Copyright 1997 IEEE. Published in the Proceedings of Micro-30, December 1-3, 1997 in Research Triangle Park, North Carolina. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Value Profiling

Brad Calder Peter Feller

Department of Computer Science and Engineering
University of California, San Diego
{calder,pfeller}@cs.ucsd.edu

Alan Eustace

Digital Equipment Corporation
Western Research Lab
eustace@pa.dec.com

Abstract

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 paper we examine the invariance found from profiling instruction values, and show that many instructions have semi-invariant values even across different inputs. We also investigate the ability to estimate the invariance for all instructions in a program from only profiling load instructions. In addition, we propose a new type of profiling called Convergent Profiling. *Estimating the invariance from loads and convergent profiling are used to reduce the profiling time needed to generate an accurate value profile. The value profile can then be used to automatically guide code generation for dynamic compilation, adaptive execution, code specialization, partial evaluation and other compiler optimizations.*

1 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 paper examines using profile feedback information to identify which variables have 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. Value profiling is an approach that can identify these semi-invariant variables.

The goal of value profiling is different from value prediction. Value prediction is used to predict the next result value (write of a register) for an instruction. This has been shown to provide predictable results by using previously cached values to predict the next value of the variable using a hardware buffer [5, 9, 10]. This approach was shown to work well for a hardware value predictor, since values produced by an instruction have a high degree of temporal locality.

Our research into the semi-invariance of variables is different from these previous hardware predication studies. For compiler optimizations, we are more concerned with the invariance of a variable, the top N values of the variable, or a popular range of values for the variable over the life-time of the program, although the temporal relationship between values can provide useful information. The value profiling techniques presented in this paper keep track of the top N values for an instruction and the number of occurrences for each of those values. This information can then be used to automatically guide compilation and optimization.

In the next section, we examine motivation for this paper and related work. Section 3 describes a method for value profiling. Section 4 describes the methodology used to gather the results for this paper. Section 5 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 order to reduce the time to generate a value profile for optimization, §6 investigates the ability to estimate the invariance for all non-load instructions by value profiling only load instructions and propagating their invariance. Section 7 examines a new type of profiler called the *Convergent Profiler* and its use for value profiling. The goal of a convergent profiler is to reduce the amount of time it takes to gather detailed profile information. For value profiling, we found that the data being profiled, the invariance of instructions,

often reaches a steady state, and at that point profiling can be turned off or sampled less often. This reduces the profiling time, while still creating an accurate value profile. We conclude by summarizing the paper in §8.

2 Motivation and Related Work

This paper was originally motivated by a result we found when examining the input values for long latency instructions. A divide on a DEC Alpha 21064 processor can take 60 cycles to execute, and a divide on the Intel Pentium processor can take up to 46 cycles. Therefore, it would be beneficial to special case divide instructions with optimizable numerators or denominators. In profiling `hydro2d` from the SPEC92 benchmark suite, we found that 64% of the executed divide instructions had either a 0 for its numerator or a 1 for its denominator. In conditioning these divide instructions on the numerator or denominator, with either 0 or 1 based on profiling information, we were able to reduce the execution time of `hydro2d` by 15% running on a DEC Alpha 21064 processor. In applying the same optimization to a handful of video games (e.g., Fury3 and Pitfall) on the Intel Pentium processor, we were able to reduce the number of cycles executed by an estimated 5% [1] for each of these programs. These results show that value profiling can be very effective for reducing the execution time of long latency instructions.

The recent publications on *Value Prediction* in hardware provided further motivation for our research into value profiling [5, 9, 10]. The recent paper by Lipasti et al. [9] showed that on average 49% of the instructions wrote the same value as they did the last time, 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 equal the instruction's degree of invariance, which is needed for certain compiler optimizations.

2.1 Uses for Value Profiling

Value profiling can benefit several areas of research including dynamic compilation and adaptive execution, performing compiler optimizations to specialize a program for certain values, and providing hints for value prediction hardware.

[1] This estimation is a static calculation using a detailed pipeline architecture model of the Pentium processor. The estimation takes into consideration data dependent and resource conflict stalls.

2.1.1 Dynamic Compilation, Adaptive Execution and Code Specialization

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 [1, 3, 4, 7, 8]. 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 [8] and Knoblock and Ruf [7] 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. In addition, Autrey and Wolfe have started to investigate a form of staging analysis for automatic identification of semi-invariant variables [2]. Consel and Noel [3] 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. [1] 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 generating run-time 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.

The above approaches used for dynamic compilation, to determine optimizable code regions, can also be applied to static optimization. These regions can benefit from code specialization if a variable or instruction has the same value across multiple inputs. If this is the case, the code could be

duplicated creating a specialized version optimized to treat the variable as a constant. The execution of the specialized code would then be conditioned on that value. Value profiling can be used to determine if these potential variables or instructions have the same value across multiple inputs, in order to guide code specialization.

2.1.2 Hardware-based Optimizations

In predicting the most recent value(s) seen, an instruction's future value has been shown to have good predictability using tag-less hardware buffers [9, 10]. Our results show that value profiling can be used to classify the invariance of instructions, so a form of value profiling could potentially be used to improve hardware value prediction. Instructions that are shown to be variant can be kept out of the value prediction buffer, reducing the number of conflicts and aliasing effects, resulting in a more accurate prediction using smaller tables. Instructions shown to have a high invariance with the value profiler could even be given a "sticky" replacement policy.

The Memory Disambiguation Buffer [6] (MDB) is an architecture that allows a load and its dependent instructions to be hoisted out of a loop, by checking if store addresses inside the loop conflict with the load. If a store inside the loop is to the same address, the load and its dependent instructions are re-executed. A similar hardware mechanism can be used to take advantage of values, by checking not only the store address, but also its value. In this architecture, only when the value of the load hoisted out of the loop changes should the load and its dependent instructions be re-executed. Value profiling can be used to identify these semi-invariant load instructions.

3 Value Profiling

In this section we will discuss a straight forward approach to value profiling. This study concentrates on profiling at the instruction level; finding the invariance of the written register values for instructions. The value profiling information at this level can be directly mapped back to the corresponding variables by the compiler for optimization. There are two types of information needed for value profiling to be used for compiler optimizations: (1) how invariant is an instruction's value over the life-time of the program, and (2) what were the top N result values for an instruction.

Determining the invariance of an instruction's resulting value can be calculated in many different ways. The value prediction results presented by Lipasti et al. [9, 10] used a tag-less table to store a cache of the most recently used values to predict the next result value for an instruction. Keeping track of the number of correct predictions equates to the number of times an instruction's destination register

```
void InstructionProfile::collect_stats (Reg cur_value) {
    total_executed ++;
    if (cur_value == last_value) {
      mrv_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 1: A simple value profiler keeping track of the N most frequent occurring values, along with the most recent value (MRV-1) metric.

was assigned a value that was the most recent value or one of the most recent M values. We call this the *Most Recent Value - M* (MRV-M) metric, where M is the history depth of the most recent values kept.

The MRV metric provides an indication of the temporal reuse of values for an instruction, but it does not equate exactly to the invariance of an instruction. By *Invariance - M* (Inv-M) we mean the percent of time an instruction spends executing its most frequent M values. 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 MRV-1 (which stores only the most recent value) of 0%, but the instruction has an invariance Inv-1 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 MRV-1 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-1. The MRV differs from invariance because it does not have state associated with each value indicating the number of times the value has occurred. Therefore, the replacement policy it uses, least recently used, cannot tell what value is the most common. We found the MRV metric is at times a good prediction of invariance, but at other times it is not because of the examples described above.

3.1 A Value Profiler

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 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. Therefore, there is a TNV table for every register being profiled. 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.

3.2 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 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. In this paper we use the approach of *clearing* out the bottom half of the TNV table after profiling the instruction for a specified *clear-interval*. After an instruction has been profiled more than the clear-interval, the bottom half of the table is cleared and the clear-interval counter is reset.

We made the number of times sampled for the clear-interval dependent upon the frequency of the middle entry in the TNV table. This middle entry is the LFU entry in the top half of the table. The clear-interval needs to be larger than the frequency count of this entry, otherwise a new value could never work its way into the top half of the table. In our profiling, we set the clear-interval to be twice the frequency of the middle entry each time the table is cleared, with a minimum clear-interval of 2000 times.

4 Evaluation 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 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 [11] 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.

	Data Set 1		Data Set 2	
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

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

5 Invariance of 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 2), we used a TNV table of size 50. For all the other results we used a TNV table of size 10 for each instruction (register).

5.1 Metrics

We now describe some of the metrics we will be using throughout the paper. When an instruction is said to have an "Invariance-M" of X%, this is calculated by taking the number of times the top M 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).

In order to examine the invariance for an instruction we look at Inv-1 and Inv-5. For *Inv-1*, the frequency count of

Program	ILd	FLd	LdA	St	IMul	FMul	FDiv	IArth	FArth	Cmp	Shft	CMov	FOps
compress	44(27)	0(0)	88(2)	16(9)	15(0)	0(0)	0(0)	11(36)	0(0)	92(2)	14(9)	0(0)	0(0)
gcc	46(24)	83(0)	59(9)	48(11)	40(0)	30(0)	31(0)	46(28)	0(0)	87(3)	54(7)	51(1)	95(0)
go	36(30)	100(0)	71(13)	35(8)	18(0)	100(0)	0(0)	29(31)	0(0)	73(4)	42(0)	52(1)	100(0)
ijpeg	19(18)	73(0)	9(11)	20(5)	10(1)	68(0)	0(0)	15(37)	0(0)	96(2)	17(21)	15(0)	98(0)
li	40(30)	100(0)	27(8)	42(15)	30(0)	13(0)	0(0)	56(22)	0(0)	93(2)	79(3)	60(0)	100(0)
perl	70(24)	54(3)	81(7)	59(15)	2(0)	50(0)	19(0)	65(22)	34(0)	87(4)	69(6)	28(1)	51(1)
m88ksim	76(22)	59(0)	68(8)	79(11)	33(0)	53(0)	66(0)	64(28)	100(0)	91(5)	66(6)	65(0)	100(0)
vortex	61(29)	99(0)	46(6)	65(14)	9(0)	4(0)	0(0)	70(31)	0(0)	98(2)	40(3)	20(0)	100(0)
applu	65(1)	33(23)	19(8)	26(10)	3(0)	11(21)	6(1)	28(13)	5(13)	76(4)	54(0)	58(0)	5(2)
apsi	65(4)	13(19)	17(11)	11(11)	18(0)	9(15)	4(1)	25(13)	4(17)	94(3)	34(0)	46(0)	45(1)
fpppp	58(2)	27(31)	79(2)	17(12)	1(0)	4(24)	3(0)	46(4)	2(21)	85(1)	35(0)	19(0)	54(1)
hydro2d	76(3)	62(24)	5(13)	63(8)	68(0)	79(11)	32(1)	27(5)	73(11)	95(7)	77(1)	68(6)	81(4)
mgrid	77(1)	4(37)	0(14)	6(2)	3(0)	12(5)	50(0)	9(2)	1(35)	97(2)	63(0)	48(0)	64(0)
su2cor	37(4)	13(16)	13(16)	11(9)	15(0)	4(17)	3(0)	31(11)	2(13)	97(3)	62(2)	100(0)	36(2)
swim	56(0)	1(24)	0(18)	1(9)	45(0)	1(15)	0(1)	1(2)	2(28)	100(2)	16(0)	2(0)	64(0)
tomcatv	62(2)	0(27)	2(7)	3(8)	31(0)	2(17)	0(1)	24(3)	2(25)	99(3)	51(0)	1(1)	1(2)
turb3d	54(6)	37(17)	9(9)	39(16)	25(0)	31(13)	2(0)	25(14)	38(15)	86(3)	52(1)	47(0)	57(1)
wave5	22(4)	10(38)	16(5)	8(15)	6(0)	2(34)	2(18)	22(17)	1(31)	99(3)	51(19)	33(1)	32(2)
Avg	54(13)	43(14)	34(9)	30(11)	21(0)	26(10)	12(1)	33(18)	15(12)	91(3)	49(4)	40(1)	60(1)

Table 2: Breakdown of invariance by instruction types. These categories include integer loads (ILd), floating point loads (FLd), load address calculations (LdA), stores (St), integer multiplication (IMul), floating point multiplication (FMul), floating point division (FDiv), all other integer arithmetic (IArth), all other floating point arithmetic (FArith), compare (Cmp), shift (Shft), conditional moves (CMov), and all other floating point operations (FOps). The first number shown is the percent invariance of the top most value (Inv-1) for a class type, and the number in parenthesis is the dynamic execution frequency of that type. Results are not shown for instruction types that do not write a register (e.g., branches).

the most frequently occurring value in the final TNV table is divided by the number of times the instruction was profiled. For *Inv-5*, the number of occurrences for the top 5 values in the final TNV table are added together and divided by the number of times the instruction was profiled.

When examining the difference in invariance between the two profiles, for either the two data sets or between the normal and convergent profile, we examine the difference in invariance and the difference in the top values encountered for instructions executed in both profiles. *Diff-1* and *Diff-5* are used show the weighted difference in invariance between two profiles for the top most value in the TNV table and the top 5 values. The difference in invariance is calculated on an instruction by instruction basis and is included into a weighted average based on the first input, for only instructions that are executed in both profiles. The metric *Same-1* shows the percent of instructions profiled in the first profile that had the same top value in the second profile. To calculate Same-1 for an instruction, the top value in the TNV table for the first profile is compared to the top value in the second profile. If they are the same, 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 total number of times these instructions were profiled based on the first input. Two other metrics, *Find-1* and *Find-5*, are calculated in a similar manner. They show the percent of time the top 1 element or the top 5 elements in the first profile for an instruction appear in the top 5 values for that instruction in the second profile.

When calculating the results for Same-1, Find-1, and Find-5 we only look at instructions whose invariance in the first profile are greater than 30%. The reason for only looking at instructions with an Inv-1 invariance larger than 30% 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 30% since it is large enough to avoid variant instructions when looking at the top 5 values. For these results two numbers are shown, the first number is the percent match in values found between the two profiles, and the second number in parenthesis is the percent of profiled instructions the match corresponds to because of the 30% invariance filter. Therefore, the number in parenthesis is the percent of instructions profiled that had an invariance greater than 30%.

When comparing the two different data sets, *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.

5.2 Breakdown of Instruction Type Invariance

Table 2 shows the percent invariance for each program broken down into 14 different and disjoint instruction categories using data set 1. The first number represents the average percent invariance of the top value (Inv-1) for a given instruction type. The number next to it in parenthesis is the percent of executed instructions that this class

	Data Set 1							Data Set 2		Comparing Params in Data Set 1 to Data Set 2					
	Procedure Calls					Params		Params		Over-	Invariance		Top Values		
Program	%Instr	30%	50%	70%	90%	Inv1	Inv5	Inv1	Inv5	lap	diff1	diff5	same1	find1	find5
compress	0.27	0	0	0	0	0	1	1	2	99	0	0	96 (0)	96	95 (0)
gcc	1.23	54	48	34	17	31	43	31	43	99	1	0	96 (25)	98	98 (28)
go	1.08	9	9	1	0	10	18	12	24	99	3	1	85 (4)	95	96 (7)
ijpeg	0.15	77	21	19	19	36	72	36	72	100	0	0	34 (10)	34	25 (10)
li	2.45	45	31	25	13	33	42	42	54	99	7	2	95 (29)	96	96 (33)
perl	1.23	54	45	45	45	46	67	38	48	97	23	5	73 (28)	73	76 (34)
m88ksim	1.18	52	50	37	12	34	50	65	79	99	9	1	85 (26)	97	97 (34)
vortex	1.66	99	95	92	92	63	68	63	70	100	2	0	69 (42)	69	69 (42)
applu	0.00	91	91	90	10	80	93	53	73	100	25	1	99 (79)	99	99 (79)
apsi	0.11	99	99	99	68	61	75	71	91	100	2	1	72 (42)	72	70 (42)
fpppp	0.01	99	99	64	64	95	100	86	96	100	3	0	93 (88)	99	99 (95)
hydro2d	0.00	98	98	97	97	90	92	91	93	100	0	0	99 (89)	99	99 (89)
mgrid	0.00	80	75	62	58	65	95	66	96	100	1	0	98 (60)	100	100 (69)
su2cor	0.03	99	99	99	99	84	85	83	85	100	0	0	99 (83)	99	99 (84)
swim	0.00	57	55	54	53	46	52	45	53	100	2	1	66 (30)	67	68 (32)
tomcatv	0.02	99	99	85	85	80	91	80	91	100	0	0	99 (80)	99	99 (90)
turb3d	0.11	99	99	99	0	47	96	47	96	100	0	0	99 (41)	99	99 (48)
wave5	0.02	99	99	99	53	73	96	63	99	85	3	0	52 (31)	58	64 (46)
average	0.53	73	67	61	44	54	69	54	70	99	4	1	84 (44)	86	86 (48)

Table 3: Invariance of parameter values and 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-1 invariance greater than 30, 50, 70 and 90%. The rest of the metrics are in terms of parameters and are described in detail in §5.1.

type accounts for when executing the program. For the store instructions, the invariance reported is the invariance of the value being stored. The results show that for the integer programs, that 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.

5.3 Invariance of 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 3 shows the predictability of parameters. Instr shows the percent of instructions executed which were procedure calls for data set 1. The next four columns show the percent of procedure calls that had at least one parameter with an Inv-1 invariance greater than 30, 50, 70, and 90%. These first five columns show results in terms of procedures, and the remaining columns show results in terms of parameter invariance and values. The remaining metrics are described in detail in §5.1. 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 44% of the parameters executed (passed to procedures) for data set 1 had the same value 84% of the time when that same parameter was passed in a procedure for the second data set.

5.4 Invariance of Loads

The graphs in Figure 2 show the invariance for loads in terms of the percent of dynamically executed loads in each program. The left graph shows the percent invariance calculated for the top value (Inv-1) in the final 10 entry TNV table for each instruction, and the right graph shows the percent invariance for the top 5 values (Inv-5). The invariance shown 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-1 invariance greater than 50% for `li`, this would account for around 40% of the executed loads. The Figure shows that some of the programs `compress`, `vortex`, `m88ksim`, and `perl` have 100% Inv-1 invariance for around 50% of their executed loads, and `m88ksim` and `perl` have a 100% Inv-5 invariance for almost 80% of their loads. It is interesting to note from these graphs the bi-modal nature of the load invariance for many of the programs. Most of the loads are either completely invariant or very variant.

Table 4 shows the value invariance for loads. The invariance Inv-1 and Inv-5 shown in this Table for data set 1 is the average of the invariance shown in Figure 2. Mrv-1 is the percentage of time the most recent value was the next value encountered by the load. Diff M/I is the weighted difference in Mrv-1 and Inv-1 percentages on an instruction by instruction basis. The rest of the metrics are described in §5.1. The results show that the MRV-1 metric has a 10% difference in invariance on average, but the difference is

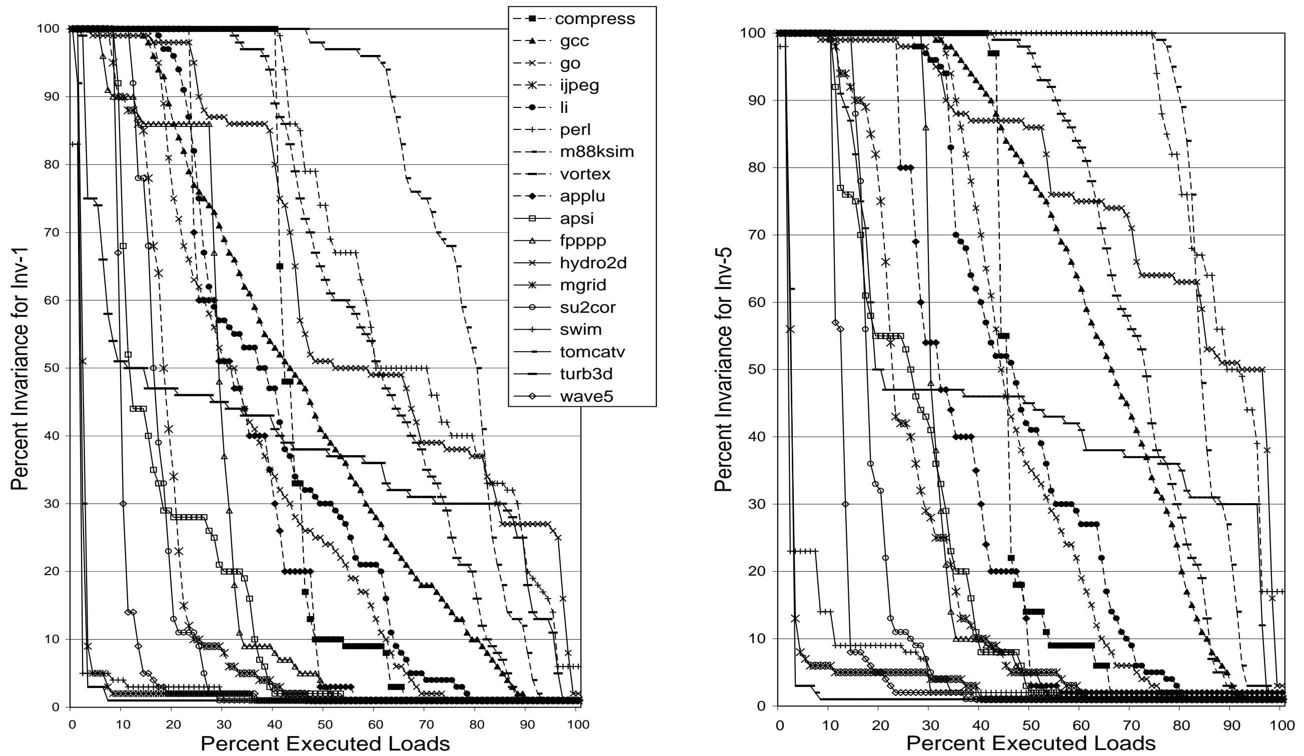

Figure 2: Invariance of loads. The graph on the left shows the percent invariance of the top value (Inv-1) in the TNV table, and graph on the right shows the percent invariance of the top 5 values (Inv-5) in the TNV table. The percent invariance is shown on the y-axis, and the x-axis is the percent of executed load instructions. The graph is formed by sorting all the instructions by their invariance, and then putting the instructions into 100 buckets filling the buckets up based on each load's execution frequency. Then the average invariance, weighted by execution frequency, of each bucket is graphed.

									Comparing Data Set 1 and Data Set 2				
	Data Set 1				Data Set 2				%	Invariance		Top Values		
Program	Mrv1	Inv1	Inv5	diff M/I	Mrv1	Inv1	Inv5	diff M/I	Overlap	diff1	diff5	same1	find1	find5
compress	49	44	46	8	53	48	50	9	99	2	0	96 (40)	96	96 (40)
gcc	54	45	64	19	53	44	64	18	99	2	0	92 (38)	93	93 (40)
go	47	35	47	19	48	38	53	18	99	3	1	93 (29)	98	98 (33)
ijpeg	45	19	26	28	47	19	26	30	99	1	0	94 (16)	94	93 (17)
li	37	39	48	11	45	43	53	17	99	6	1	94 (33)	95	95 (35)
perl	59	67	87	11	54	58	77	12	91	13	2	80 (47)	86	81 (55)
m88ksim	74	76	84	4	81	83	89	3	99	4	0	95 (71)	98	98 (75)
vortex	65	60	71	11	66	61	76	11	99	5	0	81 (47)	85	85 (51)
applu	35	33	34	13	36	35	36	12	99	0	0	71 (22)	71	71 (22)
apsi	17	18	25	5	26	29	40	7	99	4	1	55 (7)	55	55 (7)
fpppp	46	28	32	18	45	23	36	23	99	10	1	96 (25)	97	97 (25)
hydro2d	82	61	78	23	83	62	79	23	99	0	0	99 (58)	99	99 (58)
mgrid	3	4	5	0	3	4	5	0	99	0	0	99 (2)	99	99 (2)
su2cor	18	17	19	1	18	17	19	1	99	0	0	98 (16)	98	98 (16)
swim	3	1	3	1	22	7	11	15	99	0	0	99 (0)	99	99 (0)
tomcatv	3	2	2	1	4	2	3	2	100	0	0	98 (1)	99	98 (2)
turb3d	36	38	48	8	40	42	52	8	99	3	0	99 (34)	99	99 (36)
wave5	15	11	13	5	33	22	25	12	99	5	1	80 (7)	80	80 (8)
average	38	33	41	10	42	35	44	12	99	3	0	90 (27)	91	91 (29)

Table 4: Invariance of load values using a TNV table of size 10. Mrv1 is the average percent of time the current value for a load was the last value for the load. Diff M/I is the difference between Mrv1 and Inv1 calculated instruction by instruction. The rest of the metrics are described in detail in §5.1.

large for a few of the programs. The difference in invariance of instructions between data sets is very small. The results show that 27% of the loads executed in both data sets (using the 30% invariance filter) have the same top invariant value 90% of the time. Not only is the invariance between inputs similar, but a certain percentage (24%) of their values are the same.

The clearing interval and table size parameters we used affect the top values found for the TNV table more than the invariance. When profiling the loads with a 10 entry TNV table, if clearing the bottom half of the table is turned off, the average results showed a 1% difference in invariance and the top value was different 8% of the time in each TNV table using the 30% filter. In examining different table sizes (with clearing on), a TNV table of size 4 had on average a 1% difference in invariance from a TNV table of size 10, and the top value found was different 2% of the time. When using a table size of 50 for the load profile, on average there was a 0% difference in invariance and the top value was different 4% of the time when compared to the 10 entry TNV table when only examining loads that had an Inv-1 invariance above 30%.

6 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.

To calculate the invariance for the instructions within a basic block we developed a set of simple heuristics. The default heuristic used for instructions with two input registers is to set the def register invariance to the invariance of first use register times the invariance of second use register. If one of the two input registers is undefined, the invariance of def register is left undefined in the RegMap. For instructions with only one input register (e.g., MOV), the invariance of the def register is assigned the invariance of the use. Other heuristics used to propagate the invariance included the loop depth, induction variables, stack pointer, and special instructions (e.g., CMOV), but for brevity we will not go into these.

Table 5 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-1) for these instructions using the profile used to form Table 2. The fourth column is the overall estimated invariance for these instructions, and the fifth column is the weighted difference in invariance Inv-1 between the real profile and the estimation on an instruction by instruction basis. The next 7 columns show the percent of executed instructions that have an average invariance above the threshold of 10, 30, 50, 60, 70, 80 and 90%. Each column contains three numbers, the first number is the percent of instructions executed that had an Inv-1 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 83% of the instructions that have an invariance of 60% 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.

7 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

Program	% of Instrs	% Inv-1		Diff-1	% Instructions Found Above Invariance Threshold						
		Prof	Est		10%	30%	50%	60%	70%	80%	90%
compress	50	18	11	9	20 (41, 8)	9 (55, 8)	8 (53, 9)	5 (86, 0)	5 (95, 0)	4 (100, 0)	4 (100, 0)
gcc	47	46	38	13	33 (83, 3)	27 (77, 4)	22 (77, 4)	19 (77, 5)	17 (75, 6)	15 (75, 6)	13 (80, 6)
go	47	38	38	6	28 (89,12)	22 (91, 5)	18 (93, 5)	15 (95, 6)	14 (96, 7)	13 (97, 8)	12 (98, 8)
ijpeg	71	16	9	13	18 (33, 8)	13 (39,12)	11 (41,14)	9 (41,22)	8 (46,24)	8 (48,26)	6 (56,31)
li	32	36	34	8	19 (93, 5)	15 (99, 7)	14 (84, 7)	13 (79, 8)	11 (81, 9)	11 (77, 9)	9 (82,12)
perl	40	64	60	10	35 (98, 2)	30 (94, 2)	25 (83, 5)	22 (80,13)	17 (86,20)	17 (86,10)	16 (81, 5)
m88ksim	47	61	54	8	37 (82, 0)	32 (88, 0)	30 (91, 2)	28 (93, 2)	27 (93, 2)	25 (95, 1)	24 (94, 2)
vortex	40	55	47	14	28 (89, 7)	24 (86, 5)	23 (75, 5)	22 (73, 4)	21 (73, 5)	20 (74, 5)	20 (75, 6)
apsi	62	16	12	7	15 (56, 7)	11 (60,10)	8 (65,13)	7 (75,14)	7 (79,15)	5 (92,18)	5 (92,20)
fpppp	52	9	7	2	6 (73, 0)	4 (89, 0)	4 (96, 1)	4 (96, 0)	3 (96, 1)	3 (99, 1)	3 (96, 1)
hydro2d	59	56	39	19	41 (75, 1)	39 (66, 2)	32 (65, 1)	31 (66, 1)	31 (64, 1)	28 (65, 1)	18 (72, 6)
mgrid	59	5	3	2	3 (58, 5)	3 (56, 4)	3 (55, 4)	2 (94, 6)	2 (94, 6)	2 (98, 6)	2 (100, 7)
su2cor	64	15	13	2	12 (77, 0)	11 (84, 1)	10 (84, 1)	9 (88, 1)	9 (88, 1)	7 (98, 1)	7 (98, 1)
swim	65	4	3	1	2 (100, 0)	2 (100, 0)	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 (97, 2)	4 (98, 2)	4 (98, 2)	4 (98, 2)
turb3d	56	30	17	18	38 (54,11)	29 (26, 6)	8 (80, 7)	5 (77,27)	5 (76,13)	3 (94, 4)	3 (96, 4)
wave5	57	15	13	4	9 (79,10)	8 (86,12)	8 (86,12)	7 (88, 3)	7 (88, 3)	7 (88, 3)	7 (90, 3)
average	53	29	24	8	20 (75, 5)	17 (76, 5)	13 (78, 5)	12 (83, 7)	11 (84, 7)	10 (87, 6)	9 (89, 7)

Table 5: 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 7 columns show the percent of executed instructions that have an average invariance above the threshold of 10, 30, 50, 60, 70, 80 and 90%, and the percentage of these that the estimation profile found and the percent that were over estimated.

value profiler determines the impact on the time to profile. The problem with a straight forward profiler, as shown in Figure 1, is it could run hundreds of times slower than the original application, especially if all of the instructions are profiled. One solution we propose in this paper is to use a somewhat 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.

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.

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 paper 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. 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 instruction's 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. 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.

	Convergent Profile						Comparing Full Load Profile to Convergent				
	%	Convergence		%	Invariance		Invariance		Top Values		
Program	Prof	% Conv	% Inc	Backoff	% Inv-1	% Inv-5	% diff-1	% diff-5	% same-1	% find-1	% find-5
compress	2	71	29	6	48	51	5	1	91 (38)	95	95 (40)
gcc	24	42	58	9	46	67	2	0	94 (38)	98	98 (43)
go	1	49	51	22	40	57	6	1	91 (28)	99	99 (34)
ijpeg	0	50	50	25	21	31	3	1	99 (17)	99	99 (18)
li	1	12	88	70	43	58	6	2	93 (33)	98	98 (36)
perl	1	9	91	70	69	91	3	0	99 (65)	99	99 (75)
m88ksim	1	8	92	73	77	88	2	0	96 (72)	99	99 (76)
vortex	1	31	69	42	61	79	7	1	80 (47)	93	93 (55)
applu	0	72	28	7	34	35	0	0	99 (31)	99	99 (31)
apsi	0	45	55	19	38	47	19	4	96 (12)	96	96 (13)
fpppp	0	70	30	15	27	40	11	1	99 (26)	99	99 (26)
hydro2d	0	36	64	31	67	83	7	1	97 (56)	99	99 (58)
mgrid	0	59	41	8	38	39	35	7	99 (2)	99	99 (2)
su2cor	0	34	66	1	36	41	18	4	99 (16)	99	99 (16)
swim	0	68	32	8	3	8	2	0	97 (0)	99	99 (0)
tomcatv	0	61	39	5	5	6	2	0	87 (1)	87	87 (1)
turb3d	0	58	42	23	69	80	31	6	95 (32)	99	99 (36)
wave5	0	71	29	4	26	32	15	3	94 (9)	99	99 (10)
average	2	47	53	24	42	52	10	2	95 (29)	98	98 (32)

Table 6: Convergent profiler, where profiling continues if invariance is increasing, otherwise it is turned off. Prof is percent of time the executable was profiled. Conv and Inc are the percent of time the convergent criteria decided that the invariance had converged or was still increasing. Backoff is the percent of time spent profiling after turning profiling back on.

7.1 Performance of the Convergent Profiler

Table 6 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 (Conv) 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 (Backoff) 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 §5.1 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 2% 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 more than 60,000 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 of sampling for convergence (2000 invocations). These results show that the convergent profiler's invariance differed by only 10% from the full profile, and we were able to find the top value of the full length profile in the top 5 values in the convergent profile 98% of the time.

Table 7 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 2% and an inv-decrease threshold of 4%. If the invariance is not increasing by more than 2%, or decreasing by more than 4% then profiling is turned off. The results show that this heuristic spends more time profiling, 4% on average,

	%	Convergence			Back-	Invariance	
Program	Prof	Conv	Inc	Dec	off	diff1	diff5
compress	10	16	27	57	29	1	0
gcc	30	31	48	21	18	2	0
go	3	14	17	70	57	3	0
ijpeg	0	37	35	28	47	2	0
li	2	5	44	51	74	3	1
perl	0	43	38	19	19	3	0
m88ksim	0	15	23	62	75	2	0
vortex	1	13	53	34	70	5	0
applu	0	73	10	16	16	0	0
apsi	5	4	24	72	43	2	0
fpppp	1	12	12	76	77	7	0
hydro2d	22	0	61	39	99	3	0
mgrid	0	12	20	68	79	2	0
su2cor	1	10	7	83	72	1	0
swim	0	36	9	55	56	0	0
tomcatv	0	54	16	30	27	1	0
turb3d	1	3	54	43	96	13	2
wave5	1	20	8	72	76	0	0
average	4	22	28	50	57	3	0

Table 7: 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.

but has a lower difference in invariance (3%) in comparison to the first heuristic (10%). In terms of values this new heuristic only increased the matching of the top values by 1%. Therefore, the only advantage of using this second heuristic is to obtain a more accurate invariance. Table 7 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 awhile for the overall invariance of the instruction to reach its final variant behavior. The results also show that more of the time profiling, 57%, is spent after profiling is turned back on than using our first convergence criteria, 24%.

One problem is that after an instruction is profiled for a long time, it takes awhile for its overall invariance to change. If the invariance for an instruction converges after profiling for awhile 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 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.

8 Summary

In this paper we explored the invariant behavior of values for loads, parameters, and all register defining instructions. 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 found for instructions, when using value profiling, is very predictable even between different input sets. 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 10% of the full length profiler and found the top values 98% of the time. The idea of convergent profiling proposed in this paper can potentially be used for decreasing the profiling time needed for other types of detailed profilers.

We view value profiling as 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 lifetime of the program. This type of analysis should be used in combination with value profiling to identify optimizable code regions.

Acknowledgments

We would like to thank Jim Larus, Todd Austin, Florin Baboescu, Barbara Kreaseck, Dean Tullsen, and the anonymous reviewers for providing useful comments. This work was funded in part by UC MICRO grant No. 97-018, DEC external research grant No. US-0040-97, and a generous equipment and software grant from Digital Equipment Corporation.

References

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

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

[3] 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*. ACM, January 1996.

[4] 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*. ACM, January 1996.

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

[6] 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*, 1994.

[7] T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, January 1996.

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

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

[10] 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*, October 1996.

[11] 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.