AUTOMATIC ANNOTATION OF INSTRUCTIONS WITH PROFILING INFORMATION

BY

TERESA LOUISE JOHNSON

B.S, University of Illinois, 1993

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, 1995

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. I would also like to thank the IMPACT group, especially John Gyllenhaal and Grant Haab for their helpful discussions. John Gyllenhaal, in particular, has answered many questions and provided much advice concerning the work presented in this thesis. Also, I would like to thank Santosh Abraham at Hewlett Packard Laboratories for his valuable guidance and feedback. For their three years of financial support through a fellowship, I would like to thank the National Science Foundation. I would also like to express my appreciation to my family for all of their encouragement and support. Finally, I wish to thank Mike Stangel, who has encouraged me and kept me happy throughout the past three years.

TABLE OF CONTENTS

1. INTRODUCTION

Profiling has been used successfully to guide code optimizations [1], [2], such as branch prediction strategies [3], [4], [5]. In order to use profiling information with optimizations, the profiling information must somehow be made available to the compiler. The tool described in this thesis accomplishes the above by automatically merging profiling results into the application's low-level intermediate representation, called *Lcode*, through a process called *annotation*.

Figure 1.1 shows a flow diagram for the entire annotation process, starting from the original C source code.[1] The first phase translates C code to Lcode. The profiler is run through the simulator, so it is necessary to probe the Lcode for simulation. Probing is performed in the second step. Next the profiler is run (through the simulator), and gathers the requested information. Finally, the annotation tool merges the profiling information, contained within an *annotation file*, into the Lcode.

[1] Fortran codes should first be transformed to C via the f2c program.

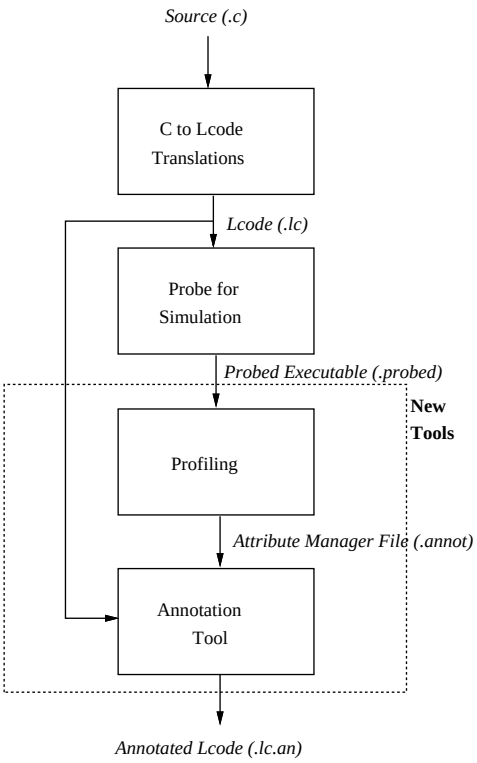


Figure 1.1: Flow diagram of the annotation process

The annotation tool, along with the format of the annotation file, will be discussed first in Chapter 2. Then the profiler will be described in Chapter 3, and an example using the profiler and annotation tool will be presented in Chapter 4. Finally, conclusions will be presented in Chapter 5. First, however, related work and the compiler framework will be discussed in Sections 1.1 and 1.2.

1.1 Related Work

Profiling has been gaining attention for use in guiding compile-time optimizations. Profile-based, or *semistatic*, branch prediction strategies have been found to achieve a level of performance comparable to the best hardware schemes [4], [5], [6], [7]. The success of semistatic branch prediction can be explained by the fact that most programs are dominated by statically predictable branches, even with different input data sets [3].

Control-flow profiling has been used to identify frequent and infrequent execution paths, which aids trace scheduling [8], ILP-enhancing optimizations, software pipelining, and classic global and loop optimizations [1], [2], [9], [10]. Profiling information has also been used to guide instruction placement [11], [12], to help the register allocator identify frequently accessed variables [13], [14], and to aid the compiler with inlining expansion [15], [16]. Memory-dependence profiling has been used to aid ILP-enhancing optimizations by allowing the compiler to reorder ambiguous memory references [2].

Profiling has also been used to identify procedures, basic blocks or source lines with high memory overheads or cache misses [17], [18], [19], [20]. Profiling information specifying the number of cache misses incurred by each access has been proposed to guide the compiler to selectively prefetch data [21], [22] and has been recently used to hand-tune code [20].

1.2 The IMPACT Compiler

The tool described in this thesis is implemented within the IMPACT compiler [23]. There are three levels of intermediate representation (IR) within IMPACT, which divide the compiler into distinct parts. *Pcode* is the highest level IR, and is based on a parallel C code representation. *Hcode* is the middle level IR, and is a flattened C representation. *Lcode* is the lowest level IR, and is a generalized register transfer language consisting of two subcomponents. These subcomponents are Lcode, which is the machine independent IR, and *Mcode*, the machine specific IR. The only difference between the two subcomponents is that there is a one-to-one correspondence between Mcode instructions and target machine assembly language instructions.

Different transformations and optimizations are performed at each IR level. The annotation tool is currently integrated with Lcode only, so profiling is performed at the Lcode level, and the results are annotated back into the Lcode.

Figure 1.2 shows the C source code for one function in the wc application, *_wcp*, and Figure 1.3 shows the corresponding Lcode. Each function in Lcode consists of a

```
wcp(wd, charct, wordct, linect)
register char *wd;
long charct; long wordct; long linect;
{
        while (*wd) switch (*wd++) {
        case 'l':
                ipr(linect);
                break;

        case 'w':
                ipr(wordct);
                break;

        case 'c':
                ipr(charct);
                break;

        }
}
```

Figure 1.2: C source code for the _wcp function of wc

hierarchy of three structures: the function itself, control blocks and operations. The function is marked with the key word *function*, control blocks with *cb*, and operations with *op*. Following each key word is an identifier, which gives the function, cb or op a unique identification. For functions this is the function name, and each cb and op is given a number unique within that function. Other information is also attached to each structure, but describing the entire format is outside the scope of this thesis.

However, one feature of the Lcode IR important to this work is the use of *attributes*. An optional structure called an *attribute list* can be attached to each function, cb and op. This list is contained within the '<' and '>' characters, and is used to pass information between passes of the Lcode optimizations. Within the attribute list are one or

```
(ms text)
(global _wcp)
(function _wcp 1.000000 <ES> <
(ARCH:HPPA)
(MODEL:PA_7100)>)
  (cb 2 1.000000 [(flow 1 9 0.000000)(flow 0 12 1.000000)] <(trace (i 4))>)
    (op 1 define [(mac $P0 i)] [])
    (op 2 define [(mac $P1 i)] [])
    (op 3 define [(mac $P2 i)] [])
    (op 4 define [(mac $P3 i)] [])
    (op 5 define [(mac $return_type i)] [])
    (op 6 define [(mac $local i)] [(i 0)])
    (op 7 define [(mac $param i)] [(i 16)])
    (op 8 prologue [] [])
    (op 9 mov [(r 1 i)] [(mac $P0 i)])
    (op 10 mov [(r 2 i)] [(mac $P1 i)])
    (op 11 mov [(r 3 i)] [(mac $P2 i)])
    (op 12 mov [(r 4 i)] [(mac $P3 i)])
    (op 13 ld_c [(r 5 i)] [(mac $P0 i)(i 0)] <(param (i 0))>)
    (op 14 beq [] [(r 5 i)(i 0)(cb 9)] <(NL_stln)>)
  (cb 12 3.000000 <S>
[(flow 108 4 1.000000)
 (flow 119 8 1.000000)
 (flow 0 7 1.000000)] <
(trace (i 0))>)
    (op 17 ld_c [(r 7 i)] [(r 1 i)(i 0)] <(param (i 0))>)
    (op 47 add [(r 1 i)] [(r 1 i)(i 1)])
    (op 18 beq [] [(r 7 i)(i 108)(cb 4)] <(NL_inner)>)
    (op 19 beq [] [(r 7 i)(i 119)(cb 8)] <(NL_inner)>)
  (cb 7 1.000000 <S> [(flow 1 5 0.000000)(flow 1 5 1.000000)] <(trace (i 1))>)
    (op 20 bne [] [(r 7 i)(i 99)(cb 5)] <(NL_inner)>)
    (op 29 mov [(mac $P0 i)] [(r 2 i)])
    (op 30 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
    (op 48 jump [] [(cb 5)])
  (cb 4 1.000000 [(flow 1 5 1.000000)] <(trace (i 3))>)
    (op 22 mov [(mac $P0 i)] [(r 4 i)])
    (op 23 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
  (cb 5 3.000000 [(flow 1 12 2.000000)(flow 0 9 1.000000)] <(trace (i 3))>)
    (op 25 ld_c [(r 9 i)] [(r 1 i)(i 0)] <(param (i 0))>)
    (op 26 bne [] [(r 9 i)(i 0)(cb 12)] <(LB_inner)(LE_inner)>)
  (cb 9 1.000000 [] <(trace (i 2))>)
    (op 27 epilogue [] [])
    (op 28 rts [] [] <(tr (mac $P15 i))>)
  (cb 8 1.000000 [(flow 1 5 1.000000)] <(trace (i 5))>)
    (op 33 mov [(mac $P0 i)] [(r 3 i)])
    (op 34 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
    (op 36 jump [] [(cb 5)] <(NL_inner)>)
(end _wcp)
```

Figure 1.3: Lcode produced for the _wcp function of wc

more attributes contained within a set of parentheses. For example, op 34 contains the following attribute list:

```
<(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>
```

which consists of three attributes:

1. `(tr (mac $P0 i))`
2. `(ret (mac $P15 i))`
3. `(param_size (i 16))`

The first items within each attribute are the attribute names, which are `tr`, `ret`, and `param_size` for the above three attributes (the meaning of these attributes is not important here). The name is followed by a list of parenthesized value expressions, the formats depending on the type of information. For example, the attribute `param_size` has the value `(i 16)`, which is the integer value 16.

As will be described in Chapter 2, the annotation tool manipulates these attribute lists. Then the annotation tool is used to pass profiling information back into Lcode, as will be described in Chapters 3 and 4.

2. THE ANNOTATION TOOL

The annotation tool described in this chapter performs the annotation of Lcode instructions with information produced by the profiler, which will be discussed in Chapter 3. Several other capabilities are also provided by the annotation tool, which operates in one of five modes.

2.1 Annotation File Format

Before describing each of these modes, it is important to understand the format of an *annotation file*. This file contains a list of attributes and associated attribute values for a program. An attribute can belong to a specific function, control block or operation, as discussed in Section 1.2. In the annotation file, attributes are divided into smaller lists by the functions they reside within. For each function, all of the corresponding function attributes are listed sequentially, one per line, followed first by control block attributes and second by operation attributes. An example of an annotation file for the _wcp function of the wc benchmark is shown in Figure 2.1. Each list is also surrounded

```
# Attribute Manager file --- Version 1
begin _wcp 0 7 3 1.000000
2 trace i 4
12 trace i 0
7 trace i 1
4 trace i 3
5 trace i 3
9 trace i 2
8 trace i 5
13 param i 0
17 param i 0
25 param i 0
end _wcp
```

Figure 2.1: Annotation file example

by a pair of delimiters. Before the first attribute associated with a particular function, there is a `begin` statement with the following syntax:

```
begin function_name num_fn_attrs num_cb_attrs num_op_attrs weight
```

The number of function, control block and operation attributes within the function function_name are contained in num_fn_attrs, num_cb_attrs and num_op_attrs, respectively. These values are necessary because there are no delimiters between the function, control block and operation attribute sublists. The weight of the function is needed for *combine* mode, and will be described in Section 2.3. As listed in the `begin` statement of Figure 2.1, there are zero function attributes, seven control block attributes and three operation attributes for that function.

Following the last attribute associated with a particular function, there is an `end` statement with the following syntax:

```
end function_name
```

Also, before the `begin` statement is the following comment:

```
# Attribute Manager file --- Version 1
```

where the Attribute Manager is a set of routines that read and write the annotation file and manipulate annotation file data, which are maintained in an internal data structure. Additionally, each Attribute Manager comment and proceeding `end` statement is separated by a blank line.

Each line of the annotation file that contains an attribute has the following syntax:

```
num attr_name attr_value
```

where num is the number of the function, control block or operation associated with this attribute, attr_name is the name of the attribute, and attr_value is the attribute value, which the Attribute Manager expects to be an integer.

In the example of Figure 2.1, the seven control block attributes are listed first, and are all *trace* attributes. Next the three operation attributes are listed, which are all *param* attributes. The meaning of these attributes is not important here. Note that these are not necessarily all of the attributes existing in function _wcp.

2.2 Annotation Modes

The modes in which the annotation tool will operate are as follows:

extract Extracts a specified list of attributes from a given Lcode file and creates an annotation file containing the extracted attributes.

index Creates an index file for a given annotation file, which lists the start index (the number of characters offset from the start of the file) of each function's attributes within the annotation file. An index file is required for insert mode.

insert Inserts attributes from a given annotation file into the given Lcode file, creating a new Lcode file in the process. An index file for the annotation file must have been created.

strip Strips off the specified list of attributes from the given Lcode file, creating a new Lcode file in the process that is identical to the original Lcode file except for the stripped-off attributes.

combine Combines two annotation files, either averaging or adding the attribute values. The two annotation files must have identical lists of attributes, but not necessarily the same attribute values. In other words, in the two files being combined, each function must have the same set of attributes in exactly the same order. The type of combining to be performed is specified as described in Section 2.3.

2.3 Using the Annotation Tool

There are two ways to run the annotation tool. The first is to directly run Lannotate, which is the name of the annotation tool executable. The second is to run the

gen_Lannotate script, which calls Lannotate and sets the necessary parameters. The latter approach is generally preferred, as it is easier to use, and requires less command line arguments. However, each will be described below.

2.3.1 Lannotate

The command line syntax of a call to Lannotate is as follows:

```
Lannotate [-i infile] [-o outfile] [-p parmfile] [-Pmacro=value]
          [-Fparm=value]
```

The input Lcode file is specified with infile, which is only needed for *extract*, *insert* and *strip* modes. Similarly, the output Lcode file is specified with outfile, which is only needed for *insert* and *strip* modes (*extract* mode does not output Lcode). If either of these parameters is not specified, stdin or stdout is assumed. The parameter file to be used is specified with parmfile. If it is not specified, the parameter file specified by the STD_PARMS_FILE environment variable is used, and if the environment variable is not set, the STD_PARMS file in the current working directory is used.

The option -Pmacro=value is used to define *macro* as *value* for parameter file preprocessing. The *macro* may be used to define parameter values. Only one macro is specific to Lannotate, *base*, which is the base name for the annotation and index files. For example, the annotation file *base*.annot will be created for *extract* mode and read for *insert*, *index* and *combine* modes. Also, the index file *base*.index will be created for *index* mode and

read for both *insert* and *combine* modes. Specifying these filename parameters will be explained in more detail below.

The option -Fparm=value is used to force parameter *parm* to *value* in the parameter file. Specific Lannotate parameters are:

mode The five possible Lannotate modes were described above in Section 2.2.

type The combine type, needed only for the *combine* mode, may be either add or average. The specified combine type is common to all attributes; it is not possible to specify a combine type for each individual attribute. For add, corresponding attribute values are simply added together. For average, attributes are averaged together, using the function weights. Each attribute is multiplied by the weight of the enclosing function. Corresponding attributes are then added, and their sum is divided by the sum of the weights of their enclosing functions, to give a weighted (or averaged) attribute value. In both cases corresponding function weights are added to form new function weights.

fn_attr_list Space-separated list of function attributes for extract and strip modes. A single wildcard character '*' may be used at the end of a name to mean any attribute with that prefix.

cb_attr_list Similar to the above item, but for control block attributes.

op_attr_list Similar to the above item, but for operation attributes.

src1_file_name Name of the annotation file to use with *insert*, *combine* and *index* modes. For *extract* mode this file is actually used as the destination file. Defaults to *base*.annot (changing the base name was described above).

src2_file_name Name of the second annotation file to use with *combine* mode. Defaults to *base*_sum.annot.

dest_file_name Name of the annotation file to create with *combine* mode. Defaults to *base*_sum.annot.

index_file_name Name of the index file to create with *index* mode, or use with *insert* mode. Defaults to *base*.annot_index.

Running Lannotate from the command line with the necessary parameters set will cause the annotation tool to operate in the specified mode.

2.3.2 gen_Lannotate

Obviously, specifying all of the necessary parameters with the correct syntax could become tedious. Therefore, the gen_Lannotate script was written to simplify running the annotation tool. The command line syntax used to call gen_Lannotate is:

```
gen_Lannotate mode [type] ["fn_attr_list" "cb_attr_list"
                "op_attr_list"] directory base < list
```

for which the arguments are as follows:

mode The five possible Lannotate modes were described above in Section 2.2.

type The combine type, needed only for *combine* mode, is either add or average. A more detailed description was given in Section 2.3.1.

fn_attr_list Space-separated (quoted) list of function attribute names. A single wildcard character '*' may be used at the end of a name to mean any attribute with that prefix. Used only with *extract* and *strip* modes.

cb_attr_list Similar to the above item, but for control block attributes.

op_attr_list Similar to the above item, but for operation attributes.

base Base name for the annotation files.

directory Directory where files will be located.

The input, *list*, is a list of Lcode files to be used with *extract*, *insert* and *strip* modes. Lannotate will be called for each file in the input list, with the given mode. The infile, outfile and parmfile arguments to Lannotate are not specified so that the default values will be used. Therefore, for *extract*, *insert* and *strip* modes the given Lcode files will be directed into Lannotate as stdin. Similarly, for *insert* and *strip* modes the stdout is redirected to the given Lcode filenames with the suffix '.an' appended.

Also, when gen_Lannotate is called with *insert* mode, Lannotate is automatically called with *index* mode first to create an index file, in order to eliminate a step for the user and to avoid unnecessary errors. The *insert* mode of gen_Lannotate is utilized

to annotate Lcode instructions with profiling information, as will be described with an example in Chapter 4.

3. PROFILING

The profiler described in this chapter was developed to run at the Lcode level of the IMPACT compiler. It currently profiles up to three types of low-level instructions:

1. memory accesses
2. branches
3. predicates

Each type of profiling will be explained in turn below. The profiler is actually run through Lsim, the Lcode simulator, when given the appropriate options. Because the simulator is directed to run as a profiler, no emulation is performed. Also, timing simulation is not needed. Uniform sampling is used to reduce profiling time [24]. However, at least 10,000,000 instructions are profiled, by evenly spacing 50 samples. If the application contains less than 10,000,00 instructions, then all instructions are profiled.

Before the simulator is called, the code must first be probed for simulation by inserting instrumenting instructions. These instructions are executed to trace events in the

program module. Events that are traced are control block entries, conditional branches, function entries, function returns, loads and stores. Some of this information is used by the profiler, depending on the type of profiling requested.

The simulator invokes the main profiling routine, *S_profile*, once for each sample. *S_profile* is given the starting program counter value as well as the profile count, which is the number of instructions to profile for that sample.

The profiler detects the types of profiling requested and keeps track of the necessary information. After profiling is completed for the entire application, all requested information is written to an annotation file, described in Section 2.1. Each piece of profiling information, given an appropriate attribute name, is written to the file as an attribute value, along with the program counter (pc) of the corresponding memory access, branch or predicate. Then the annotation tool can be used to merge the profiling information, in the form of attributes, back into the Lcode. Once in the Lcode, this information can be used to guide other transformations, such as static branch prediction and static memory disambiguation.

3.1 Profiling Memory Accesses

The profiler calls a simulation library routine to create a data cache in software using global parameters from the simulator that specify the cache size, associativity and block size. Section 3.5 will explain how to set these parameters. There are simulation routines to find a block within the cache corresponding to a particular address, find

the least recently used block in the set corresponding to a particular address, replace a particular cache block, make a particular cache block the most recently used in that set, and invalidate a particular cache block.

Using the data cache and the available routines, the profiler can update the data cache on every memory access. This can be used to keep track of several types of information pertaining to memory accesses. Currently, the profiler keeps track of the number of times each operation is executed, and when profiling memory accesses, the number of times each load, store and prefetch misses. A profiling structure for each operation holds this information. After profiling is completed, individual miss ratios for each load, store and prefetch are calculated from this information and a *dmr* (*data miss ratio*) attribute is written to the annotation file with the operation id corresponding to the pc. Because the Attribute Manager expects attribute values to be long integers, as explained in Section 2.1, the miss ratios are multiplied by 10000 to save four digits to the right of the decimal point. Therefore, the *dmr* is calculated as follows:

$$dmr = (int)prof\_info[op.pc].num\_misses * 10000.0/prof\_info[op.pc].num\_executed$$

where *prof_info* is the array of profiling information structures, one for each operation, and *op.pc* is the pc of the current operation.

Figure 3.1 shows the Annotation file created by the profiler for the _wcp function of wc when profiling memory accesses. The three *dmr* attributes correspond to the three memory operations in the _wcp function, as shown in Figure 1.3.

```
# Attribute Manager file --- Version 1
begin _wcp 0 0 3 1.000000
13 dmr 10000
17 dmr 0
25 dmr 0
end _wcp
```

Figure 3.1: Annotation file created for the _wcp function of wc when profiling memory accesses

In Chapter 4, the results of profiling memory accesses in a group of applications will be presented.

3.2 Profiling Branches

If branch profiling is requested, the profiler calls a routine to create a Branch Target Buffer (BTB) in software. Then for each branch instruction that is profiled, the profiler gets the BTB prediction and checks if it was incorrect. If so, a counter of mispredictions for that branch is incremented. After profiling is completed, the number of mispredictions for each branch is written to the annotation file. Several types of BTB models are supported. These types are two-bit counter, static and BTC (Branch Target Cache), corresponding to the attributes *MP_CTR*, *MP_STA* and *MP_BTC*, respectively. The BTB model, as well as the its size, block size, associativity and history size are specified as parameters in the parameter file. The corresponding parameter names are *BTB_model*, *BTB_size*, *BTB_block_size*, *BTB_assoc* and *BTB_history_size*.

```
#PROFILE SUMMARY:
           6 profile samples.
     1152431 total instructions profiled.
      141523 loads profiled.
      151474 stores profiled.
           0 prefetches profiled.
           0 branches profiled.
           0 predicates profiled.
           0 instructions skipped in program.
     1152431 total instruction count.
      100.00 percent of execution profiled.
      0.0039 data cache miss ratio.
```

Figure 3.2: Profile Summary for wc

3.3 Profiling Predicates

If predicate profiling is requested, then for every predicated instruction the profiler simply checks if the predicate was squashed. A counter keeps track of the number of times each predicate is squashed. After profiling is completed, the number of times each predicated instruction was squashed is written as an attribute to the annotation file with the attribute name *PS*.

3.4 Profiler Statistics

The profiler keeps track of several statistics that are then written to the *base*.profiler file after profiling is completed. Figure 3.2 shows the section of this file that gives a summary of the profiler statistics. This example was generated by profiling only memory

accesses in the wc application. Therefore, no branches or predicates were profiled. Additionally, no prefetches existed in wc, since the statistics show that the entire application was profiled, and the count of profiled prefetches is zero.

When profiling memory accesses, the total data cache miss ratio for the profiled instructions is also output.

3.5 Running the Profiler

The Lannot_bench script calls Lsim with the appropriate parameters for profiling. The command line syntax is:

```
Lannot_bench base directory [-Fprofile_memory_accesses=yes]
            [-Fprofile_branches=yes] [-Fprofile_predicates=yes]
            [-Fdcache_size=value] [-Fdcache_block_size=value]
            [-Fdcache_assoc=value] [-Fparms=value]
```

where *base* and *directory* have the same meaning as for gen_Lannotate, described in Section 2.3.2. To profile memory accesses, branches or predicates, the parameter *profile_memory_accesses*, *profile_branches* or *profile_predicates*, respectively, should be specified with the value "yes." The default value of these parameters is "no," so not specifying a parameter will cause the corresponding type of profiling to be disabled.

The cache size, block size and associativity can be set by specifying a value for *dcache_size*, *dcache_block_size* and *dcache_assoc*, respectively. These values are only used when profiling memory accesses, as will be discussed in greater detail in Chapter 4. The

cache size and block size should be specified in bytes. A value of x for dcache_assoc corresponds to an x-way associative cache, with the values 0 and 1 corresponding to fully associative and direct-mapped caches, respectively. The default is a direct-mapped, 32k-byte cache with a 64-byte block size.

Other parameters can be set by specifying -Fparms=value, where the parameter *parm* will be set to *value*. Examples of other parameters that can be set are the BTB parameters for branch profiling, discussed in Section 3.2.

The default input data set for each benchmark is specified in a file read by the Lannot_bench script. The default can be overridden by adding the optional argument `-Pexec_args=my_input_file` to the end of the Lannot_bench call, where *my_input_file* is the name of the new input data set. Then the *combine* mode of the annotation tool, described in Section 2.2, can be used to average the results of the different profiling runs.

4. MEMORY PROFILING RESULTS

In this chapter, the results of profiling memory accesses in a group of applications are presented. When profiling memory accesses, as discussed in Section 3.1, individual data cache miss ratios for each load, store and prefetch instruction are computed. These are written to an annotation file, with the attribute name *dmr*.

4.1 Experimental Environment

The applications profiled are the integer SPEC benchmarks (alvinn, compress, ear, espresso, eqntott, li, sc and cc1), as well as some other common integer programs (cccp, cmp, eqn, grep, lex, qsort, tbl, wc and yacc). These benchmarks are described in Table 4.1. Lcode for the benchmarks used in this experiment has already been through Lcode optimizations, including superblock [25] optimizations.

The profiler is invoked as described in Chapter 3. For this experiment a 32k-byte, direct-mapped cache with a 64-byte block size was used.

Table 4.1: Benchmark descriptions

name	*description*
alvinn	neural network training
cccp	GNU C preprocessor
cmp	compare files
compress	compress files
ear	inner ear simulation
eqn	format math formulas for troff
eqntott	boolean equation minimization
espresso	truth table minimization
cc1	GNU C compiler
grep	string search
lex	lexical analysis program generator
li	lisp interpreter
tbl	format tables for troff
sc	spreadsheet
wc	word count
yacc	parsing program generator

4.2 Running the Experiment

The experiment consists of three steps: probing for simulation, profiling memory accesses and annotation, which are the last three steps in the flow diagram of Figure 1.1. These steps are accomplished by using the scripts gen_Lprobe, Lannot_bench, described in Section 3.5, and gen_Lannotate, described in Section 2.3.2, respectively.

To simplify running the experiments, the gen_memannot script was written to automatically run the three steps. The command line syntax of gen_memannot is:

```
gen_memannot directory base list [size block_size assoc]
```

where *directory*, *base* and *list* are as described in Section 2.3.2, and *size*, *block_size* and *assoc* are optional parameters that specify the cache configuration. If the optional parameters are not specified, then the default values discussed in Section 3.5 are assumed. Also, if any of the parameters are specified, all must be specified.

Figure 4.1 shows the annotated Lcode for the *_wcp* function of wc created by the gen_memannot script. It differs from the Lcode shown in Figure 1.3 in that the three load instructions (op 13, op 17 and op 25) have been annotated with *dmr* attributes.

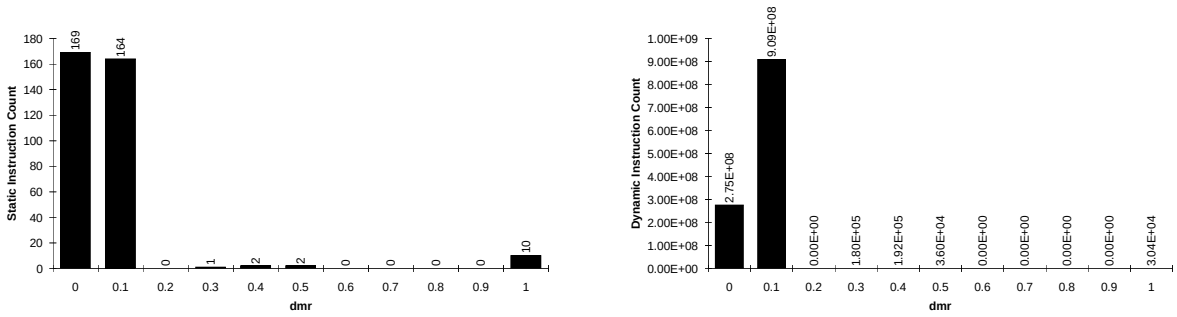
4.3 Experimental Results

Figures 4.2(a) - 4.17(a) show a histogram of the *dmr* values of statically counted load and store instructions obtained through profiling each of the applications. Each application exhibits the same behavior: almost all of the memory accesses have very low miss ratios, with a few accesses having very high miss ratios. Over 85% of the *dmr* values are below 0.2 for each application, and most of these are below 0.1. For all applications except compress and eqntott, there are a very small number of misses from around 0.2 to just below 1, and then a small group of accesses clustered at 1. These accesses are most likely causing compulsory misses that could be eliminated by prefetching. After using the annotation tool's *insert* mode, described in Section 2.2, these accesses could be identified by the compiler and marked for selective prefetching.

However, Figures 4.2(b) - 4.17(b) show a histogram of the *dmr* values of dynamically counted load and store instructions. Each time a load or store is executed, it is counted

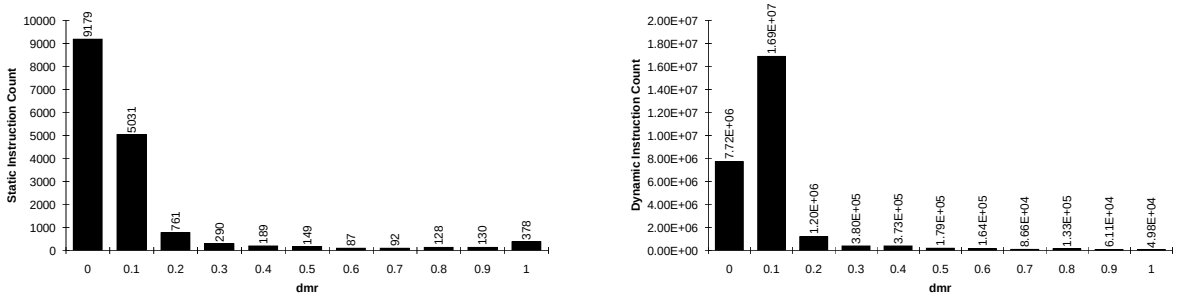
```
(ms text)
(global _wcp)
(function _wcp 1.000000 <ES> <
(ARCH:HPPA)
(MODEL:PA_7100)>)
  (cb 2 1.000000 [(flow 1 9 0.000000)(flow 0 12 1.000000)] <(trace (i 4))>)
    (op 1 define [(mac $P0 i)] [])
    (op 2 define [(mac $P1 i)] [])
    (op 3 define [(mac $P2 i)] [])
    (op 4 define [(mac $P3 i)] [])
    (op 5 define [(mac $return_type i)] [])
    (op 6 define [(mac $local i)] [(i 0)])
    (op 7 define [(mac $param i)] [(i 16)])
    (op 8 prologue [] [])
    (op 9 mov [(r 1 i)] [(mac $P0 i)])
    (op 10 mov [(r 2 i)] [(mac $P1 i)])
    (op 11 mov [(r 3 i)] [(mac $P2 i)])
    (op 12 mov [(r 4 i)] [(mac $P3 i)])
    (op 13 ld_c [(r 5 i)] [(mac $P0 i)(i 0)] <(param (i 0))(dmr (f 1.000))>)
    (op 14 beq [] [(r 5 i)(i 0)(cb 9)] <(NL_stln)>)
  (cb 12 3.000000 <S>
[(flow 108 4 1.000000)
 (flow 119 8 1.000000)
 (flow 0 7 1.000000)] <
(trace (i 0))>)
    (op 17 ld_c [(r 7 i)] [(r 1 i)(i 0)] <(param (i 0))(dmr (f 0.000))>)
    (op 47 add [(r 1 i)] [(r 1 i)(i 1)])
    (op 18 beq [] [(r 7 i)(i 108)(cb 4)] <(NL_inner)>)
    (op 19 beq [] [(r 7 i)(i 119)(cb 8)] <(NL_inner)>)
  (cb 7 1.000000 <S> [(flow 1 5 0.000000)(flow 1 5 1.000000)] <(trace (i 1))>)
    (op 20 bne [] [(r 7 i)(i 99)(cb 5)] <(NL_inner)>)
    (op 29 mov [(mac $P0 i)] [(r 2 i)])
    (op 30 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
    (op 48 jump [] [(cb 5)])
  (cb 4 1.000000 [(flow 1 5 1.000000)] <(trace (i 3))>)
    (op 22 mov [(mac $P0 i)] [(r 4 i)])
    (op 23 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
  (cb 5 3.000000 [(flow 1 12 2.000000)(flow 0 9 1.000000)] <(trace (i 3))>)
    (op 25 ld_c [(r 9 i)] [(r 1 i)(i 0)] <(param (i 0))(dmr (f 0.000))>)
    (op 26 bne [] [(r 9 i)(i 0)(cb 12)] <(LB_inner)(LE_inner)>)
  (cb 9 1.000000 [] <(trace (i 2))>)
    (op 27 epilogue [] [])
    (op 28 rts [] [] <(tr (mac $P15 i))>)
  (cb 8 1.000000 [(flow 1 5 1.000000)] <(trace (i 5))>)
    (op 33 mov [(mac $P0 i)] [(r 3 i)])
    (op 34 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
    (op 36 jump [] [(cb 5)] <(NL_inner)>)
(end _wcp)
```

Figure 4.1: Annotated Lcode for the _wcp function of wc



(a) (b)

Figure 4.2: Histogram of *dmr* values for **alvinn** (a) static and (b) dynamic instructions

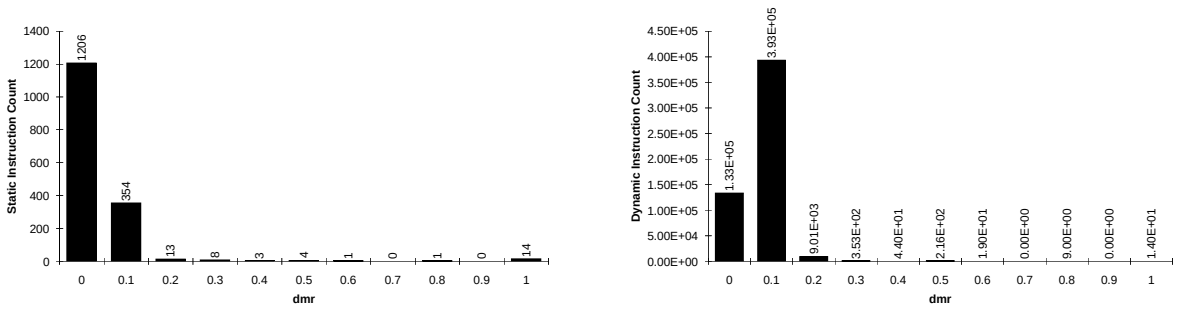


(a) (b)

Figure 4.3: Histogram of *dmr* values for **cc1** (a) static and (b) dynamic instructions

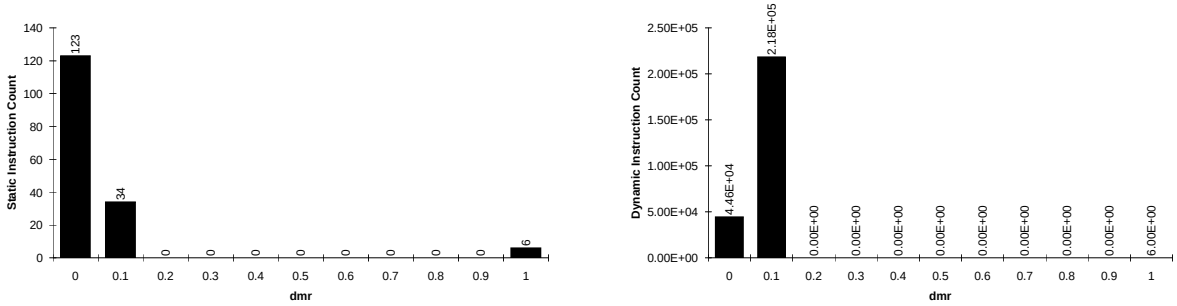
as one dynamic instruction with the miss ratio of the corresponding static instruction. Except for compress and sc, the number of dynamic instructions with high miss ratios is insignificant, suggesting that many of the instructions with high miss ratios are executed infrequently.

Compress has a group of accesses distributed evenly from 0.8 to 1. By using the annotation tool's *insert* mode to merge the miss ratios back into Lcode, the corresponding accesses were identified and studied. Some of the misses resulting from accesses with a miss ratio of 1 are compulsory. The other misses resulting from accesses with high miss ratios are most likely dominated by either conflict or capacity misses. Figure 4.18



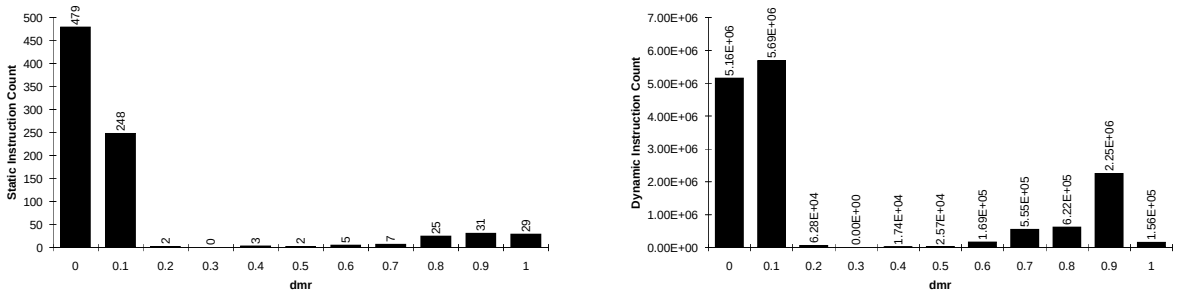
(a) (b)

Figure 4.4: Histogram of *dmr* values for **cccp** (a) static and (b) dynamic instructions



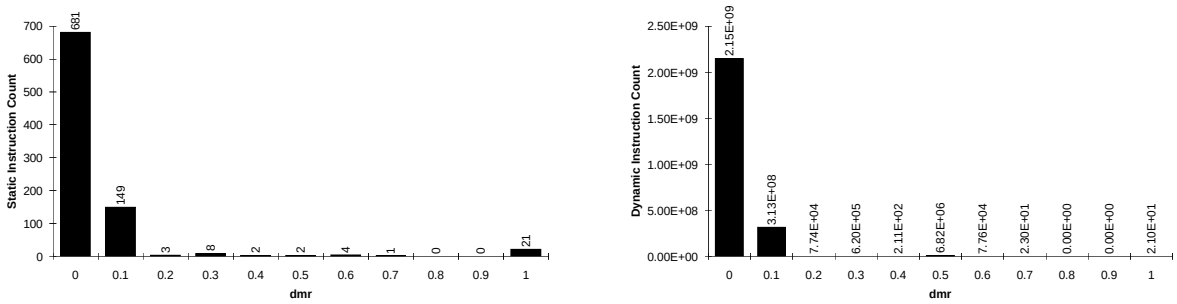
(a) (b)

Figure 4.5: Histogram of *dmr* values for **cmp** (a) static and (b) dynamic instructions



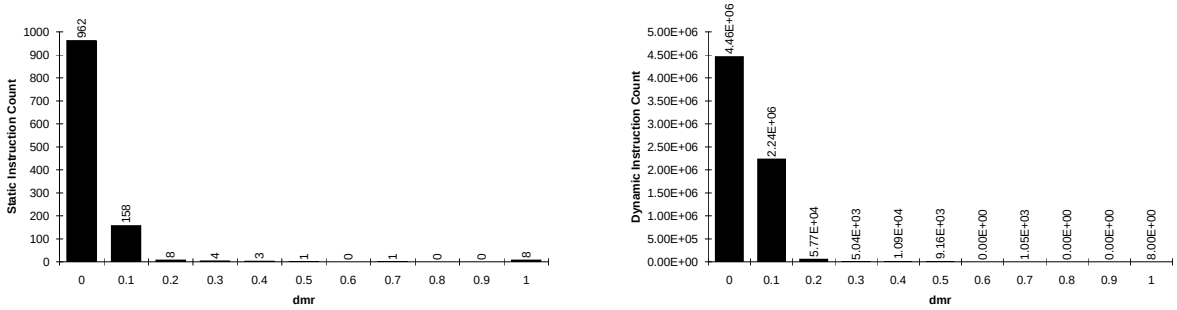
(a) (b)

Figure 4.6: Histogram of *dmr* values for **compress** (a) static and (b) dynamic instructions



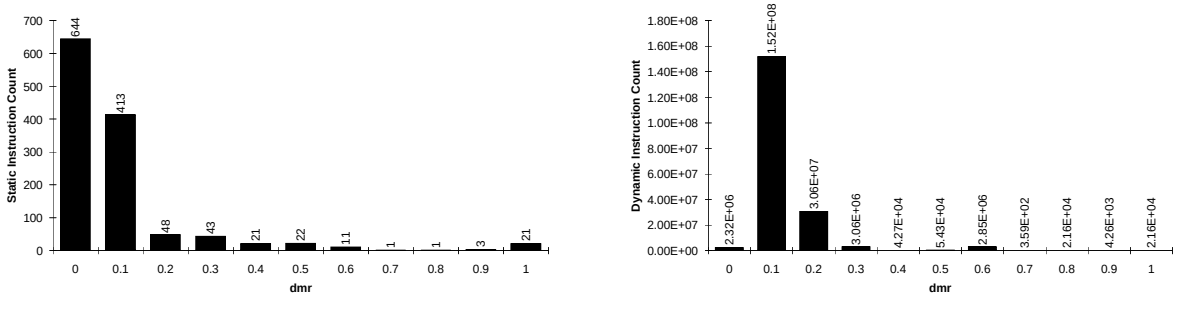
(a) (b)

Figure 4.7: Histogram of *dmr* values for **ear** (a) static and (b) dynamic instructions



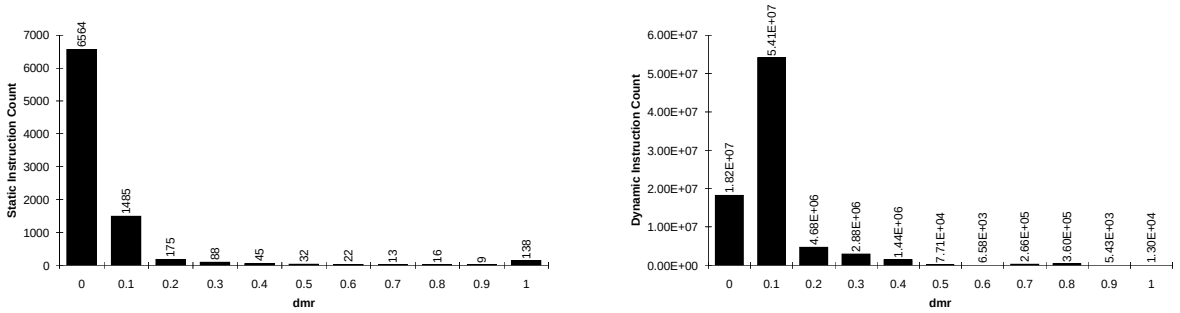
(a) (b)

Figure 4.8: Histogram of *dmr* values for **eqn** (a) static and (b) dynamic instructions



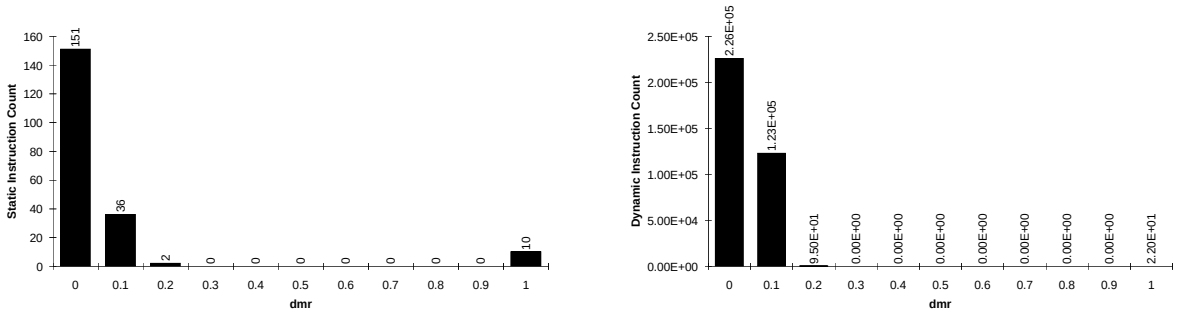
(a) (b)

Figure 4.9: Histogram of *dmr* values for **eqntott** (a) static and (b) dynamic instructions



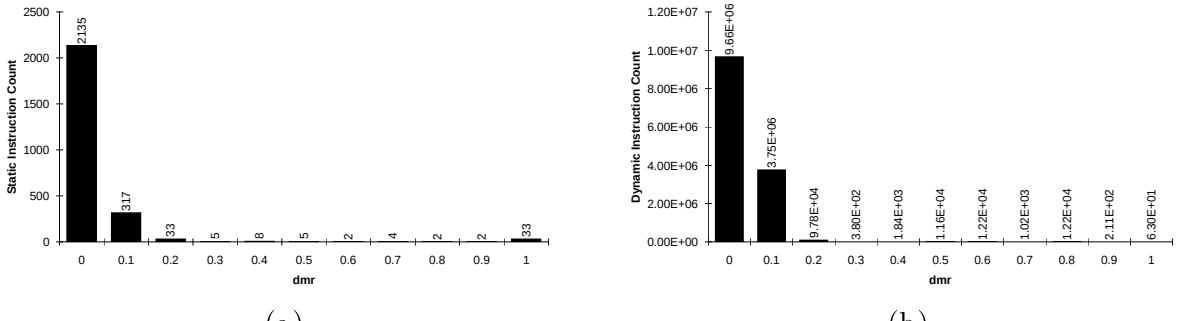
(a) (b)

Figure 4.10: Histogram of *dmr* values for **espresso** (a) static and (b) dynamic instructions



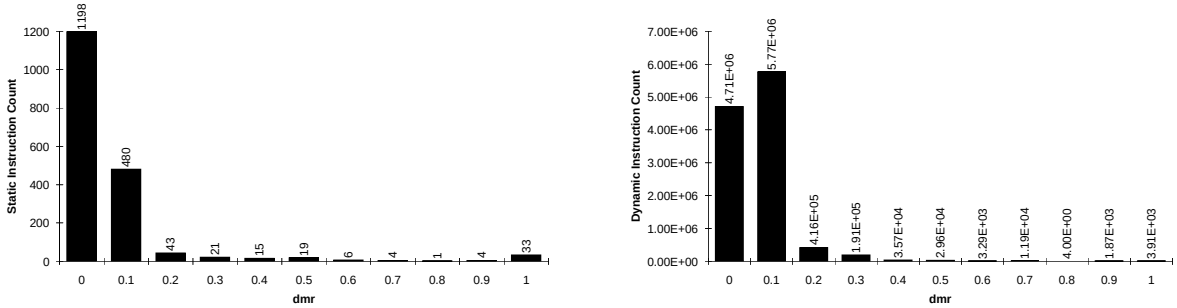
(a) (b)

Figure 4.11: Histogram of *dmr* values for **grep** (a) static and (b) dynamic instructions



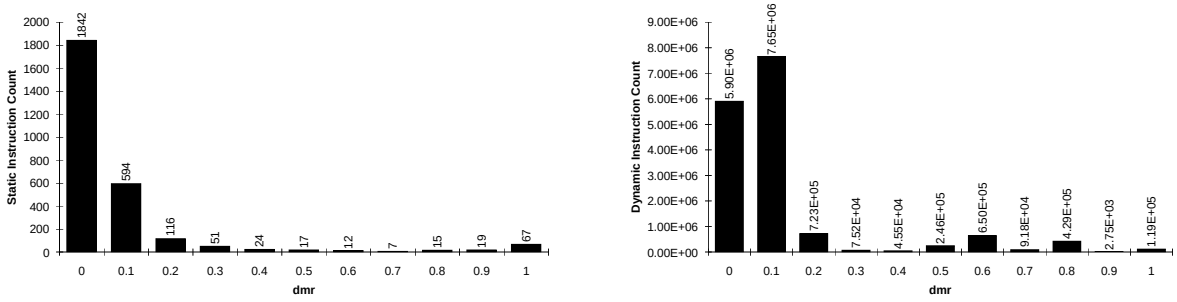
(a) (b)

Figure 4.12: Histogram of *dmr* values for **lex** (a) static and (b) dynamic instructions



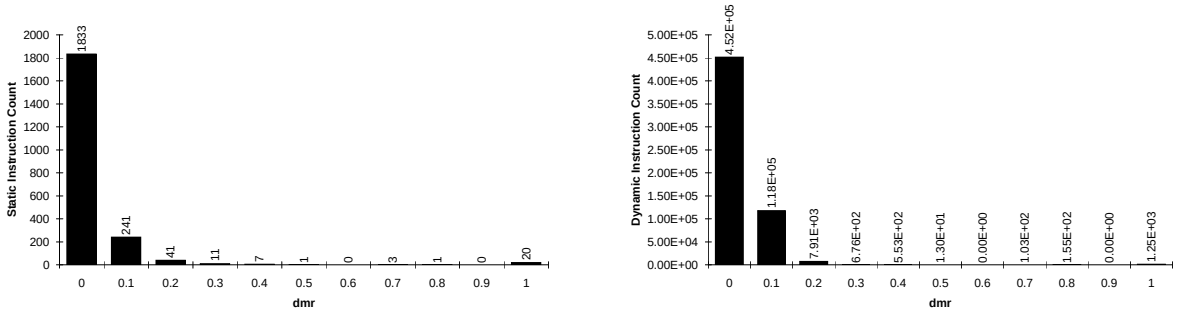
(a) (b)

Figure 4.13: Histogram of *dmr* values for **li** (a) static and (b) dynamic instructions



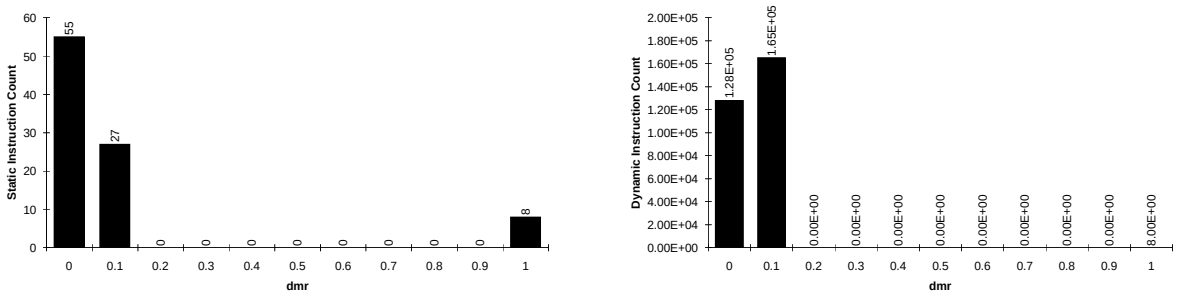
(a) (b)

Figure 4.14: Histogram of *dmr* values for **sc** (a) static and (b) dynamic instructions



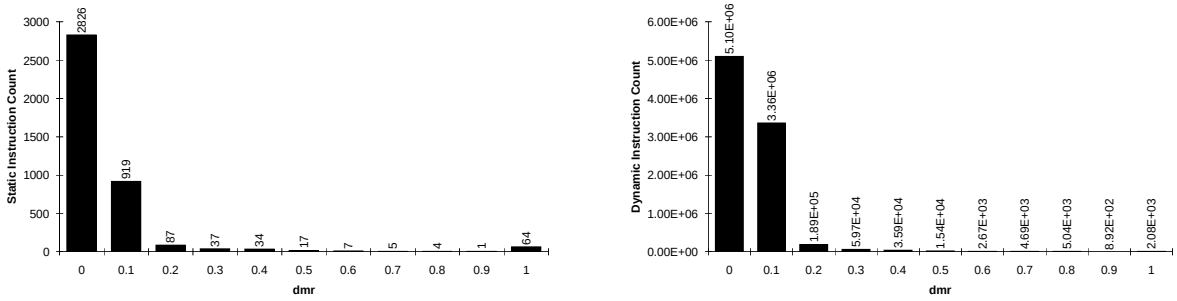
(a) (b)

Figure 4.15: Histogram of *dmr* values for **tbl** (a) static and (b) dynamic instructions



(a) (b)

Figure 4.16: Histogram of *dmr* values for **wc** (a) static and (b) dynamic instructions



(a) (b)

Figure 4.17: Histogram of *dmr* values for **yacc** (a) static and (b) dynamic instructions

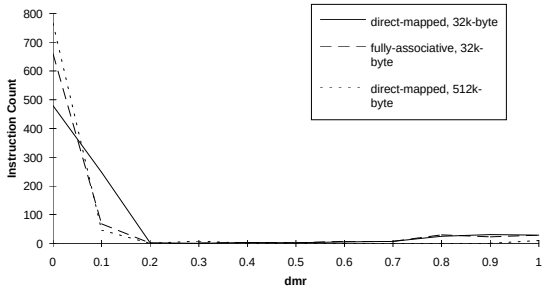


Figure 4.18: Histogram of *dmr* values for compress for three cache configurations

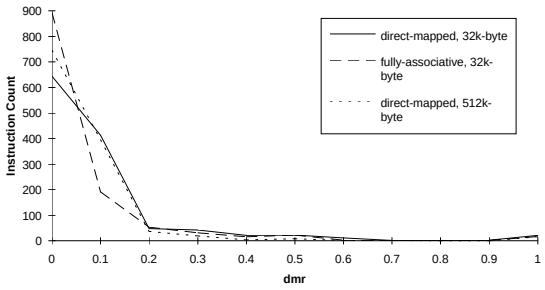


Figure 4.19: Histogram of *dmr* values for eqntott for three cache configurations

suggests that these accesses are dominated by capacity misses, for almost all of the high miss ratios disappear with a direct-mapped, 512k-byte cache, while very few are removed with a fully associative, 32k-byte cache. However, most of the small, nonzero miss ratios were reduced to zero with the fully associative cache; therefore, the corresponding misses are dominated by conflict misses.

On the other hand, eqntott has a group of accesses distributed from 0.2 to 0.6. As Figure 4.19 suggests, these misses are split between conflict and capacity misses. However, the fully associative, 32k-byte cache reduced more than half of the accesses with miss ratios originally around 0.1 to zero, whereas the direct-mapped, 512k-byte cache left the number of accesses with miss ratios around 0.1 almost unchanged. This suggests that most of the misses corresponding to these accesses are conflict misses, rather than capacity misses. Again, by using the annotation tool's *insert* mode to merge the miss ratios back into Lcode, these accesses can be identified and studied further to determine the cause of the misses and, therefore, how to best eliminate or reduce the miss ratios.

The behavior exhibited by these applications follows that observed in a previous study [21], [22]. In their study, it was found that a small number of instructions in the SPEC89 benchmarks cause a majority of the misses, as well as that a small number of instructions have high miss ratios. Their experiments were profiled using a fully associative, 16k-byte cache with a 4-byte line size. They also suggested using their profiling results to mark instructions with high miss ratios, above a specified threshold value, as long latency instructions to prefetch.

5. CONCLUSIONS

This thesis has described profiling and annotation tools that allow profiling information to be automatically annotated into an intermediate representation of the code. First, the profiling tool is run through the simulator, and gathers requested information for each instruction. Memory access, branch and predicate profiling are currently supported. The profiling information is written to an annotation file which is then read by the annotation tool, which merges the profiling information into Lcode, a low-level intermediate representation in the IMPACT compiler. Each profiled statistic is attached to the corresponding operation as an Lcode attribute.

Additionally, the results of a preliminary study of memory accesses were presented. The profiling tool was used to gather the miss ratios of load and store operations in 16 benchmark applications, and the annotation tool was then used to merge the miss ratios back into each benchmark's Lcode. Histograms show that only a small number of instructions have high miss ratios, while the rest of the instructions have miss ratios close to zero.

Further studies are needed to better characterize the accessing behavior of individual instructions. These can include miss counts, which is the total number of misses caused by an instruction, and reuse ratios. The memory accessing information will be used in the future to guide cache hierarchy management, such as promotion/demotion between cache levels and cache bypassing.

Also, it may be desirable to annotate profiling information into higher levels of the intermediate representation, or into the source code itself. This would be useful to better study the types of accesses which frequently miss in the cache, and to develop heuristics for identifying and optimizing those accesses in the compiler when profiling information is not available.

REFERENCES

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

[2] W. Y. Chen, S. A. Mahlke, N. J. Warter, and W. W. Hwu, "Using profile information to assist advanced compiler optimization and scheduling," in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[3] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of the 5rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[4] A. Krall, "Improving semi-static branch prediction by code replication," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp. 97–106, June 1994.

[5] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, October 1994.

[6] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 396–403, June 1986.

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

[8] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[9] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: The MIT Press, 1985.

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

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

[12] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[13] D. W. Wall, "Global register allocation at link time," in *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pp. 264–275, June 1986.

[14] D. W. Wall, "Register windows vs. register allocation," in *Proceedings of the 1988 SIGPLAN Symposium on Compiler Construction*, pp. 264–275, June 1988.

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

[16] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Software Practice and Experience*, vol. 22, pp. 349–370, May 1992.

[17] A. J. Goldberg and J. Hennessy, "Performance debugging shared-memory multiprocessor programs with mtool," in *Proceedings of Supercomputing '91*, pp. 481–490, 1991.

[18] D. Callahan, K. Kennedy, and A. Porterfield, "Analyzing and visualizing performance of memory hierarchies," in *Instrumentation for Visualization*, New York: ACM Press, 1990.

[19] A. Gupta, M. Martonosi, and T. Anderson, "Memspy: Analyzing memory-system bottlenecks in programs," *Performance Evaluation Review*, vol. 20, pp. 1–2, June 1992.

[20] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *IEEE Computer*, pp. 15–26, October 1994.

[21] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of load/store instruction latencies," in *Proceedings of the 26th International Symposium on Microarchitecture*, pp. 139–152, December 1993.

[22] S. G. Abraham and B. R. Rau, "Predicting load latencies using cache profiling," Tech. Rep. HPL-94-110, Hewlett-Packard Laboratory, 1501 Page Mill Road, Palo Alto, CA 94304, November 1994.

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

[24] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

[25] 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 structure for VLIW and superscalar compilation," *Journal of Supercomputing*, pp. 229–248, July 1993.