to create a single tree structure to encode all potential instruction patterns [4].

3. **Table-driven code generation:** Table-driven approaches try to automate the process of generating code generators. They use a formal machine description and a code-generator generator. These approaches break down the problem of code generation into two parts: register allocation and instruction selection. In previous approaches described above, the code generator usually tries to do register allocation too.

   The work by Graham and Glanville [7], made a significant breakthrough in automating the code generation phase in compilers. Their work assumes a very low-level IR in the form of prefix expressions. Their code generator then generates assembly/machine code from this low-level IR. Their algorithm uses a method similar to that used during parsing source code and the machine instructions are also described in the prefix notation. The idea is to perform a pattern match similar to source code parsing, translating the IR sequence of prefix expressions into assembly instructions. The complication of this algorithm over standard syntax analysis is that machine descriptions are usually ambiguous since operations can access their operands in many ways. Besides, the code generator has to select from a variety of instructions. One limitation of this approach is that a one-to-one mapping is assumed between the machine instruction templates and the target machine instructions. This assumes a lot about the IR and effectively requires the IR to be changed when targeting a new machine.

Other work by Gordon [8], involved writing a code generator for an established Coral 66 compiler for the Intel 8080/8085, the new code generator being targeted for Intel 8086. The optimizer used here was also table driven but the difference from the work by Graham *et al* [7] was that the table was hand-coded with macro-defined symbols for all the intermediate codes. Each item in the table had an original sequence of codes to be identified and a sequence to replace the original. This is similar in idea to the munger presented in this work but the munger only does either a one-to-many mapping or a many-to-one mapping of instructions. All other mappings (one-to-one) are specified through the machine description file. The basic idea behind this optimizer is that it would be recursive and work in parallel on upto four different layers or levels of tables. Each layer is chained to the next. Processing persists within each layer until no further action can be taken. The functions of the different layers is as follows:

- The first table expands intermediate code to more closely resemble the way that the target machine works. It also combines successive simple codes that can be handled by a single target instruction.
- The second table optimizes successive logical and arithmetic instructions and absorbs some manipulation of operand addresses into the addresses themselves.
- The third table inserts push or pop instructions needed if the top of the stack resides in a register in the target machine.
- The fourth table specifies optimizations to generate operations which can be performed directly on variables without using the stack or registers.

Yet another table-driven approach tries to separate the pattern-matching process and the pattern-selection process [9]. Here a code generator was developed called UNH-CODEGEN which takes two inputs, a machine description in YACC-like format and a set of C structure definitions for valid tree nodes. The output of the system is C source for a code generator. The basic idea is to avoid expensive cost analysis of the machine description at code generation time. This cost analysis is done while generating the code-generator. In essence, cost analysis examines all competing sequences of patterns (those that might match the same pattern tree) and removes those that are suboptimal from later consideration (at code generation). Some contextual information can also be passed to the code generator allowing some "goal-oriented" code generation. This is unique as compared to the system developed by Graham *et al* [7].

Besides these broad classes of code generators, there has been some research of very simple (naive) code generators and optimizing the resulting generated code using *peephole optimizers*.

One of the earliest works done in this area was by Fraser *et al* [10]. It describes an optimizer that operates on object code in order to replace certain sequences of instructions with better sequences. This peephole optimizer used a symbolic machine description to simulate pairs of adjacent instructions, replacing them with an equivalent single instruction. The symbolic machine description describes the syntax of an instruction and its effect (both explicit and implicit, such as the setting/clearing of condition codes). Once the optimizer knows the effect of individual instructions, it passes forward over the program and considers the effects of physically adjacent instructions; where possible, these pairs are replaced by a single instruction that has the same effect. The authors didn't find the need to combine more than a sequence of triples of instructions. More complex replacement strategies such as replacing a triplet with a sequence of two instructions, were also unnecessary. An implementation of the peephole optimizer in a real cross-compiler called “YC” has been done by Fraser *et al* [11].

A similar effort [12] uses a classical rule-directed peephole optimizer which runs after a naive code generation pass. The advantage of using classical peephole optimization over more portable methods [10, 11] above is speed. Classical optimizers are usually very simple and very quick. Yet, because the rules are hand-written, not all possible optimizations may be covered.

A hybrid of the two approaches [13] may exploit the speed advantage of classical peephole optimizers while retaining the more thorough approach of machine description driven optimizers [10]. The basic idea is to build a rule generator on top of a fast classical peephole optimizer. The retargetable peephole optimizer [10] simulates adjacent instructions and where possible replaces them with an equivalent single instruction. This optimizer is used to compile a large “training set” of rules by initially processing a large sample program with the rule-generator. More rules can then be inferred and added to the complete set of rules as the compilation progresses. Rules may also be pre-loaded from an earlier compilation thereby minimizing the amount of work that needs to be done by the retargetable optimizer.

Clearly, there are two primary directions which code generation research has taken, hand-written, and automated. The literature also shows, that, as a rule, while automated code generators are usually slower than hand-written ones, they are more thorough and produce more optimal code. The advantages of the two approaches can be combined if a naive code generator is used and the resulting code optimized using a peephole optimizer. This is the approach that my work has taken. The munger behaves like a peephole optimizer which optimizes code generated for the Tinker architecture, to run on the IA-64.

Chapter 3

The IA-64 processor architecture

The IA-64 architecture is **Intel's** 64-bit EPIC style architecture. It is designed with a number of new features to extract greater instruction level parallelism (ILP), such as speculation, predication, large register files, a register stack, advanced branch architecture, *etc.* These features may be categorized as those that support explicit parallelism and those that enhance parallelism [3].

3.1 Explicit parallelism

3.1.1 Conveying compile time information to the processor

The IA-64 architecture provides mechanisms, such as instruction templates which explicitly specify groups of independent instructions that can be executed simultaneously. An instruction template is a bundle of three instructions of a particular type (integer, float, memory, or multimedia). The number of templates are limited to thirty one, in IA-64. They don't cover all possible combinations of instruction types. In any implementation of IA-64 more than one bundle may be executed simultaneously. These bundles can be specified in the source code, by the compiler, using templates.

In addition, some features such as branch hints and cache hints, allow the compiled code to manage the hardware resources of the processor using run-time information. Every memory load/store operation has a 2-bit cache hint field which specifies the spatial and/or temporal locality of the memory area being accessed. The processor could use this information to determine placement of cache lines in the cache hierarchy. This leads to better utilization of the cache hierarchy thereby minimizing the penalties incurred by cache misses. Branch hints provide information to the hardware to improve branch prediction performance by suggesting how the hardware should predict a certain branch.

3.1.2 Multiple resources

ILP is the ability to execute multiple instructions simultaneously which implies the presence of parallel resources - multiple functional units and a large number registers. Both these are present in the IA-64 architecture. This architecture is inherently scalable since the instructions are issued in *bundles* of three instructions, and, depending on the implementation more than one bundle may be issued in a single cycle.

The number of registers also needs to be large due to the parallel nature of the programs running on the processor. The IA-64 architecture has 128 64-bit integer and floating-point registers, 64 1-bit predicate registers, and 8 64-bit branch target registers.

IA-64 also avoids the unnecessary spilling and filling of registers at procedure call and return interfaces. At a call site, a new frame of registers is available for the new procedure without need for register spill or fill (either by the caller or callee). Register access occurs by renaming the virtual register identifiers in the instructions through a base register into the physical registers. The callee can freely use available registers without having to spill and eventually restore the caller's registers. If there is an underflow or an overflow at a call or a return site respectively, the register stack engine (RSE) stalls the processor and manages all spills and fills to/from the memory.

3.2 Enhancing parallelism

3.2.1 Speculation

Both control and data speculation are supported by the IA-64 architecture. In both cases the compiler exposes ILP by issuing an operation early and removing the latency of this operation from the critical path. Thus the compiler can overlap instructions and tolerate their latencies.

Control speculation

Control speculation is the execution of an operation before the branch that guards it. When the compiler speculates an operation it leaves a check operation at the original location of the operation. This check verifies whether an exception has occurred and branches to recovery code if it has. An exception is represented in different ways for integer and floating-point operations. For general purpose registers, an extra bit called the *NaT*, or *Not a Thing*, is defined for each register. When this bit is set it means it was set by an operation that was speculated and caused an exception. These bits are propagated down a dependence chain. Thus, any operation that uses a register whose NaT bit is set also sets the NaT bit of its destination. If a floating-point instruction excepts, the destination register stores a specific pseudo-zero encoding called the *NaTVal* which represents a deferred exception.

Data speculation

Data speculation is the execution of a load prior to a store that precedes it and potentially aliases with it. Analogous to control speculation when a compiler speculates a load it leaves behind a load check instruction at the original location.

The check verifies if an overlap occurred between the load and the aliasing store instruction and branches to recovery code if it did.

When an advanced load is executed it allocates an entry in a structure called the Advanced Load Address Table (ALAT). Later when a corresponding check is executed, the presence of an entry in the table indicates that the data speculation was successful. An entry in this table is invalidated by a store or a load to the same location.

3.2.2 Predication

Predication is the conditional execution of instructions based upon the set/unset state of a predicate bit. Predication replaces branches used for conditional execution resulting in larger basic blocks and the elimination of associated mispredicts. The larger basic blocks in turn provide a greater scope to extract parallelism. A basic analysis of the predicates can easily detect disjoint sets of predicates thereby allowing instructions guarded by disjoint predicates to be issued simultaneously.

3.2.3 Software pipelining

Support for modulo-scheduling of loops is provided through rotating registers and special branches. Register support is provided through a fixed size area of of the predicate and floating-point register files and a programmable sized area of the general purpose register file.

3.3 Comparison with the Tinker architecture

The Tinker architecture is a VLIW processor. It is very similar to the IA-64 architecture. Tinker has support for all ILP optimization techniques that are supported in the IA-64 architecture, such as, speculation, predication, modulo scheduling, and parallel execution.

The primary difference between the two architectures is that Tinker allows any combination of instructions to issue simultaneously, as long as resource and dependency constraints are satisfied. This is unlike IA-64 where instructions are issued in bundles of three instructions. The allowed combination of instruction types in a bundle is fixed and has to be one of thirty one predefined templates.

The instruction sets of the two architectures are also fairly similar and most Tinker instructions have an equivalent in the IA-64 instruction set. Only about twelve instructions in the Tinker instruction set do not have a direct map onto equivalent IA-64 instructions. One difference between the two architectures that affects the performance of the munger, as is shown in Chapter 5, is that Tinker supports more operand formats than IA-64 in arithmetic and non-arithmetic instructions.

Chapter 4

Munger

4.1 Introduction

In order to leverage the LEGO optimizing back-end compiler, the Tinker specific Rebel code generated from the Elcor pass (Fig. 1.1) needs to be transformed into IA-64 specific Rebel code. As it turns out, the IA-64 instruction set is fairly similar to the Tinker instruction set [2, 3]. Most instructions in the Tinker instruction set can be directly (one-to-one) mapped to equivalent instructions in IA-64. The equivalence between these instructions and the Tinker instructions is described in the machine description of the IA-64 architecture. Some instructions may need to be expanded or mapped into sequences of Tinker instructions that have one-to-one mappings with equivalent IA-64 instructions. Conversely, by adding new (IA-64) instructions to LEGO itself, it is possible to detect sequences of instructions in the

Tinker specific Rebel that could be replaced by a single IA-64 instruction. This would optimize the Rebel code for IA-64 without making any changes to the optimizations that the back-end performs. The primary focus of this work was to generate code for the IA-64 architecture in the simplest way. Performance of the resulting code was not considered.

4.2 Structure of the munger

The munger is a peephole optimizer which reads in the code generated for an abstract machine (Tinker) and "munges" (transforms) it to generate code suitable to be scheduled on the IA-64 architecture. The munger was designed to be as machine independent as possible. It reads in two forms of inputs: one is the Rebel file that needs to be transformed; the other is a file that contains rules which specify how certain instructions from the Tinker instruction set have to be either expanded into equivalent sequences of instructions or how certain sequences of instructions have to be merged into a single instruction. These rules are stored in the form of an associative map, for quick lookup, with the key to the map being either the Tinker instruction that has to be expanded or the single IA-64 instruction that is formed by merging a sequence of Tinker instructions.

4.2.1 One-to-many mapping

The format for writing the rules for expanding a single instruction into a sequence of instructions is described below.

```
/
<Tinker op> destn1 destn2 source1 source2 source3 predicate
<Tinker op> destn1 destn2 source1 source2 source3 predicate

⋮

/
```

The complete rule for expanding a single instruction fits within two "/" symbols. Here, the first Tinker instruction that follows the "/" symbol is the instruction that needs to be expanded into a sequence of Tinker instructions. This sequence is defined by the remainder of the Tinker instructions in the rule above. An example of such a rule is shown below,

```
/
CMPP.W.FALSE.UN.UC  D1 D2 S1 S2 S3 PR
MOVE                G1 NULL 1 NULL NULL PR
CMPP.W.EQ.UN.UC D1  D2 GPR0 G1 NULL PR
/
```

All the operands are defined for each instruction in the rule. If some operands don't exist they are defined to be NULL. In the example above D1 is the first destination, D2 is the second destination. S1, S2, and S3 are the source operands and PR is the predicate operand. The example above transforms a single CMPP operation into two operations. The CMPP operation stands for "compare to predicate" and sets the value of two predicate destinations based on a comparison between the source operands. In this case the comparison operator is FALSE, which means the result of the comparison is always "0", irrespective of the source operands. The values stored into the predicate destinations further depend on the qualifiers, in this case UN (unconditional) and UC (unconditional complemented). A qualifier UN implies that the result of the comparison is stored as it is in the predicate destination, while UC implies that the result of the comparison is first complemented and then stored in the predicate destination. Thus in this example D1 gets a value "0" while D2 gets a "1".

This instruction gets mapped into a sequence of two instructions which moves the literal "1" into a register and compares it with zero, thereby always clearing the predicates. In many cases new operands need to be created while forming the rule. This is accommodated by specifying what kind of operand to create, using mnemonics. Thus, in the example above a new general purpose register (`gpr`) is created by the MOVE instruction, denoted by `G1`, which is further used by the CMPP.W.EQ.UN.UC instruction. This sequence of *defines* and *uses* is also kept track of in the munger by using an associative map. A list of mnemonics that have been currently defined is shown in table 4.1.

The algorithm to perform this one-to-many mapping is described below. The rules to perform the mapping are initially read and stored in the form of an associative map, with the first instruction in the rule being a key. Then, for each procedure in the Rebel code module, the mapping function is called on each region. A region can be one of three types

- Treegion: A treegion is a tree-like region with multiple basic blocks or hyper blocks in it. A treegion contains no merge points by definition.
- Hyper block: A hyper block is a large region formed by combining basic blocks through predication. It has a single entry point and multiple exit

Mnemonic	Operand type
NIL	NULL operand
D1	Original destn1
D2	Original destn2
S1	Original source1
S2	Original source2
S3	Original source3
PR	Original predicate
Gn	New general purpose register
Fn	New floating point register
Pn	New predicate register
Bn	New branch register
Cn	New control register
CBn	New control block
U	New undefined operand
n	New int/float literal
GPRn	general purpose register with the specified number “n”
FPRn	floating point register with the specified number “n”

Table 4.1: Set of mnemonics used for creating operands.

```
/* Map the instructions in a procedure */
int
mapper(proc, n)
{
   for(int i=0;i < num_regions_in_proc;i++)
   {
      // If the region is a treegion....
      if(region_type == ``treegion'')
         for(int j = 0;j < num_subregions;j++)
            n = mapper(sub_region, n);

      // If the region is a basic-block or hyper-block....
      if(region_type == ``basic-block'' || region_type == ``hyper-block'')
         n = mapper(region, n);
   }

   return n;
}
```

Figure 4.1: Pseudo-code to perform one-to-many mapping at a procedural level

points.

- Basic block: A basic block is a contiguous section of code with a single entry point and a single exit point.

If the region is a treegion, the mapper function is called again for each basic block or hyper block in the treegion. "n", below, is an integer that is used to provide a unique index to each new operation created. The pseudo-code for this is shown in Fig.4.1,

The mapping function for each basic block looks at each instruction starting from the top of the basic block, and searches the set of rules to see if that instruction needs to be expanded. If it does, then that instruction is expanded as per the rules otherwise the next instruction is processed. Once the instructions have been translated, the order of operands in each instruction is checked to see that it is in accordance with the allowed addressing formats in the IA-64 instruction set architecture (ISA). The pseudo-code for this is shown in Fig.4.2,

4.2.2 Many-to-one mapping

The rule format for merging a sequence of instructions into a single instruction is very similar to the one-to-many case and is described below.

```
%
<Tinker op> destn1 destn2 source1 source2 source3 predicate
<Tinker op> destn1 destn2 source1 source2 source3 predicate

⋮

%
```

The complete rule for merging a sequence of instructions fits within two "%" symbols. Here, the first Tinker instruction that follows the "%" symbol is the single instruction obtained by merging a sequence of Tinker instructions. This sequence is defined by the remainder of the Tinker instructions in the rule above. An example of such a rule is shown below:

```
%
BR NULL NULL S1 NULL NULL PR
PBRR  D1 D2 S1 S2 S3 PR
BRU NULL NULL D1 NULL NULL PR
%
```

In the example above, a new instruction BR has been added to the Tinker instruction set. It is a combination of two Tinker operations, a PBRR instruction and a BRU instruction. A PBRR is a "prepare to branch" instruction that loads the branch address into a branch target register. A BRU is an unconditional branch,

```
/* Map the instructions in a region */
int
mapper(region, n)
{
   // If the region is a basic-block or hyper-block....
   if(region_type == ``basic-block'' || region_type == ``hyper-block'')
   {
      // Initial mapping
      op = first_op_in_region;
      while(op)
      {
         // Expand "op" if necessary.
         n = translate_op(op, n);
         op = next_op;
      }

      // Check for erroneous op-formats, and correct them
      op = first_op_in_region;
      while(op)
      {
         n = operand_chk(op, n);
         op = next_op;
      }
   }

   return n;
}
```

Figure 4.2: Pseudo-code to perform one-to-many mapping at a basic-block level

which branches to the location specified by the branch register in its source operands. This is easily expressed through the rule shown above. In the many-to-one case there is no need to create new operands. This is because the single instruction which is formed will take its sources from the instruction sequence being merged and its destination from the last instruction in the sequence.

The algorithm to perform a many-to-one mapping is described below. The rules have already been read in before the merging is attempted and those rules that are specific to merging are stored in an associative map separate from the one for performing the expanding of an instruction into instruction sequences. Then, for each procedure, the merge function is called on each basic-block region. The pseudo-code for this is shown in Fig.4.3,

The merging function for each basic-block first generates the DAG (directed acyclic graph) for the region. This DAG contains all data dependence information of the region being considered. Then for each pattern specified in the rules it searches through the DAG for the first instruction in the pattern. If the first instruction is found, it goes on to search for the rest of the pattern in the DAG, in a recursive depth-first manner. The use of the DAG to search for sequences of instructions, automatically ensures that only valid sequences of instructions are found *i.e* only those sequences which contain the definitions as well as uses of variables. This is guaranteed because, by definition, a DAG stores dependence chains, which means every instruction which defines a value is connected to all other instructions which use this value in the future. This pattern of instructions is then replaced by a single instruction in the Rebel code. While replacing this pattern of instructions it is important to ensure that if any destination registers generated in the instruction sequence are live outside this sequence, they have to be re-generated before they are used. There may be other

```
/* Merge insn's in a procedure */
int
merge (proc, n)
{
   /* call merge() for each region */
   for(int i=0;i < num_regions_in_proc;i++)
   {
      // If the region is a treegion....
      if(region_type == ``treegion'')
         for(int j = 0;j < num_subregions;j++)
            n = merge(sub_region, n);

      // If the region is a basic-block or hyper-block....
      if(region_type == ``basic-block'' || region_type == ``hyper-block'')
         n = merge(region, n);
   }

   return n;
}
```

Figure 4.3: Pseudo-code to perform many-to-one mapping at the procedure level

potential advantages of using a DAG to search for patterns, but these haven't been investigated in this report.

Every block may have more than one merges possible. The merges are performed on the basic-block in the order in which the rules are read in. This ordering of the rules may impact code size since, some rules may overlap with other rules. Thus, the larger the number of instructions that a rule merges, the sooner it should be used. This would result in the smallest code size. Code performance hasn't been considered here and may even worsen depending on resource constraints. If the resulting merged instruction requires a particular functional unit, which isn't available in large numbers, the code performance may actually go down. The pseudo-code for this is shown in Fig.4.4

```
/* Merge instructions in a basic-block */
int
merge(region, n)
{
   // If the region is a basic-block or hyper-block....
   if(region_type == ``basic-block'' || region_type == ``hyper-block'')
   {
      /* form the DAG for the region */
      BuildDagForRegion(region, machine, Knobs);

      // for each pattern in the ``merge_rules'' associative map...
      for (all_patterns)
      {
         Front = find_first_op_in_pattern();
         match = getMatch (Front, vec+1);
         replace_sequence_with_single_op();
      }
   }

   return n;
}
```

Figure 4.4: Pseudo-code to perform a many-to-one mapping at the basic-block level

Chapter 5

Results and conclusions

The munger was run on six benchmarks in the SPEC95 benchmark suite. These are *go*, *m88ksim*, *compress*, *li*, *ijpeg*, and *vortex*. Each of these benchmarks was "munged" to produce code suitable for scheduling on the IA-64 architecture. This code was then scheduled using the LEGO back-end compiler with most ILP optimizations turned on , such as global treegion scheduling, speculation, operation combining, and multi-way branching. The scheduler obtained machine specific information from a machine description of the Itanium processor, procured from **Intel**.

5.1 Results

The primary aim of the munger is to generate code for IA-64 *without* rewriting the back-end portion of the compiler. A number of statistics were collected to verify that the IR expansion due to the one-to-many mappings is not excessive. A IR expansion weakens the effects of any optimizations that may be performed by the back-end compiler. The percentage increase in code size for each benchmark is shown in Fig.5.1. From this figure, it may be seen that the average IR expansion is about

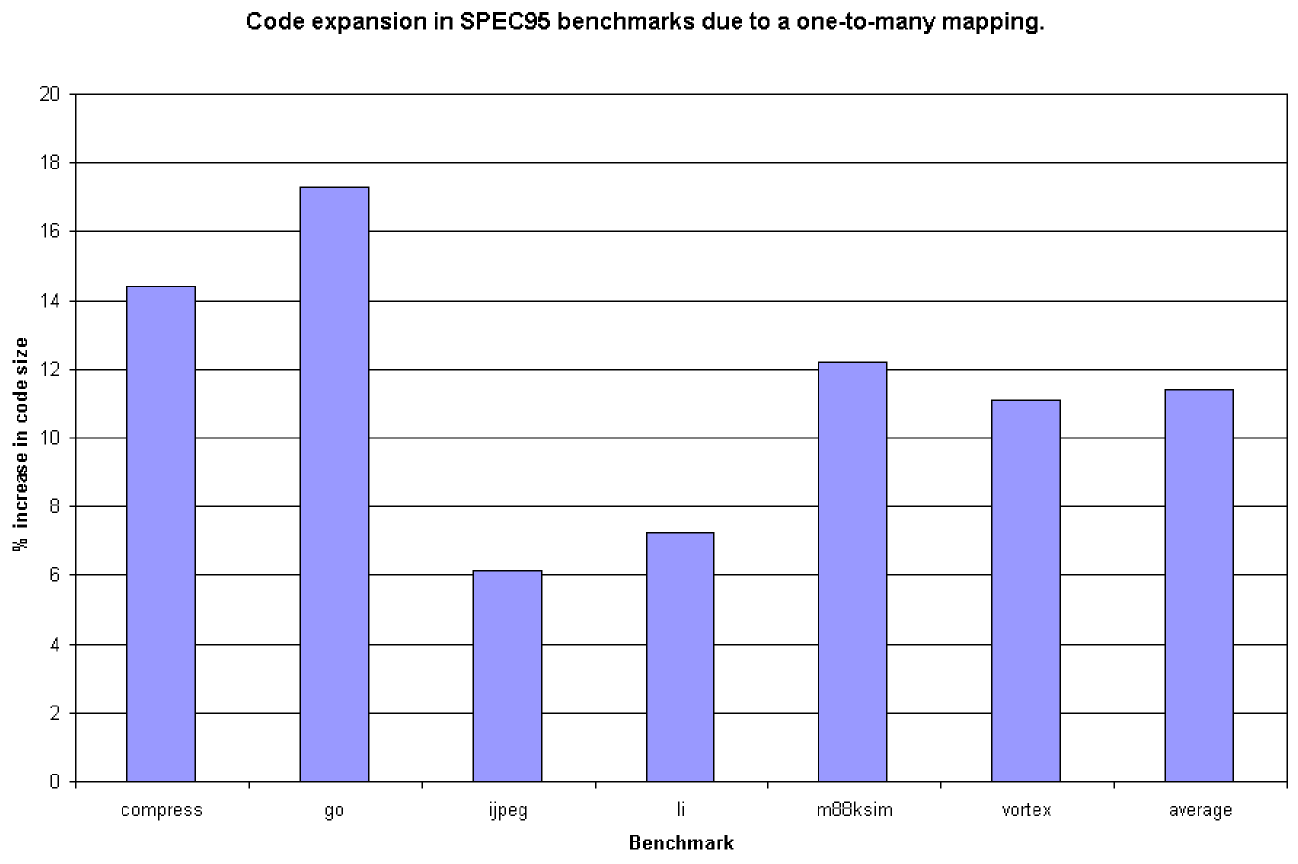

Figure 5.1: Percent increase in code size in the SPEC95 benchmarks.

11% and the maximum IR expansion occurs in the **go** benchmark, where the code grows by about 17%.

New operations are introduced into the code for a number of reasons besides the specified one-to-many mapping rules. Some of the possible reasons for the addition of operations are mentioned below:

1. The register allocator currently doesn't recognize special purpose registers like GPR0, FPR0, FPR1 and PR0, which store fixed values in them and cannot be written to. This necessitates the addition of a MOVE instruction to store

this fixed value into the designated register before actually using these special registers.

2. The scheduler currently doesn't support predication in its most general form, *i.e*, where a single conditional branch with short paths can be eliminated by predicating the instructions on both the true and false paths. As a result, some mapping rules which could have been written much more effectively by exploiting predication now need to be written using conditional branches and adding new basic-blocks.

3. The IA-64 architecture doesn't allow the use of (application) control registers as source registers in all but a few instructions, whereas Tinker does. This situation also has to be remedied by first adding a MOVE instruction with the control register as the source and a general purpose register as the destination, followed by an analogous MOVE after the actual operation is done, to store the result back into the control register.

4. The same situation as above applies to address labels in the code too.

5. The operand formats in IA-64 integer ALU instructions don't allow the use of a literal as the first operand in the set of source operands. This situation is easily remedied by interchanging the two operands for commutative operations. For non-commutative operations, on the other hand, the source operands cannot simply be interchanged since that might alter the result. Instead a MOVE instruction needs to be added before the original operation to store the literal into a register.

As can be seen from the list above, a lot of additional instructions are added

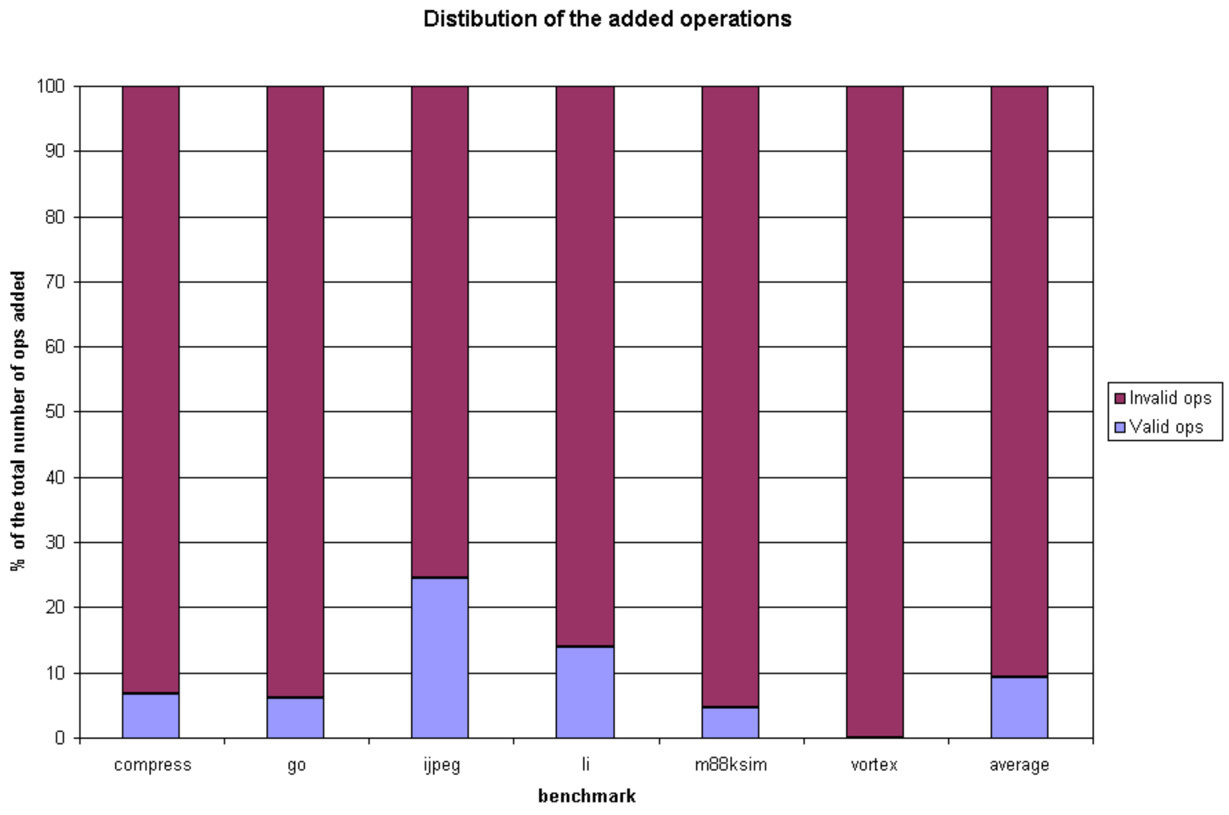

Figure 5.2: Distribution of added instructions in the SPEC95 benchmarks due to mapping rules and other reasons

to the original code largely due to the inadequacies of the current version of the back-end compiler. Future enhancements to the back-end compiler could easily remove a lot of these redundant additions. It was also observed that a majority of the operations are added because of one of the reasons mentioned above, rather than the mapping rules themselves. This is shown in Fig.5.2, where, on average, the number of additional (valid) operations added due to the mapping rules is around 10% of the total number of added operations. The breakdown of additional operations (causes mentioned above) is shown in Fig.5.3. It may be seen from the figure that, in all the benchmarks, the two primary causes for addition of instructions are:

1. a literal operand occurring in the wrong position in the list of source operands for an operation, requiring a MOVE instruction to be added, and

2. the use of an address label as a source operand in an instruction other than a MOVE, once again requiring a MOVE to be added to first store this address label in a general purpose register.

The only other significant cause for instruction addition is the predication problem requiring the formation of conditional branches and new basic blocks. This can be completely eliminated if the scheduler is enhanced to support predication. That would further reduce the number of instructions added by about 6% on average over all the benchmarks.

An analysis of the rules used by the munger shows that only a few of the rules are used most of the time. The top five rules used by the munger, for each benchmark, are shown in Figs.5.4-5.9. The top seven rules over all the benchmarks are also shown in Fig.5.10. Table, 5.1, below, gives the description of the actual rule corresponding to a particular rule number.

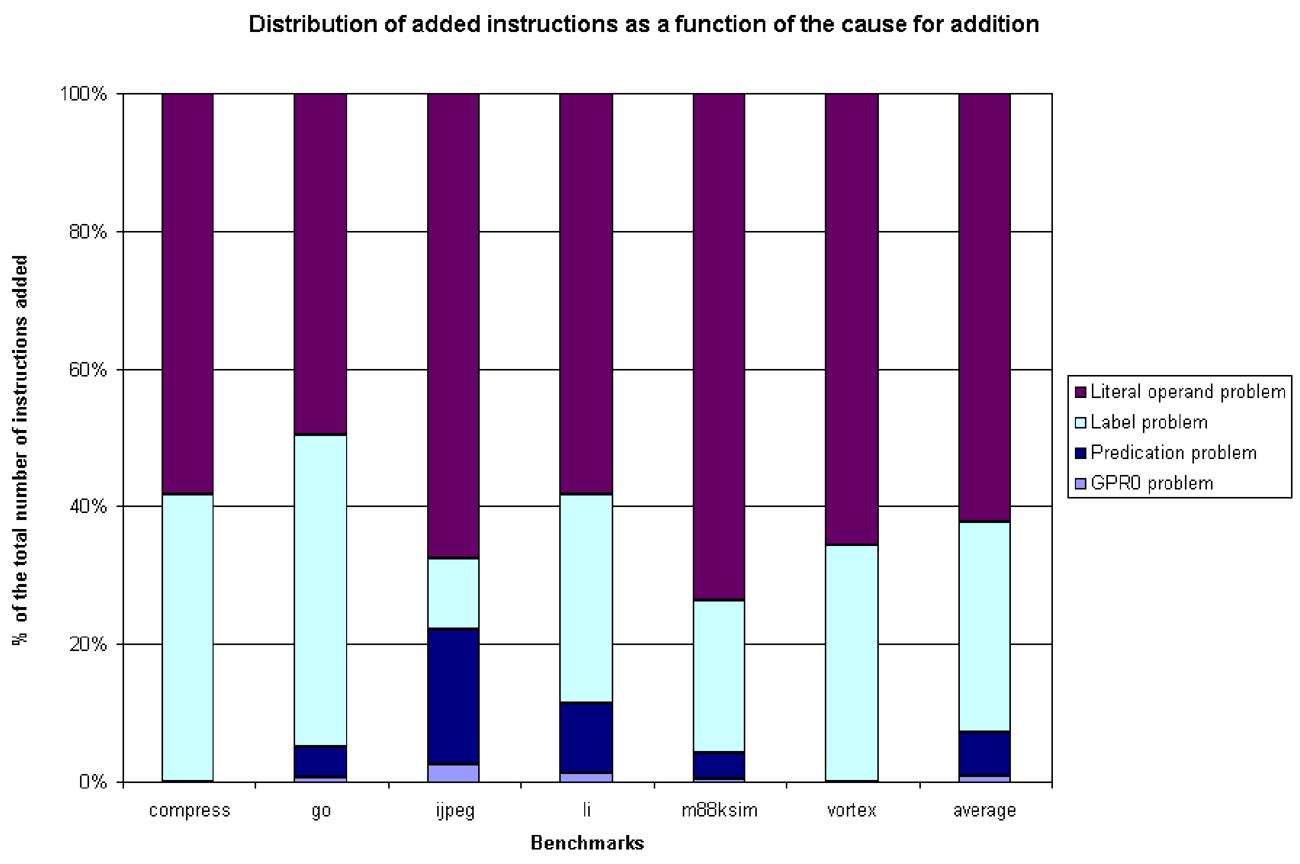

Figure 5.3: Distribution of added instructions in the SPEC95 benchmarks

Rule number	Syntax
1	ADDL.W : ADD.W
57	CMPP.W.EQ.UN.UN: CMPP.W.EQ.UN.UC CMPP.W.EQ.UN.UC
245	CMPP.W.GEQ.UN.UN: CMPP.W.GEQ.UN.UC CMPP.W.GEQ.UN.UC
306	CMPP.W.GT.UN.UN: CMPP.W.GT.UN.UC CMPP.W.GT.UN.UC
429	CMPP.W.LGEQ.UN.UN: CMPP.W.LGEQ.UN.UC CMPP.W.LGEQ.UN.UC
674	CMPP.W.LT.UN.UN: CMPP.W.LT.UN.UC CMPP.W.LT.UN.UC
729	CMPP.W.NEQ.UN.UN: CMPP.W.NEQ.UN.UC CMPP.W.NEQ.UN.UC

Table 5.1: Rule syntax corresponding to the rule numbers shown in Figs.5.4-5.9.

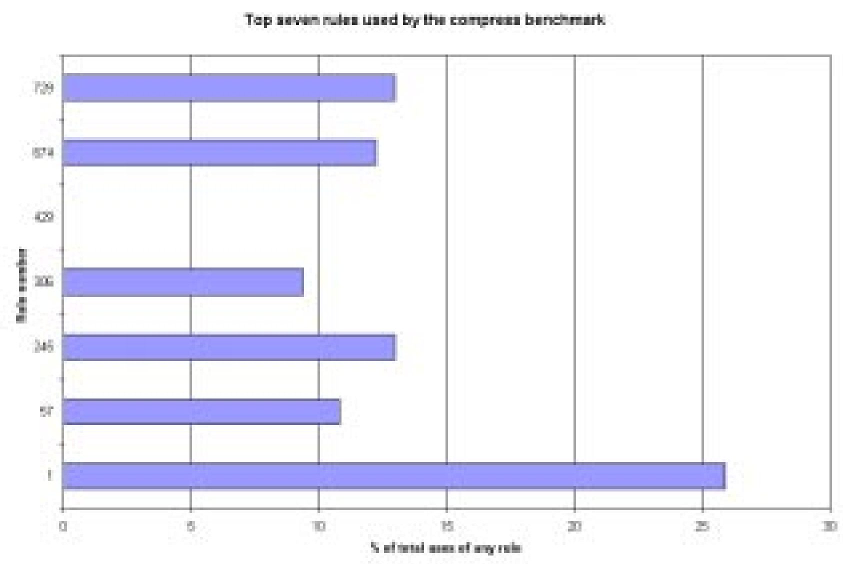

Figure 5.4: Top seven rules used by the **compress** benchmark

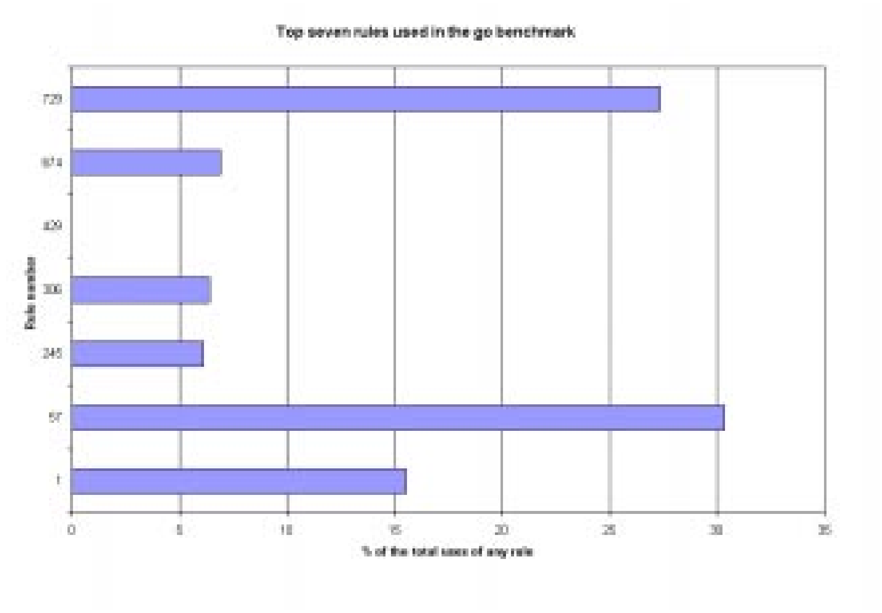

Figure 5.5: Top seven rules used by the **go** benchmark

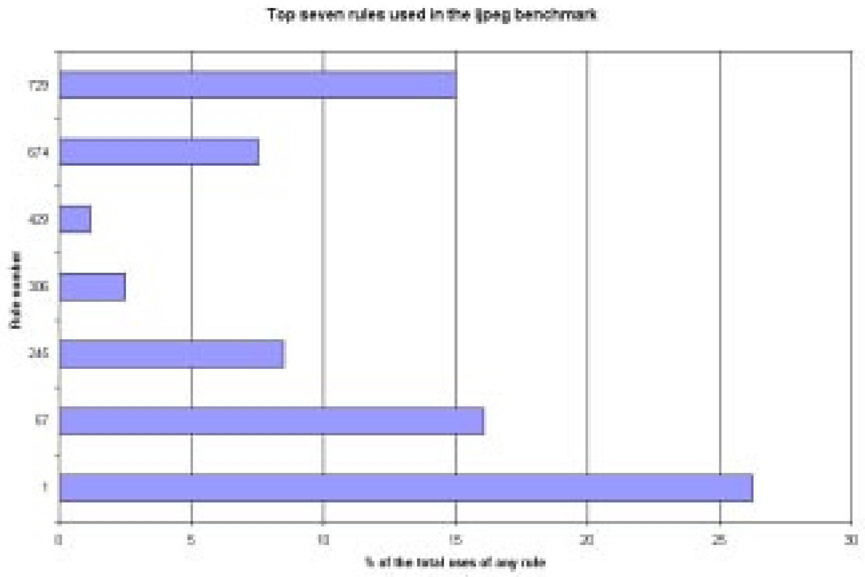

Figure 5.6: Top seven rules used by the **ijpeg** benchmark

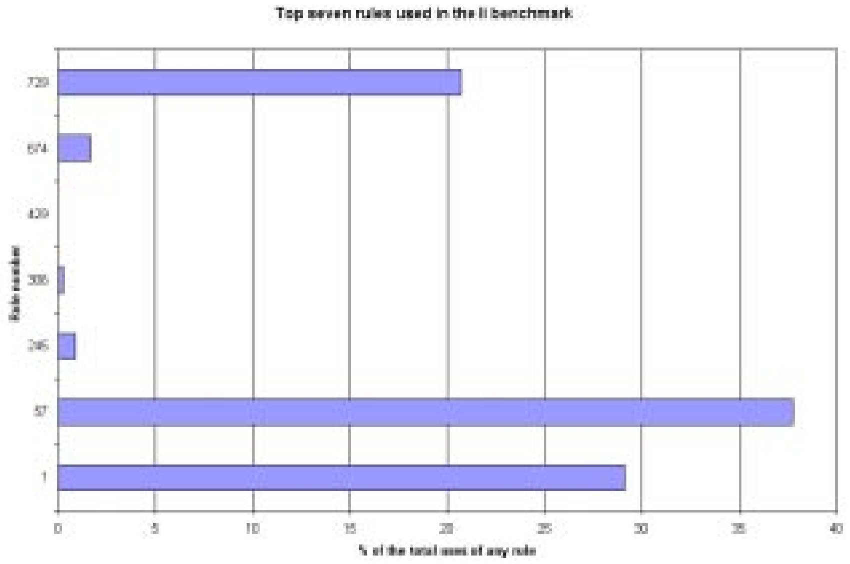

Figure 5.7: Top seven rules used by the **li** benchmark

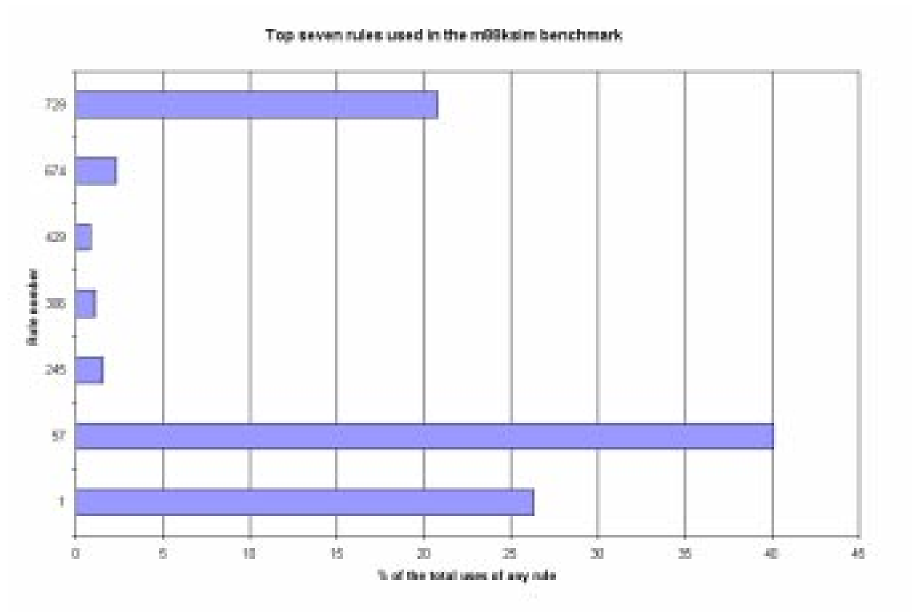

Figure 5.8: Top seven rules used by the **m88ksim** benchmark

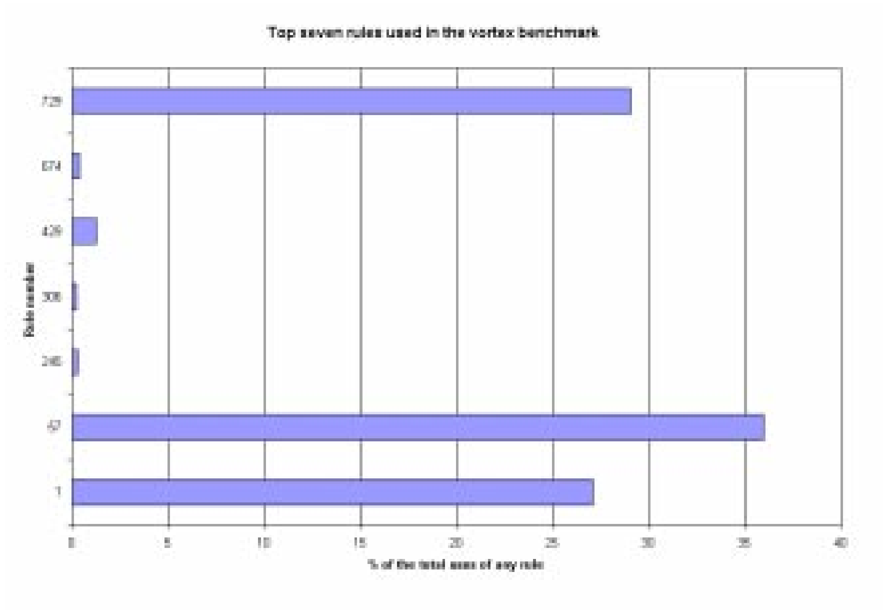

Figure 5.9: Top seven rules used by the **vortex** benchmark

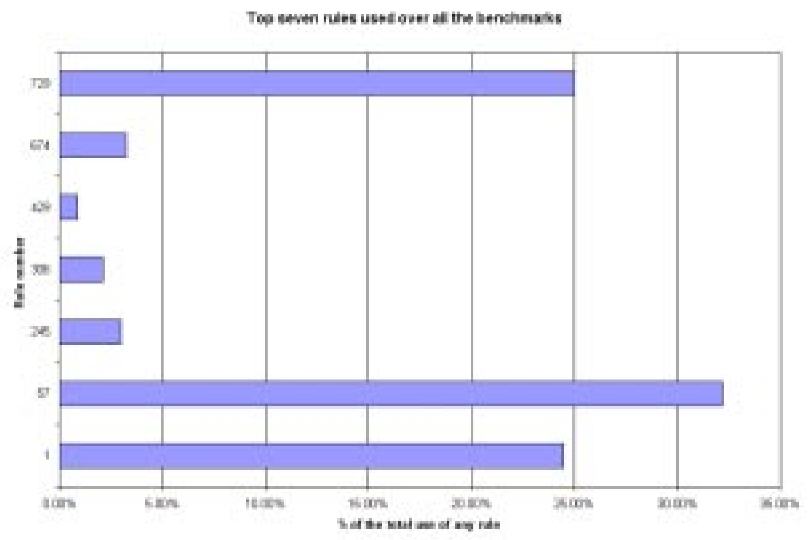

Figure 5.10: Top seven rules used over all the benchmarks

As can be seen from table 5.1, six of the top seven rules used in each of the benchmarks is a translation of a CMPP (compare to predicate) operation. The CMPP operation in Tinker sets the value of two predicate destinations based on the comparison of the source operands. The comparison operator is specified in the operation itself (*e.g.*, the .EQ., .GEQ., .NEQ. *etc.*) above. The way in which each predicate is set is also specified in the operation (*e.g.*, .UN. specifies that the predicate is to be set unconditionally). Since IA-64 doesn't allow the setting of both its predicate destinations unconditionally, every instance of the above mentioned CMPP rules needs to be converted into two instances of the same rule, except for the fact that the replacement CMPP operations set one predicate unconditionally while the second is inverted (.UC. signifies an unconditional complement). Table 5.2 shows the top seven rules used in each benchmark as a fraction of the total number of rules used.

benchmark	fraction of all rules used by the top seven rules
compress	84
go	93
ijpeg	77
li	90
m88ksim	93
vortex	94

Table 5.2: The top seven rules used as a fraction of the total number of rules

5.2 Conclusions

Owing to the large similarities between the Tinker and the IA-64 architectures, code generation for the IA-64 architecture was attempted within the framework of the LEGO back-end compiler, which currently supports only the Tinker architecture. A rule-based munger was developed to transform the code generated for the Tinker architecture to IA-64 code. The munger is currently designed to be machine independent, in that, by rewriting the rules one could transform Tinker IR to resemble more closely any other architecture specific IR, as long as that architecture is a VLIW/EPIC style machine.

As shown in my results section 5.1, the munger does not cause excessive IR explosion in the SPEC95 benchmarks. On average the amount of IR explosion was about 11%. This value is further reducible by enhancing the back-end compiler and making it fully capable of supporting predication. This would avoid the need for generating conditional branches and new basic-blocks. The register allocator also needs to be updated so that it understands which registers are bound to certain values. This measure would reduce the number of added MOVE instructions which contribute significantly to the IR expansion.

Further work needs to be done in specifying rules that merge one or more operations to emit a single operation. There is a lot of scope for improving the specification of the rules for the munger. Currently a separate pass in the munger corrects for any wrong operand formats. This pass could actually be avoided by specifying the correct operand formats through the syntax of the rules. The munger could then substitute any instructions with wrong operand formats with the correct ones, as specified in the rules.

This approach to IA-64 code generation is very effective, as, development of the back-end ILP compiler can occur independent of any retargeting of the compiler to new architectures. ILP optimizations implemented in the back-end can be tested out on any new architecture by simply munging the Rebel code into code compatible with the target architecture.

List of References

[1] Aho, A.V., and Ullman, J.D. *Principles of Compiler Design.* Addison Wesley 1977.

[2] *Tinker: A PlayDoh VLIW architecture.* Computer Architecture Research Laboratory, Department of Electrical and Computer Engineering March 1986.

[3] *IA-64 Application Developer's Architecture Guide.* Intel May 1999.

[4] Ganapathi Mahadevan, Fischer C. N. and Hennessy J. L. "Retargetable Compiler Code Generation." *Computing Surveys*, 14(4):573-592, Dec 1982.

[5] Aho A. V. and Johnson S. C. "Optimal Code Generation for Expression Trees." *Journal of the Association for Computing Machinery*, 23(3):488-501, July 1976.

[6] Aho A. V., Ganapathi Mahadevan and Tjiang S. W. K. "Code Generation Using Tree Matching and Dynamic Programming." *ACM Transactions on Programming Languages and Systems*, 11(4):491-516, Oct 1989.

[7] Glanville R. S. and Graham S. L. "A New Method for Compiler Code Generation." *Proc. 5th ACM Symp. Principles of Programming Languages* (Tuscon, Ariz., Jan 23-25), ACM, New York, 231-240, Jan 1978.

[8] Stevenson Gordon "Code Generation with a Recursive Optimizer." *Software Practice & Experience*, 10:393-403, 1980.

[9] Hatcher P. J. and Tuller J. W. "Efficient Retargetable Compiler Code Generation." *Proc. Intl. Conf. on Computing Languages*, IEEE, New York, 25-30, 1988.

[10] Davidson J. W. and Fraser C. W. "The Design and Application of a Retargetable Peephole Optimizer." *ACM Transactions on Programming Languages and Systems*, 2(2):191-202, April 1980.

[11] Davidson J. W. and Fraser C. W. "Code Selection through Object Code Optimization." *ACM Transactions on Programming Languages and Systems*, 6(4):505-526, Oct 1984.

[12] Davidson J. W. and Whalley D. B. "Quick Compilers Using Peephole Optimization." *Software Practice & Experience*, 19(1):79-97, Jan 1989.

[13] Davidson J. W. and Fraser C. W. "Automatic Inference and Fast Interpretation of Peephole Optimization Rules." *Software Practice & Experience*, 17(1):801-812, Nov 1987.

[14] *LEGO IR User's Manual.* Computer Architecture Research Laboratory, Department of Electrical and Computer Engineering April 1998.

[15] Mahlke, Scott A. "Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches." *Ph.D. dissertation*, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, Sept 1996