Journal of Instruction-Level Parallelism 1 (1999) 1-6 Submitted 6/98; published 3/99

Value Profiling and Optimization

Brad Calder CALDER@CS.UCSD.EDU
Peter Feller PFELLER@HOME.COM
Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114 USA

Alan Eustace EUSTACE@PA.DEC.COM
Western Research Lab
Compaq Computer Corporation
250 University Avenue
Palo Alto, CA 94301 USA

Abstract

Variables and instructions that have invariant or predictable values at run-time, but cannot be identified as such using compiler analysis, can benefit from value-based compiler optimizations. Value-based optimizations include all optimizations based on a predictable value or range of values for a variable or instruction at run-time. These include constant propagation, code specialization, optimizations assuming the value predictability of an instruction, continuous optimization, and partial evaluation.

This paper explores the value behavior found from profiling load instructions and memory locations. We compare the value predictability and invariant behavior of instructions (registers) and variables (memory locations) found from value profiling across different inputs. We use the value profiles to perform code-specialization for a couple of programs, showing that value profiles can be used to reduce a program's execution time up to 21%. The ability to accurately and efficiently generate value profiles is also examined using convergent profiling and random sampling.

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 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 list or range of a variable's top values. *Value Profiling* can be used to identify the invariance and the top N values or range of a variable.

Value Profiling can also be used to identify the predictability of instructions for value prediction and value-based optimization. Value prediction [1, 2, 3, 4] enables programs to exceed the limits which are placed upon them by their data-dependencies. The goal is to break true data-dependencies by predicting the outcome value of instructions before they are executed, and forwarding these speculated values to instructions which depend on them.

We originally proposed and evaluated Value Profiling in [5]. In that work, we examined the invariance of parameters, load instructions, and the breakdown of invariance between different instruction types. We showed that the invariance and last value predictability of instructions was similar between different inputs. In addition, we presented two approaches for reducing the time for value profiling. The first approach, called

Convergent Profiling, used an intelligent form of sampling to determine when the profiling information (invariance) converged. When this happened, profiling was turned off, decreasing the time to obtain an accurate value profile. The second approach presented results for estimating the invariance of all instructions using only a basic block profile and a value profile for load instructions. We showed that an accurate estimation of invariance for the remaining instructions could be generated by propagating the invariance from the load instructions through the data dependency graph. Therefore, when profiling instructions in this paper we concentrate only on the load instructions.

In this paper we:

- Provide a detailed design analysis of the value profiler used, showing how many values need to be kept track of to capture value and invariance information.
- Examine memory location (variable) value profiling and how it differs from value profiling instructions.
- Use value profiles to perform code-specialization compiler optimizations, obtaining a 21% execution time speedup for `m88ksim` and a 15% speedup for `hydro2d`.
- Compare the performance of our previously proposed convergent value profiling to random sampling, and provide timing results for convergent profiling.

Section 2 describes related work and the potential uses for value profiling. Section 3 will describe value profiling, the algorithms and data structures used to gather value profiles, and how changing the parameters in our data structure affects the invariance and the top values found. Section 4 describes the programs and methodology used to gather the value profiles, and the metrics used to evaluate the results. Section 5 examines the invariance and the value predictability behavior of load instructions. Section 6 shows the results for profiling memory locations (variables). Section 7 provides code samples and speedup results using value profiles for value-based code specialization. Section 8 compares the performance of convergent profiling to random sampling, and Section 9 summarizes this paper.

2. Motivation and 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 indicating which instructions show a high degree of predictability. Value profiling can also be used to provide an automated approach for identifying semi-invariant variables for guiding dynamic compilation, adaptive execution, and code specialization.

2.1 Value Prediction

The recent publications on value prediction [1, 2, 3, 4] in hardware provided motivation for our research into value profiling. Lipasti et al. [3] introduced the term value locality, which describes the likelihood 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.

Last Value Prediction (LVP) is implemented in hardware using an N entry value history table [2]. The table 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 is used to hash into that table to retrieve the last value. Several additional value predictor models have been proposed [6, 7, 8, 9]. These include stride prediction, context prediction, and several hybrid approaches.

Gabbay et al. [6] studied the applicability of program profiling to aid 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. Then only instructions marked predictable were considered for value prediction. The main advantage of this approach was better usage of the prediction table, which resulted in decreased number of mispredictions.

Value prediction can benefit in several ways from value profiling. By classifying instructions into predictable, not predictable, or hard to predict, one can determine which instructions to statically predict or not to predict. Value profiling can be used to even classify instructions indicating which type of predictor would better predict the instruction for a hybrid predictor. This increases the prediction accuracy and decreases the conflicts or aliasing in a prediction table. In addition, critical path analysis can be combined with value profiling, to identify which instructions should consume a predicted value [1].

2.2 Load Speculation and Value Speculation Optimizations

The Memory Conflict Buffer (MCB) proposed by Gallagher et al. [10] provides a hardware solution with compiler support to allow load instructions to speculatively execute before stores. The compiler inserts a check instruction at the point where the load is known to be non-speculative. The check instruction checks to see if a store wrote to the same address since the speculative load was executed. If the speculation was incorrect, recover code has to be executed.

Moudgill and Moreno [11] proposed a similar approach but instead of comparing addresses as in the MCB approach, they compare the speculated value and the real value of the load. They speculatively execute the load and its dependent instructions, and they also re-execute the load in its original location. They then check the value of the speculative load with the correctly loaded value. If they are different a recovery sequence must be executed.

Fu et al. [12, 13] examined breaking dependency chains by having the compiler insert new instructions to provide predicted values for value prediction. They examined inserting add instructions to provide compiler-based stride value prediction for loads. Value profiling was shown in this work to be a valuable tool in guiding this type of value speculation scheduling.

Reinman et al. [14] examined using value profiling and memory disambiguation profiling to identify the loads that will have communication with stores, and the predictability for those loads during execution. These profiles were used to find the loads that will benefit from either memory renaming [15] or last value prediction, and then to classify the loads to use the more accurate form of prediction.

2.3 Compiler Analysis for Run-Time Optimization

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 continuous optimization. For these techniques to be effective the compiler can help determine which sections of code to concentrate on for run-time optimization. Existing techniques for dynamic compilation and adaptive execution require the user to identify run-time invariants using user guided annotations [16, 17, 18, 19, 20]. One of the goals of *value profiling* is to provide an automated approach for identifying semi-invariant variables and to use this to guide run-time optimization.

Staging analysis has been proposed by Lee and Leone [20] 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 [19] used a form of staging analysis and annotations to guide data specialization.

Autrey and Wolfe [21] have started to investigate a form of staging analysis for automatic identification of semi-invariant variables. Consel and Noel [17] 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. [16] 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 code to identify code regions that could potentially benefit from run-time code generation and optimization. Value profiling can then be used to determine which of these code regions have variables with semi-invariant behavior. Then those code regions identified as profitable by value profiling would be candidates for run-time optimization.

2.4 Code Specialization

Code specialization is a compiler optimization that selectively executes an optimized version of the code conditioned on the value of a variable. Given an invariant variable and its value, the original code is duplicated and optimized using the value. There will be one general version of the code, and a special version of the code. The two versions of the code will be conditioned on the semi-invariant variable, to choose which version to execute.

Calder and Grunwald [22] found that up to 80% of all function calls in C++ languages are made indirectly. These indirect function calls are virtual function calls, and can have multiple branch destinations. They pose a serious performance bottleneck for future processors that try to exploit instruction level parallelism. One method to reduce the penalty of virtual function calls is to create a specialized version of the method, and have the code conditioned on the type of the object used in the virtual function call. Calder and Grunwald [23] found that on average 66% of the virtual function calls had only a single destination, and these could benefit from code specialization. Hölzle et al. [24] 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. They implemented their system in Self [25], which dynamically compiles or recompiles the code applying the optimization with polymorphic inline caches. Dean et al. [26, 27] 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.

Richardson [28] 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 long latency calculations, with semi-invariant and optimizable inputs. These long latency calculations could then be specialized based on the optimizable inputs. 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 a special case 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.

```
void InstructionProfile::collect_stats (Reg cur_value) {
    total_executed ++;
    if (cur_value == last_value) {
      lvp_1_metric ++;
      num_times_profiled ++;
    } else {
      LFE_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 last value prediction (LVP) metric.

3. Value Profiling

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 value predictability of the instruction.

3.1 TNV Table

The value profiling information required 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. There is always a TNV table associated with the entity that is the target of profiling. In the case of the loads, 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 encountered (LFE) 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 can also be other useful fields as part of the structure, including a counter for the number of 0-values encountered, and a counter for stride prediction.

3.2 TNV Table Design Alternatives

For this research, we analyzed all of the parameter settings for the TNV table. The goal was to determine what settings provide the most accurate value profiling. We varied every design parameter, trying many different possibilities for table sizes and time intervals. In this paper we only provide a summary for each parameter, discussing the configurations that performed well. The detailed results for each program from varying these parameters can be found in [29].

3.2.1 REPLACEMENT POLICY FOR TOP N VALUE TABLE

We chose not to use an LRU replacement policy for the TNV table, since replacing the least recently used value does not take into consideration the number of occurrences for that value. Instead, we use a Least

Frequently Encountered (LFE), replacement policy for the TNV table. LFE chooses to replace the table entry that has the smallest frequency count. This is the value that has been encountered during profiling, according to the TNV table, the least amount of time. A straight forward LFE 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 a TNV table of size N already contains N entries, each profiled more than once, then using a least frequently encountered 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 paper 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* amount of time 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-Used* (LFU) 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 LFU entry. The clear-interval is computed by taking the maximum of the minimum clear interval size, and twice the number of times the LFU value was encountered. In this paper we used a minimum clear interval size of 2000 times. The table size is equal to the total number of steady entries added to the number of clear entries.

3.2.2 BASELINE TNV TABLE CONFIGURATION

There are three different parameters that can affect value profiling and the replacement policy. The steady part size, clear size, and the clear interval size. The invariance and top values found for these different value tables are compared to a larger TNV reference table with a table size of 50 entries and a clear interval size of 2000. In the table, 25 of these entries were for the steady part and the other 25 were for the clear part. We chose a large TNV table size for comparison, since the goal is to determine the minimum sizes for the steady part and clear part of the table for efficient, but accurate, value profiling. We use the following two metrics to examine the different TNV parameters:

- Find-Top. The percent of time the top value for an instruction in the 50 entry table is equal to one of the values for that instruction in the steady part of the smaller table being examined. This metric indicates the ability of the smaller table sizes we examine to find the top values found in a much larger table.

- Diff-All. The weighted difference in invariance between two profiles for all values in the steady part of 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 50 entry table profile, for only those instructions that are executed in both profiles. The metric shows how close the invariance found for the smaller TNV profiler matched that of the large 50 entry TNV table.

3.2.3 LEAST FREQUENTLY ENCOUNTERED VS LRU

As described above, we used a least frequently encountered replacement policy for the TNV table instead of least recently used. When a value is encountered and it is not in the TNV table, an entry in the TNV table needs to be removed in order to insert the new value. To compare the performance of LFE and LRU we present results for a 6 entry TNV table, and compare their performance to a 50 entry table using LFE. For LRU replacement, all 6 entries are candidates for replacement. For LFE replacement, we use a size of 3 for

	LFE Replacement Policy				LRU Replacement Policy		
Program	Inv-Top	Diff-Top	Diff-All	Find-Top	Diff-Top	Diff-All	Find-Top
compress	44.2	0.1	0.1	100.0	3.3	1.6	91.6
gcc	45.6	0.3	0.4	99.8	13.5	7.7	84.5
go	35.4	0.5	0.8	99.3	9.4	4.7	95.6
ijpeg	19.0	0.4	0.4	100.0	8.4	4.1	74.0
li	38.9	1.8	2.6	98.0	7.7	5.4	77.5
perl	66.9	1.3	2.6	100.0	4.5	4.7	93.1
m88ksim	76.3	0.2	0.4	99.9	14.3	5.6	93.9
vortex	60.4	0.9	1.5	100.0	12.7	7.0	86.8
applu	33.7	0.0	0.0	100.0	0.8	0.3	100.0
apsi	18.3	0.0	0.1	100.0	8.2	4.3	96.5
fpppp	28.2	0.1	0.1	100.0	2.0	0.9	97.5
hydro2d	61.9	0.2	0.2	100.0	56.7	24.0	78.3
mgrid	3.9	0.1	0.1	100.0	3.7	1.5	99.4
su2cor	17.5	0.0	0.0	100.0	3.8	1.7	90.7
swim	1.1	0.0	0.2	100.0	1.0	1.0	91.3
tomcatv	2.1	0.0	0.0	100.0	0.2	0.1	86.9
turb3d	37.9	0.4	0.5	100.0	28.4	10.9	96.7
wave5	11.0	0.0	0.0	100.0	1.1	0.7	97.8
Average	33.5	0.3	0.6	99.8	10.0	4.8	90.7

Table 1: The difference in invariance and top values found when using LFE and LRU for the TNV table replacement policy, when profiling load instructions.

the steady part of the table, and the other 3 entries are the clear part of the table. Therefore, only the 3 entries with the least frequently encountered values are candidates for replacement when using LFE replacement.

Figure 1 shows the ability for a 6 entry TNV table to capture the invariant behavior and top values when using LFE and LRU replacement policy, when compared to the 50 entry table mentioned above. The first column (Inv-Top) shows the percentage of references on average the top value found in the 50 entry table accounts for when profiling load instructions. An Inv-Top of 76% for `m88ksim` means that on average the most frequently occurring value for a load accounts for 76% of the references performed by that load. The column labeled Diff-Top shows the percent difference in Inv-Top between the 50 entry table and the 6 entry TNV table for the two replacement policies. Diff-All and Find-Top are described above. The difference metrics are weighted and calculated on an instruction by instruction basis. The results show that a 6 entry TNV table with LFE replacement finds almost all of the top values and has a degree of invariance within .3% on average when compared to the 50 entry table. Using LRU replacement for a 6 entry TNV table finds only 90% of the top values, and has an average difference in invariance of 10%. Since LFE and LRU have roughly the same implementation cost, we use LFE in all of our results.

3.2.4 STEADY STATE ENTRIES

Increasing the steady size increases the number of values that 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. Figure 2 shows the percent of executed loads captured via value profiling for different sizes for the steady part of a TNV table. For each program there are four bars. The first bar uses only one entry, the second bar uses three entries, the third bar uses five entries, and the fourth bar shows the percent of executed loads (values)

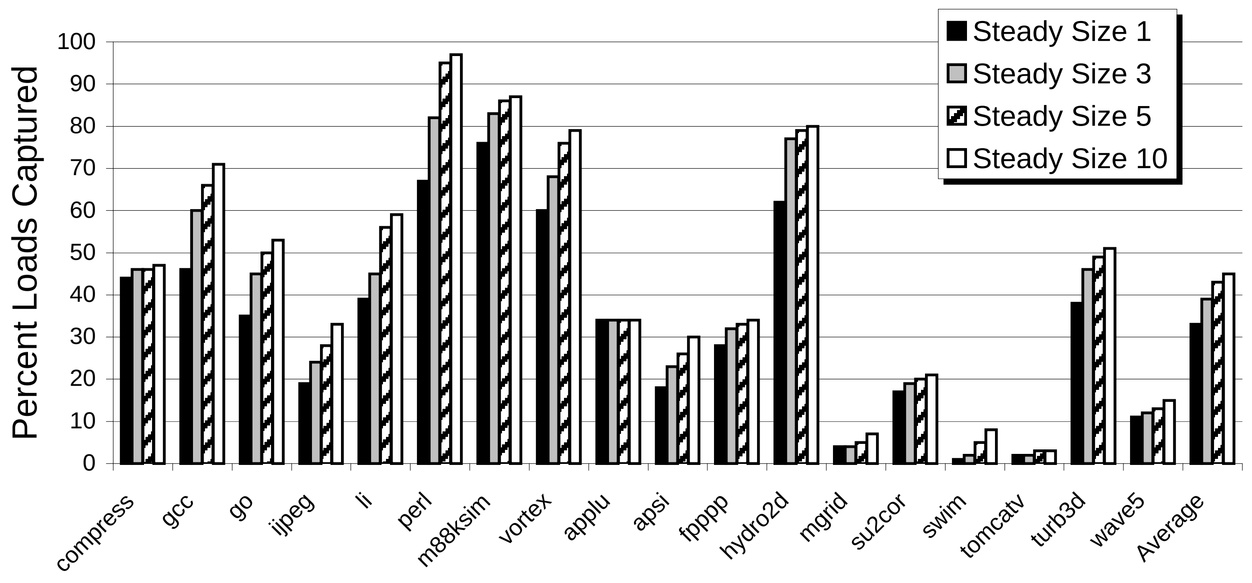

Figure 2: Percentage of Loads Executions (Values) Captured for Top N Values. This graph shows the percentage of executed loads captured using a TNV table with one, three, five, and 10 entries for its steady part.

captured when having a table with a steady part of size 10. Some programs show substantial increases in values captured when comparing the use of only one entry to using 3 or 5 steady part entries, and only a moderate gain when using ten entries. `Perl` and `m88ksim` illustrate this behavior.

Table 2 shows the difference in invariance and values found for different steady sizes in comparison to a 50 entry TNV table with a steady size of 25. The results show that the most significant performance increase resulted from increasing a steady size of 1 entry to a steady size of 2 entries. A steady size of 1 had an average 3.1% for Diff-All and 92.0% for Find-Top. Increasing the table size to 2 entries reduced the Diff-All difference to 0.8% and increased the Find-Top to 99.5%. Once the steady size reaches 4 entries, Find-Top is 100% and Diff-All is 0.4%.

3.2.5 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. In examining the clear sizes, we varied the number of table entries cleared from 0 to 5 for a TNV table of size 6. 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. Not clearing any entries in the TNV table resulted in decreased performance (number of values captured) for almost all programs. For a table size of 6 entries, the best clear size was 2 to 3 entries, which resulted in a Diff-All of 0.4% and a Find-Top of 99.9%. Detailed results for examining the clear size can be found in [29].

	Steady Sizes									
	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)
compress	0.4	100.0	0.1	100.0	0.1	100.0	0.1	100.0	0.1	100.0
gcc	3.2	92.4	0.7	99.5	0.4	99.8	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
ijpeg	1.0	96.0	0.5	100.0	0.4	100.0	0.4	100.0	0.3	100.0
li	7.2	87.0	3.1	97.7	2.6	98.0	2.0	99.5	1.6	99.5
perl	11.7	83.3	3.2	99.3	2.6	100.0	1.9	100.0	1.5	100.0
m88ksim	2.6	97.1	0.7	99.9	0.4	99.9	0.3	100.0	0.2	100.0
vortex	6.9	89.6	1.9	99.8	1.5	100.0	1.1	100.0	1.0	100.0
applu	0.3	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
fpppp	11.1	51.7	0.2	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
mgrid	0.3	100.0	0.2	100.0	0.1	100.0	0.1	100.0	0.1	100.0
su2cor	0.6	97.2	0.1	99.8	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
tomcatv	0.1	92.3	0.0	99.8	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
wave5	0.1	99.2	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
Average	3.1	92.0	0.8	99.5	0.6	99.8	0.4	100.0	0.3	100.0

Table 2: Steady Size Table. This Table shows how changing the number of entries in the steady part of the TNV table affects the accuracy of the profiling data. For each steady size the following metrics are given: D(a)=Diff-All, F(t)=Find-Top. The sizes shown are compared against a 50 entry TNV table with a steady size of 25.

	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

Table 3: Data sets used in gathering results for each program, and the number of instructions executed in millions for each data set.

3.2.6 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 LFU value. For a value to make it into the steady part of the table, the clear interval has to at least be larger than the LFU entry in the steady part of the table. We examined several different criteria for setting the clear interval and several minimum clear interval values. Having a minimum clear interval between 2,000 to 500,000 instruction invocations and setting the clear interval to be two to three times larger than the LFU entry of the steady part provided reasonable results. Overall, the results were not that sensitive to different clear interval sizes. Detailed results for examining the clear interval can be found in [29].

For the results presented in the remainder of the paper, we use a TNV table size of 6, with a steady size of 3, a clear part of 3, and a clear interval of 2000 profiled instructions.

4. 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 3 shows the two data sets we used in gathering results for each program, and the number of instructions executed in millions.

We used ATOM [30] 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.

4.1 Metrics

We now describe some of the metrics we will use in the remainder of this paper. Certain metrics are used to describe the characteristics of one particular profile, others are used to compare two profiles.

4.2 Profile Metrics

The following metrics 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 overall 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).*
     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).*
     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 steady part of 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).*
   This metric measures the number of last value 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 same as the last value for that instruction. 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 last 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.

4.3 Metrics for Comparing Two Profiles

In this paper we compare profiles of the same program profiled with different inputs, as well as profiles generated by a full profiler to that of a convergent and random profiler. When doing this, we examine both the differences in their invariances and also the difference in their top values.

4.3.1 DIFFERENCE IN INVARIANCES

The following are the two 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 steady part of the TNV table. As in Diff-Top, the difference in invariance for Diff-All 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.

4.3.2 DIFFERENCE IN VALUES

To compare two profiles we would also like to compare the top values found in both profiles. When calculating this metric, 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 (semi-random) instructions, there is a high likelihood that the top values in the two profiles are different, and we are not interested in these instructions since they would not be candidates for optimization. We use an invariance threshold of 30%. This was shown to filter out the instructions with random behavior. Results for additional thresholds can be found in [29].

1. *Finding the Top Value (Find-Top).*
   The Find-Top metric shows the percent of time the top value for an instruction in the first profile is equal to one of the values in the steady part of the table in the second profile. An instruction is only included in this metric if its Inv-Top in the first profile is greater than the invariance threshold of 30%.

4.4 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 on the x-axis. The graph is formed by sorting all the instructions by the metric on the y-axis, and then putting the instructions into 100 buckets, filling the buckets in sorted order. Each bucket represents 1% of the executed load instructions for the program. The number of instructions put into each bucket depends upon the number of times each instruction was executed. Finally, the average result, weighted by execution frequency, of each bucket is graphed. Therefore, the y-axis metric is non-accumulative, and the x-axis represents the percent of executed instructions.

5. Profiling Instructions

This section shows the results from profiling load instructions. Results are shown for the invariance, the accuracy of LVP, and the percent of zero values encountered during profiling. We conclude this section by comparing the invariance, LVP accuracy, and Zero values for two different input sets.

5.1 Invariance of Loads

Figures 3 and 4 show the invariance for load instructions in terms of the percent of dynamically executed loads in each program. Each figure is broken up into two graphs. One graph showing the invariance for the C programs, and the other graph showing the invariance for the FORTRAN programs. Figure 3 shows the percent invariance calculated for the top value (Inv-Top) in the steady part of the TNV table for each instruction, and Figure 4 shows the percent invariance for the top three values (Inv-All). See §4.4 for an explanation on how the graphs were 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 `gcc`, this would account for 45% of the executed loads. Figure 3 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.

5.2 Percent Prediction Accuracy

Figure 5 shows the Last Value Prediction accuracy for each program. It is interesting to see for programs like `compress` and `m88ksim` that their LVP graph looks very similar to their Inv-Top graph. Whereas, the LVP graph and Inv-Top graph are very different for `hydro2d` and `ijpeg`. The values for loads in these two programs have much higher prediction accuracy than invariant behavior.

5.3 Percent Zeroes

The value of zero was by far the most popular value found during value profiling. Figure 6 shows the percent of load instructions whose resulting register value is a zero value. The results for `m88ksim` show that almost 50% of the executed load instructions loaded a value of zero. The reason for this will be shown in section 7.

5.3.1 ZERO, LVP, AND INVARIANCE ACROSS DIFFERENT INPUT SETS

Table 4 compares the LVP accuracy, the percent zeroes, and the invariance between different inputs. The difference is calculated on an instruction by instruction basis subtracting the value of the metric being calculated, and then taking the weighted average. The results show that there is a high degree of similarity in the profiles for the two data sets. `Perl` showed the largest difference between inputs for all three metrics. Whereas, `swim` and `wave5` have a large difference between inputs for only LVP. For the rest of the programs, the results show that profiles from different data sets have a high correlation when using value profiling.

6. Profiling Memory Locations

In this section we extend the profiling of load instructions to profiling of variables and memory locations. The motivation for this research was to compare the predictability and invariance of load instructions with

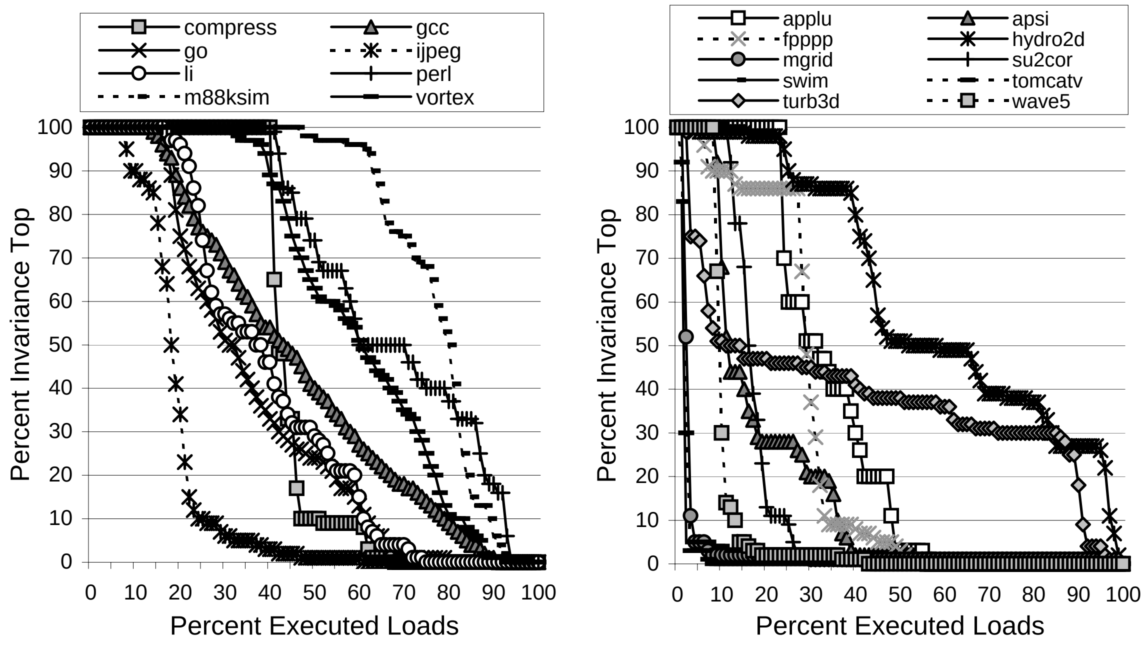

Figure 3: Invariance of Loads (Top Value). The graphs show the percent invariance of the top value (Inv-Top) in the TNV table. The percent invariance is shown on the y-axis, and the x-axis is the percent of executed load instructions.

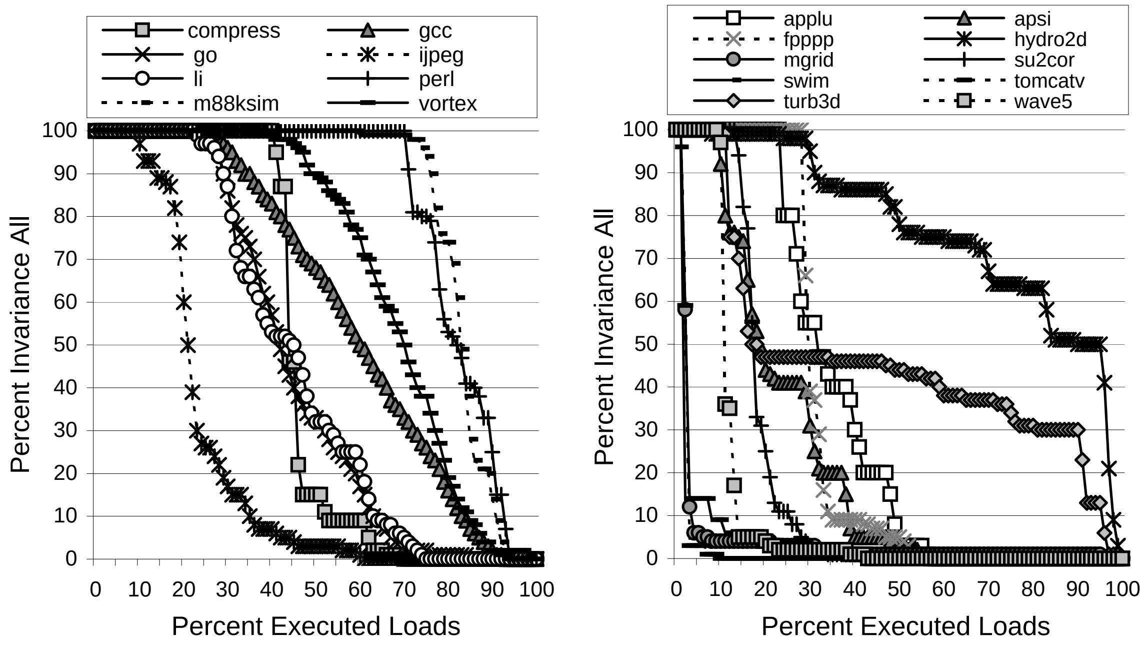

Figure 4: Invariance of Loads (All Values). The graphs shows the percent invariance of the top 3 values (Inv-All) in the TNV table. The percent invariance is shown on the y-axis, and the percent of executed load instructions on the x-axis.

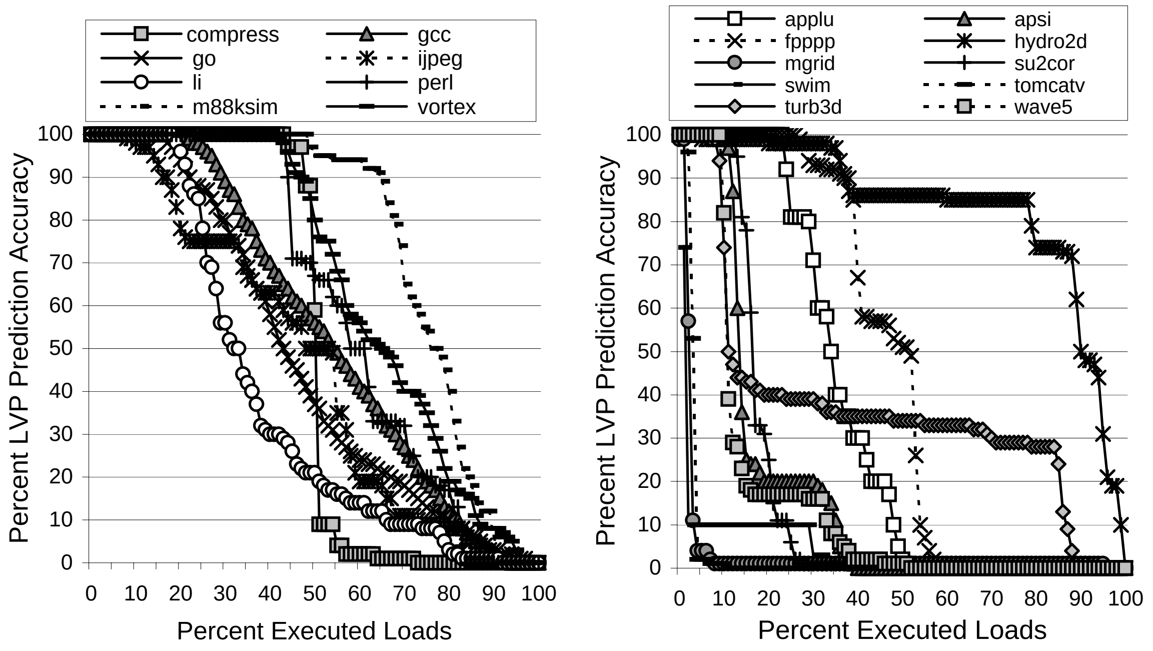

Figure 5: Last Value Predictability of Loads. The graph shows the percent prediction accuracy using LVP. The percent prediction accuracy is shown on the y-axis, and the percent of executed load instructions on the x-axis.

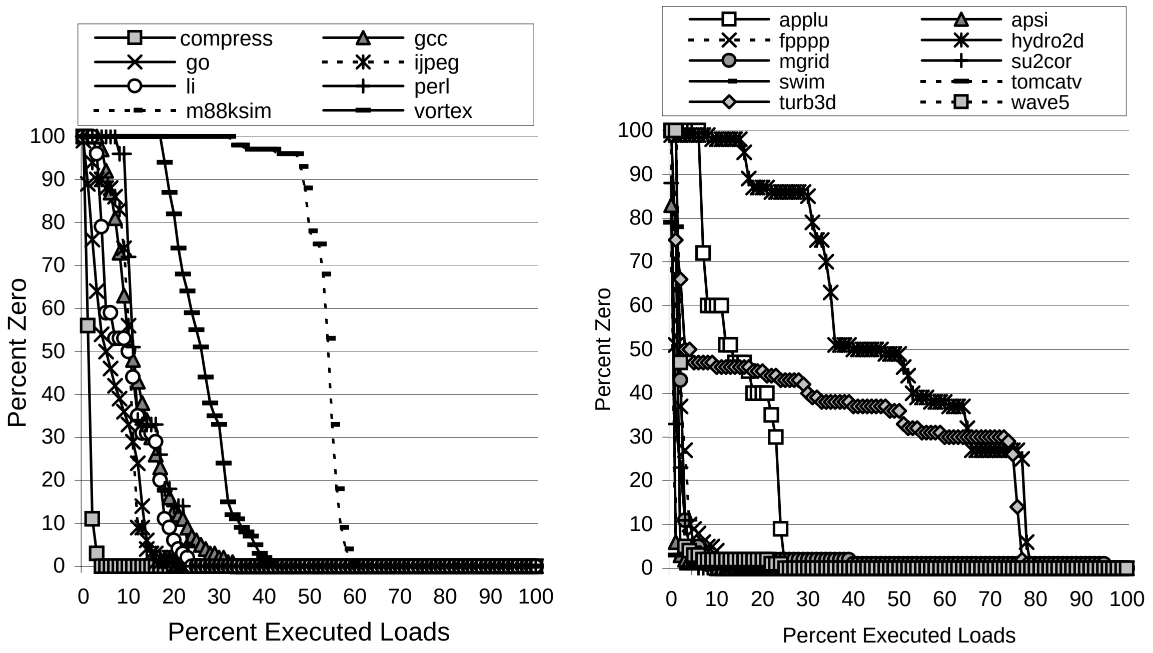

Figure 6: Zero Value for Loads. The graph shows the percent of time the result of the load instruction had a zero value. The percent zero values is shown on the y-axis, and the percent of executed load instructions on the x-axis.

	LVP			ZERO			INV		
Program	test	train	Diff	test	train	Diff	test	train	Diff-Top
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
fpppp	46	46	2	3	4	2	28	23	10
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	39	43	4	13	14	1	33	36	4

Table 4: LVP/ZERO/INV Comparison. Shows the difference between LVP, ZERO and Inv-Top from the test input to the train input.

the variables used by those instructions. Our prior research [5] focused only on profiling instructions. In conducting that research, we found that many instructions that are hard to predict or that have variant behavior actually access data (variables) that are invariant or are very predictable. The best example of this behavior are load instructions inside of a procedure that is called from several different points in the program with different variables as parameters.

A simple example of this can be seen in Figure 7. The example illustrates how it is possible for an instruction (the load of record field `p->foo` inside `ProcA`) to operate on invariant data, but to have changing values. The procedure `ProcA` is called twice inside the while loop each time passing in one of the two different objects. Another procedure `ProcB` defined in a third party library is called which may or may not modify the variables. Therefore, at compile-time it may be unknown whether the value of `foo` will change or not. Profiling the load instruction (`p->foo`) inside of `ProcA` results in a load instruction whose top value occurs only 50% of the time and a load which has 0% last value prediction accuracy. This is because the load instruction sees the value sequence `v1,v2,v1,v2,v1,v2,etc`. If the function were inlined or specialized, the load and add instruction would be 100% LVP predictable and invariant. Profiling the variables `v1` and `v2` being passed into `ProcA` would reflect this invariance correctly, rather than profiling only the load instruction. If this behavior was identified, the compiler could then specialize the different procedure sites, increasing the invariance and predictability of the instructions.

If the invariance and predictability of the parameters are profiled, compiler analysis can also be used to determine if the invariance and predictability would be higher if the procedure was specialized/inlined

```
....
while () {
   ProcA (&v1);
   ProcA (&v2);
   ProcB (&v1,&v2);
}
...

void ProcA (struct REC *p)
{
   a = b + p->foo;
   ...
}
```

Figure 7: Example of Memory Locations Being More Invariant than the Instructions. ProcA is called two times from within the while loop, with invariant parameters, but the load inside of ProcA (p->foo) only has an Inv-Top invariance of 50%.

or not. Compiler analysis cannot always determine this because of global variables, indirect calls, third party routines, or many definitions that could potentially reach a use in a procedure via long call chains. In these situations, the value profiling information would need to be kept track of using a form of *call path value profiling*, or using a memory location (variable) profile as described in this section. One could even implement a value profiler that kept track of the invariance and values for different addresses encountered by a load, on a per load basis.

6.1 Memory Location Value Profiler

The goal is to determine how the invariance and predictability of load instructions correspond to the variables they load. To profile the values stored in variables, we chose to profile at the granularity of memory locations. For our purposes, a *memory location* represents any simple data type, field within a record, or element within an array.

Instead of trying to profile the variable names, we chose to profile the values stored in memory locations (addresses) because memory locations can be dynamic, being allocated and freed during program execution. In addition, heap objects are not associated with any one variable name during execution. For the purposes of this study, we treat each consecutive eight-byte piece of memory (on the 64-bit Alpha architecture) from the start of each global or heap object as a variable and profile the top values for each 8-byte memory location. This is not exact because the size of some fields for an object and some types may be smaller than eight-bytes. When profiling stack variables, we concatenated a unique procedure ID with the load's offset, in order to create a unique address when gathering the value profile information. In addition, each allocated heap memory address is given its own TNV table. When a heap object is deallocated, its TNV information is stored to the profile on disk, and a new TNV table is created for the same memory addresses if the addresses are used again in a later memory allocation.

The memory location profiler is a two-step profiler. Step one generates a profile which contains the address, and number of times each memory location was accessed. Step two reads in the description file

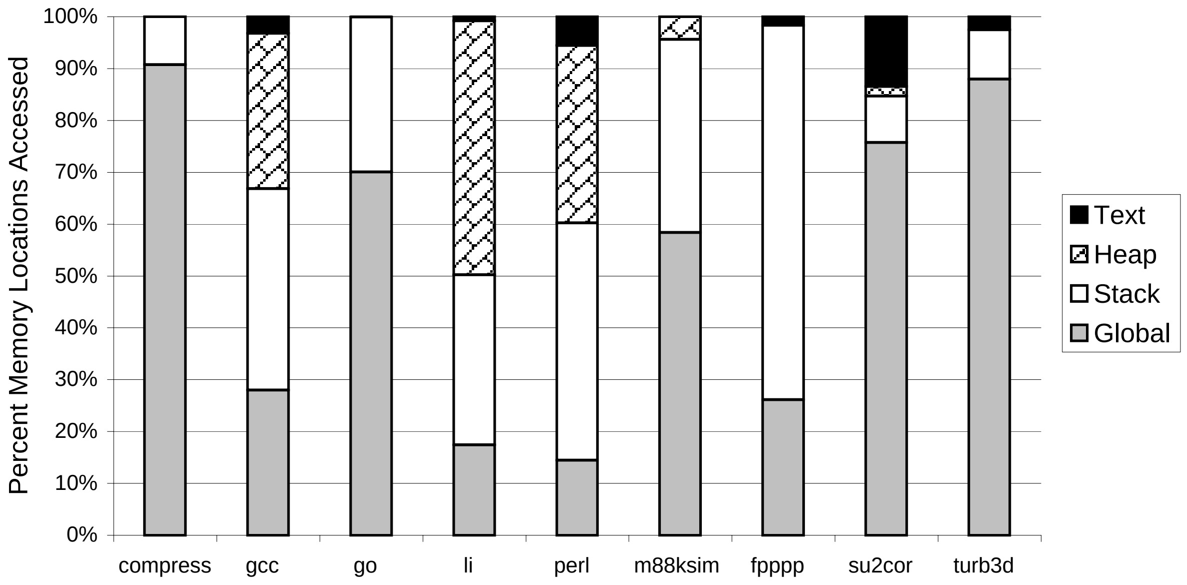

Figure 8: Memory References by Type. The graph shows the percentage of references each variable type (Global, Stack, Heap, and loads to the Text segment) accounts for in each program.

generated by the first step and then value profiles the top 99% percent of accessed memory locations. We only profile the top 99% of the accessed memory locations to reduce memory usage and time to profile.

During profiling, the memory address is first hashed and checked to see if it should be profiled (in the top 99%). If so, a TNV table exists for that memory location, and the value being loaded is recorded as in a normal TNV table update described in Section 3. In addition to the values being stored in the TNV table, for each value we keep track of the list of loads that had that value and the number of times the load had that value for the memory location during execution. With this information, we can determine exactly which load PCs were responsible for what value and how many times.

Figure 8 shows the breakdown of memory references in terms of the type of data being accessed for each program. The references are broken up into those to the global data segment, stack, heap objects, and those loads accessing the text segment.

6.2 Invariance of Memory Locations

Figure 9 displays the invariance of memory locations with respect to percent of memory locations referenced. It is interesting to note, that for certain programs, the invariance is much higher for memory locations than it is for load instructions. 15% of the loads for `su2cor` are shown to be 100% invariant in Figure 3. In comparison, 33% of the memory locations accessed by `su2cor` are shown to have 100% invariance in Figure 9. The values seen by the load instructions varied more than the memory locations because the loads accessed several different addresses (memory locations). Another interesting observation is the extreme *bi-polar* nature of the invariance of the memory locations. The figure shows that either the value in the memory location almost never changed (100% invariant) or it was very random (0% invariant).

Figure 10 shows the same information as Figure 9 for a history depth of three. This shows the invariance of the top three values found for each memory location.

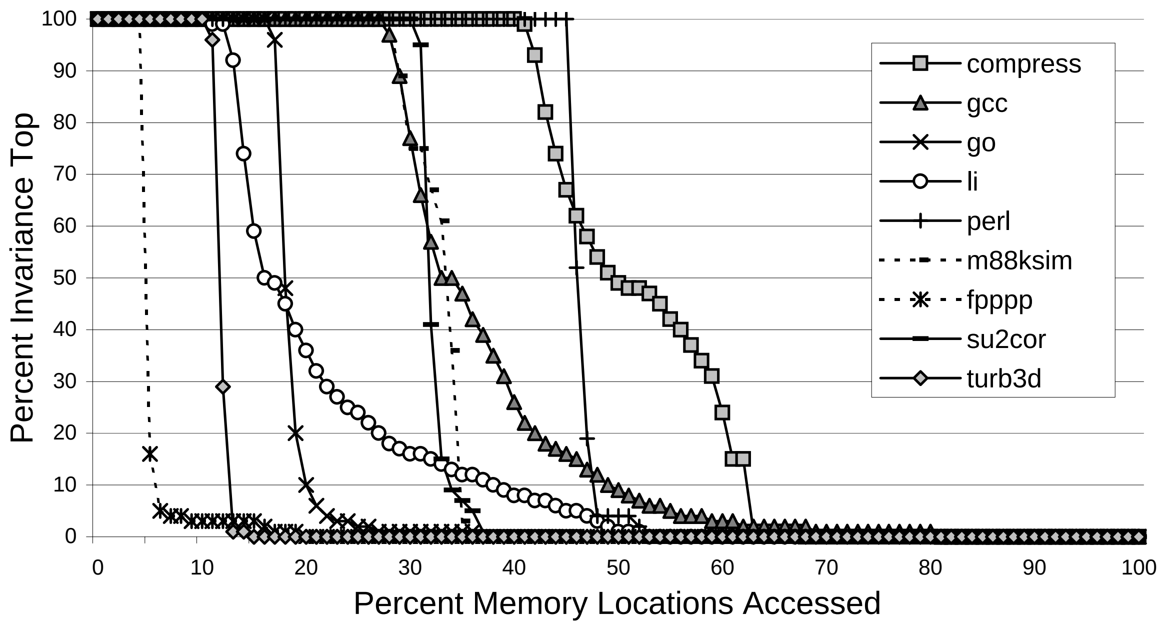

Figure 9: Invariance of Memory Locations (Top). The graph shows the percent invariance of the top value in the TNV table. The percent invariance is shown on the y-axis, and the percent of accessed memory locations on the x-axis.

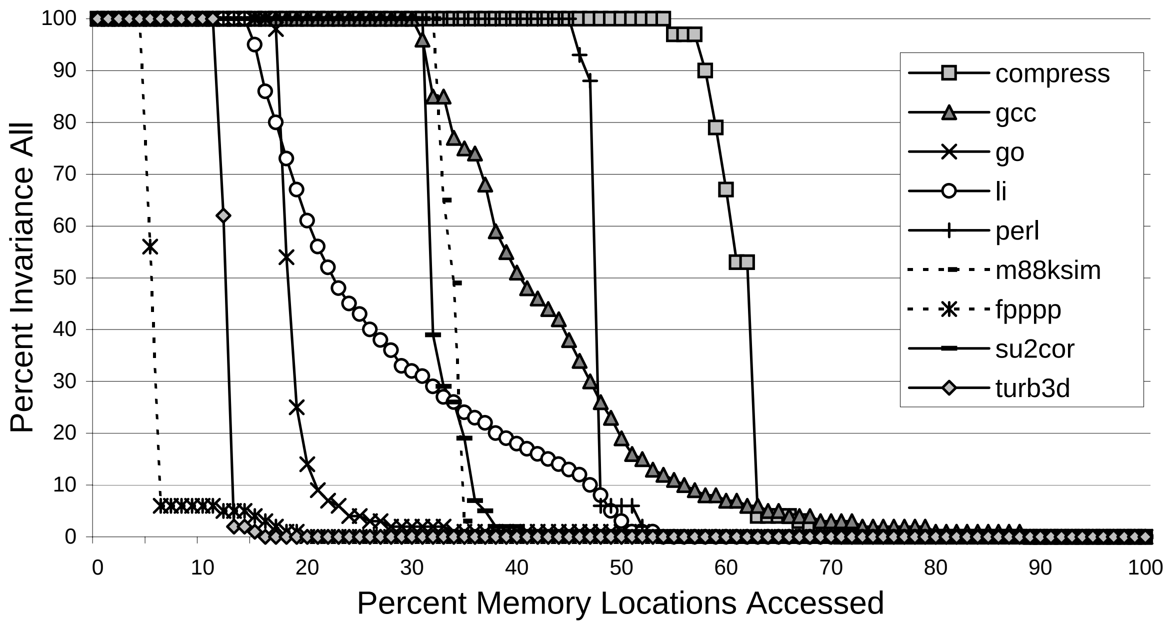

Figure 10: Invariance of Memory Locations (All). The graph shows the percent invariance of all the values (Inv-3) in the TNV table. The percent invariance is shown on the y-axis, and the percent of accessed memory locations on the x-axis.

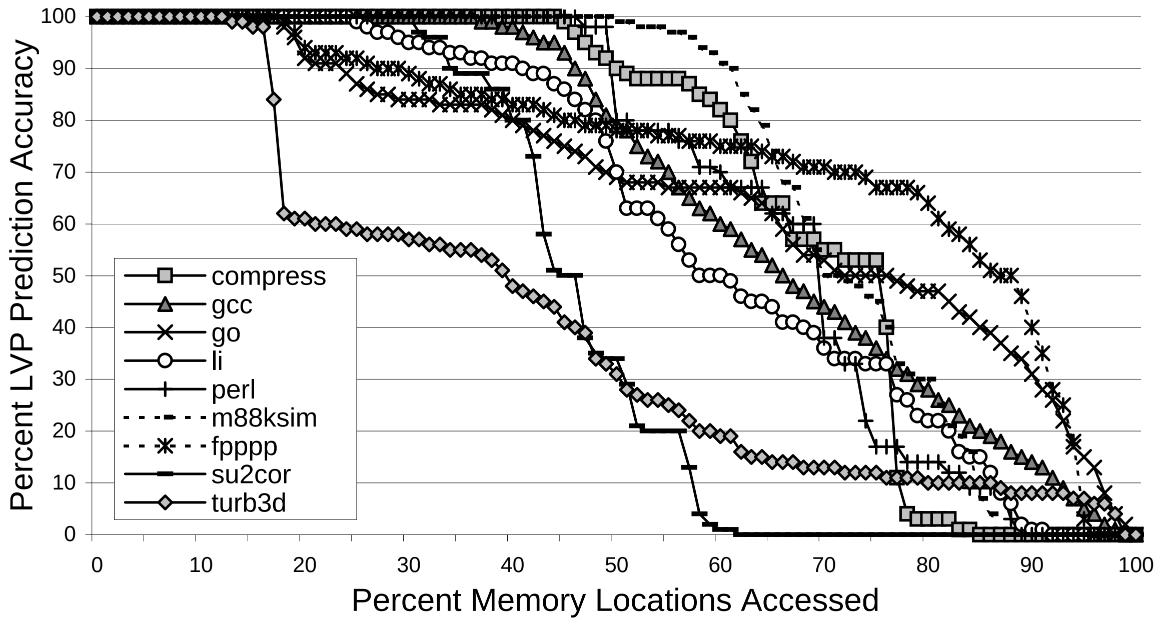

Figure 11: LVP of Memory Locations. The graph shows the percentage of time the value of the Memory Location was the same as for the previous time the memory location was accessed. The percent LVP is shown on the y-axis, and the percent of dynamically referenced memory locations on the x-axis.

6.3 LVP Accuracy of Memory Locations

Figure 11 illustrates the last value prediction accuracy for accessed memory locations. 50% of all memory locations accessed have an LVP of 80% or higher for almost all the programs. It is interesting to note that the LVP prediction accuracy is much higher than the invariance found in Figures 9 and 10. This occurs because even when a new value is stored to a given memory location, it still might be loaded by a handful of different load instructions before its value changes via a store. This can easily create a LVP accuracy in the 80% even though the top value seen for a memory location is not significant. This can be seen for `fpppp` which has 0% invariance for more than 90% of its memory references, but it still has higher than a 70% LVP prediction accuracy for 80% of its references.

6.4 Number of Memory Locations Accessed by a Load

Table 5 illustrates how many different memory locations a given load instruction accesses. The first number in each column shows the number of static loads, and the number in parenthesis shows the weighted percent of executed loads they account for. Results show that 51% of the executed loads access only one memory location. 24% of all executed static loads, accessed between two and ten different memory locations. An extreme case is `su2cor` for which 8% of the executed static loads accessed between 100,000 and 1,000,000 different memory locations. In a case like this it would be very unlikely to achieve a high degree of invariance for those load instructions, unless all those memory locations contain the same values.

7. Value-based Code Specialization

In this section we examine the performance potential of using value profiling to guide code specialization. We took the instruction value profile for the top C and FORTRAN program with the highest invariances,

Programs	1	10	100	1000	10000	100000
compress	210 (77.0)	27 (17.6)	0 (0.0)	0 (0.0)	2 (1.9)	6 (3.5)
gcc	12445 (55.1)	7608 (32.5)	3385 (10.7)	2210 (1.4)	30 (0.3)	0 (0.0)
go	4312 (41.4)	2552 (27.7)	2617 (8.1)	1165 (9.8)	740 (13.0)	0 (0.0)
li	634 (57.2)	256 (14.1)	54 (6.3)	52 (0.4)	87 (18.9)	17 (3.0)
perl	290 (53.5)	80 (46.5)	0 (0.0)	0 (0.0)	0 (0.0)	0 (0.0)
m88ksim	800 (43.4)	226 (39.7)	68 (16.9)	3 (0.0)	0 (0.0)	0 (0.0)
fpppp	2630 (67.0)	584 (32.4)	59 (0.2)	22 (0.4)	0 (0.0)	0 (0.0)
su2cor	1128 (51.5)	167 (3.2)	38 (0.4)	16 (6.5)	382 (30.2)	36 (8.3)
turb3d	237 (13.9)	34 (4.8)	62 (81.3)	0 (0.0)	0 (0.0)	0 (0.0)
Average	2520 (51.1)	1281 (24.3)	698 (13.8)	385 (2.1)	137 (7.1)	6 (1.6)

Table 5: Percent Static Loads Accessing Different Memory Locations. This table shows the number of static loads accessing 1 to 1,000,000 different addresses. The number in parenthesis shows the weighted percent of dynamic references the static loads account for. The last column shows the static loads referencing between 100,000 to 1,000,000 different memory locations.

`m88ksim` and `hydro2d`, as shown in Figure 3. We then used the value profile information to guide specialization performed by hand. The programs were run on an Alpha 21164 500 MHZ processor, and they were compiled with the DEC compilers using full optimization (-O5 -ifo -om). The execution times were recorded using the SPEC runspec scripts. Our results show that value profiling can be an effective tool for providing feedback to profile-driven optimizers, and it can be a useful performance profiling tool for programmers pointing out potential places for optimization.

7.1 M88ksim

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

Figure 12 illustrates the code specialization performed on the routine `alignd`, where `amantlo` and `amanthi` are parameters to the function. Our value profiler indicated 100% invariance for `amantlo` and `amanthi`, with a value of zero. Using the value profile, the compiler could perform the simple code transformation, as shown in Figure 12(b). Specializing the if-clause to only execute when `amantlo` and `amanthi` are non-zero decreases the execution time of `m88ksim` by 13%.

Figure 13 shows the `killtime` routine in `m88ksim`. The value profile indicated that the load which was responsible for loading `m88000.time_left[i]` had a zero value 99% of the time, and so did the load for `funct_units[i].busy`. Straight forward compiler code specialization did not provide a performance benefit. Instead, this is an example where a performance tool pointed out a potential performance problem (a lot of redundant computation), which could easily be optimized by a programmer.

Since only a few registers if any had a non-zero `time_left` value during the execution of the first loop in `killtime`, we restructured the algorithm by using a small array that keeps a map of the non-zero `time_left` elements. This array would only contain those elements of `time_left` that are non-zero. Since this list is short, the while loop only needs to be executed a few times. The optimized code is shown in Figure 14. The map is added to the structure `PROCESSOR`. Additional code in `killtime` is required

```
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 12: `m88ksim: alignd` - The original 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%.

```
  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 );
  }
}
```

```
int execute(...)
{
...
  if ((f->rsd_used) && (ir->dest ! = 0)) {
    m88000.Regs[ir->dest] = m88000.ALU;
    if ((usecmmu) && (ir->op >= (unsigned)LDB)
      && (ir->op <= (unsigned)LDHU)) {
      m88000.time_left[ir->dest] = f->fu_latency + Dcmmutime;
      if ((ir->op == LDD) && ((ir->dest) <= 30)) {
        m88000.time_left[(ir->dest)+1]=f->fu_latency+Dcmmutime+1;
      }
    } else {
      m88000.time_left[ir->dest] = f->fu_latency;
      if ((ir->op == LDD) && ((ir->dest) <= 30)) {
        m88000.time_left[(ir->dest)+1] = f->fu_latency + 1;
      }
    }
    m88000.wb_pri[ir->dest] = f->wb_pri;
  }
...
}
```

Figure 13: `m88ksim`: `killtime` - The original function `killtime` accounts for 23% of all executed instructions in `m88ksim`.

```
struct PROCESSOR {
...
  WORD time_left_map_next, /* next map register */
  WORD time_left_map[REGs], /* map time_left regs */
...
}

void killtime (unsigned int time_to_kill) {
 register int i;
 Clock += time_to_kill;
 i = 0;
 while (i < m88000.time_left_map_next) {
  int index = m88000.time_left_map[i];
  m88000.time_left[index] -= MIN(m88000.time_left[index], time_to_kill);
  if (m88000.time_left[index] == 0) {
   m88000.time_left_map[i] =
    m88000.time_left_map[--m88000.time_left_map_next];
  } else
   i++;
 }
 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 );
 }
}

int execute(...) {
...
 if ((f->rsd_used) && (ir->dest ! = 0)) {
  m88000.Regs[ir->dest] = m88000.ALU;
  if ((usecmmu) && (ir->op >= (unsigned)LDB) && (ir->op <= (unsigned)LDHU)) {
   if (m88000.time_left[ir->dest] == 0)
    m88000.time_left_map[m88000.time_left_map_next++] = ir->dest;
   m88000.time_left[ir->dest] = f->fu_latency + Dcmmutime;
   if ((ir->op == LDD) && ((ir->dest) <= 30)) {
    if (m88000.time_left[(ir->dest)+1] == 0)
     m88000.time_left_map[m88000.time_left_map_next++] = (ir->dest)+1;
    m88000.time_left[(ir->dest)+1]=f->fu_latency+Dcmmutime+1;
   }
  } else {
   if (m88000.time_left[ir->dest] == 0)
    m88000.time_left_map[m88000.time_left_map_next++] = ir->dest;
   m88000.time_left[ir->dest] = f->fu_latency;
   if ((ir->op == LDD) && ((ir->dest) <= 30)) {
    if (m88000.time_left[(ir->dest)+1] == 0)
     m88000.time_left_map[m88000.time_left_map_next++] = (ir->dest)+1;
    m88000.time_left[(ir->dest)+1] = f->fu_latency + 1;
   }
  }
  m88000.wb_pri[ir->dest] = f->wb_pri;
 }
...
}
```

Figure 14: `m88ksim: killtime` - This is an optimized version a programmer could create by changing their data structures. The changes in are shown in bold text. This optimization resulted in a 9% execution speedup.

```
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 15: `hydro2d: filter` - the function `filter` accounts for 13% of the executed instructions in `hydro2d`. The original code is shown in (a) and the specialized version in (b). This modification led to a speedup of 11%.

to make sure that items get deleted from the map when their `time_left` becomes zero. The routine `execution` had to be modified to make sure that the map was updated properly. A register was added to the map every time its `time_left` value changed.

The `killtime` optimization provided a 9% execution time speedup. Performing both the `alignd` and `killtime` optimizations together resulted in an overall execution speedup of 21% for `m88ksim`.

7.2 Hydro2d

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.

Figure 15(a) shows a small fraction of the function `filter` from the SPEC95 FORTRAN program `hydro2d`. This code fragment is inside of a loop that accounts for 13% of all executed instructions. Our value profiler found `QP` and `QM` to be zero 98% of the time. Performing the simple code specialization shown in Figure 15(b) resulted in 11% speedup.

Figure 16(a) shows the original 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% using the optimized code in Figure 16(b). There were a few additional 1-2% speedup cases we found via the value profile for `hydro2d` resulting in an overall execution speedup of 15%.

```
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 16: `hydro2d: tistep` - the function `tistep` accounts for 4% of the executed instructions in `hydro 2d`. The original code is shown in (a) and the specialized version in (b). This modification led to a speedup of 2%.

8. 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 1, is it could run a hundred times slower than the original application, especially if all of the instructions are profiled. One solution we proposed in [5] was 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 17 shows the invariance of load values for `compress` throughout program execution. Each static load has its execution broken up into 10 time intervals, and the invariance is kept track of separately for each time interval and then plotted. Each interval in this example represents 10% of a load's execution. The figure shows that about 40% of all executed loads are *always* invariant, and 30% are random across all time intervals. The convergent profiler would classify these loads quickly into their respective category. The remaining 30% of the loads have and invariance that varies over time, and these loads pose a greater challenge for a sampling profiler to obtain an accurate invariance. The goal for using this intelligent sampler is to quickly classify those instructions that are invariant or random, while continuing to profile the harder

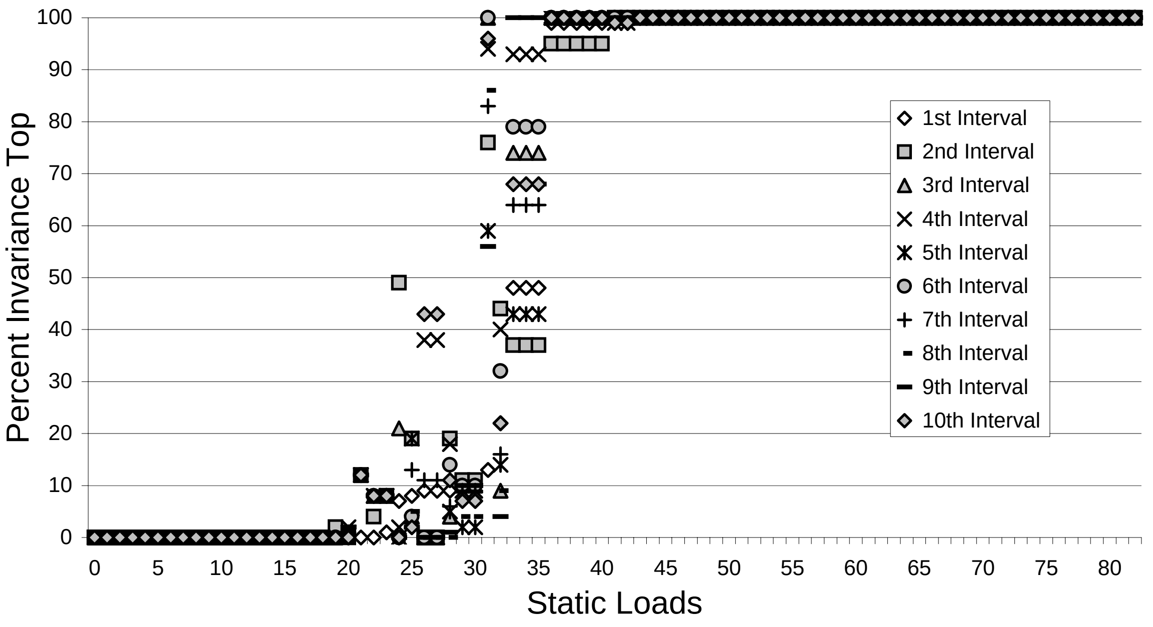

Figure 17: Interval Invariance for Compress. This graph shows the invariance during different parts of execution for each instruction. Each instructions execution is broken up into ten equal chunks of time based on number of load instructions executed. The invariance was then gathered for each static load instruction for each of the ten intervals. The invariance for the top value found for each interval is plotted for each static load. Only load instructions executed more than 100 times are shown.

to classify instructions. For example, Figure 17 shows that the static loads numbered 33, 34, and 35 had a wide range of invariance during their execution. Over the 10 sampled intervals their invariance oscillated with the following Inv-Top invariances: 48%, 37%, 74%, 93%, 42%, 79%, 64%, 100%, 100%, and 68%.

8.1 Convergent Value Profiler

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 *backoff* time period has elapsed. This is used to periodically turn profiling back on to see if the invariance is at all changing.

We examined the performance of several heuristics for the convergence criteria for value profiling. The heuristic we found to provide the best performance only continues to profile 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

	Convergence Algorithm							
	Conv(Inc=10%/Dec=10%)				None (Random)			
Program	Prof	Diff-Top	Diff-All	Find-Top	Prof	Diff-Top	Diff-All	Find-Top
compress	4.2	1.7	0.3	97.7	4.1	3.3	0.7	99.7
gcc	21.8	2.7	0.6	98.0	23.0	3.0	0.5	97.0
go	1.3	4.2	1.0	99.8	1.1	4.4	1.3	99.8
ijpeg	0.2	2.3	0.7	99.7	0.3	2.6	0.8	99.9
li	0.3	5.1	1.6	98.0	0.2	5.6	1.8	98.1
perl	0.1	3.0	0.4	99.7	0.2	2.3	0.5	100.0
m88ksim	0.1	2.3	0.3	99.6	0.1	2.6	0.6	100.0
vortex	0.4	6.7	1.0	98.6	0.4	7.2	1.1	99.0
applu	0.1	0.2	0.1	100.0	0.3	0.3	0.1	100.0
apsi	0.2	4.2	1.0	99.9	0.5	8.4	1.9	100.0
fpppp	0.5	7.9	0.6	100.0	0.3	9.3	0.9	100.0
hydro2d	8.2	4.7	1.4	95.7	0.2	6.3	1.2	99.8
mgrid	0.1	5.3	1.3	99.8	0.0	25.0	5.1	99.8
su2cor	0.3	3.0	0.7	99.9	0.3	12.9	2.8	99.7
swim	0.0	0.9	0.4	100.0	0.0	1.3	0.6	100.0
tomcatv	0.1	1.2	0.4	100.0	0.1	1.8	0.5	100.0
turb3d	0.2	9.0	1.9	99.9	0.1	24.3	5.0	100.0
wave5	0.3	2.1	0.7	99.9	0.4	10.5	2.5	100.0
Average	2.1	3.7	0.8	99.2	1.8	7.3	1.5	99.6

Table 6: Results for different converging algorithms using the fast-converging backoff. The invariance and values found for convergent profiling and random sampling are compared to a value profile of the complete execution of the program.

within these bounds. When calculating the invariance the total frequency of the steady part of the TNV table is examined. For the results, we use a convergence-interval for testing the criteria of 2000 instruction executions, and 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.

Once profiling is turned off, it will be turned on after the *backoff* period has elapsed. We examined several different backoff methods. The technique we used in this paper is a fast-converging backoff method, which is computed by multiplying the total number of instructions profiled by twice the number of times the instruction's invariance has converged. The more times the instruction converges, the longer the backoff period.

Table 6 shows the performance of the convergent profiler, when using the upper and lower change in invariance bounds for determining convergence. The Prof column shows the percentage of executed load instructions that had profiling turned on. The results show that this heuristic spends on average 2.3% of its time profiling. When comparing the invariance found to a profiler that profiles the full length of a program's execution, there was only a 3.7% difference in invariance. Results for additional thresholds and different backoff periods/algorithms can be found in [29].

	Time	Slowdown		
Program	Original	Freq	Conv	Full
compress	0.5s	3.0	6.4	20.4
gcc	4.6s	3.7	11.0	30.8
go	120.4s	4.7	13.2	45.3
ijpeg	74.2s	5.5	10.5	35.0
li	57.1s	5.5	11.9	40.5
perl	53.0s	5.0	14.9	37.7
m88ksim	206.7s	4.4	9.8	28.2
vortex	141.6s	4.7	12.6	55.0
applu	294.2s	2.3	4.7	15.6
apsi	108.9s	3.1	6.5	22.8
fpppp	286.2s	7.6	31.7	104.3
hydro2d	276.2s	2.1	4.3	11.1
mgrid	253.2s	4.7	11.7	43.8
su2cor	180.1s	2.4	4.8	16.9
swim	173.9s	2.6	5.6	21.5
tomcatv	197.2s	2.3	4.7	16.9
turb3d	215.2s	3.8	8.3	26.7
wave5	163.9s	2.4	5.3	19.6
Average	156.0s	3.8	10.0	32.9

Table 7: The time it takes to execute and profile each program. Execution time is shown in seconds. Simulation time is shown as multiples of the original execution time.

8.2 Comparing Convergent Profiling and Random Sampling

We now analyze how the convergent algorithm compares to a random sampling method. Unlike a traditional random sampler which randomly samples instructions every so often taking the value, our random sampler continuously samples instructions for a given time-interval, and then backs off for a random amount of time, and then samples again. The number of instructions to backoff, is a random number where the upper bound to the random function increases exponentially. The upper bound starts at 2000 and increases exponentially each time a new sampling interval takes place. Therefore, the random function chooses a random number between 2000 and the upper bound of instructions to backoff. When profiling starts, a sample of 2000 values is taken. Then profiling for the instruction is turned off a random amount of time using the two bounds. This sampling and random backoff continues until the program finishes execution. Table 6 shows that when using the Conv(Inc/Dec) heuristic, better results are achieved for the difference in invariance. However, both the convergent and random algorithm found all the top values when compared to the full length profile.

In terms of invariance, the Conv(Inc/Dec) heuristic performs better for all programs. For `mgrid`, `su2cor`, `turb3d`, and `wave5`, the difference in invariance is 11% to 25% for random sampling, compared to 2% to 9% for the Conv(Inc/Dec) heuristic. These results show the need for some form of intelligent sampler for these programs, but random sampling provides good results for many of the programs.

Random sampling can be either as accurate as convergent profiling, less accurate, or more accurate. It depends on the sampling regimen, and the amount of time the program is profiled. We actually examine several different backoff algorithms and different sampling periods, but only showed results in Table 6 for a sampling configuration that gave good performance and the same profiling time as convergent profiling. Results for additional random sampling algorithms and random backoff thresholds can be found in [29].

8.3 Convergent Profiling Timings

Figure 7 shows the time it takes to execute and profile the SPEC programs. The first column shows the time in seconds to execute the programs on a 500 MHZ Alpha 21164. All of the rest of the results are shown as multiples of the original execution time. As described in section 4, all programs were compiled with full optimization (`-O4 -ifo`), and all programs were profiled using ATOM [30] with the `-A1` option for optimization. ATOM is made to be a general purpose profiling tool for research, so a performance hit is taken when using it for profiling. To see this effect, the column labeled Freq shows the performance when using ATOM to only keep track of the number of times each load was executed. We insert an ATOM call for every load instruction. ATOM then creates a procedure call to our analysis routine which increments a counter for the load, when the load is executed. A commercial profiler could perform in-lining to help reduce the overhead of performing the procedure call for each load, and perform additional optimizations on the data structures after in-lining. Since ATOM does not perform/allow this in-lining, it has a fixed amount of additional overhead. The results show that it takes 3.8 times longer to execute the program when gathering these frequency counts using ATOM. The column labeled Conv shows that the convergent profiler results in an average 10 time slowdown when profiling. Profiling the load instructions for the complete execution of the program results in an average slowdown of 33 times.

9. Summary

In the paper we presented value profiling and examined the design of a simple value profiler. Our results showed that using a value profiling table that keeps two to four values in the steady part of the table throughout profiling captures the invariance and the top values for the programs examined. Having two to three clear entries reserved to allow new values to make it into the steady part of the table, also resulted in an accurate value profile. Therefore, only a table size of 4 to 6 entries is needed to accurately capture the invariant information.

This paper explored the invariant behavior of values for loads, and memory locations, as well as their value predictability. Our results show that the invariance and LVP, found for instructions when using value profiling, are very predictable between different input sets.

As an alternative to profiling instructions, we implemented a technique for profiling memory locations (variables). Value profiling memory locations (variables) could expose more optimization potential than only using an instruction value profiler. This can expose invariant behavior for instructions inside of procedures called from several locations in the program. Our results showed higher prediction accuracy and more invariant behavior for the memory locations over profiling instructions. Also, the memory locations were much easier to classify as very invariant/predictable or very random/unpredictable than the load instruction profiles. Using compiler analysis with value profiling of parameters or value profiling with path information could also potentially expose more opportunities for optimization.

Convergent value profiling was examined as an accurate technique to decrease profiling time. We compared the performance of convergent profiling to a random sampler and found that on average both profiled about 2% of the executed instructions, with an average 10 time slowdown in execution. Convergent profiling had an invariance that was only 3.7% different from a full length profiler, whereas the random sampling had an average difference in invariance of 7.3% from the full length profiler. For half of the programs the random sampler was just as accurate as the convergent profiler, but for the other half the convergent profiler was up to 20% more accurate than random sampling.

We also showed the potential of value profiling for optimization. Using value profiling for compiler-based code specialization resulted in a 13% speedup for `alignd` in `m88ksim` and a total 15% speedup for `hydro2d`. In addition, we showed the benefit of using value profiling as a performance analysis tool. Value profiling identified a significant amount of redundant computation for `killtime` in `m88ksim`, and with a simple change in algorithm and modification of a data structure, we were able to reduce the execution time by 9% for `m88ksim`. Combining both optimizations for `m88ksim` resulted in a combined 21% reduction in execution time.

9.1 Future Directions

The convergent profiler we examined was not fully optimized for memory usage and instruction count, and had additional overhead from using ATOM that a commercial profiler would not necessarily have. Therefore, one should be able to further reduce the overhead shown in this paper due to value profiling. In addition, using hardware sampling, as in DCPI [31], could potentially reduce this overhead to a few percent.

Different algorithms could also be used for value profiling to reduce the profiling time, but at a tradeoff of a lose in accuracy. An example of this would be to use random index into the TNV table. If the index is a hit (same value), then the frequency counter is incremented, and if it is a miss the value would be replaced. If the goal of the profiler is to only find the most invariant load instructions, random index/replacement could provide accurate results.

Another future direction for this research is motivated by the value prediction accuracies found for the memory locations. The last value prediction accuracies shown in Figure 11 are much higher than seen when predicting the last value based on the instruction's PC as shown in Figure 5. This suggests that hardware value prediction could possibly benefit from predicting some load instructions using a form of memory address to index into the value table instead of the load instruction's PC. To make such an architecture worthwhile, an address other than the effective address would probably be needed. Two studies [32, 33] have suggested predictive techniques for generating fast or approximate memory addresses early in the pipeline, which could then be used to index into a value prediction table.

Acknowledgments

We would like to thank the anonymous reviewers for the useful comments. This work was funded in part by NSF CAREER grant No. CCR-9733278, UC MICRO grant No. 97-018, and a Compaq Computer Corporation external research grant No. US-0040-97.

References

[1] B. Calder, G. Reinman, , and D. Tullsen, "Selective value prediction," in *26th Annual International Symposium on Computer Architecture*, May 1999.

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

[3] M. Lipasti and J. Shen, "Exceeding the dataflow limit via value prediction," in *29th International Symposium on Microarchitecture*, pp. 226–237, Dec. 1996.

[4] M. Lipasti, C. Wilkerson, and J. Shen, "Value locality and load value prediction," in *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, Oct. 1996.

[5] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *30th International Symposium on Microarchitecture*, pp. 259–269, Dec. 1997.

[6] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?," in *30th International Symposium on Microarchitecture*, pp. 270–280, Dec. 1997.

[7] Y. Sazeides and J. E. Smith, “The predictability of data values,” in *30th International Symposium on Microarchitecture*, pp. 248–258, Dec. 1997.

[8] K. Wang and M. Franklin, “Highly accurate data value prediction using hybrid predictors,” in *30th International Symposium on Microarchitecture*, pp. 281–290, Dec. 1997.

[9] G. Reinman and B. Calder, “Predictive techniques for aggressive load speculation,” in *31st International Symposium on Microarchitecture*, Dec. 1998.

[10] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W. Hwu, “Dynamic memory disambiguation using the memory conflict buffer,” in *Six International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183–193, Oct. 1994.

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

[12] C. Fu, M. Jennings, S. Larin, and T. Conte, “Value speculation scheduling for high performance processors,” in *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[13] C. Fu, M. Jennings, S. Larin, and T. Conte, “Software-only value speculation scheduling,” tech. rep., Department of Electrical and Computer Engineering, North Carolina State University, June 1998.

[14] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin, “Classifying load and store instructions for memory renaming,” in *International Conference on Supercomputing*, June 1999.

[15] G. Tyson and T. M. Austin, “Improving the accuracy and performance of memory communication through renaming,” in *30th Annual International Symposium on Microarchitecture*, pp. 218–227, Dec. 1997.

[16] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, “Fast, effective dynamic compilation,” in *Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pp. 149–159, ACM, May 1996.

[17] 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*, pp. 145–156, ACM, Jan. 1996.

[18] D. Engler, W. Hsieh, and M. Kaashoek, “‘C: A language for high-level efficient, and machine-independent dynamic code generation,” in *Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 131–144, ACM, Jan. 1996.

[19] T. Knoblock and E. Ruf, “Data specialization,” in *Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pp. 215–225, ACM, Jan. 1996.

[20] 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*, pp. 137–148, ACM, May 1996.

[21] T. Autrey and M. Wolfe, “Initial results for glacial variable analysis,” in *9th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.

[22] B. Calder, D. Grunwald, and B. Zorn, “Quantifying behavioral differences between C and C++ programs,” *Journal of Programming Languages*, vol. 2, no. 4, pp. 313–351, 1994.

[23] B. Calder and D. Grunwald, “Reducing indirect function call overhead in C++ programs,” in *1994 ACM Symposium on Principles of Programming Languages*, pp. 397–408, Jan. 1994.

[24] 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*, pp. 326–336, ACM, June 1994.

[25] U. Hölzle, C. Chambers, and D. Ungar, “Optimizing dynamically-types object-oriented languages with polymorhic inline caches,” in *ECOOP’91, Fourth European Conference on Object-Oriented Programming*, pp. 21–38, July 1991.

[26] 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*, pp. 93–102, ACM, June 1995.

[27] 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*, pp. 108–123, ACM, Oct. 1995.

[28] S. Richardson, “Exploiting trivial and redundant computation,” in *Proceedings of the Eleventh Symposium on Computer Arithmetic*, 1993.

[29] P. Feller, “Value profiling for instructions and memory locations,” Tech. Rep. CS98-581, Department of Computer Science and Engineering, University of California, San Diego, Apr. 1998. MS Thesis.

[30] 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*, pp. 196–205, ACM, 1994.

[31] 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*, Oct. 1997.

[32] T. Austin, D. Pnevmatikatos, and G. Sohi, “Streamlining data cache access with fast address calculation,” in *22nd Annual International Symposium on Computer Architecture*, June 1995.

[33] B. Calder, D. Grunwald, and J. Emer, “Predictive sequential associative cache,” in *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, Feb. 1996.