Parameter Value Characterization of Windows NT-based Applications

John Kalamatianos
David Kaeli

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA

Ronnie Chaiken

Advanced Development Group
Microsoft Research
Redmond, WA

Abstract

Compiler optimizations such as code specialization and partial evaluation can be used to effectively exploit identifiable invariance of variable values. To identify the invariant variables that the compiler misses at compile time, value profiling can provide valuable information.

In this paper we focus on the invariance of procedure parameters for a set of desktop applications run on MS Windows NT 4.0. Most of those applications are non-scientific and execute interactively through a rich GUI. Due to the dynamic nature of this workload one would expect that parameter values would exhibit an unpredictable behavior. Our work attempts to address this question by measuring the invariance and temporal locality of parameter values. We also measure the invariance of parameter values for four benchmarks from the SPECINT95 suite for comparison.

1 Introduction

Many compiler optimizations exploit the *invariance* commonly exhibited by variables during program execution. Some of these optimizations include: constant propagation and folding, function inlining, code specialization and partial evaluation [10]. This paper investigates the use of profile information to identify which parameters have invariant or semi-invariant behavior. The goal is to provide the compiler with an accurate picture about which parameters exhibit almost constant behavior and where in the program is the best place to apply parameter value-driven optimizations.

If a parameter is assigned either a single or a small set of values, we can characterize this behavior as invariant or semi-invariant, respectively. Since parameter passing is a commonly used method to allow procedures to communicate, the likelihood that control or data-flow within the called procedure depends on a passed parameter value is high. Furthermore, software engineering techniques encourage code modularity, increasing the need for parameter passing that affects control and data flow in the called procedure. Parameter value invariance needs to be assessed carefully before the compiler is going to attempt to perform value-driven optimizations.

Value profiling has been used extensively to expose the predictability of several values such as load target addresses [8], indirect branch targets [5, 7] and register-based values [11]. In this paper we use value profiling to expose parameter predictability and invariance. While our characterization technique is similar to the approach described by Calder et al. in [1], we only focus on procedure parameters. Our work extends Calder's by capturing data on both a Call Site and a Procedure basis, and then using specific data types as filters since not all parameters can be used in value-driven optimizations (e.g., floating-point parameters).

The rest of this paper is organized as follows. In Section 2 we describe our parameter value profiling tool while Sections 3 and 4 discuss the experimental results for invariance and temporal locality respectively. Section 5 concludes the paper and suggests future directions for our work.

2 Parameter Value Profiling Tool

Next, we describe our approach to profiling parameter values. Profile information for parameter values can be available at compile-time, link-time or run-time (assuming the application was compiled with adequate debug information). We work at the binary level (i.e., after the code has been compiled) so that we do not compete with the code generator and the optimizer.

There are three types of information that our profiling tool can provide to a value-driven optimizer: (i) the degree of invariance of the parameter over the lifetime of a program, (ii) the degree of temporal locality of a parameter during the execution of the application, and (iii) the most frequently accessed values for the given parameter. Since the procedure associated with a parameter may be called from different call sites, we also provide with the above information for either each call site or on a procedure basis where the information from all call sites is combined.

In order to determine the invariance of a parameter, we followed the same approach proposed in [1]. We computed the *Invariance-k* for every parameter where k = 1, 2, 3 and 4. Invariance-k is defined as the percentage of time a parameter referenced its k most frequent values. Thus, the higher the invariance for a smaller value of k, the better candidate a parameter makes for value-driven optimizations.

We use a 10-entry *Top N Value* (TNV) table [6] for every parameter in order to approximate its top N values. Every TNV entry contains a parameter value and an associated frequency count. Entries are sorted in decreasing order of frequency count. A TNV table consists of a 5-entry steady state portion and a 5-entry clear portion and uses a *Least Frequently Used* (LFU) policy to update/replace entries. This replacement policy purges the clear part after a specified interval. The interval is set to twice the frequency value of the middle entry of the TNV table each time the table is cleared, with a minimum value of 2000. Based on the experimental results in [6], we believe that a 10-entry TVN table is sufficient to ensure that the top 4 most-frequent values are recorded for every parameter. For more information about the TNV table and its update policies, see [6].

The temporal locality of a parameter is captured by assessing its *predictability*. We define parameter predictability to be the number of times a parameter value changes, divided by the number of times this parameter is passed. For example, if a parameter value changed its value on each call, the predictability for this parameter would be 0%. If the value changed on every other call, the predictability would be 50%. Hence, the higher the predictability the lower the entropy of the parameter contents. Predictability is one way to capture the temporal locality of parameter values. Accurate temporal information is critical in order to perform memory and cache-conscious code specialization [2] and inlining [3, 12] that will not severely increase the working set of the application or thrash the cache by forcing the specialized and the original body of a procedure to collide.

We believe that parameter invariance will identify the most frequently used values of a parameter, but invariance alone does not capture the access patterns of those values. Predictability is a useful measure to capture this missing information. In the future, we plan to gather more elaborate temporal information such as the locality of a group of parameters that are activated by different call sites close in time.

2.1 Tracing Methodology

To perform our evaluation, we collected parameter values for a set of Microsoft NT 4.0 applications and 4 SPECINT95 benchmarks. The programs were compiled with the Microsoft Visual C/C++ v6.0 compiler on a PentiumPro-based machine. All applications were compiled, instrumented and traced under the Windows NT4.0 operating system, with debug information enabled. Table 1 shows the applications, the number of traced procedure calls, the static count of activated call sites and the static count of activated procedures for every scenario. There exist two lines for Word and the Mso9.dll library (the MS Office DLL) because we ran two different scenarios for those two applications.

We use a tracing tool that can collect parameter values for every procedure in an executable/DLL. Instrumentation code is inserted at the beginning of every procedure, capturing procedure parameter values and an id that uniquely identifies the caller/callee pair. Available debug information allows the tool to keep around information on a parameter's data type and allows us to selectively trace parameters based on their types. For the experiments presented in this paper, we did not trace any floating-point parameters. If a procedure has no parameters or floating-point parameters only, it is not instrumented, thereby reducing the overall profiling overhead. After a profile was generated, we measure the parameter invariance and temporal locality. During this step, we treat pointers as booleans, having either a NULL or a non-NULL value. Their predictability is measured by counting the number of times a pointer-based parameter value changes from NULL to non-NULL and vice-versa. This was done in order to first examine whether code specialization with respect to a NULL value of a pointer can be a feasible optimization. By setting a flag in our tool we can re-evaluate the invariance and predictability of pointers modeled as multi-value entities. However, we will not address the results from those experiments in this paper. Furthermore, our statistics do not include any call sites that were activated only once during profiling.

On every experiment we instrumented a single executable (or DLL in the case of the Mso9.dll) and collected the parameter values of all the calls made to procedures lying inside that binary image. Therefore, the numbers in Table 1 do not include calls made to non-instrumented DLLs.

3 Parameter Value Invariance

In this section we present the results associated with non-pointer parameter value invariance. We classify non-pointer parameters based on their Invariance-1 and 4. Invariance-1 is used because most value-driven optimizations are highly likely to use the most frequently accessed value of a param-

Program	Proc. Dyn.Count	C.S. static count	Proc. Static Count
Word 97 (SOS)	6,701,151	24,949	5,897
Word 97 (breadth)	8,250,441	16,730	4,553
Mso9.dll (W97-SOS)	5,927,768	12,969	4,297
Mso9.dll (Access97)	9,064,113	9,669	2,738
Foxpro v6.0	137,516,865	8,125	2,268
SQLserver v7.0	17,402,025	25,161	5,470
VC/C++ v6.0 Linker	113,140,314	2,144	631
Excel 97	1,258,949	10,066	2,971
PowerPoint 97	11,363,411	18,429	5,346
Access 97	17,219,900	26,196	6,172
Gcc	13,138,648	5,860	1,013
Go	4,136,288	1,699	315
Perl	42,386,645	644	185
Ijpeg	50,651,550	476	217

Table 1. Static and dynamic procedure count and static call site count for all benchmarks.

eter. Invariance-4 is used to limit the search space. Parameters that frequently reference more than 4 values can be considered highly variant.

The classification scheme shown in Table 2 indicates how many parameters have Inv-1 and 4 in the range of 0-10% (variant parameters) and in the range of 90-100% (invariant parameters). Since these parameters lie at the opposite ends of the Invariance plot we will refer to them as *biased* throughout this paper. For each range, we list the static and dynamic count of parameters, labeled as (s/d) in Table 2. Both counts are expressed as percentages over all activated and traced parameters (including pointers). Columns 2-5 of Table 2 present the results. For example, in *Access97* 14.6% of all parameters in the program (including pointers) have Invariance-1 more than 90%. These parameters represent 7.1% of all parameter references generated during program execution. Column 6 shows how many non-pointer parameters were activated (static and dynamic count) in order to clarify the proportion of pointer parameters in the application. We can also use those percentages to find how many non-pointer parameters have Invariance-1 or 4 in the range 10-90%. For example, in *Access97*, 36.1% of all parameters have a non-pointer data type. They represent 35.6% of all parameter references in the program trace. Since 0.6% (27.0%) of non-pointer parameters have Invariance-4 in the range of 0-10% (90-100%), the remaining 8.5% have Invariance-4 in the range of 10-90%.

All statistics presented in Table 2 are gathered at the Procedure site. Therefore, the invariance statistics for a particular parameter are combined for all the call sites that called the procedure associated with this parameter.

As we can see from Table 2, the percentage of parameters having low or high Invariance-1 and 4 is very similar between Windows applications and SPECINT95 benchmarks. For all benchmarks, there are more parameters with high Invariance-1 compared to the ones with low. *Go* is the only outlier since it has more parameters with very low Invariance-1. The high variance of its parameter values is also verified in the findings of [1]. More parameters are found to be highly invariant when we move from Inv-1 to 4 (as we would expect). However, the total dynamic count of parameters with Inv-4 more than 90% does not exceed 50% of all parameter references for most of the applications. A significant number of parameters reference a large set of values.

In addition, the large static counts for highly invariant parameters may be due to parameters which are activated infrequently, probably referencing the same value. These parameters are not good candidates for value-driven optimizations. Although we do not present detailed data about the frequencies of each parameter, dynamic counts from experiments like *Word97* using the SOS scenario and *Excel* support this explanation.

Almost all applications have approximately half of their non-pointer parameters being either highly invariant or highly variant. This is particularly true for *FoxPro* and the *VC++ Linker* (*FoxPro* and the *VC++ Linker* have the highest number of dynamically executed calls among the Windows applications and thus the highest percentage of frequently referenced parameters). The respective dynamic count though is less than half of the dynamic count of all non-pointer parameters. The two exceptions to this observation are again *FoxPro* and the *VC++ Linker*, where the dynamic counts of the biased parameters are significant (approximately 22% of all parameter references and more than 50% of all non-pointer parameter references). This may be

Program	Procedure basis				Non-Pointer
	Inv-1		Inv-4		Parameters
	0-10%	90-100%	0-10%	90-100%	Total Count (%)
	(s/d)	(s/d)	(s/d)	(s/d)	(s/d)
Word97 (SOS)	2.4/6.4	20.0/7.7	0.8/1.8	32.5/18.6	41.7/49.1
Word97 (breadth)	1.4/4.3	18.3/8.8	0.3/2.2	33.1/17.3	41.5/44.4
Mso9.dll (W97-SOS)	1.4/8.1	16.4/9.6	0.2/2.1	25.7/18.5	32.8/39.6
Mso9.dll (Access97)	1.1/1.8	17.1/13.7	0.3/0.1	27.6/20.9	35.1/35.9
FoxPro v6.0	4.9/6.3	26.3/15.2	1.7/5.6	42.0/25.8	57.6/46.0
SQLserver v7.0	0.8/2.1	10.8/10.6	0.3/1.2	17.6/21.5	21.6/31.6
VC++ v6.0 Linker	3.8/15.6	13.8/7.2	2.7/11.7	19.7/9.9	30.6/29.3
PowerPoint97	1.1/1.3	10.8/5.2	0.3/0.7	15.9/8.1	20.3/22.4
Excel97	1.4/7.1	21.4/8.5	0.3/3.9	37.7/20.8	47.7/52.3
Access97	1.7/4.8	14.6/7.1	0.6/1.9	27.0/15.8	36.1/35.6
Gcc	2.4/4.9	12.5/7.5	1.0/0.7	24.4/19.6	31.6/33.0
Go	24.7/43.1	9.8/1.6	13.6/35.7	38.1/14.1	93.3/70.1
Perl	0.4/2.0	18.2/12.7	0.2/0.2	30.5/19.5	35.5/28.6
Ijpeg	1.9/4.5	20.9/3.1	1.9/4.5	27.8/4.3	35.3/52.6

Table 2. Non-pointer parameter distribution based on invariance measured on a procedure basis.

Program	Inv-1		Inv-4	
	0-10%	90-100%	0-10%	90-100%
	(s/d)	(s/d)	(s/d)	(s/d)
Word97 (SOS)	2.4/6.0	36.3/15.6	0.9/3.2	53.6/33.1
Word97 (breadth)	1.9/6.9	34.4/18.0	0.4/4.2	54.1/29.3
Mso9.dll (W97-SOS)	1.3/11.3	28.9/16.6	0.2/3.4	41.7/31.0
Mso9.dll (Access97)	0.8/2.6	29.1/23.6	0.2/0.2	43.0/34.8
FoxPro v6.0	5.3/9.2	44.0/28.4	2.1/7.8	62.5/42.3
SQLserver v7.0	1.5/2.1	20.4/18.0	0.9/0.7	30.4/29.5
VC++ v6.0 Linker	5.2/26.9	24.5/10.5	4.1/20.8	34.2/14.4
PowerPoint97	1.6/1.9	18.0/8.6	0.5/1.1	24.8/12.2
Excel97	1.5/7.3	34.0/15.6	0.3/5.0	52.3/33.6
Access97	1.6/5.8	25.2/11.3	0.6/2.7	39.9/22.4
Gcc	3.6/6.6	22.0/14.0	1.5/1.3	35.6/36.1
Go	24.4/58.5	16.2/2.4	11.1/51.5	44.4/13.4
Perl	1.1/4.3	29.7/25.3	0.5/0.0	45.4/38.3
Ijpeg	3.2/11.8	33.1/8.8	3.2/11.8	40.1/10.6

Table 3. Procedure distribution based on max Invariance of their non-pointer parameters.

an indication of the bimodal behavior of parameters in desktop applications when they are referenced frequently: they access either very few or too many values. The Go benchmark also has more than half of its parameter references being biased, but this is because most of its parameters are highly invariant. *Ijpeg*, *Perl* and *Gcc* do not seem to exhibit this type of behavior, at least when Invariance-1 is studied. Most parameters in *Gcc* and *Perl* seem to have high Invariance-1 and 4, while the opposite is found for *Ijpeg* (although only 1.9% of its parameters are highly invariant, the majority of the non-biased parameters have relatively low Invariance-1). On the other hand, the *VC++ Linker* has the largest percentage of parameter references being highly invariant from all Windows applications.

Table 3 shows the distribution of procedures based on the maximum invariance of their non-pointer parameters. If a procedure has only pointer arguments, it is not classified here, but it participates in the total static/dynamic count. In other words, all percentages are expressed over the total (static/dynamic) count of activated procedures in the application. The classification scheme remains the same: two ranges are presented (0-10% and 90-100%) for Invariance-1 and 4.

As we can see from Table 3, there is a large number of static procedures having at least one parameter with high invariance. But their contribution to the total dynamic procedure count is rather small (the largest is *FoxPro* at 28.4%). That does not necessarily mean that optimizations such as code specialization will not be beneficial. If a parameter is used inside an intensive loop, it may be possible to significantly improve performance by applying several loop optimizations initiated by the invariance of the parameter's values.

Table 4 presents the statistics, previously shown in Table 2, on a Call Site basis. The only difference lies in the computation of the static counts. Since we measure the invariance of a parameter when its associated procedure is called from a specific call site, we instantiate a parameter N, N times, with N being the number of call sites. The total static count is the sum of all parameter instances. All percentages refer to parameter instances where each parameter instance is uniquely associated with a call site. Notice, that the dynamic count remains the same.

The primary observation made from Table 4 is that the percentages of highly invariant parameter instances are higher than those of highly invariant biased parameters for all benchmarks. That means that parameter instances with high invariance are often combined to generate low invariance for their associated parameter. In other words, a call site often calls a procedure passing a single value in a parameter most of the time, but different call sites may pass different values to the same parameter. The number of highly variant parameter and parameter instances differs from benchmark to benchmark. In 3 cases (*Word97-SOS*, the *VC++ Linker*,and *Perl*) the static count percentage was increased. This corresponds to the cases where variant parameter instances combine to produce a less variant parameter. This can occur when parameter instances access individually the same large set of values.

Clearly, there is more opportunity for applying value-driven optimizations on a call site rather than on a procedure basis since the invariance of parameter instances is much higher. However, one must be careful where to optimize since call sites are much more frequent than procedures and code specialization/inlining can lead to code explosion and increase of the memory footprint of the application [9, 4, 3]. These optimization could actually degrade performance if we are not careful (due to page faults, cache and TLB thrashing, etc).

Table 5 provides data associated with call sites (similar to the data in Table 3). A Call Site is similarly classified based on the maximum invariance of the non-pointer parameters of the procedure associated with it. Again, call sites calling procedures with pointer arguments only are not classified but are included in the overall static/dynamic count.

As was the case with parameters and parameter instances, there are more call sites calling a procedure with at least one non-pointer parameter with more than 90% invariance-1 and 4. The dynamic count contributions of those call sites is also significantly higher for all benchmarks making a call site-based optimization an attractive solution, provided that call sites are carefully selected.

4 Temporal Locality of Parameter Values

In this section we present results associated with the temporal behavior of parameters and parameter instance values. Table 6 presents these results. The Table structure is similar to the invariance statistics in Tables 2 and 4. The total static/dynamic counts for non-pointer parameters and parameter instances remain exactly the same. Columns 2-3 and 5-6 show the predictability of parameters and parameter instances respectively.

As we can see from Table 6, a fair portion of the parameters in both Windows and SPECINT95 applications exhibit excellent predictability. We should be careful though to distinguish between parameters that are referenced infrequently, and may have good temporal locality, from those that actually do present good predictability. Parameter instances do have higher predictability and are definitely more biased than parameters. The former indicates that parameter instances infrequently change their value. Their associated parameter though, seems to be less predictable since the reference patterns of parameter instances overlap in time causing the parameter value to frequently change. The biasing effect is interpreted as follows: a parameter is formed

Program	Call Site basis				Non-Pointer
	Inv-1		Inv-4		Parameter Instances
	0-10%	90-100%	0-10%	90-100%	Total Count (%)
	(s/d)	(s/d)	(s/d)	(s/d)	(s/d)
Word97 (SOS)	2.5/8.1	29.2/16.3	0.9/3.3	37.7/23.2	45.4/49.1
Word97 (breadth)	1.2/5.1	27.7/17.2	0.3/2.5	37.4/22.9	45.1/44.4
Mso9.dll (W97-SOS)	0.9/5.0	25.8/15.3	0.1/2.3	31.0/21.6	35.2/39.6
Mso9.dll (Access97)	0.7/1.9	25.5/19.5	0.1/0.1	30.5/23.7	34.8/35.9
FoxPro v6.0	4.8/6.9	35.5/22.0	2.1/5.9	47.0/29.4	60.3/46.1
SQLserver v7.0	0.4/1.8	20.8/19.8	0.1/1.2	24.0/24.6	25.9/31.6
VC++ v6.0 Linker	4.5/15.0	26.2/9.6	2.9/14.1	29.5/11.6	39.2/29.3
PowerPoint97	0.8/1.6	17.5/9.2	0.2/0.8	19.9/10.9	22.9/22.4
Excel97	1.2/7.9	35.4/18.1	0.1/3.4	45.1/26.7	52.0/52.3
Access97	1.3/5.6	24.9/13.4	0.4/1.6	31.6/18.0	37.5/35.6
Gcc	2.1/3.9	24.9/17.7	0.9/2.3	30.1/23.3	35.0/33.0
Go	16.3/38.4	25.2/4.7	7.9/22.2	42.1/16.8	79.6/70.1
Perl	0.7/0.3	26.4/15.6	0.5/0.3	32.9/19.7	37.6/28.6
Ijpeg	1.6/4.5	31.7/3.1	1.2/4.5	38.3/17.7	43.6/52.6

Table 4. Non-pointer parameter instance distribution based on invariance measured on a call site basis.

by instances that have strong or poor predictability. When their reference patterns are combined in time, the poorly predictable instances are interleaved with the others to produce a parameter with predictability that lies somewhere in the middle (30-70%). The only benchmark that did not follow this rule was ijpeg. A large number of instances with medium predictability were combined to form a number of unpredictable parameters. Overall, the temporal locality of parameters should be considered during the selection of parameters/call sites/procedures where value-driven optimizations will be applied. Since specialized code will be activated based on the occurrence of a value, the temporal locality will likely define the time span during which the specialized code will be accessed and more importantly reside in the cache.

5 Conclusions

In this paper we investigated the behavior of parameter values for several applications with different characteristics. We studied parameter value behavior by measuring their value invariance and temporal locality using profiling. This information can be directly employed by the compiler/linker to guide value-driven optimizations.

We showed that there exist parameters which possess high invariance even in applications possessing a rich GUI, and which are used in highly interactive environments. Furthermore, we examined parameter entropy and found that some parameters have very high predictability. This is quite encouraging because value-driven optimizations such as code specialization and inlining need to be used carefully since they tend to increase the working set of an application and apply pressure on the memory hierarchy. We also found that invariance/predictability of parameter values are higher when measured at the call site versus measured at the procedure. Therefore, it may be worthwhile considering applying optimizations on a call site basis.

We view parameter value profiling and behavior characterization as important tools for applying value-driven compiler optimizations. A complementary approach could be to use further analysis to identify parameters that affect program control flow (e.g., participate in conditional statements) or data-flow (e.g., specify the value of the index variable of a loop). This type of analysis could decrease the tracing overhead by selecting which parameters should be instrumented. A natural extension of this work could also be a sensitivity analysis of the invariance/locality of parameter values between different input sets.

References

[1] B. Calder, P. Feller, and A. Eustace. Value Profiling. In *Proceedings of the International Symposium on Microarchitecture*, pages 259–269, December 1997.

[2] B. Calder, P. Feller, and A. Eustace. Value Profiling and Optimization. Technical Report CS98-592, University of California, San Diego, July 1998.

Program	Inv-1		Inv-4	
	0-10%	90-100%	0-10%	90-100%
	(s/d)	(s/d)	(s/d)	(s/d)
Word97 (SOS)	3.5/11.4	47.8/32.6	1.4/5.0	59.3/43.0
Word97 (breadth)	1.8/9.4	47.7/33.9	0.4/4.7	60.1/40.1
Mso9.dll (W97-SOS)	0.9/6.0	38.9/24.7	0.1/4.0	44.3/34.2
Mso9.dll (Access97)	0.7/2.2	37.8/31.1	0.1/0.2	42.9/37.1
FoxPro v6.0	5.4/8.9	53.7/38.4	2.8/7.7	65.4/43.4
SQLserver v7.0	1.0/2.0	39.0/26.1	0.7/0.8	43.3/32.5
VC++ v6.0 Linker	4.8/25.2	40.7/13.9	3.6/24.9	44.6/16.3
PowerPoint97	0.9/2.5	24.3/14.3	0.3/1.5	26.9/16.1
Excel97	1.1/9.2	46.9/28.8	0.2/4.2	55.6/39.8
Access97	1.3/6.8	36.3/20.1	0.4/2.1	42.5/25.1
Gcc	3.1/5.9	43.8/34.8	1.4/4.4	49.0/43.8
Go	23.4/50.6	24.9/5.3	10.9/27.5	40.4/14.9
Perl	2.3/0.4	42.8/31.1	1.8/0.4	51.6/38.9
Ijpeg	2.6/11.9	50.0/8.7	1.9/11.9	55.4/46.1

Table 5. Call Site distribution based on max Invariance of the non-pointer parameters of their associated procedures.

[3] P. Chang and S. M. et.al. Profile-Guided Automatic Inline Expansion for C Programs. *Software, Practice and Experience*, 22(5):349–370, May 1992.

[4] J. Davidson and A. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, 18(2):89–101, February 1992.

[5] K. Driesen and U. Holzle. Accurate Indirect Branch Prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 167–178, June 1998.

[6] P. Feller. Value Profiling for Instructions and Memory Locations. Master's thesis, University of California, San Diego, April 1998.

[7] J. Kalamatianos and D. Kaeli. Predicting Indirect Branches via Data Compression. In *Proceedings of the International Symposium on Microarchitecture*, pages 272–281, December 1998.

[8] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[9] S. McFarling. Procedure Merging with Instruction Caches. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 71–79, June 1991.

[10] S. Muchnick. *Advanced Compiler Design*. Morgan Kaufman Publishers, 1998.

[11] Y. Sazeides and J. Smith. The Predictability of Data Values. In *Proceedings of the International Symposium on Microarchitecture*, December 1997.

[12] W.Y.Chen, P.P.Chang, T.M.Conte, and W.W.Hwu. The Effect of Code Expanding Optimizations on Instruction Cache Design. *IEEE Transactions on Computers*, 42(9):1045–1057, September 1993.

Program	Procedure basis			Call Site basis		
	Predictability		Total Count	Predictability		Total Count
	90-100%	0-10%	(%)	90-100%	0-10%	(%)
	(s/d)	(s/d)	(s/d)	(s/d)	(s/d)	(s/d)
Word97 (SOS)	21.2/16.1	4.4/5.9	41.7/49.1	30.0/21.3	3.9/8.8	45.4/49.1
Word97 (breadth)	19.1/11.6	2.9/4.9	41.5/44.4	28.1/19.4	2.6/6.6	45.1/44.4
Mso9.dll (W97-SOS)	16.9/12.8	4.2/8.2	32.8/39.6	26.2/17.7	2.6/9.4	35.2/39.6
Mso9.dll (Access97)	17.2/17.5	4.9/3.8	35.1/35.9	25.6/21.5	2.5/3.0	34.8/35.9
FoxPro v6.0	28.4/19.8	7.7/4.3	57.6/46.0	37.3/26.7	6.8/6.2	60.3/46.1
SQLserver v7.0	11.5/12.3	1.6/3.7	21.6/31.6	21.1/21.1	0.9/3.2	25.9/31.6
VC++ v6.0 Linker	16.3/7.6	4.4/4.7	30.6/29.3	27.9/10.2	5.0/11.1	39.2/29.3
PowerPoint97	11.0/6.3	2.4/1.5	20.3/22.4	17.6/10.2	1.5/1.8	22.9/22.4
Excel97	22.6/11.3	4.6/10.7	47.7/52.3	36.6/21.1	2.9/10.6	52.0/52.3
Access97	16.4/12.3	2.4/3.2	36.1/35.6	25.9/18.1	1.7/3.5	37.5/35.6
Gcc	13.1/8.7	2.6/2.9	31.6/33.0	25.5/18.8	1.9/3.6	35.0/33.0
Go	10.7/6.1	24.9/21.0	93.3/70.1	26.5/6.9	12.3/25.4	79.6/70.1
Perl	17.8/8.9	1.4/2.0	35.5/28.6	26.6/15.9	1.6/2.3	37.6/28.6
Ijpeg	21.1/3.1	5.0/24.4	35.3/52.6	31.6/3.1	3.2/3.7	43.6/52.6

Table 6. Non-pointer parameter distribution based on their predictability measured either on a procedure or a call site basis.