UNIVERSITY OF CALIFORNIA, SAN DIEGO

Value Profiling for Instructions and Memory Locations

UCSD Technical Report
CS98-581, April 1998

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science in
Computer Engineering

by

Peter T. Feller

Committee in charge:

Professor Bradley Calder, Chairperson
Professor Dean Tullsen
Professor Jeanne Ferrante

The thesis of Peter Feller is approved:

University of California, San Diego

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT OF THE THESIS

Value Profiling for Instructions and Memory Locations

by

Peter Feller
Master of Science in Computer Engineering
University of California, San Diego, 1998
Professor Bradley Calder, Chair

Identifying variables as invariant or constant at compile-time allows the compiler to perform optimizations including constant folding, code specialization, and partial evaluation. Some variables, which cannot be labeled as constants, may exhibit semi-invariant behavior. A *semi-invariant* variable is one that cannot be identified as a constant at compile-time, but has a high degree of invariant behavior at run-time. If run-time information was available to identify these variables as semi-invariant, they could then benefit from invariant-based compiler optimizations.

In this thesis the value behavior and invariance found from profiling register-defining instructions and memory locations, as well as their differences, are analyzed. Many instructions and memory locations have semi-invariant values even across different inputs. In addition, the predictability of register values and memory location values using a Last-Value-Prediction (LVP) scheme is investigated. The ability to estimate the invariance for all instructions in a program from only profiling load instructions is examined, and an intelligent form of sampling called *Convergent Profiling* is introduced to reduce the profiling time needed to generate an accurate value profile.

A value profile can be used to automatically guide code generation for dynamic compilation, adaptive execution, code specialization, partial evaluation and other compiler optimizations. Using value profile information to perform code specialization is shown to decrease execution time up to 13%.

Chapter I

Introduction

Many compiler optimization techniques depend upon analysis to determine which variables have invariant behavior. Variables which have invariant run-time behavior, but cannot be labeled as such at compile-time, do not fully benefit from these optimizations. This thesis examines using profile feedback information to identify which variables have invariant/semi-invariant behavior. A *semi-invariant* variable is one that cannot be identified as a constant at compile-time, but has a high degree of invariant behavior at run-time. This occurs when a variable has one to N (where N is small) possible values which account for most of the variable's values at run-time. In addition to knowing a variable's invariance, certain compiler optimizations are also dependent on knowing a variable's value information. *Value Profiling* is an approach that can identify the invariance and the top N values of a variable.

The invariance of a variable is also important when doing *Value Prediction*. Value prediction [17, 27, 28] enables programs to exceed the limits which are placed upon them by their data-dependencies. The goal is to predict at run-time the outcome value of instructions before they are executed, and forwarding these speculated values to instructions which depend on them. This approach enables data-dependent instructions to execute non-sequentially and therefore to enhance the programs Instruction-Level-Parallelism (ILP). With modern micro processor word sizes of 32-64 bits, the predictability of register values would seem to have a very low probability. However, a property of programs called *value locality* [27] shows that many instructions have only very few distinct values.

In this thesis we will examine one potential type of value predictor, the *Last-Value-Predictor* (LVP). The LVP predicts the value of an instruction to be the same as the previously encountered value for that instruction. Value profiling will also be used to keep track of the instruction's LVP, and guide value prediction.

Profiling memory location values will be examined as an alternative to instruction values. Profiled instructions could potentially access many different memory locations. An example therefore would be a load which is used within a loop to access an array. Hence, it is likely for that load instruction to then show a high degree of variance, even though every memory location it accesses might be invariant.

Profiling a program can be very time consuming. The time needed to generate a profile depends on the level of detail required from profiling. For value profiling, we found that the data being profiled, the invariance of instructions, often reaches a steady state. In this thesis we will introduce a type of intelligent sampling which profiles instructions until the steady state is reached. We call this method *Convergent Value Profiling*. As an alternative we also investigate the ability to estimate the invariance for all non-load instructions by value profiling only load instructions and propagating their invariance.

I.A Thesis Overview

In the next chapter, we examine Related Work. Chapter III will describe value profiling, the algorithms and data structures used to gather value profiles, and the metrics used to interpret the results. Chapter IV gives a brief introduction into profiling techniques, their uses, and what type of profiles we employed in this study. Chapter V examines the semi-invariant behavior of all instruction types, parameters, and loads, and shows that there is a high degree of invariance for several types of instructions. In Chapter VI we discuss how changing the parameters in our data structure affects the results. Chapter VII shows the results for profiling memory locations. Chapter VIII examines a new type of intelligent profiler, and Chapter IX investigates the ability to estimate the invariance for all non-load instructions. Chapter X will show examples of how value profiling information could be used in applying code specialization. Chapter XI summarizes the thesis and Chapter XII concludes by providing some Future Directions.

Chapter II

Related Work

Value profiling can be a benefit to several areas of current compiler and architecture research. Value profiles can be used to provide feedback to value prediction about which instructions show a high degree of invariance. Value profiling can also be used to provide an automated approach for identifying semi-invariant variables and be used to guide dynamic compilation and adaptive execution. Another use for value profiling is code specialization. Value profiling information could be used to identify invariant or semi-invariant variables and then apply code specialization to certain parts of the program as illustrated in Chapter X.

II.A Value Prediction

The recent publications on value prediction [17, 27, 28] in hardware provided motivation for our research into value profiling. Lipasti et al [27] introduced the term value locality, which describes the likely hood of the recurrence of a previously seen value within a storage location. The study showed that on average 49% of the instructions wrote the same value as they did the last time they were executed, and 61% of the executed instructions produced the same value as one of the last 4 values produced by that instruction using a 16K value prediction table. These results show that there is a high degree of temporal locality in the values produced by instructions, but this does not necessarily equate to the instruction's degree of invariance, which is needed for certain compiler optimizations.

Last value prediction is implemented in hardware using an N entry Value History Table (VHT) [17]. The VHT contains a value field and an optional tag, which would store the identity of the instruction which is mapped to the entry. The PC of the executing instruction will be used to hash into that table to retrieve the last value. A stride predictor works similarly to the last-value predictor, however a stride predictor has an additional stride field. The stride is computed by taking the value difference between the two last encountered values, and the predicted value is the stride added to the last value. If the stride is zero, the value is constant, and this equates to last value prediction.

Several different value predictor models have been proposed [18, 34, 39]. Wang et. al [39] studied the performance of two different hybrid predictors in comparison to several stand-alone predictors. The first hybrid is a combination of a LVP and a stride predictor, the second one a combination of a stride predictor and a 2-level predictor. The 2-level predictor stores the last 4 values in a VHT. A Value History Pattern, which is encoded from the previous instructions, is used to select one of these 4 values as the next value. The authors show results for five different value predictors. The five predictors are LVP, stride, 2-level, hybrid(LVP, stride), and hybrid(stride, 2-level). Their average results over the six integer programs from the SPEC92 benchmark are 42%, 52%, 52%, 60%, 69% respectively.

Sazeidas et. al [34] classifies two types of value predictors, *computational* predictors and *context based* predictors. A computational predictor is one which computes a value given information for the previous values. An example would be a stride predictor. A context-based predictor predicts values that follow a certain finite pattern. For the prediction to take effect, the pattern must repeat itself. Their results show that LVP correctly predicts values about 40% of the time, stride predictors about 56% and context based 78%. These values are all averages over the SPEC95 benchmark suite. In addition, they provide results dividing the prediction into different instruction types. Of all correctly predicted results, add/subtracts make up 41%, loads 32%, logic operations 3%, shift operations 10% and set operations 6.5%. They mention that different instruction types need to be studied separately, and therefore suggest a hybrid predictor based on different instruction types.

Gabbay et al [18] studied the applicability of program profiling to value prediction. His motivation for using profiling information was to classify the instructions tendency to be value predictable. The opcodes of instructions found to be predictable were annotated. Only instructions marked predictable were considered for value prediction. The main advantage of this approach compared to the author's previous approach [17] was better usage of the prediction table, and decreased number of mispredictions.

Value prediction can benefit in several ways from value profiling. By classifying instructions into semi/invariant, invariant or variant, one can determine which instructions not to value predict. This not only increases the utilization of the prediction table, but also reduces the number of mispredictions.

II.A.1 Load Speculation

The Memory Conflict Buffer (MCB) proposed by Gallagher et al [19] provides a hardware solution with compiler support to allow load instructions to speculatively execute before stores. The addresses of speculative loads are stored with a conflict bit in the MCB. All potentially ambiguous stores probe the MCB and set the conflict bit if the store address matches the address of a speculative load. The compiler inserts a check instruction at the point where the load is known to be non-speculative. The check instruction checks the speculative load's conflict bit in the MCB; if not set, the speculation was correct, otherwise the load was mis-speculated.

A similar approach for software-based speculative load execution was proposed by Moudgill and Moreno [29]. Instead of using a hardware buffer to check addresses, they check values. They allow loads to be speculatively scheduled above stores, and in addition they execute the load in its original location. They then check the value of the speculative load with the correct value. If they are different a recovery sequence must be executed.

Value profiling could support the approach of Moudgill and Moreno [29] to only reschedule loads with a high invariance. This could potentially decrease the number of mis-speculated loads.

II.B Compiler Analysis for Dynamic Compilation

Dynamic compilation and adaptive execution are emerging directions for compiler research which provide improved execution performance by delaying part of the compilation process to run-time. These techniques range from filling in compiler generated specialized templates at run-time to fully adaptive code generation. For these techniques to be effective the compiler must determine which sections of code to concentrate on for the adaptive execution. Existing techniques for dynamic compilation and adaptive execution require the user to identify run-time invariants using user guided annotations [2, 12, 15, 25, 26]. One of the goals of *value profiling* is to provide an automated approach for identifying semi-invariant variables and to use this to guide dynamic compilation and adaptive execution.

Staging analysis has been proposed by Lee and Leone [26] as an effective means for determining which computations can be performed *early* by the compiler and which optimizations should be performed *late* or postponed by the compiler for dynamic code generation. Their approach requires programmers to provide hints to the staging analysis to determine what arguments have semi-invariant behavior. Code fragments can then be optimized by partitioning the invariant parts of the program fragment. Knoblock and Ruf [25] used a form of staging analysis and annotations to guide data specialization.

Autrey and Wolfe [3] have started to investigate a form of staging analysis for automatic identification of semi-invariant variables. Consel and Noel [12] use partial evaluation techniques to automatically generate templates for run-time code generation, although their approach still requires the user to annotate arguments of the top-level procedures, global variables and a few data structures as run-time constants. Auslander et al [2] proposed a dynamic compilation system that uses a unique form of binding time analysis to generate templates for code sequences that have been identified as semi-invariant. Their approach currently uses user defined annotations to indicate which variables are semi-invariant.

The annotations needed to drive the above techniques require the identification of semi-invariant variables, and value profiling can be used to automate this process. To automate this process, these approaches can use their current techniques for gen-

erating code to identify code regions that could potentially benefit from run-time code generation. Value profiling can then be used to determine which of these code regions have variables with semi-invariant behavior. Then only those code regions identified as profitable by value profiling would be candidates for dynamic compilation and adaptive execution.

II.C Code Specialization

Code specialization is a type of compiler optimization, which selectively executes a different version of the code, conditioned on the value of a variable. Given an invariant variable and its value, the original code is duplicated. There will be one general version of the code, and a special version of the code. The specialized version of the code will be conditioned on the invariant variable. A selection mechanism based on the invariant variable will choose which code to execute.

Calder and Grunwald [7] found that up to 80% of all function calls in C++ languages are made indirectly. These indirect function calls are virtual function calls, also referred to as methods or dynamically dispatched functions. They pose a serious performance bottleneck due to the added overhead of having to perform a table lookup to determine the branch target. Additionally, opportunities to perform optimizations such as procedure inlining and interprocedural analysis, are lost. One technique to overcome that bottleneck is to compile a specialized version of each method, for each class inheriting the method. This process is referred to as *customization*. This allows the compiler to statically bind the specialized functions, and then perform optimizations on them. The drawback of that method is that the compile-time as well as code space requirements are increased.

Hölzle et al [23] implemented a run-time type feedback system. Using the type feedback information, the compiler can then inline any dynamically dispatched function calls, specializing the dispatch based on the frequently encountered object types. The authors implemented their system in Self [24], which dynamically compiles or recompiles the code applying the optimization with polymorphic inline caches. However, this type feedback can also be used off-line. They found that unlike in C, procedure inlining does

not add a lot of extra code space due to the smaller functions of C++. They also found that the performance gain achieved using procedure inlining in object oriented languages is higher than in FORTRAN or C.

Dean et al [13, 21] extend the approach of customization by specializing only those cases where the highest benefit can be achieved. Selective specialization uses a run-time profile to determine exactly where customization would be most beneficial. What sets this apart from type feedback [23] is knowledge of the formal parameters, which allows for additional optimizations.

Richardson [32] studied the potential performance gain due to replacing a complex instruction with trivial operands, with a trivial instruction. He profiled the operands of arithmetic operations looking for trivial calculations. A trivial instruction is defined as being able to complete in one cycle. Divisions and multiplications by a power of 2, which can be replaced by a shift operation are not included in this study. He found that these optimizations can lead to up to 22% in performance gain, and floating-point intensive programs gave the highest speedup.

The profilers needed for these techniques are just special cases of a more general form of a value profiler. Value profiling provides information on how invariant a given instruction or variable is and the instruction's top values. The invariance of a variable is crucial in determining if a particular section of code should be specialized. For some optimizations, knowing the value information is just as important as the invariance. Code specialization is one example where the invariance as well as the values are crucial. Examples for code specialization are shown in Chapter X.

Chapter III

Value Profiling Methodology

Value profiling is used to find (1) the invariance of an instruction over the life-time of the program, (2) the top N result values for an instruction, and (3) the predictability of the instruction.

III.A TNV Table

The value profiling information required for compiler optimization ranges from needing to know only the invariance of an instruction to also having to know the top N values or a popular range of values. Figure III.1 shows a simple profiler to keep track of this information in pseudo-code. The value profiler keeps a Top-N-Value (TNV) table for the register being written by an instruction. There is always a TNV table associated with the entity that is the target of profiling. In the case of the parameters, there is one TNV table for each parameter. If we profile loads, then there will be a TNV table for each load, and when profiling memory locations, there will be one TNV table for each memory location.

The TNV table stores (value, number of occurrences) pairs for each entry with a least frequently used (LFU) replacement policy. When inserting a value into the table, if the entry already exists its occurrence count is incremented by the number of recent profiled occurrences. If the value is not found, the least frequently used entry is replaced.

There are also other counters counting the number of 0-values encountered, and a counter for the LVP. The number of times the parameter or instruction was visited

```
void InstructionProfile::collect_stats (Reg cur_value) {
      total_executed ++;
      if (cur_value == last_value) {
        lvp_1_metric ++;
        num_times_profiled ++;
      } else {
        LFU_insert_into_tnv_table(last_value, num_times_profiled);
        num_times_profiled = 1;
        last_value = cur_value;
      }
}
```

Figure III.1: A simple value profiler keeping track of the N most frequent occurring values, along with the last value prediction (LVP) metric.

and the PC (if its an instruction) is also part of the structure.

III.A.1 Replacement Policy for Top N Value Table

We chose not to use an LRU replacement policy, since replacing the least recently used value does not take into consideration the number of occurrences for that value. Instead, we use a LFU replacement policy for the TNV table. A straight forward LFU replacement policy for the TNV table can lead to situations where an invariant value cannot make its way into the TNV table. For example, if the TNV table already contains N entries, each profiled more than once, then using a least frequently used replacement policy for a sequence of $...XYXYXYXY...$ (where X and Y are not in the table) will make X and Y battle with each other to get into the TNV table, but neither will succeed. The TNV table can be made more forgiving by either adding a "temp" TNV table to store the current values for a specified time period which is later merged into a final TNV table, or by just clearing out the bottom entries of the TNV table every so often.

The approach we used in this thesis was to divide the TNV table into two distinct parts, the *steady part* and the *clear part*. The steady part of the table will never be flushed during profiling, but the clear part will be flushed once a *clear-interval* has expired. The clear interval defines the number of times an instruction is profiled, before the clear part of the table is flushed. The value which was encountered the least number of times in the steady part will be referred to as the *Least-Frequently-Encountered* (LFE)

Table III.1: Data sets used in gathering results for each program, and the number of instructions executed in millions for each data set.

	test		train	
Program	Name	Exe M	Name	Exe M
compress	ref	93	short	9
gcc	1cp-decl	1041	1stmt	337
go	5stone21	32699	2stone9	546
ijpeg	specmun	34716	vigo	39483
li	ref (w/o puzzle)	18089	puzzle	28243
m88ksim	ref	76271	train	135
perl	primes	17262	scrabble	28243
vortex	ref	90882	train	3189
applu	ref	46189	train	265
apsi	ref	29284	train	1461
fpppp	ref	122187	train	234
hydro2d	ref	42785	train	4447
mgrid	ref	69167	train	9271
su2cor	ref	33928	train	10744
swim	ref	35063	train	429
tomcatv	ref	27832	train	4729
turb3d	ref	81333	train	8160
wave5	ref	29521	train	1943

value. For a new value to work its way into the steady part of the table, the clear-interval needs to be larger than the frequency count of the LFE value. The clear-interval is computed by taking the maximum of the minimum clear interval size, and twice the number of times the LFE value was encountered. In this thesis we used a minimum clear interval size of 2000. The number of entries in the steady part and the clear part of the table depends on both, the *table size* and the *clear size*. The clear part of the table has clear size entries, the steady part has table size - clear size entries.

Chapter VI will investigate how changing either the table size, clear size, or clear interval affects the results.

III.B Methodology

To perform our evaluation, we collected information for the SPEC95 programs. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and FORTRAN compilers. We compiled the SPEC benchmark suite under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table III.A.1 shows the two data sets we used in gathering results for each program, and the number of instructions executed in millions.

We used ATOM [35] to instrument the programs and gather the value profiles. The ATOM instrumentation tool has an interface that allows the elements of the program executable, such as instructions, basic blocks, and procedures, to be queried and manipulated. In particular, ATOM allows an "instrumentation" program to navigate through the basic blocks of a program executable, and collect information about registers used, opcodes, branch conditions, and perform control-flow and data-flow analysis.

III.C Metrics

This section describes some of the metrics we will be using throughout the thesis. Certain metrics are used to describe the characteristics of one particular profile, others are used to compare two profiles. The metrics used to compare two profiles can be divided into (1) comparing values and (2) comparing invariances.

III.C.1 Profile Metrics

The metrics described in this section are used to describe the characteristics of one particular profile.

1. *Instruction Invariance (Inv-M).*

   When an instruction is said to have an "Invariance-M" of X%, this is calculated by taking the number of times the top M (M is also referred to as the history depth) values for the instruction occurred during profiling, as found in the final TNV table after profiling, and dividing this by the number of times the instruction was executed (profiled). By Inv-M we mean the percent of time an instruction spends executing its most frequent M values. The invariance for a profile is computed by summing the invariances of all instructions weighted by their visit count. The resulting sum is then divided by the total visit count. For the invariance we have two special cases:

   - *Invariance of Top Value (Inv-Top or I(t)).*

     This metric computes the invariance only for the most frequently occurring value and is computed by dividing the frequency count of the most frequently

occurring value in the final TNV table, by the number of times the instruction was profiled.

- *Invariance of All values (Inv-All or I(a)).*
  This metric computes the invariance for all values in the steady part of the final TNV-table. The number of occurrences for all values in the final TNV table are added together and divided by the number of times the instruction was profiled.

2. *Instruction's Last Value Prediction (LVP or lvp).*
   This metric measures the number of correct predictions made for an instruction. Keeping track of the number of correct predictions equates to the number of times an instruction's destination register was assigned a value that was the last value. To compute the LVP over all instructions in the program, all instruction LVP's are summed up and weighted by their instruction count. The LVP metric provides an indication of the temporal reuse of values for an instruction, and it is different from the invariance of an instruction. For example, an instruction may write a register with values X and Y in the following repetitive pattern ...$XYXYXYXY$.... This pattern would result in a LVP (which stores only the most recent value) of 0%, but the instruction has an invariance Inv-top of 50% and Inv-2 of 100%. Another example is when 1000 different values are the result of an instruction each 100 times in a row before switching to the next value. In this case the LVP metric would determine that the variable used its most recent value 99% of the time, but the instruction has only a 0.1% invariance for Inv-top. The LVP differs from invariance because it does not have state associated with each value indicating the number of times the value has occurred.

3. *Difference of LVP and Inv (Diff L/I or D(L/I)).*
   This metric shows the weighted difference between the LVP and Inv-Top. The difference is calculated on an instruction by instruction basis and is included into an average weighted by execution.

4. *Zero*
   This metric illustrates the percent zero values encountered during profiling. The

zero metric over the entire program is computed by computing the sum of all zero values over all instructions, and then dividing that number by the total number of instructions visited.

III.C.2 Metrics for Comparing Two Profiles

This section illustrates the metrics used to compare two arbitrary profiles. In this thesis we compare profiles of the same program profiled with different profiling parameters, as well as profiles generated by the regular profiler and convergent profiler. We examine both, the differences in their invariances and also the difference in their values.

1. *Overlap (Ol).*
   When comparing the two different data sets, the overlap represents the percent of instructions, weighted by execution, that were profiled in the first data set that were also profiled in the second data set.

Difference in Invariances

This section illustrates the metrics used to compare two profile's invariances.

1. *Difference in Invariance of Top Value (Diff-Top).*
   This metric shows the weighted difference in invariance between two profiles for the top most value in the TNV table. The difference in invariance is calculated on an instruction by instruction basis and is included into an average weighted by execution based on the first profile, for only those instructions that are executed in both profiles.

2. *Difference in Invariance of All Values (Diff-All).*
   This metric shows the weighted difference in invariance between two profiles for all values in the TNV table. As for Diff-Top, the difference in invariance is calculated on an instruction by instruction basis and is included into an average weighted by execution based on the first profile, for only those instructions that are executed in both profiles.

Difference in Values

This section illustrates the metrics used to compare the values in both profiles. When calculating the metrics in this section, we only look at instructions whose invariance in the first profile are greater than a given invariance threshold. The reason for only looking at instructions with an Inv-Top larger than a given threshold is to ignore all the instructions with random invariance. For variant instructions there is a high likelihood that the top values in the two profiles are different, and we are not interested in these instructions. Therefore, we arbitrarily chose an invariance threshold which is large enough to avoid taking random instructions into account.

1. *Sameness in Values (Same).*

   This metric shows the percent of instructions profiled in the first profile that had the same top value in the second profile. To calculate Same, the top value in the TNV table for the first profile is compared to the top value in the second profile. If they are equal, then the number of times that value occurred in the TNV table for the first profile is added to a sum counter. This counter is then divided by the summation of the frequencies of all top values, based on the first input.

2. *Finding the Top Value (Find-Top).*

   Find-Top shows the percent of time the top value for an instruction in the first profile is equal to one of the values for that instruction in the second profile. The difference between this metric and Same, is that Same only looks at the top entry of the second profile, whereas Find-Top considers all entries in the second profile.

3. *Finding all Values (Find-All).*

   Find-All shows the percent of entries in the first profile that are found in the second profile. For each instruction executed in both profiles we loop through the top N values of the TNV table in the first profile, where N is the smaller of both profile steady parts. One counter (TotVis) is incremented with the frequency count of each value from the first profile. If the value is found in the second profile, another counter (TotFound) is incremented with the same frequency count that was used to increment TotVis. Once all instructions have been processed, Find-All is computed by dividing TotFound by TotVis.

4. *Percent Above Threshold (PAT(invariance threshold)).*

   The percentage of all the top values in the first profile which exceed the invariance threshold.

III.D Graphs

The graphs which show the invariance, prediction accuracy and percent zero, show their results in terms of overall program execution, where the program execution is represented by the x-axis. The graph is formed by sorting all the instructions by their desired result, and then putting the instructions into 100 buckets filling the buckets up based on each entry's execution frequency. Then the average result, weighted by execution frequency, of each bucket is graphed. The y-axis entry is non-accumulative.

III.E Profiling Instructions

With the use of ATOM [35], profiling instructions is straight-forward. Each instruction can be profiled either before or after the instruction is executed. The destination register value is passed to the function which records the profiling information. Within that function, we add the register value to the TNV table.

III.F Profiling Memory Locations

Profiling memory locations is not as straight-forward as profiling instructions. To profile a given memory location we need to instrument all loads, and pass the effective address of the load and its destination register value into the instrumentation function. Using the effective address of the load we lookup the memory location in our data structure, and then update the TNV table with the register value.

For the purposes of this thesis, a memory location is a consecutive eight-byte chunk of memory which can belong to one of the following four categories:

1. *Stack* - A stack memory location is a local variable which upon invocation of the function is instantiated.

2. *Global* - A global memory location is located within the global data segment.

3. *Heap* - All memory locations which are allocated via dynamic memory allocation.

4. *Text* - All memory locations which are located within the text segment.

III.F.1 Algorithm Profiling Memory Locations

Profiling memory locations is fundamentally different from profiling instructions. When profiling instructions, we know exactly how many instructions there are before profiling begins. Memory locations can be dynamic, being allocated and freed during program execution. The only memory locations which will be present throughout the program execution are the ones within the Text or Global segment. Local variables will only be around as long as we're in a particular procedure, however they still need to be kept track of across different procedure invocations. Dynamically allocated memory locations will be around until they are freed.

The memory location profiler is a multi-step profiler. Step one generates an output file which contains the address, and number of times each memory location was accessed. Step two reads in the description file generated by the first step and then profiles the top N percentile of accessed memory locations. We only profile the top 99% of the accessed memory locations to reduce memory usage and time to profile.

The data we keep track of during the first pass:

1. The type of the memory location

2. The address at which the object was allocated

3. The number of times the object was accessed

4. The address of the object where it was allocated (Heap)

This information is used for value profiling. Part of the initialization for the second pass, is to read in the file generated by the first pass, and allocate a TNV structure for each memory location we wish to profile. The reason we profile each memory location and not the entire variable is best illustrated by a load instruction which retrieves data from several different entries in an array. Each entry might be constant, however the

invariance of the load would be dependent on the number of different entries accessed. To determine which memory locations will be profiled, we sort all the memory locations according to the number of times the objects were accessed, and then assigning a TNV structure to each memory location if it is within the N-th percentile.

For each executed load, the profiler will look up the TNV associated with the effective address of the load. If there is a TNV table for that memory location, then the register value will be inserted into the TNV structure. If there is no TNV structure, then this memory location was not in the top Nth percentile and does not need to be profiled.

III.G Profiling Parameters

Profiling parameters requires adding the instrumentation calls immediately before the function is called. The parameters are passed in registers 16 through 21. If there are more parameters, they are passed on the stack. For simplicity, we only instrument the parameters passed through the registers.

To determine if a register is being used as a parameter, the function is analyzed, and if there is a use of register 16 through 21, before it is being defined, then we know it must be a parameter, and we instrument that register.

III.H Invariance Thresholds

An integral part of our Find-Top, Find-All and Same metrics is the invariance threshold. An invariance threshold of 30% means, that only instructions which have an invariance exceeding that threshold will be considered in the computation of those metrics. Table III.2 shows the results for different thresholds. For each threshold, there are four columns in the Table. P which indicates what percentage of instructions were above that threshold. Sm (Same) is the percentage of instructions with the same top value in both profiles. Ft (Find-Top) represents the percent of time the top value for an instruction in the first profile is equal to one of the values for that instruction in the second profile. Fa (Find-All) is the percent of entries in the first profile which are found in the second profile.

Table III.2: Threshold Invariance. This table shows how the percentage of instructions greater the invariance threshold, Same, Find and Find-all change when varying the invariance threshold. The column represents the invariance threshold. P=Percent Above Threshold, Sm=Same, Ft=Find top, Fa=Find-all. The metrics are described in Section III.C.

Program	10				30				50				70			
	% P	% Sm	% Ft	% Fa	% P	% Sm	% Ft	% Fa	% P	% Sm	% Ft	% Fa	% P	% Sm	% Ft	% Fa
compress	43	100	100	99	42	100	100	100	41	100	100	100	41	100	100	100
gcc	45	99	100	97	41	100	100	99	35	100	100	99	27	100	100	100
go	35	98	99	95	31	99	99	97	27	100	100	99	21	100	100	100
ijpeg	18	96	96	92	17	100	100	100	16	100	100	100	15	100	100	100
li	67	57	58	68	36	98	98	92	31	100	100	97	25	100	100	100
perl	68	98	98	90	66	100	100	95	55	100	100	100	49	100	100	100
m88ksim	76	100	100	98	75	100	100	99	74	100	100	99	69	100	100	99
vortex	61	98	98	93	58	100	100	97	53	100	100	99	45	100	100	100
applu	33	100	100	100	31	100	100	100	27	100	100	100	24	100	100	100
apsi	18	99	100	95	13	100	100	100	11	100	100	100	10	100	100	100
hydro2d	62	100	100	99	58	100	100	100	46	100	100	100	40	100	100	100
mgrid	2	100	100	100	2	100	100	100	2	100	100	100	2	100	100	100
su2cor	17	100	100	99	16	100	100	100	15	100	100	100	15	100	100	100
swim	0	100	100	100	0	100	100	100	0	100	100	100	0	100	100	100
tomcatv	2	100	100	100	2	100	100	100	2	100	100	100	1	100	100	100
turb3d	38	98	98	94	34	100	100	100	8	100	100	100	5	100	100	100
wave5	10	99	99	97	10	100	100	100	10	100	100	100	10	100	100	100
C-Prgs	52	93	94	92	46	100	100	97	42	100	100	99	37	100	100	100
F-Prgs	20	100	100	98	18	100	100	100	13	100	100	100	12	100	100	100
Average	35	97	97	95	31	100	100	99	27	100	100	100	23	100	100	100

If the values of instructions with a low invariance would be used when finding the match in values, they would be a very low match. The reason for that is that we would be trying to match up random values. This provided the motivation for an invariance threshold, as described in the previous paragraph.

It is important to realize that the number in the PAT column is not the percentage of instructions which have an invariance greater than the threshold, but the percentage of all the top values for the invariant instructions. PAT for an invariance threshold of 70% is 23%. That means that the top values in the TNV table for all instructions which are considered to be invariant, using the 70% threshold, account for 23% of all executed loads on average.

Chapter IV

Profiling

This section will give a brief overview of profiling and its uses. The different profilers and resulting profiles will also be described briefly.

IV.A Introduction to Profiling

> ... a profile is a mapping from instances of some kind of program entity, like variables or procedures, into numeric weights.
>
> David Wall [38]

Profiling a program gives information about a program's state during execution. The information a profile includes depends on the entity being profiled. Common profiles are the procedure level profile which gives information about how often a procedure is invoked and how much of the overall program execution time is spent in a procedure. The basic block profile contains information such as the number of times each basic block was executed, and which basic blocks precede or follow a basic block (call-graph). Instruction profiles include information about individual instructions in the program. Other information such as cache misses, pipeline stalls, etc. may also be part of a given profile. Our research focuses on profiling for values. We want to know how invariant the instructions that produce a register value in a program are. For each entity being

profiled, we record the values encountered, and the frequency of each value.

Profiling a program can be a very time consuming process. To speed up that process, certain profilers select instructions to profile either randomly or on some selection criteria.

The Continuous Profiling Infrastructure (CPI) implemented by Anderson et al [1] collects random samples at a rate of 5200 samples/sec with only 1-3% in profiling overhead. The profiling architecture works as follows. Each processor generates an interrupt after a specified number of events, allowing the interrupted instruction and the various event counters to be captured into a buffer. Upon overflow of that buffer a separate daemon collects that sampled data and stores it into a database.

On out-of-order processors, it is difficult to correlate the event counter information with the instruction that actually was responsible for the event. ProfileMe [14], which is an extension to the DEC CPI [1], addresses that issue. Rather than counting events and sampling the PC, ProfileMe samples instructions. The authors also use paired sampling which provides concurrency information about the instructions currently being executed in the pipeline.

In this thesis we introduce a new type of profiler, the goal of which is to minimize time spent profiling. This is an intelligent sampler, because it samples instructions until a given convergence criteria is satisfied.

IV.B Different Input Data

For a profile to be valuable in supporting compiler optimizations, the information contained within the profile needs to be representative for different input data sets. How well profile data of different runs correlate, was originally investigated by David Wall [38]. The author shows that a profile gathered with a given data set still outperforms static analysis of the program when run on a second data set, by more than a factor of two. In Chapter V we present results on how our value profiler performs using different input data sets.

IV.C Uses for Profiling

There are many different uses for run-time profiles. One common use is to use profiling information to hand tune existing software [36], which is a common technique employed by software developers. Other uses for profiling are discussed in the following paragraphs.

IV.C.1 Branch Path Profiling

Fisher and Freudenberger [16] used profiling to gather information on the branching behavior of several programs. The profiling information was used to determine if past runs of a program with certain input data could help in predicting branches for future runs with different input data. Young and Smith [40] used profiling to capture the path preceding each conditional branch in a program. Calder et al [6] used profiling to show that common C and FORTRAN library routines have predictable behavior between different applications.

Optimizations which depend upon run-time profile data share one common objective. They intend to focus their optimizations on the most common executed code sections and program paths. However, the profiles discussed in this section do not measure path frequencies. To determine the most frequently executed path, one would have to estimate the path from the basic block profile. This approach does not always give the correct result. Ball and Larus [4] introduce a method to efficiently profile a program's executed paths.

IV.C.2 Code and Data Placement

Pettis and Hansen [30] used profile information to guide code-positioning. The idea is to place frequently executed code sections next to each other in the address space, thereby reducing the chances of cache conflicts. Their approach to reordering procedures and basic-blocks led to an average reduction in execution time of 15%. By applying an algorithm which takes the cache size, cache line size and procedure size into account to intelligently place procedures, Hashemi et al [22] were able to reduce the cache miss rate by 17% compared to the mapping algorithm of Pettis and Hansen [30].

Gloy et al [20] used profiling to collect temporal ordering information of procedure calls. Along with the cache configuration and procedure sizes, the temporal ordering information is used to estimate the conflict cost of a potential procedure ordering. Calder et al [8] use profiling and placement techniques from Gloy et al [20] to guide data placement.

IV.C.3 C++ Profiling

Providing run-time type feedback via a profile was the approach Hölzle and Ungar [23] suggested for inlining virtual function calls. Calder et al [7] use profile information to quantify the difference of C++ programs and C programs, and in an independent study the authors [5] use profile feedback to help predict indirect function calls. Dean et al [13] use profiling information to selectively inline virtual functions, based on their execution count.

IV.C.4 Miscellaneous

Profiles have been used to aid the compiler in promoting variables, which are executed the most frequently, into registers [33, 37].

Reinman et al [31] used profile information to determine the load-store dependency in programs. A load which is directly dependent upon a store might be able to bypass memory by using the value of the store directly.

Additional research [21, 9] focused on inlining functions given a run-time profile. Inlining functions permits the compiler to perform additional optimizations such as register allocation, code scheduling, common subexpression elimination, constant propagation, and dead code elimination. Profiles have also been used for instruction scheduling [11]. The order in which instructions are executed, is largely dependent on the control flow of the program, and the data dependency between instructions. Chen et al [11] used a control-flow profile to guide code motion. In addition, they used a memory-dependence profile to reorder loads and stores in a program.

Other research has used profiles on a broader basis. Chang et al [10] used profile information to assist in several classic code optimizations, such as loop invariant code removal, global variable migration, loop induction variable migration etc. The advantage

Table IV.1: Basic Block Quantile Table. Shown are the number of static basic blocks which account for a given percentile of the overall executed Basic Blocks. The columns represent the different percentiles. Tot is the number of basic blocks for the program and Vis is the number of basic blocks which have been visited at least once during profiling.

Program	1	3	5	10	20	30	60	90	95	99	Tot	Vis
compress	1	1	1	1	2	3	8	20	24	27	2731	451
gcc	1	3	7	27	95	198	1018	4494	6627	11224	80989	28212
go	1	2	2	5	14	31	226	1107	1652	3078	17844	10562
ijpeg	1	1	1	2	4	5	17	89	136	220	14670	3075
li	1	1	2	3	7	14	49	147	214	385	10884	2518
perl	1	1	2	3	7	12	40	124	142	156	25350	3354
m88ksim	1	1	1	2	5	8	67	216	292	448	13722	2996
vortex	1	2	3	5	10	16	62	378	777	2195	35339	13430
applu	1	1	1	1	2	4	14	37	50	62	31683	2515
apsi	1	1	2	4	10	18	41	138	222	418	33969	3497
hydro2d	1	1	2	3	6	9	21	76	92	286	32887	3530
mgrid	1	1	1	1	1	1	2	19	48	89	31879	2901
su2cor	1	1	1	2	4	6	17	122	207	276	32818	3598
swim	1	1	1	1	1	1	2	3	3	8	31276	2187
tomcatv	1	1	1	1	1	1	5	137	208	303	29328	2190
turb3d	1	1	1	2	4	7	23	72	88	111	32508	3104
wave5	1	1	2	3	6	10	39	112	190	368	34341	3427
C-Prgs	1	1	2	6	19	39	204	885	1298	2220	28790	9228
F-Prgs	1	1	2	2	4	7	23	109	188	412	41527	2695
Average	1	1	2	4	11	20	97	429	645	1156	28954	5385

of profile-guided optimizations is that the compiler can optimize the program sections which are executed the most. Using static compiler analysis to guide in optimization, the compiler can in certain instances actually degrade the performance of a program, for example by optimizing the less frequently taken path in a loop.

IV.D Different Profiles used in this Study

This section will discuss the various profilers implemented for this study. We will also show statistical data on how many instances of the entity being profiled, account for what percentile of the overall execution.

IV.D.1 Basic-Block Profile

The basic block profile contains information about how often each basic block was visited. Each basic block also contains a list of pointers to predecessors and successors. Table IV.1 shows how many basic blocks account for the overall executed basic

Table IV.2: Load Quantile Table. Shown are the number of static loads which account for a given percentile of the overall executed loads. The columns represent the different percentiles. Tot is the number of loads for the program and Vis is the number of loads which have been visited at least once during profiling.

Program	1	3	5	10	20	30	60	90	95	99	Tot	Vis
compress	1	1	1	2	5	7	20	49	54	60	3058	571
gcc	2	4	9	36	117	236	1095	4987	7310	12303	69728	29309
go	1	2	4	8	24	57	378	1734	2471	4382	17704	13500
ijpeg	1	2	3	5	11	18	66	182	265	420	14245	3576
li	1	1	2	4	8	15	51	137	193	324	7288	2051
perl	1	1	2	5	10	18	46	141	164	183	20049	3367
m88ksim	1	1	1	2	5	17	79	212	277	431	9509	2744
vortex	1	2	4	7	14	23	81	547	1168	3195	30878	16151
applu	1	2	3	5	22	46	283	648	715	837	25796	4798
apsi	2	5	7	14	34	61	175	414	550	1181	29404	6135
hydro2d	2	4	6	11	21	60	181	369	450	577	25090	4190
mgrid	1	2	3	5	10	15	29	55	83	193	23846	2764
su2cor	1	2	4	7	15	22	64	281	462	762	25570	4624
swim	1	2	2	4	8	11	22	33	35	36	23175	2341
tomcatv	1	2	2	4	8	12	42	90	98	256	23193	2470
turb3d	1	2	3	6	12	18	49	133	195	285	24658	3597
wave5	2	5	8	16	37	59	157	423	535	851	29171	4599
C-Prgs	1	2	3	9	24	49	227	999	1488	2662	21557	8909
F-Prgs	1	3	4	8	19	34	111	272	347	553	28738	3946
Average	1	2	4	8	21	41	166	614	884	1546	23668	6282

blocks. For the C programs about 9% of the visited basic blocks account for 90% of the program execution. For the FORTRAN programs it is only about 4%. Swim executes 90% of all its instructions in only 3 basic blocks, which is only 0.14% of its visited basic block, whereas `gcc` spends 90% of its execution in 4494 basic blocks which is about 15% of its basic blocks.

IV.D.2 Load Profile

To determine which loads would be most beneficial to optimize, we need to determine which loads are executed the most often, and then we also need to know if the load result is invariant. Table IV.2 shows how many loads are required to exceed a given percentile of the total program execution. The columns in the table are the percentiles. The first column in the table shows that most all programs have an instruction which is responsible for more than 1% of all instructions executed. Overall programs we see, that only 614, or 2.1% of all static load instructions, account for 90% of all executed loads. For optimization purposes, those would be the loads we are mostly concerned with.

Table IV.3: Procedure Quantile Table. Shown are the number of static procedures which account for a given percentile of the overall executed procedures. The columns represent the different percentiles. Tot is the number of procedures for the program and Vis is the number of procedures which have been visited at least once during profiling.

Program	1	3	5	10	20	30	60	90	95	99	Tot	Vis
compress	1	1	1	1	1	1	1	1	1	1	224	148
gcc	1	1	1	2	4	6	27	103	137	216	3986	2304
go	1	1	1	1	1	1	1	1	1	1	1146	643
ijpeg	1	1	1	1	1	1	1	6	8	17	1385	741
li	1	1	1	1	1	2	3	7	14	31	737	675
perl	1	1	1	1	1	1	1	5	7	9	1059	670
m88ksim	1	1	1	1	1	1	2	5	5	9	913	568
vortex	1	1	1	1	1	1	1	2	4	11	4592	1300
applu	1	1	1	1	2	2	4	13	21	38	1875	1031
apsi	1	1	1	1	1	1	1	1	1	10	2347	1131
hydro2d	1	1	1	1	2	3	5	7	7	12	1914	1063
mgrid	1	1	1	1	1	2	6	12	14	20	1909	1030
su2cor	1	1	1	1	1	1	1	1	1	3	1919	1050
swim	1	1	1	1	2	2	4	10	16	37	1848	1022
tomcatv	1	1	1	1	1	2	4	7	7	7	1818	991
turb3d	1	1	1	1	1	1	2	3	3	3	1947	1042
wave5	1	1	1	1	1	1	1	1	2	2	2075	1114
C-Prgs	1	1	1	1	1	2	5	18	25	41	2006	1007
F-Prgs	1	1	1	1	1	2	3	6	8	14	2522	947
Average	1	1	1	1	1	2	4	11	15	25	1864	972

IV.D.3 Parameter Profile

The parameter profile keeps information about every parameter in each procedure. We keep track of the same data as for instructions. Table IV.3 shows how many different procedures account for what percentile of all procedure calls. It is interesting to note, that for all programs up to 30% of all executed instructions are in the same procedure. 90% of execution time is spent in just 11 procedures, or 1.13% of all procedures in the program. As in the case of the loads, these few procedures, that make up the bulk of the execution is where one would most likely want to optimize. Richardson [32] suggests keeping a memoization cache of recently executed function results with their inputs. If a function call is made, the cache would be searched for those input values, and if there is an entry in the cache with all the correct input parameters, then the computation of the function could be replaced with the result value in the cache. So a function call could be replaced by a cache lookup.

Table IV.4: Instruction Quantile Table. Shown are the number of static instructions which account for a given percentile of all executed instructions. The Table columns represent the different percentiles. Tot is the number of instructions for the program and Vis is the number of instructions which have been visited at least once during profiling.

Program	1	3	5	10	20	30	60	90	95	99	Tot	Vis
compress	1	2	3	5	12	20	55	147	164	181	11522	2081
gcc	4	13	23	99	352	738	3575	16836	24599	41750	270258	106104
go	2	6	10	24	89	214	1379	6028	8687	15588	70215	49499
ijpeg	3	8	13	25	60	105	472	1159	1526	2075	61300	14992
li	1	3	5	12	33	62	184	511	732	1285	34913	8806
perl	1	4	7	15	32	56	150	471	547	608	88347	13405
m88ksim	1	2	3	6	19	49	302	894	1182	1976	46695	12584
vortex	3	7	11	22	45	80	293	1932	4111	11030	135448	62612
applu	2	5	9	17	63	153	949	2423	2689	3415	121666	19294
apsi	7	19	31	61	149	253	773	1842	2408	4347	138207	23941
hydro2d	4	12	20	39	77	185	648	1461	1846	3224	120794	18553
mgrid	2	4	7	13	25	38	76	158	357	703	113982	12813
su2cor	3	9	15	32	65	102	296	1082	1570	2713	122841	21415
swim	2	5	8	15	30	44	88	132	139	145	111516	9733
tomcatv	1	3	5	9	20	32	289	958	1136	1279	110333	8934
turb3d	3	9	14	28	56	84	232	733	1031	1412	119070	16115
wave5	7	20	33	66	157	249	686	1931	2482	4233	141582	19992
C-Prgs	2	6	9	26	80	165	801	3497	5193	9312	89837	33760
F-Prgs	3	10	16	31	71	127	449	1191	1518	2386	137499	16754
Average	3	8	13	29	76	145	615	2276	3247	5645	106982	24757

Table IV.5: Memory Location Quantile Table. This table shows the number of memory locations which account for a given percentile of the overall accessed memory locations. The columns represent the different percentiles.

Program	1	3	5	10	20	30	60	90	95	99	Tot
compress	1	1	1	2	5	7	21	2340	9247	32606	50464
gcc	2	7	12	33	92	182	1097	47817	90484	165320	448223
go	1	1	1	3	12	33	286	1752	2305	3839	22761
ijpeg	1	1	1	2	11	40	1800	44783	59461	74072	138493
li	1	1	2	5	10	19	163	7106	10620	23410	41256
m88ksim	1	1	1	1	1	4	28	220	589	123198	325667
vortex	1	1	2	3	5	9	93	1149	2503	14627	595919
applu	1	3	6	15	39	72	3269	59067	90069	114871	127814
apsi	2	6	10	22	57	107	379	4623	7910	15162	27001
hydro2d	1	3	5	14	37	185	3434	43175	50362	56218	478423
mgrid	262	1447	2632	5594	11517	17441	38075	72546	81681	109430	129072
su2cor	1	2	4	9	34	142	3898	29624	35215	39949	435979
swim	1	2	3	6	13	79	77343	166237	183119	196624	425781
tomcatv	3	41	143	5568	24095	42622	98203	169122	184561	196913	461455
wave5	1	1	2	10	44	119	3799	29791	97605	163101	356505
C-Prgs	1	2	3	8	22	48	566	17336	28784	70408	270464
F-Prgs	28	155	295	1681	5994	10340	32670	74446	91759	110381	290348
Average	18	97	185	1053	3754	6480	20631	53030	68143	95391	282892

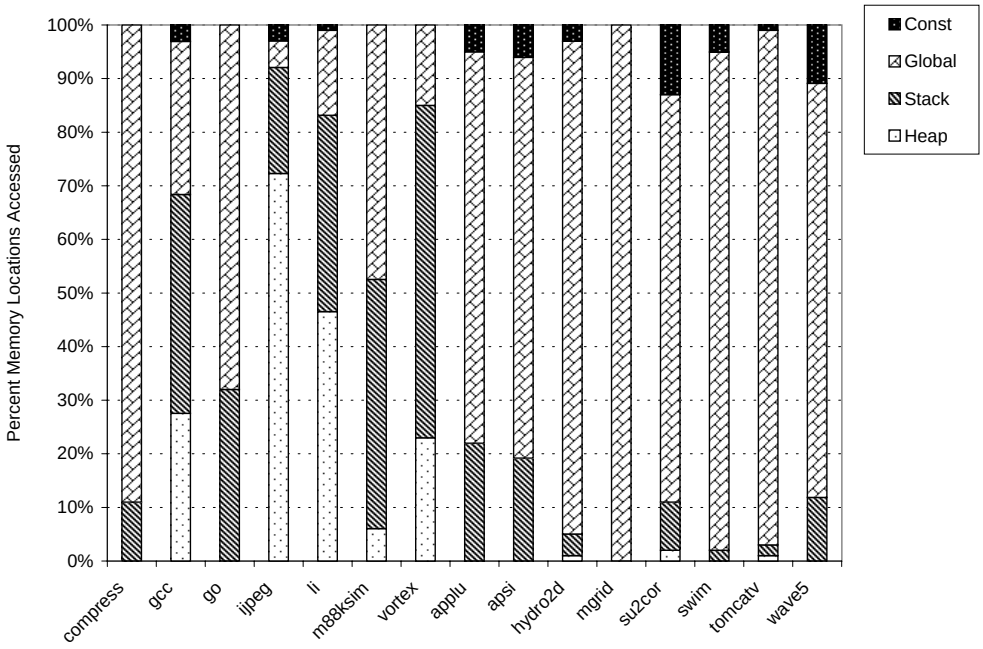

Figure IV.1: Memory Locations by Type. The graph shows the percentage each variable type (local, global, heap, const) accounts for in each program.

IV.D.4 ALL Profile

The All profile contains information about each register defining instruction of the program. Table IV.4 shows how many different instructions are necessary for a given percentile of program execution. The table shows that for the FORTRAN programs, only 1% of all instructions account for 90% of their dynamically executed instructions. For the C programs 3.1% of all static instructions are necessary to account for 90% of all dynamically executed instructions.

IV.D.5 Memory Location Profile

Table IV.5 shows how many memory locations account for a certain quantile of accessed memory locations. Only about 10 memory locations account for 10% of all

accessed memory locations. Notable exceptions are `mgrid` and `tomcatv`. Those programs perform repetitive accesses to huge global arrays. It is interesting to see how different the memory access pattern of `mgrid` is compared to all other programs.

Figure IV.1 shows the percentage each type of accessed memory location accounts for. Some programs, such as `mgrid`, `hydro2d`, `compress` operate on primarily one type of variable (Global), whereas others, such as `gcc` and `li` have a more balanced distribution.

Chapter V

Profiling Parameters and Instructions

This section examines the invariance and predictability of values for instruction types, procedure parameters and loads. When reporting invariance results we ignored instructions that do not need to be executed for the correct execution of the program. This included a reasonable number of loads for a few programs. These loads can be ignored since they were inserted into the program for code alignment or prefetching for the DEC Alpha 21164 processor.

For the results we used two sizes for the TNV table when profiling. For the breakdown of the invariance for the different instruction types (Table V.1) and (Table V.2), we used a TNV table which captured the top 25 values per instruction. For all the other results we used a TNV table which captured the top 3 values of each instruction.

V.A Breakdown of Instruction Type Invariance

Table V.1 and Table V.2 show the percent invariance for each program broken down into 14 different and disjoint instruction categories using the test data set. Each instruction category has two values associated with it. I represents the average percent invariance of the top value (Inv-Top) for a given instruction type, and X represents the percent of executed instructions that this class type accounts for when executing the

Table V.1: Breakdown of invariance by Integer instruction types. These categories include integer loads (ILd), load address calculations (LdA), stores (St), integer multiplication (IMul), all other integer arithmetic (IArth), compare (Cmp), shift (Shft), conditional moves (CMov). I is the percent invariance of the top most value (Inv-Top) for a class type, and X the dynamic execution frequency of that type. Results are not shown for instruction types that do not write a register (e.g., branches).

	ILd		LdA		St		IMul		IArth		Cmp		Shift		CMov	
Program	% I	% X	% I	% X	% I	% X	% I	% X	% I	% X	% I	% X	% I	% X	% I	% X
compress	44	27	88	2	16	9	15	0	11	36	92	2	14	9	0	0
gcc	46	24	59	9	48	11	40	0	46	28	87	3	54	7	51	1
go	36	30	71	13	35	8	18	0	29	31	73	4	42	0	52	1
ijpeg	19	18	9	11	20	5	10	1	15	37	96	2	17	21	15	0
li	40	30	27	8	42	15	30	0	56	22	93	2	79	3	60	0
perl	70	24	81	7	59	15	2	0	65	22	87	4	69	6	28	1
m88ksim	76	22	68	8	79	11	33	0	64	28	91	5	66	6	65	0
vortex	61	29	46	6	65	14	9	0	70	31	98	2	40	3	20	0
applu	65	1	19	8	26	10	3	0	11	21	76	4	54	0	58	0
apsi	65	4	17	11	11	11	18	0	25	13	94	3	34	0	46	0
fpppp	58	2	79	2	17	12	1	0	46	4	85	1	35	0	19	0
hydro2d	76	3	5	13	63	8	68	0	27	5	95	7	77	1	68	6
mgrid	77	1	0	14	6	2	3	0	9	2	97	2	63	0	48	0
su2cor	37	4	13	16	11	9	15	0	31	11	97	3	62	2	100	0
swim	56	0	0	18	1	9	45	0	1	2	100	2	16	0	2	0
tomcatv	62	2	2	7	3	8	31	0	24	3	99	3	51	0	1	1
turb3d	54	6	9	9	39	6	25	0	25	14	86	3	52	1	47	0
wave5	22	4	16	5	8	15	6	0	22	17	99	3	51	19	33	1
Avg	54	13	34	9	30	11	21	0	33	18	91	3	49	4	40	1

program. For the store instructions, the invariance reported is the invariance of the value being stored. The results show that for the integer programs, the integer loads (ILd), the calculation of the load addresses (LdA), and the integer arithmetic instructions (IArth) have a high degree of invariance and are frequently executed. For the floating point instructions the invariance found for the types are very different from one program to the next. Some programs `mgrid`, `swim`, and `tomcatv` show very low invariance, while `hydro2d` has very invariant instructions.

V.B Results for Parameters

Specializing procedures based on procedure parameters is a potentially beneficial form of specialization, especially if the code is written in a modular fashion for general purpose use, but is used in a very specialized manner for a given run of an application.

Table V.2: Breakdown of invariance by FP instruction types. These categories include floating point loads (FLd), floating point multiplication (FMul), floating point division (FDiv), all other floating point arithmetic (FArith), and all other floating point operations (FOps). I=%Invariance, X=% of executed instructions

	FLd		FMul		FDiv		FArth		FOps	
Program	% I	% X	% I	% X	% I	% X	% I	% X	% I	% X
compress	0	0	0	0	0	0	0	0	0	0
gcc	83	0	30	0	31	0	0	0	95	0
go	100	0	100	0	0	0	0	0	100	0
ijpeg	73	0	68	0	0	0	0	0	98	0
li	100	0	13	0	0	0	0	0	100	0
perl	54	3	50	0	19	0	34	0	51	1
m88ksim	59	0	53	0	66	0	100	0	100	0
vortex	99	0	4	0	0	0	0	0	100	0
applu	33	23	11	21	6	1	5	13	5	2
apsi	13	19	9	15	4	1	4	17	45	1
fpppp	27	31	4	24	3	0	2	21	54	1
hydro2d	62	24	79	11	32	1	73	11	81	4
mgrid	4	37	12	5	50	0	1	35	64	0
su2cor	13	16	4	17	3	0	2	13	36	2
swim	1	24	1	15	0	1	2	28	64	0
tomcatv	0	27	2	17	0	1	2	25	1	2
turb3d	37	17	31	13	2	0	38	15	57	1
wave5	10	38	2	34	2	18	1	31	32	2
Avg	43	14	26	10	12	1	15	12	60	1

Table V.3 shows the invariance of procedure calls. Instr shows the percent of instructions executed which were procedure calls for the test data set. The next four columns show the percent of procedure calls that had at least one parameter with an Inv-Top greater than 30, 50, 70, and 90%. This table shows results in terms of procedures, whereas Table V.4 shows the results in terms of parameter invariance and values.

Table V.4 shows the predictability of parameters. The metrics are described in detail in §III.C. The results show that the invariance of parameters is very predictable between the different input sets. The Table also shows that on average the top value for 48% of the parameters executed (passed to procedures) for the test data set had the same value 84% of the time when that same parameter was passed in a procedure for the train data set.

Table V.3: Invariance of procedure calls. Instr is the percent of executed instructions that are procedure calls. The next four columns show the percent of procedure calls that had at least one parameter with an Inv-Top invariance greater than 30, 50, 70 and 90%.

Program	% Instr	30%	50%	70%	90%
compress	0.27	0	0	0	0
gcc	1.23	54	49	34	18
go	1.08	10	9	2	0
ijpeg	0.15	78	21	19	19
li	2.45	46	31	25	14
perl	1.23	55	45	45	45
m88ksim	1.18	52	51	38	12
vortex	1.66	100	96	92	92
applu	0.00	92	91	91	11
apsi	0.11	100	100	100	68
hydro2d	0.00	99	98	98	98
mgrid	0.00	81	75	62	59
su2cor	0.03	100	100	100	100
swim	0.00	58	55	54	54
tomcatv	0.02	100	100	86	86
turb3d	0.11	100	100	100	0
wave5	0.02	100	100	100	53
Average	0.56	72	66	62	43

Table V.4: Invariance of parameter values. The results shown are I(t)=Inv-Top, I(a)=Inv-All, Ol=Overlap, D(t)=Diff-Top, D(a)=Diff-All, PAT=Percent Above Threshold, Sm=Same, F(t)=Find-Top, F(a)=Find-All. The metrics are in terms of parameters and are described in detail in §III.C.

	Test Params		Train Params		Comparing Params in Test Train Data Set						
						Invariance		Top Values			
Program	% I(t)	% I(a)	% I(t)	% I(a)	% Ol	% D(t)	% D(a)	% PAT(30)	% Sm	% F(t)	% F(a)
compress	0	0	1	1	100	0	0	0	96	96	96
gcc	30	39	30	40	100	2	1	26	97	99	98
go	9	15	11	19	100	3	2	5	86	96	95
ijpeg	36	64	36	64	100	0	1	31	35	35	35
li	33	41	42	51	99	7	3	31	95	95	91
perl	46	57	37	44	97	21	15	39	73	73	72
m88ksim	34	47	65	77	100	9	3	31	86	98	98
vortex	63	67	62	67	100	2	1	61	69	70	70
applu	80	91	53	69	100	26	3	80	99	99	99
apsi	61	71	71	89	100	3	3	59	72	72	68
hydro2d	90	92	91	92	100	0	0	90	100	100	100
mgrid	65	86	66	89	100	2	1	62	98	100	99
su2cor	84	85	83	84	100	0	0	84	100	100	100
swim	46	51	45	51	100	3	2	45	67	67	69
tomcatv	80	91	80	91	100	0	0	80	100	100	100
turb3d	47	82	47	82	100	0	0	41	100	100	100
wave5	71	88	63	95	86	6	2	57	55	55	64
Average	52	63	52	65	99	5	2	48	84	86	86

V.C Results for Loads

In this section we will discuss the results from profiling the load instructions. We will show figures for the invariance, the accuracy of LVP, and the percent of zero values encountered during profiling. In addition there will be results comparing the invariance, LVP accuracy, and Zero values for two different input sets. Finally, there will be results comparing the performance of LVP using different LVP thresholds.

V.C.1 Invariance of Loads

The graphs in Figure V.1 and Figure V.2 show the invariance for loads in terms of the percent of dynamically executed loads in each program. Figure V.1 shows the percent invariance calculated for the top value (Inv-Top) in the keep part of the TNV table for each instruction, and Figure V.2 shows the percent invariance for the top three values (Inv-All). See §III.D for an explanation on how the graph was created. The invariance is non-accumulative, and the x-axis is weighted by frequency of execution. Therefore, if we were interested in optimizing all instructions that had an Inv-Top invariance greater than 50% for `li`, this would account for around 30% of the executed loads. The Figure shows that some of the programs `compress` and `vortex` have 100% Inv-Top invariance for 40% or more of their executed loads, and `m88ksim` and `perl` have a 100% Inv-All invariance for almost 75% of their loads. It is interesting to note from these graphs the bi-polar nature of the load invariance for many of the programs. Most of the loads are either completely invariant or very variant. This indicates that one could use value profiling to accurately classify the invariance.

V.C.2 Percent Prediction Accuracy

The graph in Figure V.3 shows the Last Value Prediction accuracy graph for each program. Even though the LVP accuracy and the invariance measure different information, their average is almost identical, as can be seen by comparing Figure V.3 and Figure V.1. This is a result of the high number of instructions which are either constant or random throughout the program. The LVP accuracy metric and the invariance metric give the same result for completely invariant and completely random instructions. Given

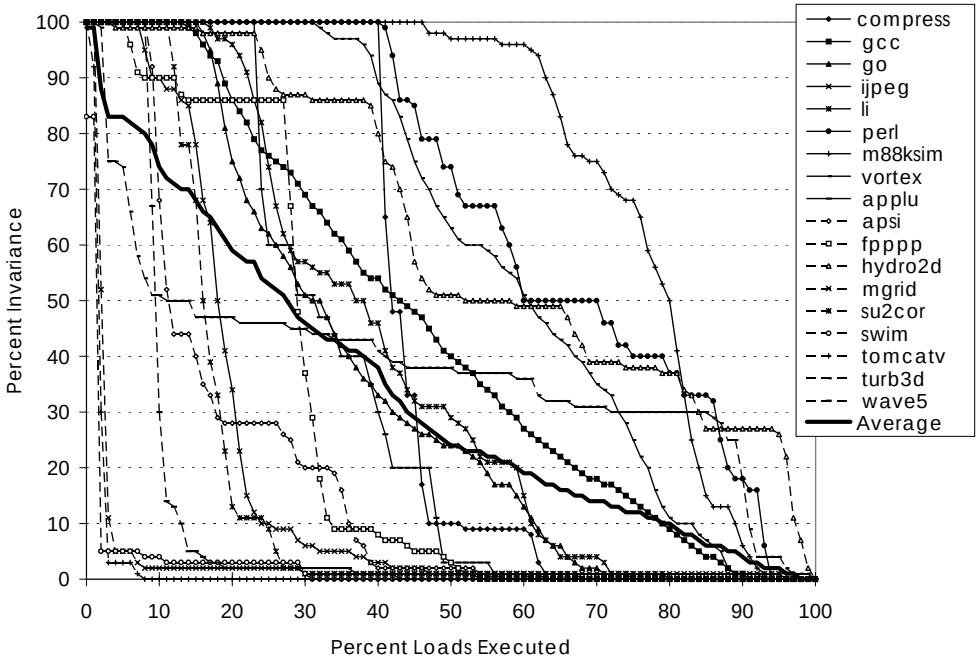

Figure V.1: Invariance of Loads (Top Value). The graph shows the percent invariance of the top value (Inv-Top) in the TNV table. The percent invariance is shown on the y-axis, and the x-axis is the percent of executed load instructions.

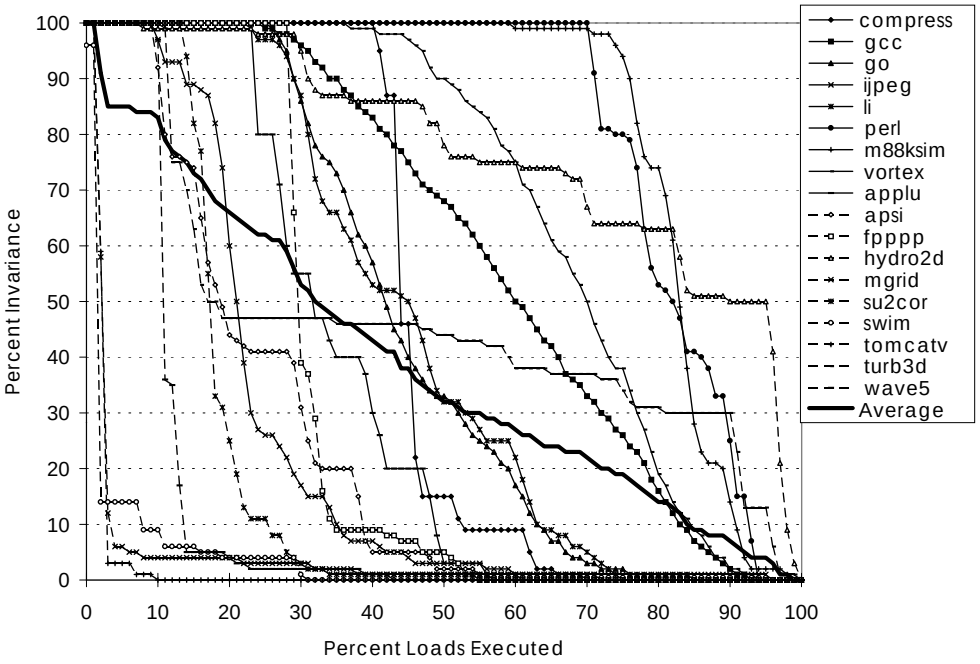

Figure V.2: Invariance of Loads (All Values). The graph shows the percent invariance of the top 3 values (Inv-All) in the TNV table. The percent invariance is shown on the y-axis, and the percent of executed load instructions on the x-axis.

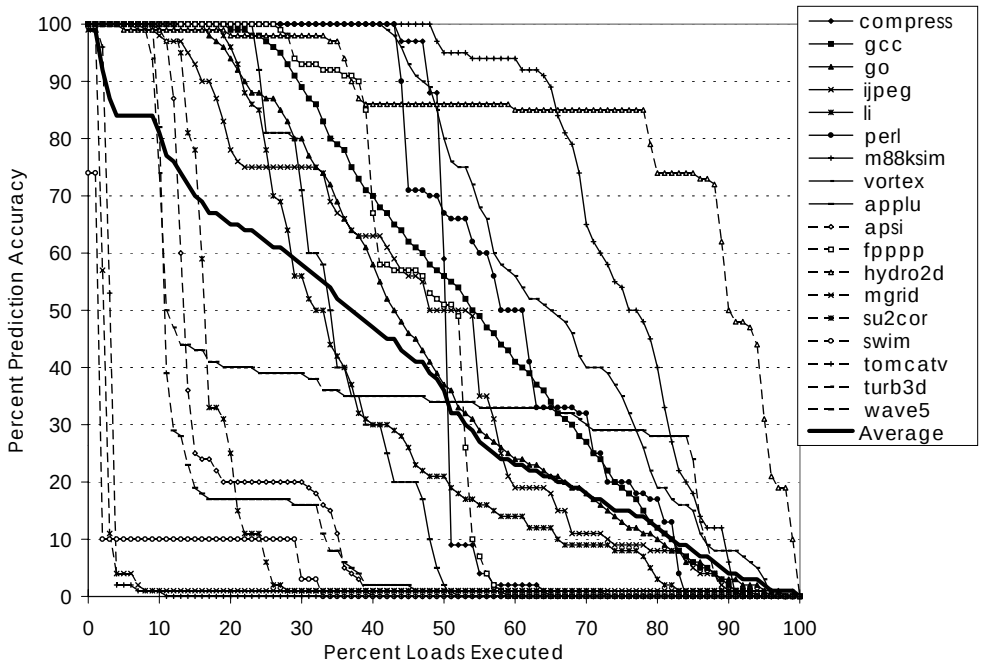

Figure V.3: Last Value Predictability of Loads. The graph shows the percent prediction accuracy using LVP. The percent prediction accuracy is shown on the y-axis, and the percent of executed load instructions on the x-axis. See §III.D for an explanation on how the graph was created.

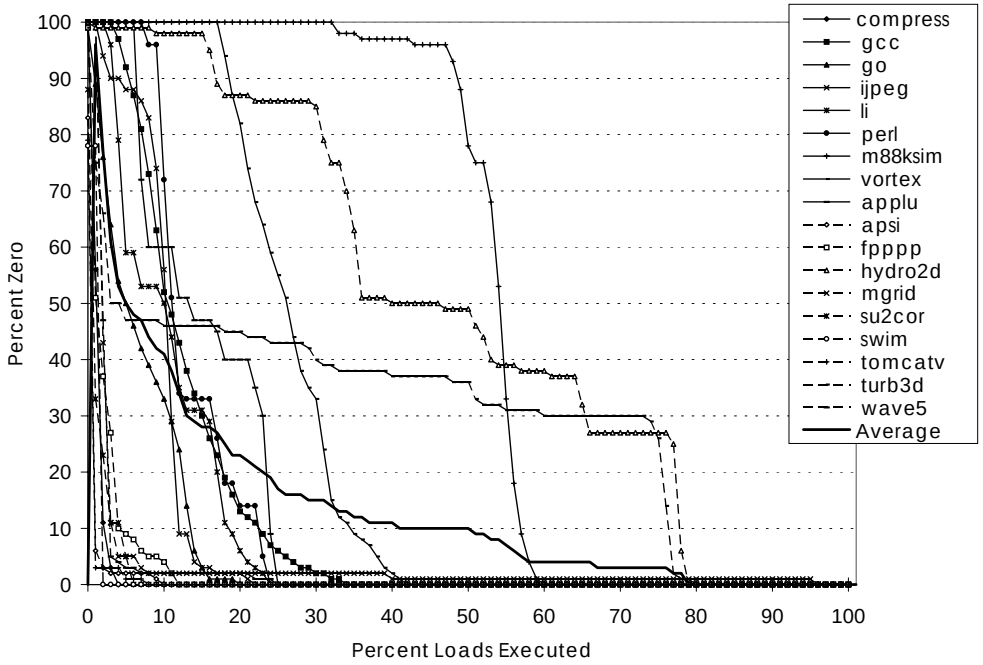

Figure V.4: Zero Value for Loads. The graph shows the percent of time the result of the load instruction had a zero value. The percent zero values is shown on the y-axis, and the percent of executed load instructions on the x-axis. See §III.D for an explanation on how the graph was created.

that those two categories account for almost 70% of all instructions, there are only 30% left, for which LVP and invariance can differ.

V.C.3 Percent Zeroes

The graph in Figure V.4 shows the percent of load instructions whose resulting register value is a zero value. `M88ksim` shows that almost 50% of each executed load instruction, loaded a value of zero.

V.C.4 Comparing Different Data Sets

This section will make comparisons between results for the test and train input, and it will also compare the predictability of instructions for different LVP thresholds.

Table V.5 shows the value invariance for loads. The invariance Inv-Top and Inv-All shown in this Table for the test data set is the average of the invariance shown in Figure V.1 and Figure V.2. The metrics are described in §III.C. The results show that the LVP metric has a 10% difference compared to the invariance on average, but the difference is large for a few of the programs. Table V.6 shows the difference in invariance of instructions between data sets to be very small. The results show that 31% of the loads executed in both data sets (using the 30% invariance threshold) have the same top invariant value 90% of the time.

Zero, LVP, and Inv

Table V.7 compares the invariance of the LVP accuracy, the percent zeroes and the invariance. Some of the results have already been shown in Table V.5 but are displayed here in a different format, to simplify the comparisons we will make. The percent zeroes and the percent invariance are very similar in both data sets. The results indicate that there is a high degree of similarity in the profile for the two data sets. This results confirms the observation made by David Wall [38], that profiles from different data sets have a high correlation.

Table V.5: Results for Load Values of Test and Train Data Set. The results shown are LVP, I(t)=Inv-Top, I(a)=Inv-All and D(l/i)=Diff(L/I) for the two data sets. The metrics are described in §III.C.

	Test				Train			
Program	% lvp	% I(t)	% I(a)	% D(l/i)	% lvp	% I(t)	% I(a)	% D(l/i)
compress	50	44	46	8	53	48	50	9
gcc	55	46	60	19	53	45	59	19
go	47	35	45	19	49	38	49	19
ijpeg	46	19	24	29	48	19	24	30
li	38	39	45	12	46	43	49	17
perl	60	67	82	11	55	57	71	12
m88ksim	75	76	83	5	81	84	89	4
vortex	66	60	68	12	66	61	73	12
applu	35	34	34	13	36	35	36	13
apsi	18	18	23	5	27	29	38	7
hydro2d	83	62	77	23	84	63	78	23
mgrid	4	4	4	0	4	4	5	0
su2cor	18	17	19	2	18	17	19	2
swim	3	1	2	2	23	7	10	16
tomcatv	3	2	2	2	4	3	3	2
turb3d	36	38	46	8	41	42	50	8
wave5	15	11	12	5	33	22	24	13
Average	38	34	39	10	42	36	43	12

Table V.6: Comparing Loads for Test and Train Input. The results shown are Ol=Overlap, D(t)=Diff-Top, D(a)=Diff-All, PAT=Percent Above Threshold, Sm=Same, F(t)=Find top, F(a)=Find-All. The metrics are described in §III.C.

Program	% Ol	% D(t)	% D(a)	% PAT(30)	% Sm	% F(t)	% F(a)
compress	100	2	1	42	96	96	97
gcc	100	3	1	41	95	96	94
go	100	4	2	31	94	98	98
ijpeg	100	1	1	17	94	94	92
li	100	7	3	35	94	95	94
perl	91	13	6	59	81	85	80
m88ksim	100	4	2	75	96	97	96
vortex	100	5	2	58	82	86	85
applu	100	1	0	31	71	71	70
apsi	100	5	2	13	55	55	50
hydro2d	100	1	0	58	100	100	100
mgrid	100	0	0	2	100	100	100
su2cor	100	0	0	16	99	99	99
swim	100	1	1	0	100	100	100
tomcatv	100	0	0	2	99	99	99
turb3d	100	3	1	34	100	100	100
wave5	99	6	2	10	80	80	81
Average	99	3	1	31	90	91	90

Table V.7: LVP/ZERO/INV Comparison. Shows the difference between LVP, ZERO and Inv-Top from the test input to the train input.

	LVP			ZERO			INV		
Program	test	train	Diff	test	train	Diff	test	train	Diff
compress	50	53	1	1	2	0	44	48	2
gcc	55	53	2	13	12	1	46	45	3
go	47	49	3	7	8	1	35	38	4
ijpeg	46	48	3	10	10	0	19	19	1
li	38	46	9	10	11	2	39	43	7
perl	60	55	14	14	14	6	67	57	13
m88ksim	75	81	4	53	51	3	76	84	4
vortex	66	66	2	27	28	1	60	61	5
applu	35	36	0	15	15	0	34	35	1
apsi	18	27	2	1	2	1	18	29	5
hydro2d	83	84	1	48	48	1	62	63	1
mgrid	4	4	0	4	4	0	4	4	0
su2cor	18	18	0	2	2	0	17	17	0
swim	3	23	16	0	0	0	1	7	1
tomcatv	3	4	0	0	1	0	2	3	0
turb3d	36	41	3	30	34	3	38	42	3
wave5	15	33	16	3	7	5	11	22	6
Average	38	42	5	14	15	1	34	36	3

LVP

In this section we present the results for comparing the LVP of two different data sets for different LVP thresholds. An LVP threshold is used to limit the instructions which will be predicted, based on their prediction accuracy. By limiting the instructions to be predicted, the Value History Table (VHT) will only be occupied by highly predictable instructions, which leads to an increased utilization of the VHT, and a reduced number of mispredictions.

Table V.8 shows (1) what percentage of loads have an LVP value greater than a threshold, (2) for those loads in (1), how does the second input compare. For each LVP threshold the four results shown are.

- The percentage of loads with an LVP greater than the threshold for the train input.
- The average LVP over all loads which exceeded the threshold in the train input.
- The percentage of loads in the test input classified above LVP threshold using only those loads which were marked as invariant by the train data set.
- The average LVP for all loads in the test input identified for prediction.

The results show that by not using an LVP threshold the prediction accuracy of the test and train input are 38% and 42%. Using a threshold of 70% decreases the number of predicted loads to 33%, but in turn increases the prediction accuracy of those loads to 94%. This reduces the mispredictions from 58% to 6%.

Table V.8: LVP Threshold Table. For each LVP threshold this table shows (1) th = the percentage of load instruction greater a LVP threshold, (2) lvp = the average LVP of those loads, (3) th = the loads in the second dataset corresponding to the loads in the first dataset which are included in (1), and (4) lvp = the average LVP over the loads in (3).

	LVP Threshold													
	all		10%				30%				50%			
	Train	Test	Train		Test		Train		Test		Train		Test	
Program	lvp	lvp	th	lvp	th	lvp	th	lvp	th	lvp	th	lvp	th	lvp
compress	54	50	56	94	52	94	53	99	50	99	53	99	50	99
gcc	53	55	81	65	81	67	67	75	67	76	53	84	54	85
go	49	47	81	59	81	58	59	74	57	74	44	86	43	86
ijpeg	48	46	82	57	83	55	57	73	58	74	44	82	44	82
li	46	38	62	72	59	60	54	80	50	67	39	96	33	81
perl	55	57	80	67	78	66	67	78	65	74	54	87	45	85
m88ksim	81	75	87	93	85	87	85	95	81	90	84	95	80	91
vortex	66	66	90	73	88	74	77	82	76	83	65	90	65	90
applu	36	35	48	75	47	74	39	87	38	87	34	93	33	93
apsi	27	18	39	67	33	52	26	92	15	90	24	97	13	97
hydro2d	84	83	98	85	98	84	94	88	94	87	88	91	89	90
mgrid	4	4	4	72	4	69	3	94	2	94	3	94	2	94
su2cor	18	18	25	70	25	70	20	86	20	86	17	95	17	95
swim	23	3	34	66	31	10	33	69	31	10	33	69	31	10
tomcatv	4	3	5	88	4	89	5	89	4	90	4	96	3	96
turb3d	41	37	87	46	87	42	84	47	84	43	14	90	12	91
wave5	33	15	44	74	42	34	40	80	38	38	33	90	29	45
C-Prgs	56	54	77	73	76	70	65	82	63	80	54	90	52	87
F-Prgs	30	24	43	71	41	58	38	81	36	69	28	90	25	79
Average	42	38	59	72	58	64	51	82	49	74	40	90	38	83

	LVP Threshold															
	70%				90%				95%				99%			
	Train		Test		Train		Test		Train		Test		Train		Test	
Program	th	lvp	th	lvp	th	lvp	th	lvp	th	lvp	th	lvp	th	lvp	th	lvp
compress	53	99	50	99	47	100	44	100	47	100	44	100	47	100	44	100
gcc	38	94	39	94	28	99	29	99	26	99	26	100	20	100	20	100
go	36	92	34	92	23	99	22	99	21	100	20	99	17	100	17	100
ijpeg	35	87	35	87	16	98	16	98	12	99	12	99	8	100	8	100
li	38	97	32	82	33	99	28	82	29	99	24	92	26	100	22	94
perl	43	93	39	88	30	100	28	95	29	100	28	97	29	100	28	97
m88ksim	78	98	72	95	73	99	63	97	70	99	54	98	53	100	49	100
vortex	55	96	55	96	46	99	47	99	42	100	44	100	38	100	40	100
applu	31	96	30	96	25	100	24	100	25	100	24	100	24	100	24	100
apsi	23	98	12	99	21	100	11	100	21	100	11	100	19	100	9	100
hydro2d	87	91	88	90	39	99	38	98	37	100	36	99	34	100	34	99
mgrid	3	95	2	94	3	95	2	94	1	100	0	100	1	100	0	100
su2cor	16	97	16	97	13	100	13	100	13	100	13	100	13	100	13	100
swim	6	100	0	100	6	100	0	100	6	100	0	100	6	100	0	100
tomcatv	4	98	3	99	4	100	3	100	4	100	3	100	4	100	3	100
turb3d	12	95	10	99	10	100	10	100	10	100	10	100	8	100	8	100
wave5	31	92	27	48	19	99	13	84	18	100	12	89	17	100	10	100
C-Prgs	47	94	44	92	37	99	35	96	34	100	31	98	30	100	28	99
F-Prgs	24	96	21	91	15	99	13	97	15	100	12	98	14	100	11	100
Average	35	95	32	91	26	99	23	97	24	100	21	98	21	100	19	99

Chapter VI

Value Profiling Design Alternatives

This section analyzes the parameter settings for the TNV table. The goal is to determine what settings provide the most accurate value profiling. There are three different parameters that can affect value profiling. The clear size, the clear interval size and the table size, whereas the table size and clear size have more influence than the clear interval size.

For all the results we will show two metrics.

- The percent of time the top value for an instruction in the configuration profiled is equal to one of the values for that instruction in the reference profile (Find-Top).
- The weighted difference in invariance between two profiles for all values in the TNV table (Diff-All).

The reference table used throughout this chapter has a table size of 50 entries, a clear interval size of 2000 and a clear size of half the table size, i.e. 25 entries.

VI.A Steady State Entries

Increasing the steady size increases the number of values which are stored in the TNV table. A steady size of 1 means that only 1 value will retire from the steady part of the table after profiling is finished. The results show, that the most significant

Table VI.1: Steady Size Table. This Table shows how changing the number of entries in the steady part of the TNV table affects the results. For each steady size the following metrics are given: D(a)=Diff-All, F(t)=Find-Top. The metrics are described in detail in §III.C.

	Steady Sizes											
	1		2		3		4		5		10	
Program	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)
compress	0.4	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.0	100.0
gcc	3.2	92.4	0.7	99.5	0.4	99.8	0.0	100.0	0.0	100.0	0.0	100.0
go	4.4	85.3	1.1	98.2	0.8	99.3	0.6	99.8	0.5	99.8	0.2	100.0
ijpeg	1.0	96.0	0.5	100.0	0.4	100.0	0.4	100.0	0.3	100.0	0.2	100.0
li	7.2	87.0	3.1	97.7	2.6	98.0	2.0	99.5	1.6	99.5	1.0	100.0
perl	11.7	83.3	3.2	99.3	2.6	100.0	1.9	100.0	1.5	100.0	0.2	100.0
m88ksim	2.6	97.1	0.7	99.9	0.4	99.9	0.3	100.0	0.2	100.0	0.0	100.0
vortex	6.9	89.6	1.9	99.8	1.5	100.0	1.1	100.0	1.0	100.0	0.4	100.0
applu	0.3	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
apsi	0.3	99.3	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0
hydro2d	3.5	90.0	0.6	99.1	0.2	100.0	0.1	100.0	0.1	100.0	0.0	100.0
mgrid	0.3	100.0	0.2	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.0	100.0
su2cor	0.6	97.2	0.1	99.8	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
swim	0.0	97.9	0.2	100.0	0.2	100.0	0.2	100.0	0.1	100.0	0.0	100.0
tomcatv	0.1	92.3	0.0	99.8	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
turb3d	1.4	97.2	1.0	98.4	0.5	100.0	0.4	100.0	0.2	100.0	0.1	100.0
wave5	0.1	99.2	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
C-Prgs	4.7	91.3	1.4	99.3	1.1	99.6	0.8	99.9	0.6	99.9	0.3	100.0
F-Prgs	0.7	97.0	0.3	99.7	0.1	100.0	0.1	100.0	0.1	100.0	0.0	100.0
Average	2.6	94.3	0.8	99.5	0.6	99.8	0.4	100.0	0.3	100.0	0.1	100.0

performance increase results from increasing a steady size of 1 entry to a steady 2 of two entries. Once the steady size reaches 4 entries, Find-Top is 100% and Diff-All is 0.4%, whereas the Diff-All for the FORTRAN programs has been reduced to 0.1%.

Figure VI.1 shows the percentage of loads captured when using different steady sizes. For each program there are four bars. The first bar uses only the top value, the second bar the top three values, the third bar the top five values, and the fourth bar shows the loads captured for the top 10 values. Some programs show substantial increases in loads captured when comparing using only the top value to using the top 3 or 5 values, and only a moderate gain to using the top ten values. `Perl` and `m88ksim` illustrate that behavior. Other programs such as `apsi` and `ijpeg` also have substantial increases from the top five to the top ten entries.

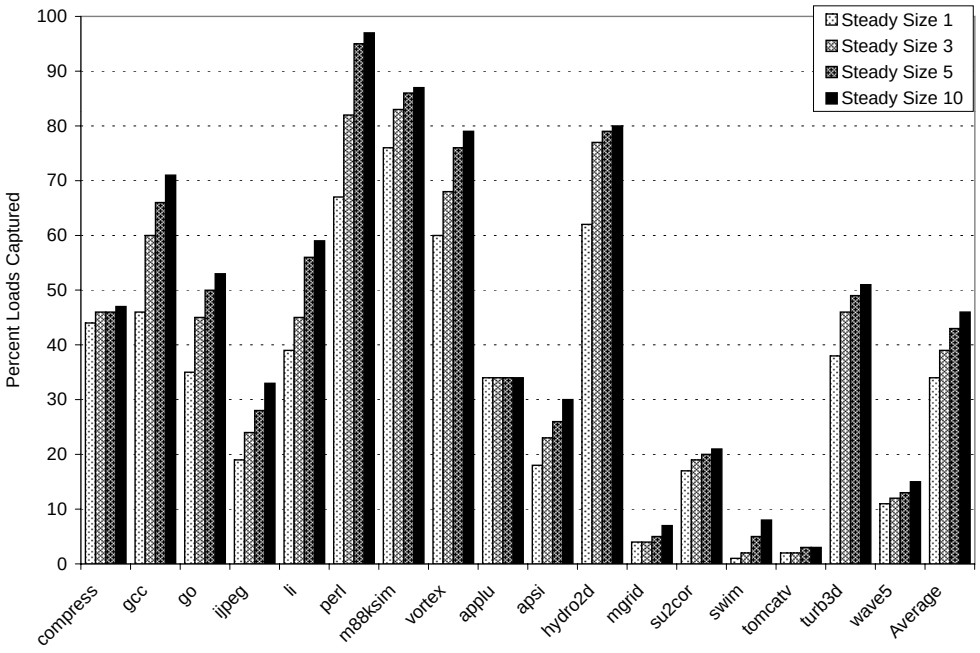

Figure VI.1: Percentage of Loads Captured for Top N Values. This graph shows the percentage of loads captured using the top N values from the TNV table with table size 50. The first bar shows the loads captured by the top entry in the TNV table, the second bar shows the loads for the top three entries, the third bar for the top five, and the fourth bar for the top ten entries.

VI.B Clear Size

Increasing the clear size, increases the number of values that are cleared from the TNV table. This gives values occurring later during program execution a better chance to get into the steady part of the TNV table. For a more detailed description on the replacement policy used see §III.A.1.

Table VI.2 illustrates the results for different clear sizes. A clear size of 0 means, that once the TNV table is filled, no new values can make it into the table. Even if these first encountered values were completely random and have only been encountered once.

For all FORTRAN and C programs the optimal clear size for a table size of

Table VI.2: Clear Size Table. This Table shows how changing the clear size affects the results. The clear size represents the number of entries which are cleared from the TNV Table after the clear interval has completed, for a table size of six. The lower this number the lower the chance for values which occur later during a programs execution to make it into the final TNV table. For each clear size the metrics are D(a)=Diff-All, F(t)=Find-Top. The metrics are described in detail in §III.C.

	Clear Sizes											
	0		1		2		3		4		5	
Program	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)
compress	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.2	100.0	0.4	100.0
gcc	1.0	99.0	1.0	99.0	0.0	100.0	0.4	99.7	2.0	100.0	4.0	98.0
go	0.6	99.0	0.6	99.0	0.5	99.7	0.8	99.1	1.7	97.9	3.7	90.7
ijpeg	0.3	100.0	0.4	100.0	0.3	100.0	0.4	100.0	0.9	100.0	1.6	99.9
li	3.7	92.1	3.7	92.1	2.1	99.5	2.6	98.0	3.3	95.9	4.2	95.1
perl	3.0	98.7	3.0	98.7	2.0	100.0	2.6	100.0	3.6	100.0	6.9	95.4
m88ksim	0.4	99.8	0.4	99.8	0.3	100.0	0.4	99.9	0.9	99.9	2.3	99.8
vortex	1.6	98.8	1.7	98.6	1.2	100.0	1.5	100.0	2.2	99.8	3.5	97.0
applu	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.2	100.0
apsi	0.2	99.8	0.2	99.8	0.0	100.0	0.1	100.0	0.8	100.0	1.4	99.5
hydro2d	0.2	100.0	0.2	100.0	0.2	100.0	0.2	100.0	1.6	100.0	4.1	100.0
mgrid	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.2	100.0	0.3	100.0
su2cor	0.1	99.5	0.1	99.7	0.0	99.8	0.0	99.9	0.2	99.8	0.6	99.6
swim	0.0	100.0	0.0	100.0	0.1	100.0	0.2	100.0	0.5	100.0	0.7	97.9
tomcatv	0.0	99.6	0.0	99.5	0.0	99.5	0.0	99.6	0.0	99.6	0.1	95.8
turb3d	0.5	100.0	0.5	100.0	0.5	100.0	0.5	100.0	0.9	100.0	2.6	99.9
wave5	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.2	100.0	0.4	99.8
C-Prgs	1.3	98.4	1.4	98.4	0.8	99.9	1.1	99.6	1.9	99.2	3.3	97.0
F-Prgs	0.1	99.9	0.1	99.9	0.1	99.9	0.1	99.9	0.5	99.9	1.2	99.2
Average	0.7	99.2	0.7	99.2	0.4	99.9	0.6	99.8	1.1	99.6	2.2	98.1

six entries is two. Not clearing any entries in the TNV table results in a decreased performance for almost all programs. Notably for `li`, the clear size makes a significant difference. For a more detailed discussion on the metrics, please refer to §III.C.

VI.C Clear Interval

Increasing the clear interval gives less frequently occurring values a higher chance to make it into the TNV-table, once the TNV-table has been filled. This results from increasing their chances of getting their visit count incremented beyond the visit count of the LFE value. For a more detailed description on the replacement policy used see §III.A.1.

Table VI.3 shows the results when varying the interval size. For the FORTRAN programs, the optimal clear interval size is achieved with 100,000 instructions. Find-Top

Table VI.3: Clear Interval Table. This Table shows how changing the clear interval size affects the results. The clear interval is the number of instructions profiled before checking for convergence. The greater this time period is, the more chances values have to make it into the TNV table. The results shown are D(a)=Diff-All and F(t)=Find-Top. The metrics are described in detail in §III.C.

	Clear Intervals									
	500		2000		100000		500000		500000000	
Program	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)	D(a)	F(t)
compress	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0
gcc	1.0	100.0	0.4	99.8	1.0	100.0	1.0	100.0	1.0	100.0
go	0.8	99.8	0.8	99.3	0.6	100.0	0.5	99.9	0.6	99.0
ijpeg	0.5	100.0	0.4	100.0	0.3	100.0	0.3	100.0	0.3	100.0
li	3.1	97.4	2.6	98.0	2.0	98.5	1.9	100.0	3.7	92.1
perl	3.2	100.0	2.6	100.0	0.0	100.0	0.1	100.0	3.0	98.7
m88ksim	0.4	99.9	0.4	99.9	0.4	99.9	0.4	100.0	0.5	99.8
vortex	1.6	100.0	1.5	100.0	1.1	100.0	0.8	100.0	1.6	98.9
applu	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
apsi	0.1	100.0	0.1	100.0	0.0	100.0	0.1	100.0	0.2	99.8
hydro2d	0.2	100.0	0.2	100.0	0.1	100.0	0.1	100.0	0.2	100.0
mgrid	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0
su2cor	0.1	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.1	99.5
swim	0.2	100.0	0.2	100.0	0.0	100.0	0.0	100.0	0.0	100.0
tomcatv	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	99.8
turb3d	0.5	100.0	0.5	100.0	0.5	100.0	0.2	100.0	0.5	100.0
wave5	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
C-Prgs	1.3	99.6	1.1	99.6	0.7	99.8	0.6	100.0	1.4	98.6
F-Prgs	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.1	99.9
Average	0.7	99.8	0.6	99.8	0.4	99.9	0.3	100.0	0.7	99.3

is 100% and Diff-All 0.1%. For the C programs, the clear interval needs to be larger to achieve optimal results. For the extremely large clear interval of 500M instructions, we notice a drop in Find-Top, as well as an increase in Diff-All. This indicates, that some values make it into the table at later stages during program execution. The C-Programs seem to be more sensitive to changes in the clear interval than the FORTRAN programs. The results for the C-Programs indicate that a longer clear interval leads to better results, however an upper bound is necessary.

Chapter VII

Profiling Memory Locations

In this section we extend the profiling of loads to profiling of memory locations. We want to know how the loads correspond to the memory locations they load. For our purposes, a *memory location* represents any eight-byte piece of memory and a *variable* any consecutive part of memory.

Profiling memory locations was motivated by instructions which access invariant memory locations but result in being variant. Two possible scenarios are given to illustrate this.

- A load which accesses any complex data structure in which the values of the different memory locations are different, but each memory location is invariant.
- Instructions in a procedure which access different data depending on the caller.

See Figure VII.1 for an example of the latter. The example illustrates how it is possible for an instruction (the add instruction) to operate on invariant data, and have a variant result. The procedure `ProcA` is called from two locations each passing in a different memory location. The invariance of the add instruction in the procedure depends on the values in those memory locations. For this example, the invariance of the add instruction would be 50%, even though the add instruction works on constant data. Profiling memory locations would reflect the invariance correctly.

In addition to the data kept in the TNV table for profiling instructions, for profiling memory locations, we keep a list of PC's which accessed the memory location. As for the values, there is a counter indicating how often that particular PC accessed

```
long v1 = 1;
long v2 = 2;
....
ProcA (&v1);
ProcA (&v2);
...

void ProcA (long *p)
{
   long a = 2 + *p;
}
```

Figure VII.1: Procedure Called with Different Addresses. This Figure illustrates how an instruction (the add instruction) which operates on constant data, can be variant.

the memory location. Additionally we keep a list of PCs for each individual value. With this information we can determine exactly which PCs were responsible for what value.

VII.A Number of Loads Accessing a Memory Location

Table VII.1 shows the percentage of memory locations which are accessed by a certain number of static loads. Each column in the Table represents a range of static load instructions. The entries represent the percentage of memory locations which are accessed by the given number of distinct loads.

The Table shows that 30% of all accessed memory locations are accessed by either one or two different load instructions. For `wave` that same metric results in 79% and for `go` in 6%. For `vortex`, 42% of all accessed memory locations are accessed 20 to 50 different static load instructions.

VII.B Number of Memory Locations Accessed by a Load

Table VII.2 illustrates how many different memory locations are accessed by a percentage of static load instructions. 16% of all static load instructions accessed only one memory location. 29% of all static loads executed, accessed between two and ten different memory locations. An extreme case is `fpppp` for which 61% of the static loads executed, accessed 100,000 different memory locations. In a case like this it would

be very unlikely to achieve a high degree of invariance for the load instructions, unless all those memory locations contain the same values. Looking at the load invariance in Figure V.1 for `fpppp` we notice how the invariance drops off after about 30% of all loads executed. This corresponds to the result presented in this paragraph.

VII.C Invariance of Memory Locations

Figure VII.2 displays the invariance of memory locations with respect to program execution. It is interesting to note, that for certain programs, the invariance is much higher for memory locations as for the loads of the particular program. The invariance of the loads for `su2cor` and `wave5`, as shown in Figure VII.2, is 17% and 11%. The invariance of the memory locations accessed by `su2cor` and `wave5` are shown in Figure VII.2 to have an invariance greater 60%. Another observation we can make by comparing the invariance of memory locations with loads is that the memory locations have a higher invariance for a longer execution time, but then become variant fast. This abrupt change from being invariant to variant is what we refer to as the *bi-polarity* of the invariance.

Results show that 30% of all loads have an average invariance of 50% (Figure V.1). For the memory locations about 50% have an average invariance of 50% (Figure VII.2). This results from the nature of a load which can access several memory locations, in a loop for example.

Figure VII.3 shows the same information as Figure VII.2 for a history depth of three. Using a history depth of 1 we notice that over all programs profiled, about 8% of all memory locations accessed were completely invariant. A history depth of three increases this number to about 16%. Comparing the invariance results for memory locations with a history depth of three in Figure VII.3 with the invariance for load values with a history depth of three from Figure V.2 shows that merely 2% of all executed loads are 100% invariant, whereas 16% of all memory locations accessed are 100% invariant.

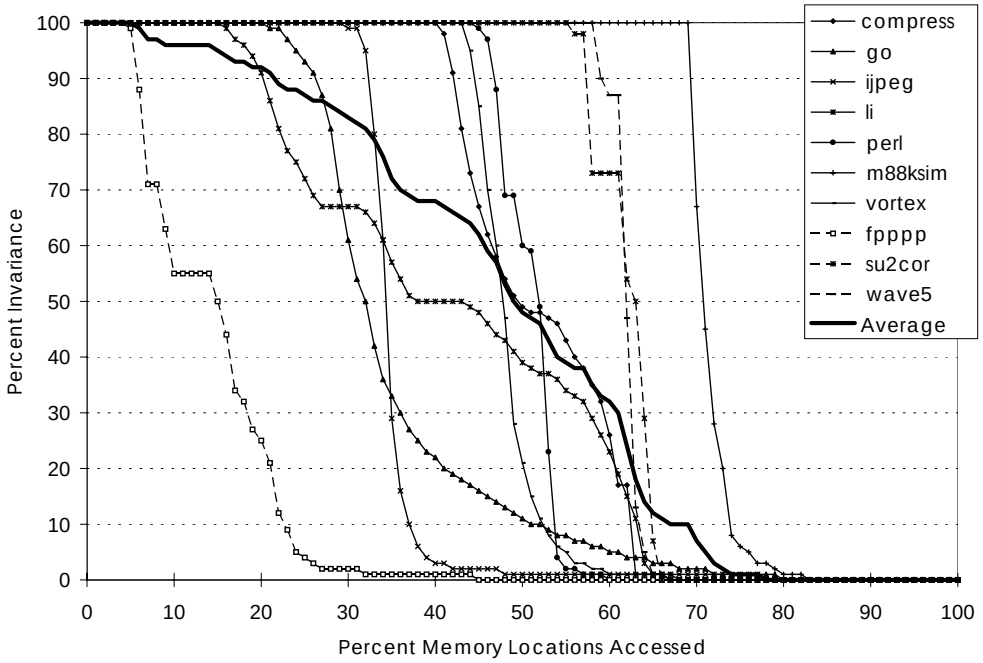

Figure VII.2: Invariance of Memory Locations (Top). The graph shows the percent invariance of the top value in the TNV table. The percent invariance is shown on the y-axis, and the percent of accessed memory locations on the x-axis.

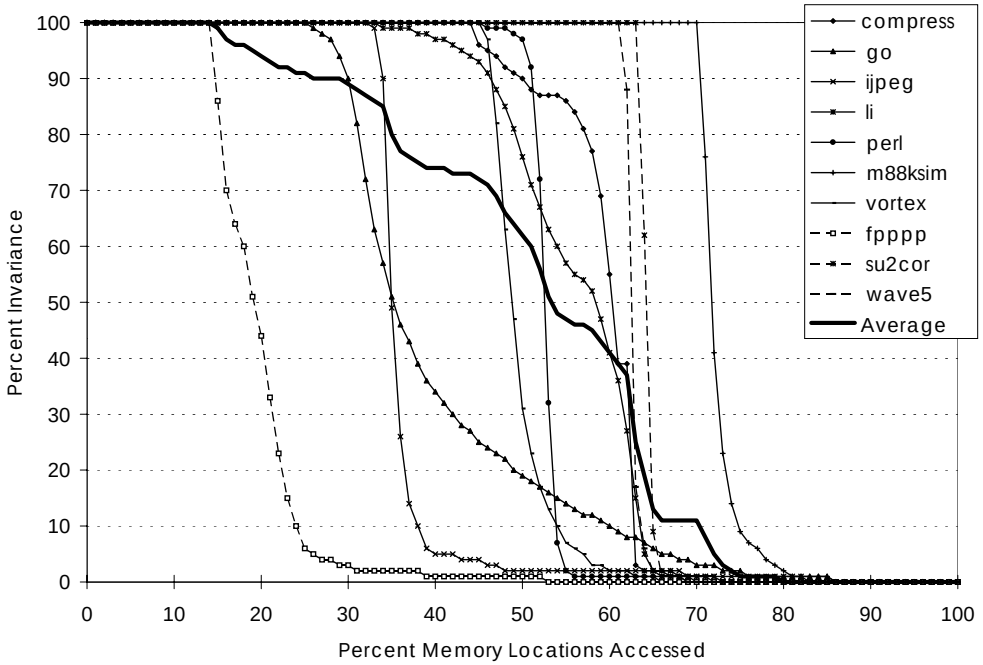

Figure VII.3: Invariance of Memory Locations (All). The graph on the left shows the percent invariance of all the values (Inv-3) in the TNV table. For a table size of six, there would be three values left in the TNV-table. The percent invariance is shown on the y-axis, and the percent of accessed memory locations on the x-axis.

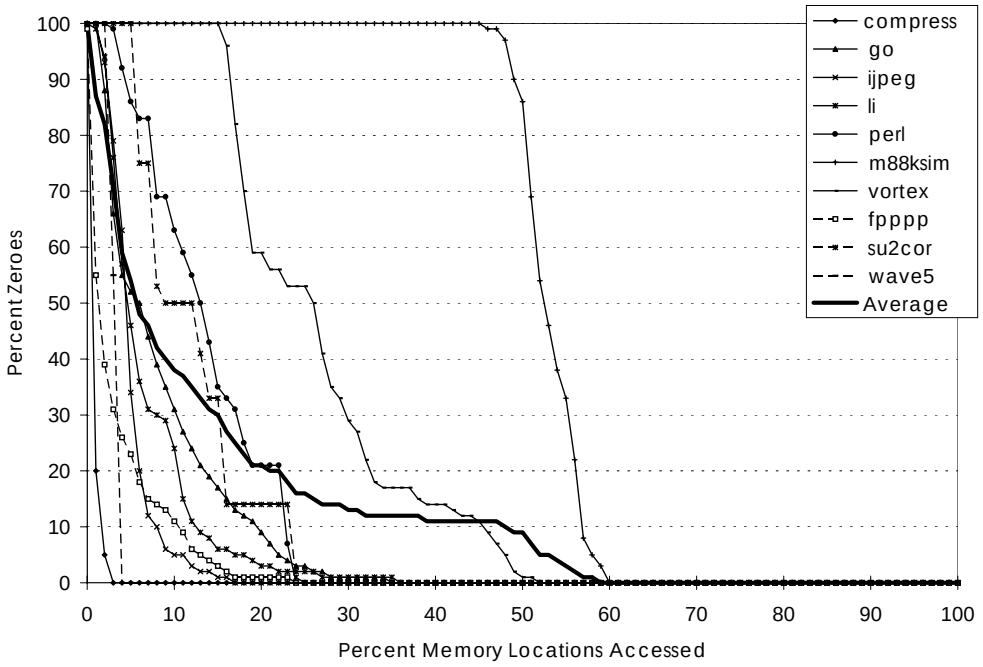

Figure VII.4: Zero Value for Data locations. The graph shows the percentage of time the Value of the data location had a zero value. The percent of zero values is shown on the y-axis, and the percent of accessed memory locations on the x-axis.

VII.D Memory Locations with Value Zero

This section shows results for memory locations with a zero value. Having 16% completely invariant memory locations would imply that a significant fraction would have a value of zero. Figure VII.4 shows this not to be the case. `Compress` which showed in Figure VII.2 that close to 50% of all memory locations accessed were invariant has essentially no memory locations with a zero value.

VII.E LVP Accuracy of Memory Locations

When comparing the last value prediction accuracy for memory locations to the LVP accuracy of loads, we see similar results to comparing their respective invariances. Figure VII.5 illustrates the LVP accuracy vs accessed memory locations. 50% of all memory locations accessed have an LVP of 80% or higher. For the loads in Figure: VII.5

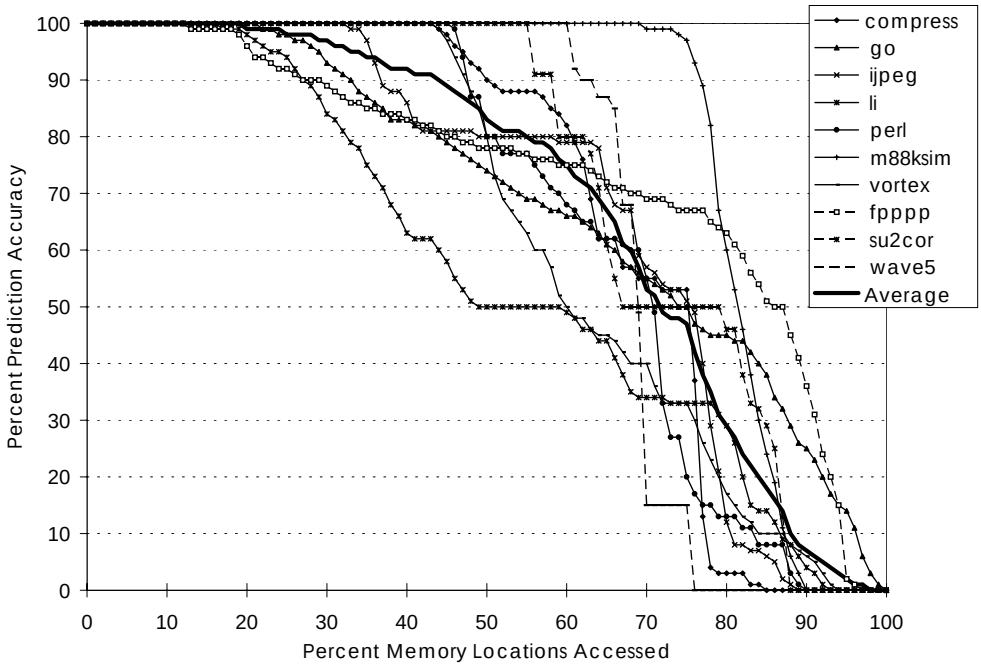

Figure VII.5: LVP of Memory Location. The graph shows the percentage of time the value of the Memory Location was the same as for the previous time the memory location was accessed. The percent LVP is shown on the y-axis, and the percent of accessed data locations on the x-axis.

the LVP accuracy for 50% of all loads executed is merely about 35%.

VII.F Memory Locations vs Load Invariance Difference

The difference in invariance between a memory location and all loads that access that memory location is shown in Figure VII.6. This difference is computed by subtracting the average invariance of all loads that access a memory location from the invariance of the memory location. The absolute value of this difference is weighed by the number of times the memory location was accessed.

The graph shows that on average for 50% of all accessed memory locations, the difference of invariance to all loads which accessed that memory location is 10%. 25% of all memory locations accessed have the same invariance as their respective loads. This

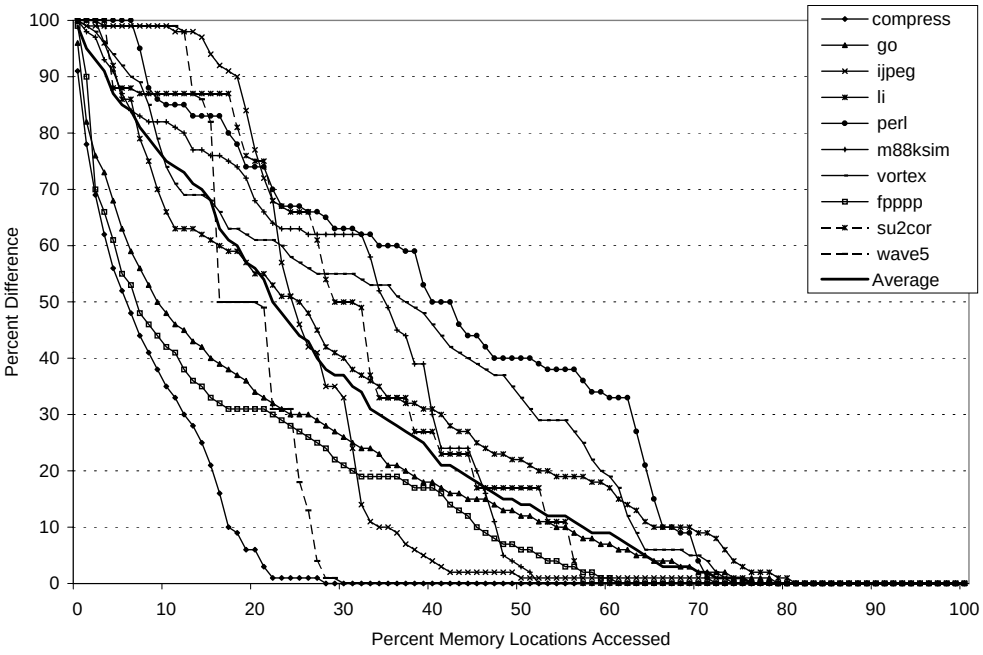

Figure VII.6: Memory Location Invariance vs Load invariance. This graph illustrates the difference in invariance between a memory location and all the loads that access that location. The y-axis is the absolute value of the difference of invariance and the x-axis is the percent of memory locations accessed.

result would indicate that it would suffice to profile just loads in a program to get a good feel for the invariance of their accessed memory.

One reason for the difference can result from code reuse, where a procedure may be called from many different locations (Figure VII.1), resulting in many different values for some of its instructions. Whereas the memory locations being accessed by the instructions, could possibly have been very invariant.

Table VII.1: Percent Memory Locations Accessed by Static Loads. This Table shows the percentage of memory locations by a given number of static loads. The column headers represent the number of static loads, and the values represent the percent memory locations accessed by that number of static loads.

	Number of Static Loads														
Programs	1	2	3	4	5	6	7	8	9	10	15	20	30	50	100
compress	22	9	21	19	5	8	4	4	3	0	0	4	0	0	0
go	3	3	2	3	2	3	2	3	2	3	7	6	11	5	12
ijpeg	15	9	6	16	11	3	1	2	3	0	2	21	9	0	0
li	10	7	5	7	11	6	3	4	4	4	14	12	13	3	0
perl	17	14	9	11	5	9	16	5	1	0	8	1	0	2	0
m88ksim	10	22	6	14	8	1	8	2	1	0	1	2	24	0	0
vortex	11	5	6	4	3	2	2	1	2	2	9	4	14	28	6
fpppp	17	10	8	7	4	6	5	5	5	4	19	7	5	0	0
su2cor	24	13	12	4	8	7	4	6	9	6	5	0	0	1	0
wave5	39	40	6	7	0	5	0	0	0	0	2	0	0	0	0
Average	17	13	8	9	6	5	4	3	3	2	7	6	8	4	2

Table VII.2: Percent Static Loads Accessing Different Memory Locations. This Table shows the percent of static loads that access 1 to 100,000,000 different addresses. Each column header represents a range of different memory locations, and the entries show the percentage of static loads.

	Number of Different Memory Locations								
Programs	1	10	100	1000	10000	100000	1000000	10000000	100000000
compress	43	24	14	3	0	2	13	0	0
go	4	16	19	19	23	16	2	0	0
ijpeg	16	28	17	13	10	9	6	1	0
li	10	19	13	12	28	9	7	2	0
perl	18	39	20	4	9	3	5	1	0
m88ksim	10	26	15	3	22	11	11	3	0
vortex	15	15	16	21	19	12	2	0	0
fpppp	7	17	4	1	7	61	2	0	0
su2cor	18	52	14	3	2	0	10	0	0
wave5	20	50	18	2	3	1	4	1	0
Average	16	29	15	8	12	12	6	1	0

Chapter VIII

Convergent Value Profiling

The amount of time a user will wait for a profile to be generated will vary depending on the gains achievable from using value profiling. The level of detail required from a value profiler determines the impact on the time to profile. The problem with a straight forward profiler, as shown in Figure III.1, is it could run hundreds of times slower than the original application, especially if all of the instructions are profiled. One solution proposed in this thesis is to use an intelligent profiler that realizes the data (invariance and top N values) being profiled is converging to a steady state and then profiling is turned off on an instruction by instruction basis.

Figure VIII.1 shows the invariance of load values for `compress` throughout program execution for different time intervals. Each load has been partitioned into time intervals. The intervals in this example are 10% of the loads execution. The Figure shows that about 40% of all executed loads are very invariant, and 30% are random. The other loads, are the ones we are concerned with. The goal for using the intelligent sampler is to quickly classify those loads that are not obviously invariant or random.

In examining the value invariance of instructions, we noticed that most instructions converge in the first few percent of their execution to a steady state. Once this steady state is reached, there is no point to further profiling the instruction. By keeping track of the percent change in invariance one can classify instructions as either "converged" or "changing". The convergent profiler stops profiling the instructions that are classified as converged based on a convergence criteria. This convergence criteria is tested after a given time period (convergence-interval) of profiling the instruction.

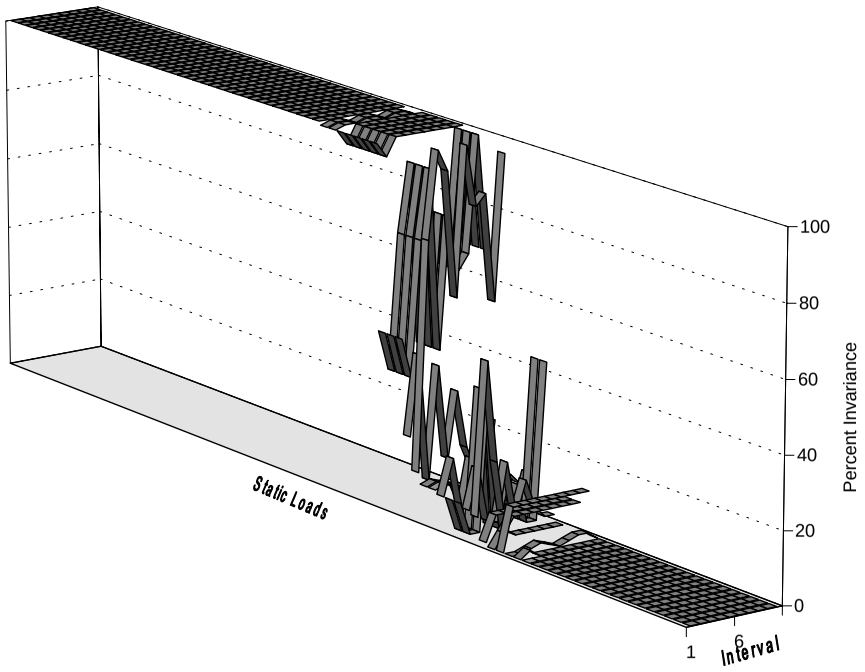

Figure VIII.1: Interval Invariance for compress. This graph shows how the invariance of the load values change during program execution. The x-axis represents the variable, the y-axis the Invariance for a given time interval, and the z-axis shows the time interval. The first set of data points along the x-axis shows the average over all time intervals. The other lines each represent ten percent of program execution. The graph shows, that about thirty percent of all variables are completely variant, and forty percent are invariant. Ideally our profiler would classify these loads quickly into their respective category. The other instructions vary over time, and pose a greater challenge for the profiler to determine their invariance.

To model this behavior, the profiling code is conditioned on a boolean to test if profiling is turned off or on for an instruction. If profiling is turned on, normal profiling occurs, and after a given convergence interval the convergence criteria is tested. The profiling condition is then set to false if the profile has converged for the instruction. If profiling is turned off, periodically the execution counter is checked to see if a given *retry* time period has elapsed. When profiling is turned off the retry time period is set to a number $total\_executed * backoff$, where back-off can either be a constant or a random number. This is used to periodically turn profiling back on to see if the invariance is at all changing.

In this thesis we examine the performance of two heuristics for the convergence criteria for value profiling. The first heuristic concentrates on the instructions with an increasing invariance. We will refer to that heuristic as *Conv(Inc)*. For instructions whose invariance is changing we are more interested in instructions that are increasing their final invariance than those that are decreasing their final invariance for compiler optimization purposes. Therefore, we continue to profile the instructions whose final invariance is increasing, but choose to stop profiling those instructions whose invariance is decreasing. When the percent invariance for the convergence test is greater than the percent invariance in the previous interval, then the invariance is increasing so profiling continues. Otherwise, profiling is stopped. When calculating the invariance the total frequency of the top half of the TNV table is examined. For the results, we use a convergence-interval for testing the criteria of 2000 instruction executions.

The second heuristic examined for the convergence criteria, is to only continue profiling if the change in invariance for the current convergence interval is greater than an *inv-increase* bound or lower than an *inv-decrease* bound. This heuristic will be referred to as *Conv(Inc/Dec)*. If the percent invariance is changing above or below these bounds, profiling continues. Otherwise profiling stops because the invariance has converged to be within these bounds.

Once profiling is turned off, it will be turned on after the *backoff* period elapsed. We examine 3 different backoff methods. (1) The fast-converging backoff method, which is computed by multiplying the total number of instructions profiled by twice the number of times converged. This results in a backoff faster than exponential. (2) The second backoff is purely exponential, and (3)random backoff, whereas the upper bound for the random range is one order of magnitude higher than the exponential backoff.

VIII.A Convergent Profile Metrics

To illustrate the performance of the convergence profiler several additional metrics are provided.

1. *Percent Profiled (Prof).*

   These results show the percentage of the total number of instructions profiled.

2. *Percent Converged (Cnv).*

   The percentage of time the profiler classified the instructions as converged.

3. *Percent Increasing (Inc).*

   The percentage of time the profiler classified the instructions as still increasing.

4. *Percent Decreasing (Dec).*

   The percentage of time the profiler classified the instructions as still decreasing.

5. *Percent Backoff (Bo).*

   The percentage of time spent profiling after turning profiling back on.

VIII.B Performance of the Convergent Profiler

This section will present all the results gathered for convergent profiling. The convergent profiles are compared to load profiles with the same table size.

VIII.B.1 Heuristic Conv(Inc)

Table VIII.1 shows the performance of the convergent profiler, which stops profiling the first instance the change in invariance decreases. The second column, percent of instructions profiled, shows the percentage of time profiling was turned on for the program's execution. The third column (Cnv) shows the percent of time profiling converged when the convergence criteria was tested, and the next column (Inc) is the percent of time the convergence test decided that the invariance was increasing. The fifth column (Back-off) shows the percent of time spent profiling after turning profiling back on using the retry time period. The rest of the metrics are described in §III.C and they compare the results of profiling the loads for the program's complete execution to the convergent profile results. The results show that on average convergent profiling spent 1.9% of its time profiling and profiling was turned off for the other 98% of the time. In most of the programs the time to converge was 1% or less. `gcc` was the only outlier, taking 24% of its execution to converge. The reason is `gcc` executes almost 30,000 static load instructions for our inputs and many of these loads do not execute for long. Therefore, most of these loads were fully profiled since their execution time fit within the time interval

Table VIII.1: Convergent profiler, where profiling continues if invariance is increasing, otherwise it is turned off. Prof is percent of time the executable was profiled. Cnv and Inc are the percent of time the convergent criteria decided that the invariance had converged or was still increasing. Bo=Backoff is the percent of time spent profiling after turning profiling back on. I(t)=Inv-Top, I(a)=Inv-All, D(t)=Diff-Top, D(a)=Diff-All, PAT=Percent loads above 30% invariance threshold, Sm=Same, F(t)=Find-Top, F(a)=Find-All.

	Convergent Profile						Comparing Full Load Profile to Convergent					
	%	Convergence		%	Invariance		Invariance		Top Values			
Program	Prof	Cnv	Inc	Bo	I(t)	I(a)	D(t)	D(a)	PAT	Sm	F(t)	F(a)
compress	2.4	71	29	6	48	51	6	1	42	96	100	98
gcc	23.9	42	58	9	46	67	3	1	41	94	99	98
go	1.0	49	51	22	40	57	6	2	31	91	100	100
ijpeg	0.2	51	49	24	21	31	3	1	17	100	100	100
li	0.7	12	88	71	44	65	8	4	36	94	99	98
perl	0.8	9	91	70	69	91	3	1	66	99	100	100
m88ksim	0.5	8	92	76	77	88	3	1	75	97	100	100
vortex	0.6	31	69	41	61	79	8	2	58	87	99	98
applu	0.2	72	28	7	34	35	1	0	31	100	100	100
apsi	0.5	44	56	22	38	47	20	4	13	100	100	100
hydro2d	0.2	36	64	31	67	83	7	2	58	98	100	97
mgrid	0.0	59	41	8	38	39	35	7	2	100	100	100
su2cor	0.3	34	66	1	36	41	19	4	16	100	100	100
swim	0.0	68	32	8	3	8	2	1	0	98	100	100
tomcatv	0.1	61	39	5	5	6	3	1	2	96	100	100
turb3d	0.1	58	42	23	69	80	32	7	34	96	100	100
wave5	0.2	71	29	4	26	32	15	4	10	95	100	100
Average	1.9	46	54	25	43	53	10	3	31	97	100	99

of sampling for convergence (2000 invocations). These results show that the convergent profiler's invariance differed by 10% from the full profile, and we were able to find the top value of the full length profile in the top values in the convergent profile 100% of the time.

VIII.B.2 Heuristic Conv(Inv/Dec)

Table VIII.2 shows the performance of the convergent profiler, when using the upper and lower change in invariance bounds for determining convergence. A new column (Dec) shows the percent of time the test for convergence decided to continue profiling because the invariance was decreasing. For these results we use an inv-increase threshold of 10% and an inv-decrease threshold of 10%. If the invariance is not increasing or decreasing by more than 10%, then profiling is turned off. The results show that this heuristic spends on average 2.3% time profiling (0.4% percent more than first heuristic),

Table VIII.2: Convergent profiler, where profiling continues as long as the change in invariance is either above the inv-increase or below the inv-decrease bound. The new column Dec shows the percent of time the invariance was decreasing when testing for convergence. Bo=Backoff is the percent of time spent profiling after turning profiling back on. I(t)=Inv-Top, I(a)=Inv-All, D(t)=Diff-Top, D(a)=Diff-All, PAT=Percent loads above 30% invariance threshold, Sm=Same, F(t)=Find-Top, F(a)-Find-All.

	Convergent Profile							Comparing Full Load Profile to Convergent					
	%	Convergence			%	Invariance		Invariance		Top Values			
Program	Prof	Cnv	Inc	Dec	Bo	I(t)	I(a)	D(t)	D(a)	PAT	Sm	F(t)	F(a)
compress	4.2	39	11	49	27	45	47	2	0	42	100	100	98
gcc	21.8	52	39	9	6	46	66	3	1	41	94	99	98
go	1.3	37	19	45	39	38	52	4	1	31	91	100	100
ijpeg	0.2	61	28	11	18	20	28	2	1	17	100	100	100
li	0.3	27	39	33	61	40	52	5	2	36	96	99	98
perl	0.1	70	26	4	7	67	89	3	0	66	99	100	100
m88ksim	0.1	50	21	29	36	76	85	2	0	75	97	100	100
vortex	0.4	46	32	23	35	58	73	7	1	58	87	100	99
applu	0.1	81	10	10	9	34	34	0	0	31	100	100	100
apsi	0.5	40	18	42	42	21	29	4	1	13	100	100	100
hydro2d	8.2	1	59	40	99	64	76	5	1	58	95	99	96
mgrid	0.1	32	17	52	52	8	8	5	1	2	100	100	100
su2cor	0.3	36	8	56	56	20	23	3	1	16	100	100	100
swim	0.0	63	13	24	27	2	6	1	0	0	98	100	100
tomcatv	0.1	71	16	13	11	3	5	1	0	2	96	100	100
turb3d	0.2	14	39	47	82	41	50	9	2	34	96	100	100
wave5	0.3	44	8	48	48	13	16	2	1	10	95	100	100
Average	2.3	45	24	31	39	35	44	3	1	31	97	100	99

but has a lower difference in invariance 3% (7% less than first heuristic). In terms of values this heuristic performs only slightly better than the previous heuristic. Therefore, the advantage of using this second heuristic is to obtain a more accurate invariance. Table VIII.2 shows a lot of the time is spent on profiling the decrease in invariance. The reason is that a variant instruction can start out looking invariant with just a couple of values at first. It then can take a while for the overall invariance of the instruction to reach its final variant behavior. The Backoff percentages indicate the time spent profiling after profiling has been turned back on. For the second convergence method 39% of the profiling time, is spent after profiling is turned back on. For the first method that same value is 25%.

One problem is that after an instruction is profiled for a long time, it takes a while for its overall invariance to change. If the invariance for an instruction converges after profiling for a while and then it changes into a new steady state, it will take a lot of profiling to bring the overall invariance around to the new steady state. One possible

Table VIII.3: Conv(Inc/Dec) - Different Convergent Criteria. This Table shows results using Conv(Inc/Dec) with different convergent criterias. The change in invariance must lie in between the two bound values for profiling to stop. This example uses the fast-converging backoff method. The numbers in the column headers show the thresholds. The results shown for each bound are Prof=time-to-classify (time spent profiling), D(t)=Diff-Top, and F(a)=Find-All.

	Convergence Bounds											
	1%/1%			10%/10%			20%/20%			40%/40%		
Program	Prof	D(t)	F(a)	Prof	D(t)	F(a)	Prof	D(t)	F(a)	Prof	D(t)	F(a)
compress	12.3	0.8	100.0	4.2	1.7	97.7	2.7	2.6	97.7	2.1	4.2	97.7
gcc	39.1	1.8	99.0	21.8	2.7	98.0	20.0	2.9	96.3	19.1	3.3	96.9
go	7.1	3.0	99.8	1.3	4.2	99.8	0.8	4.8	99.8	0.7	5.3	99.8
ijpeg	0.7	1.9	99.7	0.2	2.3	99.7	0.1	2.8	99.7	0.1	2.9	99.7
li	6.2	4.4	99.0	0.3	5.1	98.0	0.2	6.3	98.3	0.1	6.6	98.3
perl	0.7	2.3	100.0	0.1	3.0	99.7	0.1	3.5	99.7	0.1	3.6	99.7
m88ksim	0.7	1.9	99.6	0.1	2.3	99.6	0.1	2.3	99.6	0.1	3.7	95.6
vortex	3.8	5.1	99.1	0.4	6.7	98.6	0.3	7.0	98.7	0.2	7.8	98.6
applu	0.6	0.0	100.0	0.1	0.2	100.0	0.1	0.5	100.0	0.1	0.5	100.0
apsi	7.6	1.1	99.9	0.5	4.2	99.9	0.4	6.7	99.9	0.3	10.0	99.9
hydro2d	29.6	2.0	99.2	8.2	4.7	95.7	4.5	6.2	92.4	0.1	6.8	92.3
mgrid	0.3	1.4	99.8	0.1	5.3	99.8	0.0	9.7	99.8	0.0	16.4	99.8
su2cor	4.8	0.3	100.0	0.3	3.0	99.9	0.2	5.7	99.9	0.2	10.1	99.7
swim	0.1	0.5	100.0	0.0	0.9	100.0	0.0	1.7	100.0	0.0	1.5	100.0
tomcatv	0.2	1.4	100.0	0.1	1.2	100.0	0.1	2.1	99.6	0.1	2.4	99.6
turb3d	5.4	9.8	100.0	0.2	9.0	99.9	0.1	8.0	99.9	0.1	14.9	99.9
wave5	2.8	0.6	99.9	0.3	2.1	99.9	0.3	3.9	99.9	0.2	6.7	99.9
C-Prgs	8.8	2.6	99.5	3.6	3.5	98.9	3.0	4.0	98.7	2.8	4.7	98.3
F-Prgs	5.7	1.9	99.9	1.1	3.4	99.5	0.6	4.9	99.1	0.1	7.7	99.0
Average	7.2	2.2	99.7	2.3	3.4	99.2	1.8	4.5	98.9	1.4	6.3	98.7

solution is to monitor if this is happening, and if so dump the current profile information and start a new TNV table for the instruction. This would then converge faster to the new steady state. Examining this, sampling techniques, and other approaches to convergent profiling is part of future research.

Different Bounds

This section will show results for using different bounds with the Conv(Inc/Dec) convergence criteria. Increasing the bounds, allows instructions to converge faster. By increasing the bounds, and therefore decreasing time spent profiling, we can expect to see an increase in invariance difference and a decrease in the match of values.

Table VIII.3 shows the results for the Conv(Inc/Dec) heuristic using different bounds. The smaller the bounds, the smaller the change in invariance must be for the profiler to regard an instruction as being converged. The smaller the bounds are,

the longer the program will be profiled, and the more accurate the results will be. The results in the table show the time spent profiling, Diff-All, and Find-Top. When profiling the programs until their change in invariance falls between 1%, 7.2% of all executed loads are profiled. Profiling until +/-10% reduces this to 2.3%, so less than a third of the instructions as for +/-1% are profiled. The difference between the Find-All values however, is merely 0.5%. If only considering the FORTRAN programs the results are even more impressive, using bounds of +/-10% versus +/-1% reduces profiling time by more than a factor of 5.

As already explained for the case of `gcc`, the longer profiling times for the C programs result from the load instructions in the C programs being executed less often than in the FORTRAN programs. `Gcc` being least favorable for this convergence mechanism due to the small number of times each static load instruction was executed. `Gcc` has about 70,000 static loads and 1,000,000,000 dynamic loads. This gives an average of about 14,000 times, that each load was executed. The same computation for `swim` leads to 1,500,000 times each load was executed. See Table VIII.4 for an overview of how the loads are distributed for all the programs. `Gcc` shows that almost 60% of its loads are executed less than 100,000 times, whereas swim shows that 100% of all its loads are executed between 10^8 and 10^9 times.

Table VIII.4 shows what percentage of dynamic loads are loads which were executed for a certain number of times. The column of the table represents a range of execution frequencies. The column labeled 10^7 contains all load instructions which were executed anywhere from 10^6 up to 10^7 times. 55% of all loads executed in `applu` stem from loads which are executed between 10^6 and 10^7 times. For `su2cor` only 3% and `swim` 0% of loads executed have that execution frequency. Our convergent profiler will have a lower time-to-classify for programs which have loads with higher execution frequencies.

Different Backoff

There is one more parameter that we can manipulate to examine the effectiveness of convergent profiling, which is the back-off. The back-off accounts for the number of instructions to stop profiling once an instruction converged. After the back-off time has expired, profiling will get turned on again to verify that the invariance is still within

Table VIII.4: Dynamic Load Distribution. This Table shows the percentage of load instructions are accounted for by loads that are executed a certain number of times. Each column shows the percentage of dynamic load instruction, which are executed for the number of times defined by the column. Column 3, which is labeled 10^2, represents the fraction of static load instructions which were executed between 11 and 100 times. For column 4 the range would be 101 to 1000, etc.

Programs	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^10
compress	0	0	0	0	0	1	92	7	0	0	0
gcc	0	0	0	1	11	46	38	4	0	0	0
go	0	0	0	0	0	1	10	46	32	11	0
ijpeg	0	0	0	0	0	0	1	11	73	15	0
li	0	0	0	0	0	0	2	12	70	16	0
perl	0	0	0	0	0	0	0	24	72	3	0
m88ksim	0	0	0	0	0	0	1	6	53	32	8
vortex	0	0	0	0	0	0	5	13	26	55	0
applu	0	0	0	0	0	0	1	55	34	10	0
apsi	0	0	0	0	0	0	4	18	78	0	0
hydro2d	0	0	0	0	0	0	1	11	69	19	0
mgrid	0	0	0	0	0	0	0	5	4	91	0
su2cor	0	0	0	0	0	0	2	21	73	4	0
swim	0	0	0	0	0	0	0	0	0	100	0
tomcatv	0	0	0	0	0	0	2	1	52	45	0
turb3d	0	0	0	0	0	0	0	3	16	80	0
wave5	0	0	0	0	0	0	3	31	65	0	0
Average	0	0	0	0	1	3	10	16	42	28	0

the defined bounds. Here we studied three methodologies. The fast-converging backoff method, the exponential backoff, and a random backoff. Table VIII.5 illustrates the results for the different backoff methods. The results follow the pattern we've seen for the previous results, which indicate that the accuracy of the results increase with profiling time, if the convergence method used is the same.

VIII.B.3 Comparing Convergent vs Random Sampling

In this section we will analyze how well the convergent algorithm compares to a random sampling method. Unlike random sampling which randomly samples instructions every so often, our random sampler continuously samples instructions for a given time-interval, and then backs off for a random time interval. Using convergence we hope to achieve greater accuracy than by using a random sampler.

We implemented a random sampling profiling algorithm, that samples convergence interval size instructions, and then backs off for a random time interval, and then samples again. The number of instructions to back off, is a random number where the

Table VIII.5: Results for different Backoffs. The backoff specifies the number of samples which are skipped after a value has converged. The first columns shows the fast-converging algorithm which is slightly faster than exponential. The second column is the exponential backoff, and the third column a random-exponential backoff. Random-exponential will choose a random value within an exponential bound. The results shown for each backoff type are Tm=time-to-classify (time spent profiling), D(t)=Diff-Top, and F(a)=Find-All.

	Backoff Type								
	Fast-Converging			Exponential			Random		
Program	Prof	D(t)	F(a)	Prof	D(t)	F(a)	Prof	D(t)	F(a)
compress	4.2	1.7	97.7	9.1	1.1	99.8	9.1	1.1	100.0
gcc	21.8	2.7	98.0	32.0	2.0	99.0	32.5	2.0	99.4
go	1.3	4.2	99.8	2.7	3.2	99.8	2.7	3.2	99.8
ijpeg	0.2	2.3	99.7	0.6	1.4	99.9	0.7	1.5	99.9
li	0.3	5.1	98.0	1.2	4.2	99.3	1.2	3.7	99.4
perl	0.1	3.0	99.7	0.2	2.2	99.7	0.2	2.4	99.5
m88ksim	0.1	2.3	99.6	0.2	1.9	100.0	0.2	1.7	100.0
vortex	0.4	6.7	98.6	0.9	6.2	99.1	0.9	5.9	99.1
applu	0.1	0.2	100.0	0.3	0.1	100.0	0.3	0.1	100.0
apsi	0.5	4.2	99.9	1.2	3.4	99.9	1.2	3.2	100.0
hydro2d	8.2	4.7	95.7	12.2	3.0	100.0	8.6	3.2	99.9
mgrid	0.1	5.3	99.8	0.1	5.4	99.9	0.1	5.0	99.7
su2cor	0.3	3.0	99.9	1.0	2.8	100.0	0.9	2.8	99.9
swim	0.0	0.9	100.0	0.1	0.8	100.0	0.1	0.5	100.0
tomcatv	0.1	1.2	100.0	0.2	0.7	100.0	0.2	0.7	100.0
turb3d	0.2	9.0	99.9	0.5	5.8	100.0	0.4	6.7	100.0
wave5	0.3	2.1	99.9	0.7	1.8	100.0	0.7	1.8	100.0
C-Prgs	3.6	3.5	98.9	5.9	2.8	99.6	5.9	2.7	99.6
F-Prgs	1.1	3.4	99.5	1.8	2.6	100.0	1.4	2.7	99.9
Average	2.3	3.4	99.2	3.7	2.7	99.8	3.5	2.7	99.8

upper bound to the random function increases exponentially. This is repeated until the program quits. For convergence of invariance to be a good indicator for when profiling should stop, the results for random sampling should be worse than when using the convergence criteria. Table VIII.6 indicates that when using the Conv(Inc/Dec) heuristic, better results are achieved for the difference in invariance. However, the random algorithm shows a slightly higher match in values (Find-All).

Average Diff-Top for the C programs are very similar for all heuristics used. The FORTRAN programs however, have a high Diff-Top for the Conv(Inc) heuristic and the random sampler. The Conv(Inc/Dec) heuristic performs better for all programs except `ijpeg`. For `mgrid`, `su2cor`, `turb3d`, and `wave5`, the difference in invariance is a factor of four lower, for the Conv(Inc/Dec) heuristic when compared to the random sampler. These results show the need for some form of intelligent sampler for these

Table VIII.6: Results for different converging algorithms using the fast-converging back-off. The Random algorithm does not wait for convergence. There we just sample for clear interval size samples and then back off for the same number of samples as we backoff for the Conv(Inc/Dec) heuristic. The results shown for each algorithm are Tm=time-to-classify (time spent profiling), D(t)=Diff-Top, and F(a)=Find-All.

	Convergence Algorithm											
	Conv(Inc)				Conv(Inc=10%/Dec=10%)				None (Random)			
Program	Prof	D(t)	D(a)	F(a)	Prof	D(t)	D(a)	F(a)	Prof	D(t)	D(a)	F(a)
compress	2.4	5.5	1.1	97.7	4.2	1.7	0.3	97.7	4.1	3.3	0.7	99.7
gcc	23.9	2.7	0.7	97.9	21.8	2.7	0.6	98.0	23.0	3.0	0.5	97.0
go	1.0	6.2	1.9	99.8	1.3	4.2	1.0	99.8	1.1	4.4	1.3	99.8
ijpeg	0.2	3.2	1.1	99.7	0.2	2.3	0.7	99.7	0.3	2.6	0.8	99.9
li	0.7	8.3	3.5	97.7	0.3	5.1	1.6	98.0	0.2	5.6	1.8	98.1
perl	0.8	3.5	0.7	99.7	0.1	3.0	0.4	99.7	0.2	2.3	0.5	100.0
m88ksim	0.5	3.0	0.7	99.6	0.1	2.3	0.3	99.6	0.1	2.6	0.6	100.0
vortex	0.6	7.5	1.8	98.1	0.4	6.7	1.0	98.6	0.4	7.2	1.1	99.0
applu	0.2	0.8	0.2	100.0	0.1	0.2	0.1	100.0	0.3	0.3	0.1	100.0
apsi	0.5	19.9	4.4	99.9	0.5	4.2	1.0	99.9	0.5	8.4	1.9	100.0
hydro2d	0.2	7.2	1.6	97.0	8.2	4.7	1.4	95.7	0.2	6.3	1.2	99.8
mgrid	0.0	35.2	7.3	99.8	0.1	5.3	1.3	99.8	0.0	25.0	5.1	99.8
su2cor	0.3	18.8	4.2	100.0	0.3	3.0	0.7	99.9	0.3	12.9	2.8	99.7
swim	0.0	2.2	0.8	100.0	0.0	0.9	0.4	100.0	0.0	1.3	0.6	100.0
tomcatv	0.1	2.6	0.8	100.0	0.1	1.2	0.4	100.0	0.1	1.8	0.5	100.0
turb3d	0.1	31.5	6.6	99.8	0.2	9.0	1.9	99.9	0.1	24.3	5.0	100.0
wave5	0.2	15.1	3.8	99.9	0.3	2.1	0.7	99.9	0.4	10.5	2.5	100.0
C-Prgs	3.8	5.0	1.4	98.8	3.6	3.5	0.8	98.9	3.7	3.9	0.9	99.2
F-Prgs	0.2	14.8	3.3	99.6	1.1	3.4	0.9	99.5	0.2	10.1	2.2	99.9
Average	1.9	10.2	2.4	99.2	2.3	3.4	0.8	99.2	1.9	7.2	1.6	99.6

programs.

Chapter IX

Estimating Invariance

Out of all the instructions, loads are really the "unknown quantity" when dealing with a program's execution. If the value and invariance for all loads are known, then it is reasonable to believe that the invariance and values for many of the other instructions can be estimated through invariance and value propagation. This would significantly reduce the profiling time needed to generate a value profile for all instructions.

To investigate this, we used the load value profiles from the previous section, and propagated the load invariance through the program using data flow and control flow analysis deriving an invariance for the non-load instructions that write a register. We achieved reasonable results using a simple inter-procedural analysis algorithm. Our estimation algorithm first builds a procedure call graph, and each procedure contains a basic block control flow graph. To propagate the invariance, each basic block has an OUT RegMap associated with it, which contains the invariance of all the registers after processing the basic block. When a basic block is processed, the OUT RegMaps of all of its predecessors in the control flow graph are merged together and are used as the IN RegMap for that basic block. The RegMap is then updated processing each instruction in the basic block to derive the OUT RegMap for the basic block.

Besides using a load profile we also use a branch edge profile when propagating the invariance. When the invariance in the RegMap are merged, they are weighted by the OUT basic block execution count. For the results shown in this thesis, we break all of the loops in the procedure graph and process the procedures in a breadth-first order down the call graph, propagating the RegMaps inter-procedurally down the call edges.

Table IX.1: Invariance found for instructions computed by propagating the invariance from the load value profile. Instrs shows the percent of instructions which are non-load register writing instructions to which the results in this table apply. Prof and Est are the the percent invariance found for the real profile and the estimated profile. Diff-1 is the percent difference between the profile and estimation. The last 5 columns show the percent of executed instructions that have an average invariance above the threshold of 10, 30, 50, 70, and 90%, and the percentage of these that the estimation profile found and the percent that were over estimated.

	% of	% Inv-1			% Instructions Found Above Invariance Threshold				
Program	Instrs	Prof	Est	Diff-1	10%	30%	50%	70%	90%
compress	50	18	11	9	20 (41, 8)	9 (55, 8)	8 (53, 9)	5 (95, 0)	4 (100, 0)
gcc	47	46	38	13	33 (83, 3)	27 (77, 4)	22 (77, 4)	17 (75, 6)	13 (80, 6)
go	47	38	38	6	28 (89,12)	22 (91, 5)	18 (93, 5)	14 (96, 7)	12 (98, 8)
ijpeg	71	16	9	13	18 (33, 8)	13 (39,12)	11 (41,14)	8 (46,24)	6 (56,31)
li	32	36	34	8	19 (93, 5)	15 (99, 7)	14 (84, 7)	11 (81, 9)	9 (82,12)
perl	40	64	60	10	35 (98, 2)	30 (94, 2)	25 (83, 5)	17 (86,20)	16 (81, 5)
m88ksim	47	61	54	8	37 (82, 0)	32 (88, 0)	30 (91, 2)	27 (93, 2)	24 (94, 2)
vortex	40	55	47	14	28 (89, 7)	24 (86, 5)	23 (75, 5)	21 (73, 5)	20 (75, 6)
apsi	62	16	12	7	15 (56, 7)	11 (60,10)	8 (65,13)	7 (79,15)	5 (92,20)
fpppp	52	9	7	2	6 (73, 0)	4 (89, 0)	4 (96, 1)	3 (96, 1)	3 (96, 1)
hydro2d	59	56	39	19	41 (75, 1)	39 (66, 2)	32 (65, 1)	31 (64, 1)	18 (72, 6)
mgrid	59	5	3	2	3 (58, 5)	3 (56, 4)	3 (55, 4)	2 (94, 6)	2 (100, 7)
su2cor	64	15	13	2	12 (77, 0)	11 (84, 1)	10 (84, 1)	9 (88, 1)	7 (98, 1)
swim	65	4	3	1	2 (100, 0)	2 (100, 0)	2 (100, 0)	2 (100, 0)	2 (100, 0)
tomcatv	60	8	7	2	4 (91, 3)	4 (94, 4)	4 (97, 2)	4 (98, 2)	4 (98, 2)
turb3d	56	30	17	18	38 (54,11)	29 (26, 6)	8 (80, 7)	5 (76,13)	3 (96, 4)
wave5	57	15	13	4	9 (79,10)	8 (86,12)	8 (86,12)	7 (88, 3)	7 (90, 3)
Average	53	29	24	8	20 (75, 5)	17 (76, 5)	13 (78, 5)	11 (84, 7)	9 (89, 7)

After this forward pass, we propagate the invariance for return values backwards through the call graph via return edges. Intra-procedurally, the RegMaps are propagated across the basic blocks until the invariance of the OUT RegMaps for each basic block no longer change.

IX.A Propagating Invariance

To calculate the invariance for the instructions within a basic block we used a set of simple heuristics. The following description is of some of the more interesting heuristics we used. In this description we will call the definition register DEF and the two possible input registers for the instruction USE1 and USE2.

If the instruction has two input registers, the invariance of DEF is assigned the invariance of USE1 times the invariance of USE2. If one of the two input registers is

undefined, the invariance of DEF is left undefined in the RegMap. For instructions with only one input register (e.g., MOV), the invariance of DEF is assigned the invariance of USE1.

Loop induction variables can be used in many calculations inside of a loop. Therefore, it is important to correctly calculate their invariance. From the basic block profile we can determine the number of iterations a loop is executed. Using this along with the control flow graph we can determine the number of times the induction variable is executed after being initialized outside of the loop(s). For these induction registers a value equal to one divided by the number of times the instruction was executed between initializations is assigned to DEF for its invariance.

The calculation of load data addresses comprise a non-trivial amount of instruction execution for the SPEC95 programs, as can be seen in Table V.1 and Table V.2. Some programs have many of these LDA's from the stack pointer, so the invariance for these LDA instructions is basically the invariance of the stack pointer. Therefore, to calculate the invariance for the LDA's off of the stack pointer we also profile the invariance of the SP at each procedure call site.

For the conditional move instruction the value of the def register will either be the input register USE1 or it will be the original DEF register, where USE2 is the condition to be tested. We assign the DEF register an invariance of USE1 plus DEF divided by two times the invariance of USE2.

IX.B Estimation Results

Table IX.1 shows the invariance using our estimation algorithm for non-load instructions that write a register. The second column in the table shows the percent of executed instructions to which these results apply. The third column Prof shows the overall invariance (Inv-top) for these instructions using the profile used to form Table V.1 and Table V.2. The fourth column is the overall estimated invariance for these instructions, and the fifth column is the weighted difference in invariance Inv-Top between the real profile and the estimation on an instruction by instruction basis. The next 5 columns show the percent of executed instructions that have an average invariance

above the threshold of 10, 30, 50, 70, and 90%. Each column contains three numbers, the first number is the percent of instructions executed that had an Inv-Top invariance above the threshold. The second number is the percent of these invariant instructions that the estimation also classified above the invariant threshold. The last number in the column shows the percent of these instructions (normalized to the invariant instructions found above the threshold) the estimation thought were above the invariant threshold, but were not. Therefore, the last number in the column is the normalized percent of instructions that were over estimated. The results show that our estimated propagation has an 8% difference on average in invariance from the real profile. In terms of actually classifying variables above an invariant threshold our estimation finds 84% of the instructions that have an invariance of 70% or more, and the estimation over estimates the invariant instructions above this threshold by 7%.

Our estimated invariance is typically lower than the real profile. There are several reasons for this. The first is the default heuristic which multiplies the invariance of the two uses together to arrive at the invariance for the def. At times this estimation is correct, although a lot of time it provides a conservative estimation of the invariance for the written register. Another reason is that at times the two uses for an instruction were variant but their resulting computation was invariant. This was particularly true for logical instructions (e.g, AND, OR, Shift) and some arithmetic instructions.

Chapter X

Preliminary Results

In this section we examine the performance potential of code specialization. This was achieved by taking the top C and FORTRAN program with the highest invariances, `mmksim` and `hydro2d`, as shown in Figure V.1. Then we used the value profile information to guide specialization performed by hand. To evaluate the performance increase, we compiled the code with the modifications and compared the run times of the modified vs. unmodified programs.

The results in this chapter show, that value profiling can be an effective tool for providing feed back to profile-driven optimizers.

X.A Code Specialization

The value profile for `m88ksim` showed that most of its invariant instructions were executed in `killtime` and `alignd`. The routine `killtime` accounts for 23% of all executed instructions, and `alignd` for 11% of all executed instructions.

The value profile for `hydro2d` showed that most of its invariant instructions were executed in `filter` and `tistep`. The routine `filter` accounts for 41% of all executed instructions, and `tistep` for 4% of all executed instructions.

X.A.1 `M88ksim`

Figure X.1 shows the `killtime` routine in `m88ksim`. This routine executes 23% of all instructions. The value profile indicated, that the load, which was responsible

```
void killtime (unsigned int time_to_kill)
{
   register int i;
   Clock += time_to_kill;

   for (i = 0; i < 32; i++) {
      m88000.time_left[i] -= MIN(m88000.time_left[i], time_to_kill);
   }
   if (usecmmu) {
      Dcmmutime - = MIN(Dcmmutime, time_to_kill);
      mbus_latency - = MIN(mbus_latency, time_to_kill);
   }
   for (i=0; i<=4; i++) {
      funct_units[i].busy -= MIN(funct_units[i].busy, time_to_kill );
   }
}
```

Figure X.1: `m88ksim: killtime` - The function `killtime` accounts for 23% of all executed instructions in `m88ksim`. With the current data structures used, there is not much that could be optimized.

for loading `m88000.time_left[i]` and the load for `funct_units[i].busy`, has values of zero 99% of the time. Straight forward code specialization did not provide a performance benefit. Being aware of this at design time of the program could motivate the programmer to choose a different data structure or algorithm, to take advantage of the invariance of that variable. One way to achieve that would to have a boolean variable which would indicate if any entry in `m88000.time_left` has changed since the last time `killtime` was called.

Figure X.2 illustrates the code specialization performed on the routine `alignd`. That routine accounts for 11% of all executed instructions. `amantlo` and `amanthi` are parameters to the function. Our value profiler indicated 100% invariance for `amantlo` and `amanthi`. Performing simple code specialization, as shown in Figure X.3 (b), by not executing all the contents in the if-clause whenever `amantlo` and `amanthi` are zero, decreases the execution time of `m88ksim` by 13%.

X.A.2 `Hydro2d`

Figure X.3 shows the function `filter` from the SPEC95 FORTRAN program `hydro2d`. This code fragment accounted for 13% of all executed instructions. Our value

```
if (expdiff >= 0) {
 for (*s = 0 ; expdiff > 0 ; expdiff –) {
  *s |= *bmantlo & 1;
  *bmantlo >>= 1;
  *bmantlo |= *bmanthi << 31;
  *bmanthi >>= 1;
 }
 *resexp = aexp;
} else {
 expdiff = - expdiff;
 for (*s = 0 ; expdiff > 0 ; expdiff–) {
  *s |= *amantlo & 1;
  *amantlo >>= 1;
  *amantlo |= *amanthi << 31;
  *amanthi >>= 1;
 }
*resexp = bexp;
}
```

(a) Original Code

```
if (expdiff >= 0) {
 for (*s = 0 ; expdiff > 0 ; expdiff–) {
  *s |= *bmantlo & 1;
  *bmantlo >>= 1;
  *bmantlo |= *bmanthi << 31;
  *bmanthi >>= 1;
 }
 *resexp = aexp;
} else {
 if (*amantlo || *amanthi) {
  expdiff = - expdiff;
  for (*s = 0 ; expdiff > 0 ; expdiff –) {
   *s |= *amantlo & 1;
   *amantlo >>= 1;
   *amantlo |= *amanthi << 31;
   *amanthi >>= 1;
  }
 } else {
 *s = 0;
 }
 *resexp = bexp;
}
```

(b) Optimized Code

Figure X.2: `m88ksim: alignd` - The function `alignd` accounts for 11% of the executed instructions in `m88ksim`. The code in (a) was code specialized as shown in (b). This modification led to a speedup of 13%.

profiler found `QP` and `QM` to be zero 98% of the time. Performing simple code specialization resulted in 11% speedup.

Figure X.4 (a) shows the unoptimized version of the `tistep` routine. This routine executes 4% of all instructions in `hydro2d`. `DPZ(I,J)` and `DPR(I,J)` were flagged as invariant by our value profiler with an invariance of 99% and a value of zero. Code specialization resulted in an overall speedup of 2% as shown in Figure X.4 (b).

```
QP = VMAX(I,J) - UTDF(I,J)
QM = UTDF(I,J) - VMIN(I,J)

IF ( PP .EQ. 0.0D0 )
  RP(I,J)=0.0D0
ELSE
  RP(I,J)=DMIN1(DBLE(1.0D0),QP/PP)
END IF

IF ( PM .EQ. 0.0D0 )
  RN(I,J)=0.0D0
ELSE
  RM(I,J)=DMIN1(DBLE(1.0D0),QM/PM)
END IF
```

(a) Original Code

```
QP = VMAX(I,J) - UTDF(I,J)
QM = UTDF(I,J) - VMIN(I,J)

IF (QP .EQ. 0.0D0)
  RP(I,J) = 0.0D0
ELSE
  IF (PP .EQ. 0.0D0)
    RP(I,J)=0.0D0
  ELSE
    RP(I,J)=DMIN1(DBLE(1.0D0),QP/PP)
  END IF
END IF

IF (QM .EQ. 0.0D0)
  RM(I,J)=0.0D0
ELSE
  IF ( PM .EQ. 0.0D0 )
    RN(I,J)=0.0D0
  ELSE
    RM(I,J)=DMIN1(DBLE(1.0D0),QM/PM)
  END IF
END IF
```

(b) Optimized Code

Figure X.3: `hydro2d: filter` - the function `filter` accounts for 13% of the executed instructions in `hydro2d`. The code in (a) was code specialized as shown in (b). This modification led to a speedup of 12%.

```
K = 0
DO 400 J = 1,NQ
DO 400 I = 1,MQ
 VSD = SQRT ( GAM * PR(I,J) / RO(I,J) )
 VZ1 = VSD + ABS(VZ(I,J)) + DVZ(I,J)
 VZ2 = VZ1 ** 2
 VR1 = VSD + ABS(VR(I,J)) + DVR(I,J)
 VR2 = VR1 ** 2

 XPZ = DBLE(0.25D0) * DPZ(I,J) / VZ2
 XPR = DBLE(0.25D0) * DPR(I,J) / VR2

 IF ( XPZ .LT. 1.0D-3 ) DPZ(I,J) = 1.0D0
 IF ( XPR .LT. 1.0D-3 ) DPR(I,J) = 1.0D0

 TCZ = DBLE(0.5D0) * DZ(I) / DPZ(I,J) *
  ( SQRT( VZ2 + DBLE(4.0D0)*DPZ(I,J) ) - VZ1 )
 TCR = DBLE(0.5D0) * DR(J) / DPR(I,J) *
  ( SQRT( VR2 + DBLE(4.0D0)*DPR(I,J) ) - VR1 )

 IF ( XPZ .LT. 1.0D-3 )
  TCZ = DZ(I) / VZ1 * ( DBLE(1.0D0) - XPZ )
 IF ( XPR .LT. 1.0D-3 )
  TCR = DR(J) / VR1 * ( DBLE(1.0D0) - XPR )
 K = K + 1
 TST(K) = DMIN1( TCZ , TCR )
400 CONTINUE
```

(a) Original Code

```
K = 0
DO 400 J = 1,NQ
DO 400 I = 1,MQ
 VSD = SQRT ( GAM * PR(I,J) / RO(I,J) )
 VZ1 = VSD + ABS(VZ(I,J)) + DVZ(I,J)
 VZ2 = VZ1 ** 2
 VR1 = VSD + ABS(VR(I,J)) + DVR(I,J)
 VR2 = VR1 ** 2
 IF ((DPZ(I,J) .EQ. 0.0D0) .AND.
 (DPR(I,J) .EQ. 0.0D0)) THEN
  TCZ = DZ(I) / VZ1
  TCR = DR(J) / VR1
 ELSE
  XPZ = DBLE(0.25D0) * DPZ(I,J) / VZ2
  XPR = DBLE(0.25D0) * DPR(I,J) / VR2
  IF ( XPZ .LT. 1.0D-3 ) DPZ(I,J) = 1.0D0
  IF ( XPR .LT. 1.0D-3 ) DPR(I,J) = 1.0D0

  TCZ = DBLE(0.5D0) * DZ(I) / DPZ(I,J) *
   ( SQRT( VZ2 + DBLE(4.0D0)*DPZ(I,J) ) - VZ1 )
  TCR = DBLE(0.5D0) * DR(J) / DPR(I,J) *
   ( SQRT( VR2 + DBLE(4.0D0)*DPR(I,J) ) - VR1 )

  IF ( XPZ .LT. 1.0D-3 )
   TCZ = DZ(I) / VZ1 * ( DBLE(1.0D0) - XPZ )
  IF ( XPR .LT. 1.0D-3 )
   TCR = DR(J) / VR1 * ( DBLE(1.0D0) - XPR )
 END IF
 K = K + 1
 TST(K) = DMIN1( TCZ , TCR )
400 CONTINUE
```

(b) Optimized Code

Figure X.4: `hydro2d: tistep` - the function `tistep` accounts for 4% of the executed instructions in `hydro 2d`. The code in (a) was code specialized as shown in (b). This modification led to a speedup of 2%.

Chapter XI

Conclusions

In this thesis we explored the invariant behavior of values for loads, parameters, all register defining instructions and memory locations, as well as their value predictability. The invariant behavior was identified by a value profiler, which could then be used to automatically guide compiler optimizations and dynamic code generation.

We showed that value profiling is an effective means for finding invariant and semi-invariant instructions. Our results show that the invariance and LVP, found for instructions when using value profiling, is very predictable even between different input sets. As an alternative to profiling instructions, we implemented a technique for profiling memory locations. The average invariance for memory locations and instructions are similar. However, memory locations have a higher percentage of very invariant and random (not invariant) values.

In addition, we examined two techniques for reducing the profiling time to generate a value profile. The first technique used the load value profile to estimate the invariance for all non-load instructions with an 8% invariance difference from a real profile. The second approach we proposed for reducing profiling time, is the idea of creating a convergent profiler that identifies when profiling information reaches a steady state and has converged. The convergent profiler we used for loads, profiled for only 2% of the program's execution on average, and recorded an invariance within 3% of the full length profiler and found the top values 100% of the time. The idea of convergent profiling proposed in this thesis can potentially be used for decreasing the profiling time needed for other types of detailed profilers.

Our results show that using a table with four entries in the steady part after profiling, gives best results. This is best achieved using a table size of six entries with a clear size of two. Clear interval sizes from 100,000 to 500,000 samples proved reasonable performance.

Value profiling information can be used by value prediction to choose the predictable instructions to be entered into the Value History Table. This approach increases the utilization of the prediction table and reduces the number of mispredictions, which in turn increases the prediction accuracy. We also found that compared to LVP, the invariance has a slightly higher correlation between different profile runs. Using value profiling to guide LVP was shown to provide a 94% prediction accuracy for 33% of the executed loads on average.

Value profiling information has also been shown to be useful for code specialization. By identifying the invariant variables, and their associated values, and then applying code specialization, we have shown that code specialization can improve performance up to 13%.

Value profiling is an important part of future compiler research, especially in the areas of dynamic compilation and adaptive execution, where identifying invariant or semi-invariant instructions at compile time is essential. A complementary approach for trying to identify semi-invariant variables, is to use data-flow and staging analysis to try and prove that a variable's value will not change often or will hold only a few values over the life-time of the program. This type of analysis should be used in combination with value profiling to identify optimizable code regions.

Chapter XII

Future Directions

low level value information to high level code, such that programmers can Further research on value patterns for different instruction categories might lead to better heuristics for predicting future instruction values. One could implement a special predictor for each instruction type, since different instruction classes have shown to have considerable differences in invariance. A hybrid between several simple models is most likely to achieve the best results. A load instruction or an add instruction is likely to increment by a constant amount, hence for instructions like that we would use some sort of a stride predictor.

One could use an approach similar to Young and Smith [40] by using the path history when predicting values. This can be especially beneficial for procedures called from several locations in the program.

Many improvements can also be made for the intelligent sampler. The intelligence examined in this thesis was a convergence criteria based upon a change in invariance. Other approaches to be evaluated include using value information with the current convergence criteria.

When predicting instruction values at run-time it is important to minimize the number of mispredictions. One approach one could take is to identify the critical path of a program and only predict the instructions which cause the pipeline bottleneck. If the instruction causing the dependency can not be predicted then a successor of the instructions could be predicted to eliminate the bottleneck. The goal is to reduce the potential of mispredictions while maximizing the achievable speedup using value prediction.

If the predictability of a critical instruction leads to several values, it might prove beneficial to simultaneously predict multiple values, and execute them in parallel, and then discard the mispredicted ones. The benefit here is that the misprediction penalty could still be avoided. Since this leads to multiple path execution, the number of multiple values predicted would need to be heavily guided by a confidence mechanism.

Some architectures save the entire, or part of the register file in memory before procedure calls, especially for caller saves register allocation. This is necessary so the called procedure can use the registers. When returning to the procedure those registers need to be restored. Register windows address this issue, however by predicting register values one could achieve some of the benefit that register windows offer. Gabbay [17] also showed that register file prediction worked particularly well for ALU instructions on a limited number of programs.

The continuous profiling infrastructure (CPI) [1] could be extended to include value information. For continuous profiling to be a viable solution, it needs to be very effective. CPI addressed that by randomly sampling instructions. For doing accurate value profiling additional research is needed to determine if random sampling is sufficient for value profiling. Other options, such as some type of intelligent sampler might prove to be more suited. Another issue to address are the memory requirements during profiling. The memory requirements for CPI must be minimized to avoid cache misses while profiling. Keeping track of values will increase the memory requirements, however the memory required during profiling must be minimized to not impact the programs being sampled.

Bibliography

[1] J. M. Anderson, L. M. Berc, J. Dean, S. G. Ghemawat, M. R. Henzinger, S-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Continous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[2] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159. ACM, May 1996.

[3] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In *9th International Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[4] T. Ball and J. Larus. Efficient path profiling. In *29th International Symposium on Microarchitecture*, pages 46–57, December 1996.

[5] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *1994 ACM Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[6] B. Calder, D. Grunwald, and A. Srivastava. The predictability of branches in libraries. In *28th International Symposium on Microarchitecture*, pages 24–34, Ann Arbor, MI, November 1995. IEEE.

[7] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[8] B. Calder, S. John, and T. Austin. Cache-concious data placement. Technical Report, University of California, San Diego, 1998.

[9] P. P. Chang and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–376, 1992.

[10] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic compiler code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[11] W. C. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W. W. Hwu. Profile-assisted instruction scheduling. *International Journal for Parallel Programming*, 22(2):151–181, 1994.

[12] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 145–156. ACM, January 1996.

[13] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102. ACM, June 1995.

[14] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme:hardware support for instruction-level profiling on out-of-order processors. In *30th International Symposium on Microarchitecture*, pages 292–302, December 1997.

[15] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level efficient, and machine-independent dynamic code generation. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 131–144. ACM, January 1996.

[16] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, Boston, Mass., October 1992. ACM.

[17] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[18] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *30th International Symposium on Microarchitecture*, pages 270–280, December 1997.

[19] D.M. Gallagher, W.Y. Chen, S.A. Mahlke, J.C. Gyllenhaal, and W.W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Six International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[20] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *30th International Symposium on Microarchitecture*, pages 303–313, December 1997.

[21] D. Grove, J. Dean, C. Garret, and C. Chambers. Profile-guided receiver class prediction. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 108–123. ACM, October 1995.

[22] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line size. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 171–182. ACM, June 1997.

[23] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 326–336. ACM, June 1994.

[24] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-types object-oriented languages with polymorhic inline caches. In *ECOOP'91, Fourth European Conference on Object-Oriented Programming*, pages 21–38, July 1991.

[25] T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225. ACM, January 1996.

[26] P. Lee and M. Leone. Optimizing ml with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148. ACM, May 1996.

[27] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[28] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[29] M. Moudgill and J. H. Moreno. Run-time detection and recovery from incorrectly reordered memory operations. IBM Research Report, May 1997.

[30] K. Pettis and R.C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, June 1990.

[31] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Profile guided memory communication and speculative load execution. Technical Report, University of California, San Diego, 1998.

[32] Stephen E. Richardson. Exploiting trivial and redundant computation. In *Proceedings of the Eleventh Symposium on Computer Arithmetic*, 1993.

[33] Vatsa Santhanan and Daryl Odnert. Register allocation accross procedure and module boundaries. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39, June 1990.

[34] Y. Sazeides and James E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[35] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[36] Ron van der Wal. Source-code profilers for win32. *Dr. Dobb's Journal*, pages 78–88, March 1998.

[37] David W. Wall. Global register allocation at link-time. In *Proceedings of the SIGPLAN'86 Conference on Programming Language Design and Implementation*, pages 264–275, June 1986.

[38] D.W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 59–70, Toronto, Ontario, Canada, June 1991.

[39] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *30th International Symposium on Microarchitecture*, pages 281–290, December 1997.

[40] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Six International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, October 1994.