Tuning Branch Predictors to Support Virtual Method Invocation in Java

N. Vijaykrishnan
Pennsylvania State University

N. Ranganathan
University of Texas at El Paso

Tuning Branch Predictors to Support Virtual Method Invocation in Java

N. Vijaykrishnan
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, 16802
vijay@cse.psu.edu, http://www.cse.psu.edu/˜vijay

N. Ranganathan
Department of Electrical and Computer Engineering
University of Texas at El Paso
El Paso, TX
ranganat@ece.utep.edu, http://www.ece.utep.edu/faculty/webrangan

Abstract

Java's object oriented nature along with its distributed nature make it a good choice for network computing. The use of virtual methods associated with Java's object oriented behavior requires accurate target prediction for indirect branches. This is critical to the performance of Java applications executed on deeply pipelined, wide issue processors. In this paper, we investigate the use of a path history based predictor to accurately determine the target of these virtual methods. The effect of varying the various parameters of the predictor on the misprediction rates is studied using various Java benchmarks. Results from this study show that the execution of Java code will benefit from more sophisticated branch-predictors.

1 Introduction

Java is a class-based object oriented language that is used extensively for building networked applications. Some of the features of Java such as the use of virtual methods, dynamic loading and symbolic resolution that make it suitable for developing networked software applications also slow the execution speed of Java code. In this paper, we focus on addressing the performance issues involved with the use of virtual methods in Java.

The use of virtual methods as the default method invocation mechanism results in the execution of frequent indirect branch executions. The *invokevirtual* JVM bytecode [1] that is used to perform virtual method calls constitutes 5% of the Java bytecodes executed on an average for the benchmarks shown in Table 2. Many of the current JVM implementations such as Sun's JDK interpreter and CACAO Just-in-Time Compiler[2] use a dispatch table to implement the *invokevirtual* bytecode. When a virtual method is invoked, the target address is obtained from a fixed index into the the dispatch table of the current object. Finally, an indirect branch instruction is executed to jump to the fetched target address. Thus, an indirect branch is executed for every virtual method invoked. The accurate prediction of these indirect branches is critical to the performance of Java virtual machine (JVM) implementations executing on deeply pipelined systems. Speculative execution is used in such architectures to avoid the performance loss associated with the execution of branch instructions. Accurate branch predictors are essential to avoid discarding the results of the speculative execution following a misprediction.

Current processors employ a branch target buffer (BTB) based mechanism to predict the indirect branches [12]. The misprediction rates for virtual method calls using the branch target buffer is found to range up to 27% as shown in Table 1 for the studied benchmarks. Previous researchers have successfully used path history information to improve the prediction of direct branches [8, 16]. In this paper, the path history of virtual method calls is used to predict target addresses of virtual method invocations. The path history provides the capability to distinguish between different dynamic executions of the same virtual method. In the path history based predictor, a hashing function of the path history of target addresses and the virtual method call site address is used to index a target cache. The cached entry provides the predicted target address of the virtual call.

The paper is organized as follows. Section 2 discusses the background and motivation for this work. In section 3, the path history based target address predictor is introduced. Next, the experimental strategy and benchmarks used in this study are explained in section 4. The effect of the various parameters of the path-history based target address predictor on the performance of the predictor is studied using the benchmarks in Section 5. Starting from a fully associative target buffer of unlimited size, the parameters are optimized sequentially to account for the hardware constraints such as buffer size and limited associativity. The number of history buffers, the path history length, the number of target address bits, the hashing function, and the buffer structure were the parameters varied. Concluding remarks are provided in Section 6.

2 Background

The problem of target prediction for indirect branches has been investigated for C and C++ programs. Calder and Grunwald proposed a 2-bit strategy for updating the branch target buffer (BTB) [18]. The target address entry in the BTB is updated only when two consecutive predictions at that target address are incorrect. This strategy as opposed to the default strategy of updating the entry on each misprediction was shown to improve the performance. Emer and Gloy present several single-level predictors based on a combination of the values of program counter, stack pointer, register number and stack address [19]. They performed their study on SPECint95 programs.

Previous research has shown the use of correlation information from path history to predict the execution of direct branches [8, 16]. Recently, the path history information has been used to predict indirect branches [17, 20]. Chang, Hao and Patt proposed a target cache that uses the branch history to distinguish different dynamic occurrences of each indirect branch [17]. Their study was performed on select SPECint95 programs. Their work also shows the correlation between higher misprediction rates and slower execution speed. In this paper, we make use of this observation and focus on improving misprediction rates. The object oriented programs in C++ and Java use indirect branches with a much higher frequency than in SPECint95 programs. Target address prediction for indirect branches using a suite of C++ programs and SPECint95 was performed in [20]. Their study investigated the impact of various hardware constraints on the performance of a path history based predictor. Our work uses a similar approach in investigating the indirect branch behavior of Java programs. Unlike the previous efforts, the focus of this work is confined to the target prediction of indirect branches that occur due to the virtual method invocations. The best way to improve the performance of virtual method invocations is to eliminate the virtual calls by inlining or statically binding them [14]. However, only a portion of the calls can be safely bound statically [15]. We identify Java code characteristics that enable the use of path based predictors in identifying the target of the virtual calls.

Since virtual method invocation has been identified as one of the major bottlenecks for the performance of Java code [11, 13], the impact of the various parameters of the path history predictor on prediction accuracy is investigated in this work. It was observed in [11] that the proportion of virtual methods is likely to increase due to the trend towards

Table 1: Misprediction rates using normal and 2-bit replacement strategies

Benchmark	BTB misses (%)	2-bit BTB misses (%)
Javac	4.8	3.9
Javadoc	3.5	2.4
Richards	23.4	27.1
Deltablue	1.7	1.2
Heap	2.6	2.1

A 32K direct mapped BTB was used.

fine-grained object design in Java applications. In such an environment, big objects become many smaller objects. Consequently big methods become many smaller methods. This causes many more method invocations and method invocation increasingly becomes a performance bottleneck. In [13], profiling of various Java benchmarks was done to identify virtual methods as one of the bottlenecks in Java execution. This work also investigated the receiver type locality at virtual method call sites.

Java performance studies have been performed in [9], [10] to investigate the need for architectural support for Java execution. In [10], it was concluded based on their study of Java interpreters that it may be premature to provide hardware support for Java execution. The results of this paper indicate that the micro-architectural resources such as branch predictors can be enhanced to support Java execution. However, it must be noted that the support proposed here is based on the Java language characteristics rather than just the interpreter characteristics analyzed in [10].

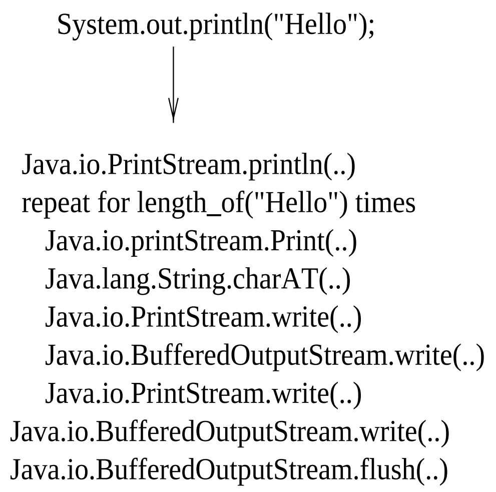
3 Path History Based Predictor

In this section, the use of path history to improve the target prediction accuracy is investigated. Path history consists of target addresses of recently executed branches. The history of target addresses provides useful correlation information that can be used to improve the branch prediction accuracy. Virtual method calls in Java programs exhibit correlation among the receiver types at call sites. This is due to the presence of correlation among consecutive call sites as shown in Figure 1. In this example, the shape object *s* invokes a series of virtual calls and the different call sites in the function drag_drop have the same receiver type. Another reason for the correlation is due to a sequence of virtual calls triggered by a single virtual method call and the presence of looping constructs as shown in Figure 2. Here, the invocation of a virtual method to print a string triggers a sequence of method invocations. The use of path history in exploiting such correlation among call sites to predict the destination of virtual calls is investigated in this paper.

```
// This is a drag drop code in GUI based programs
class mouse{
 public void drag_drop(shape &s) {
   s.invalidate_object_area();
   screen.invalidate();
   s.move(new_location);
   s.update_object_area();
   s.repaint();
   screen.update();

  } }
```

Fig 1: Correlation in receiver types among call sites



```
System.out.println("Hello");

Java.io.PrintStream.println(..)
repeat for length_of("Hello") times
   Java.io.printStream.Print(..)
   Java.lang.String.charAT(..)
   Java.io.PrintStream.write(..)
   Java.io.BufferedOutputStream.write(..)
   Java.io.PrintStream.write(..)
 Java.io.BufferedOutputStream.write(..)
 Java.io.BufferedOutputStream.flush(..)
```

Fig 2: A single virtual method invoking a series of methods

Figure 3 shows the predictor based on the use of path history information. The program counter stores the address of the virtual method call site. An indexing function of the program counter is used to access the path history information corresponding to the call site. Then a hashing function of the path history information from the history buffers and the program counter is used to form the hashing address. This address is used to index the target buffer to obtain the target address. The various parameters involved in the design of such a predictor include the number of history buffers (n), path history length (p), the number of bits of each target address registered in the history buffer (b), the hashing function, and the structure of the target buffer. The target buffer could either be tagless or a tagged buffer. In a tagless buffer, the hashing address is used to index into the buffer and no tag comparisons are involved. Hence, the mapping of two different hashing addresses in to the same target buffer location are not distinguished. In contrast, the tagged buffer has a tag associated with each entry. These tags help to distinguish different history patterns that map to the same location. The influence of these parameters on the performance of the predictor is studied in the subsequent sections.

Benchmark	Description	T1
Javac	Java compiler	215K
Javadoc	Documentation tool	274K
Richards	O.S task dispatcher [4]	1517K
Deltablue	Constraint solver [4]	12082K
Heap	Garbage collector [3]	151K

T1 - Number of virtual calls executed

Table 2: Description of Benchmarks

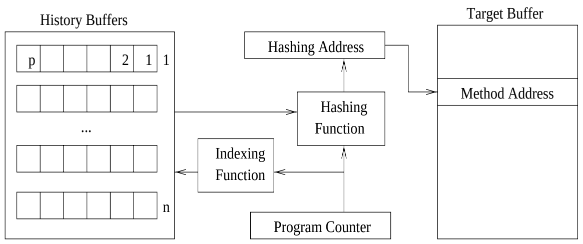


Fig 3: Path history based two level indirect branch prediction

4 Experimental Setup

The following experimental strategy was used in this study. The traces of indirect branches corresponding to the virtual method call sites were obtained through modifications to the JDK 1.0.2 source code. The benchmarks shown in Table 2 were executed using the modified JDK 1.0.2 on a Sparc-20 processor under Solaris 2.5 operating system. The *javac* and *javadoc* benchmarks are large applications with 25,400 and 28,471 lines of code respectively. The *richards* and *deltablue* benchmarks are medium size benchmarks with 410 and 984 lines of code [4]. These two benchmark were chosen as they have been used in earlier studies of polymorphic behavior of object oriented languages [5]. The *heap* benchmark is an 4495 line applet that implements incremental garbage collection.

5 Predictor Parameter Variations

In this section, the effect of varying the parameters involved in the design of the branch predictor is studied. The parameters were optimized sequentially and the following subsections report them in that order. The studied parameters include the number of history buffers (n), path history length (p), the number of bits of each target address registered in the history buffer (b), the hashing function, and the structure of the target buffer.

5.1 Number of History Buffers

The number of history buffers determines the number of virtual method call sites that share their history. When $n = 1$, all the virtual call sites share the same history buffer and the resulting predictor is called as a global history predictor. In contrast, a per-address history predictor keeps a separate history for all virtual method call sites. This is achieved when $n = 2^w$, where w is the word size. When the number of history buffers is between 1 and 2^w, a set of addresses use the same history buffer. The effect of the number of history buffers on misprediction rate was studied by using the most significant bits of the program counter to access the history buffers. To mask the effects of other parameters of the predictor, a fully associative target buffer of unlimited size was used. Further, all the bits of the target address were registered in the path history buffers and the hashing address was formed by concatenating the selected path history buffer with the program counter.

Figures 4 and 5 show the results of this investigation for *javac* and *richards* benchmarks respectively. The h most significant bits of the program counter select the history buffer corresponding to the call site. The global history predictor is simulated when $h = 0$. In contrast, $h = w$ corresponds to the per-address history predictor. It is observed from the figures that the global path history predictor performs better than those that use per-address or per-set history predictors. For example, the misprediction rates increase from 2.6% for global path history to 4.2% for the per-address scheme using *javac* with a path length of 2. This indicates that the correlation across call sites is more useful than the self history at a call site in predicting the targets. This can be ascribed to the execution of a series of virtual calls corresponding to the invocation of a single virtual call as shown earlier. A global path history can capture the effect of such constructs better than a per-address scheme. Thus, a global path history is used in refining the other parameters of the predictor.

5.2 History Path Length

The number of target addresses of the virtual methods registered in each history buffer is called as the history path length, p. When $p = 0$, the two-level path history predictor becomes a single level predictor similar to the BTB strategy. The variation in path length can help in determining whether the correlation among the target addresses of virtual methods is long-term or short-term. The effect of path length variation was studied with a fully associative target buffer of unlimited size along with a global path history. Figure 6 shows the results of this study for the different benchmarks.

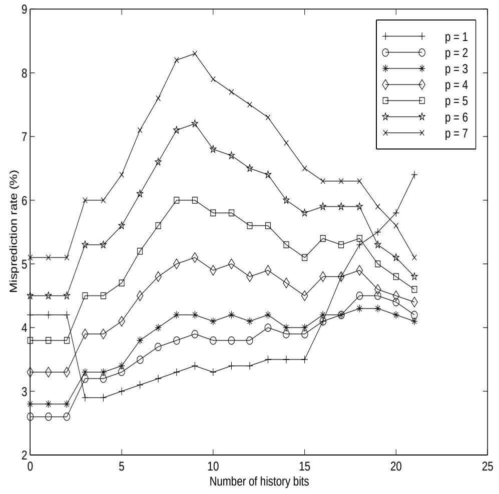


Fig 4: Variation in misprediction rate with number of history buffer sets for *javac*. A fully associative target buffer of unlimited size was used.

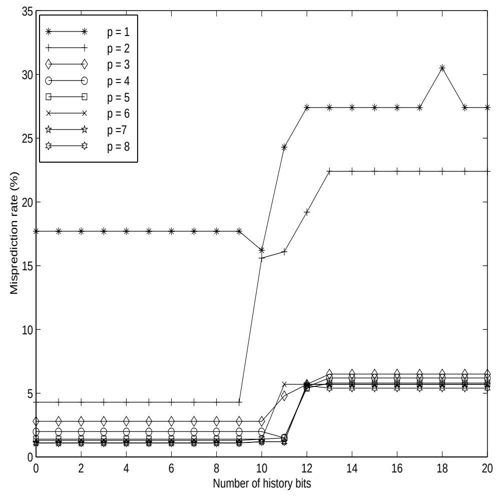


Fig 5: Variation in misprediction rate with number of history buffer sets for *richards*. A fully associative target buffer of unlimited size was used.

The path length affects the misprediction rate in two ways. Firstly, misses occur when the path length is too small to capture a long-term dependence. Secondly, longer paths take a longer time to adapt to branch behavior changes and this results in start-up misses. Thus, a longer path would capture more long term dependence but would have more start-up misses. In contrast, a shorter path fails to capture the long-term regularities in method invocation targets but adapts quickly to changes in branch behavior.

The *javac* benchmark reflects this trade-off clearly. The misprediction rate reduces from 4.6% when the path length is zero to a misprediction rate of 2.6% when the path length is two. In this phase, the effect of capturing more regularities dominates the effect of start-up misses. However, the misprediction rates increase when the path length is increased beyond two, specifically from 2.6% to 5.1% when path length is increased from two to seven. The start-up misses begin to dominate any improvement obtained by capturing virtual method history dependence longer than two. This indicates that most path history patterns used in *javac* have a relatively short period. The *javadoc* benchmark also exhibits a similar behavior.

The *heap* benchmark does not benefit from the path history information. It is observed that the branch target buffer scheme performs better than the predictor with the path history. This is due to the relatively constant target addresses at the call sites in the *heap* benchmark. Hence, the path history information only adds to the start-up misses and does not benefit from capturing any additional regularities. In contrast, the *richards* and *deltablue* benchmarks benefit from long path lengths. The misprediction rates keep decreasing as path lengths increase from 0 to 8. This shows that these benchmarks have a long-term correlation that enables the overshadowing of start-up misses associated with longer paths. These results indicate that the optimum values for the path length differ based on the benchmark.

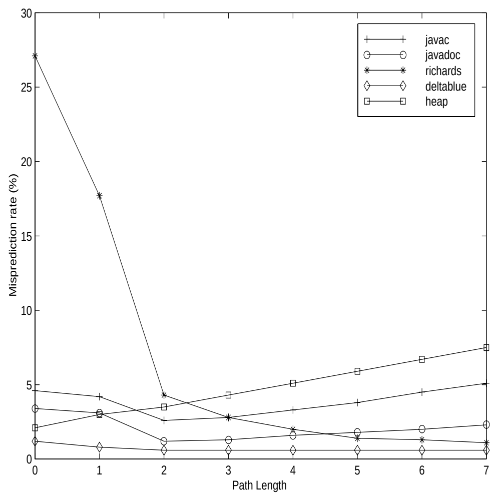


Fig 6: Variation in misprediction rate with path length using global history. A fully associative target buffer of unlimited size was used.

5.3 Path History Compression

The global history pattern along with the branch address stored in the program counter is used to index the target buffer. When all the bits of the history buffer and the program counter are used, the resulting bit pattern is long and is equal to $(p+1)*w$. The number of different path history patterns captured by this hashing address length is $2^{((p+1)*w)}$. However, most programs do not have that many patterns. Thus, the effect of varying the number of bits stored per target address stored in the history buffer on the misprediction rates was investigated. Table 3 shows the results of this investigation for the benchmarks. The least significant bits of the target addresses were used in the history patterns. It is observed that the least significant bits capture more information than the more significant bits. For *javac*, *javadoc* and *deltablue* the misprediction rates decrease when b is increased from 2 to 8 and does not change when bit size of b is increased further. This study shows that registering only the least significant bits of the target address in the history buffer could reduce the bit width of the hashing address without much loss in performance.

Table 3: Effect of history bit compression of misprediction rates

	Misses (%)			
b	Javac	Javadoc	Richards	Deltablue
2	4.7	3.4	23.4	1.6
4	3.7	2.4	25.6	1.3
6	3.3	2.0	6.0	0.8
8	2.6	1.2	6.0	0.6
10	2.6	1.2	6.0	0.6
12	2.6	1.2	6.0	0.6
32	2.6	1.2	4.3	0.6

b is the number of bits from target address used in the path history information.
A path length of two and a fully associative target buffer of unlimited size was used.

5.4 Hashing Function

The effect of the hashing function on the misprediction rates was investigated using limited size tagless target buffers. The hashing function needs to utilize both the path history and the program counter (call site) information effectively. The simplest hashing function is the concatenation scheme shown in Figure 7. Here, h bits of path history information and the program counter are concatenated to form the least significant and most significant bits of the hashing address respectively. Then, the s least significant bits of the hashing address are used to index the target buffer of size 2^s. The contents of the indexed entry provides the predicted target address.

The bit width of the path history buffer, h that constitutes the least significant bits used to index the target buffer was varied and its effect on the misprediction rate was investigated. This was performed to study the relative importance of the path history and program counter information. Figure 8 shows the results of this for an 8K entry target buffer using *javac* for different values of b. It is observed that the misprediction rate decreases, when h increases from 0 to 6. When h is increased further, the misprediction rates increase. Since the 8K target buffer is indexed using a fixed size 13-bit index, the number of bits from the program counter used in the index reduces as the value of h increases. This indicates that the call site location is relatively more important than just the path history information. The target address being primarily determined by the call site location and path history providing only additional information in the prediction accounts for this behavior. Table 4 shows the relative importance of the path history and call site location. It is observed that using only the 13-bit call site location to index the 8K target buffer, a misprediction rate of 4.9% is achieved. The misprediction rate increases to 12.9% when 12 bits of path history and 1-bit of the call site location are used for *javac*.

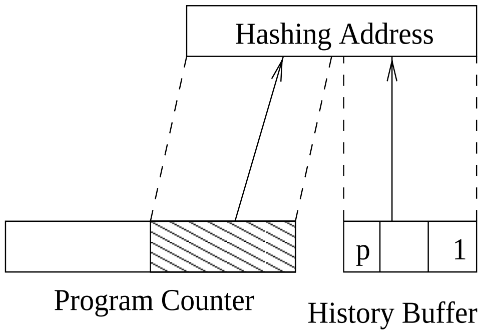


Fig 7: Tagless concatenation scheme with global path history

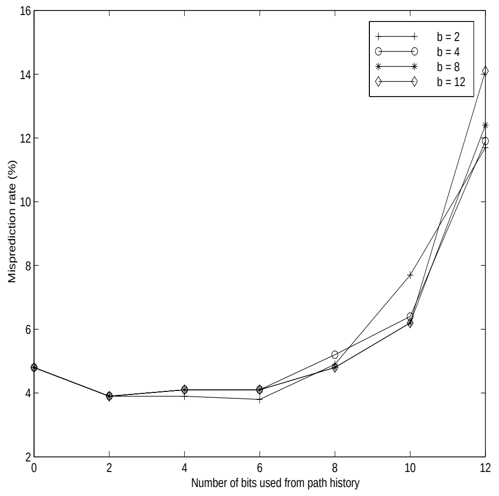


Fig 8: Variation in misprediction rate with concatenated length using tagless concatenation scheme with global path history. An 8K target buffer was used. b refers to number bits per address recorded in the history buffer

In order to utilize both the program counter and the path history bits more effectively for a fixed size target buffer, a XOR hashing scheme shown in Figure 9 was in-

Table 4: Effectiveness of hashing schemes

S	Javac Misses (%)			Richards Misses (%)		
	8K	16K	32K	8K	16K	32K
0	4.9	4.7	4.7	23.4	23.4	23.4
4	4.2	3.8	3.7	4.0	4.0	4.0
6	4.1	3.7	3.5	4.0	4.0	3.9
8	5.2	4.4	3.7	3.8	3.8	3.8
10	6.4	5.0	4.1	8.7	8.0	1.8
12	12.9	7.7	5.6	14.2	8.7	8.7
xor	3.6	3.3	3.2	2.4	2.3	2.0

An entry y in column S refers to the concatenated index formed with y bits of path history and remaining bits from program counter. All schemes register 4 bits of target address in history buffer

vestigated. The XOR hashing function helps in combining more information from the program counter and the path history bits as compared to the concatenation scheme. Here, a bitwise XOR of the program counter and the path history buffer bits is performed to obtain the hashing address. Then, the least significant bits of the hashing address are used to index the target buffer. Two replacement strategies were studied to update the target buffers using the global XOR scheme. These schemes are the same as those studied to update the BTB. It was observed that the 2-bit scheme performs better for the XOR scheme for most of the benchmarks. Figure 10 shows the results for the *javac* benchmark. The 2-bit strategy is referred to as the XOR scheme in the rest of the paper. The effectiveness of the XOR scheme as compared to the concatenation scheme is shown in Table 4. It is observed that for an 8K target buffer, the minimum misprediction rate using the concatenation scheme is 4.1% compared to the 3.6% using the XOR scheme for *javac*.

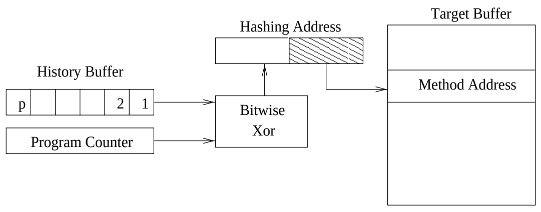


Fig 9: Tagless XOR scheme with global path history

Next, the effect of path length on the misprediction rate of the XOR scheme was investigated. In order to vary the path length,

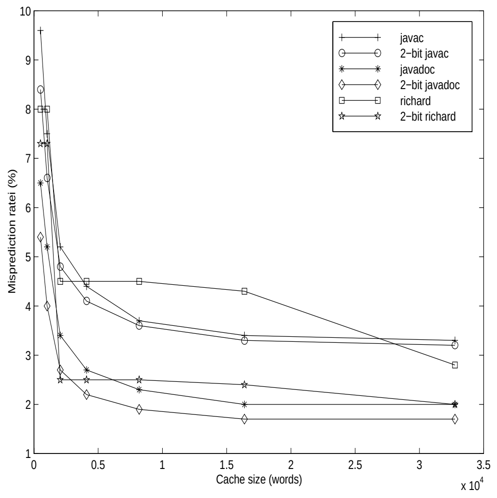


Fig 10: Comparison of normal and 2-bit update schemes using global tagless XOR. All configurations use 4-bit of method address in history buffer

the number of bits b written from each target address into the history buffer was varied. If s bits are required to index the tagless target buffer and p is the path length, b was chosen such that $b * p \leq s$. Figures 11 and 12 show the results of this study for *javac* and *richards* benchmarks respectively. The misprediction rate for *javac* exhibits a similar trend as the fully associative unconstrained target buffer size. It achieves the minimum misprediction rates for a path length of two. For the *richards* benchmark the misprediction rates are the least for a path length of two when target buffer sizes are small(0.5K to 2K). When target buffer size is increased (4K to 32K), the minimum misprediction value is achieved for a path length of three. Thus, a longer path length improves misprediction rate with an increase in the target buffer size. This is due to the greater number of bits of each target address constituting the index portion of the target buffer for a given path length.

5.5 Tagged versus Tagless Target Buffers

We also studied the impact of interference due to the presence and absence of tags with the target buffers. The XOR hashing scheme

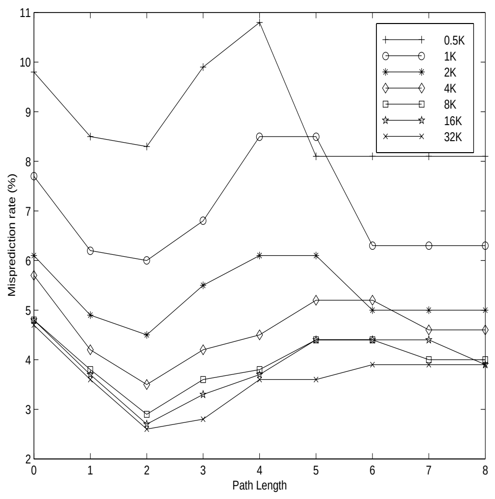


Fig 11: Variation in misprediction rate with path length for *javac* for different target buffer sizes. Uses tagless XOR scheme with global path history

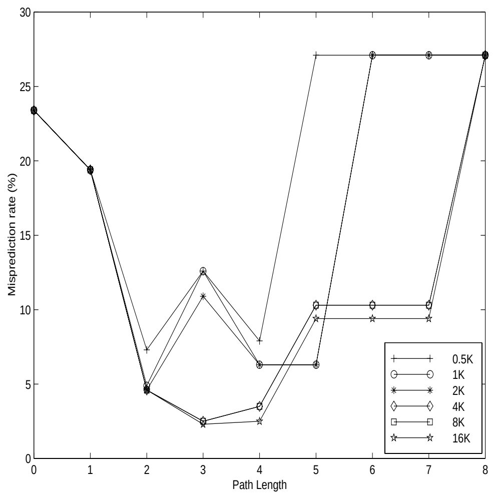


Fig 12: Variation in misprediction rate with path length for *richards* for different target buffer sizes. Uses tagless XOR scheme with global path history

was utilized in studying both the approaches. In the tagless scheme, the target address of the indirect branch is selected using the hashing address to index the target buffer. Since, no tags are associated with each target buffer entry more than one hashing address can map to the same location. Due to this interference, the target of the indirect branch is selected based on the outcome of some other branch path pattern. A positive interference occurs when the when two different patterns that map to the same target location have the same target address. Similarly, when the interference results in more misses it is called as negative interference. A tagged target buffer can be used to eliminate the effects of negative interference.

The impact of the tagged and tagless target buffers was studied for various buffer sizes by varying the associativity and the path length. Figures 13 and 14 shows the variation in misprediction rate using target buffer sizes of 2K and 4K for the tagged and tagless buffers respectively for *javac* and *richards* benchmarks. Additional entries were provided for the tagless case to account for the area overhead in maintaining tags. In these plots, an increase in the number of target address bits in history buffer corresponds to a decrease in path length. It is observed that the misprediction rates decrease with increase in associativity for the tagged buffers. Also, it is observed that there is not a significant improvement when associativity is increased beyond 8. For *javac*, it is observed that the tagless target buffer performs better than the the tagged buffers when $a = 1$ and $a = 2$ for all path lengths. It must be noted that the tagged buffers are useful only when they are able to register the alternate target address when a conflicting path history is identified. Thus, direct mapped tagged buffers perform inherently worse than the tagless buffers as they also do not benefit from positive interference. Hence, higher associativities are required in the tagged caches to benefit from the absence of negative interference.

When path lengths become large (number of target bits in history buffer becomes small), the number of different patterns generated corresponding to an indirect branch increases. Hence, a tagless buffer can benefit from the positive interference between these different patterns. Thus, it is observed that the tagless buffer performs better than the 4 and 8-way associative tagged buffers for longer path lengths for *javac*. For the *richards* benchmark the effect of positive interference is lesser, since it benefits from longer distinguishing patterns as was observed in Figure 6. Thus, the tagless target buffer performs better than only a direct mapped tagged buffer for all path lengths for the *richards* benchmark. It is also be observed that the the tagged buffers of higher associativity provide a greater improvement in prediction rates for smaller path lengths in both the benchmarks. This indicates that the conflict misses due to short term variations in targets is being reduced by the tagging mechanism.

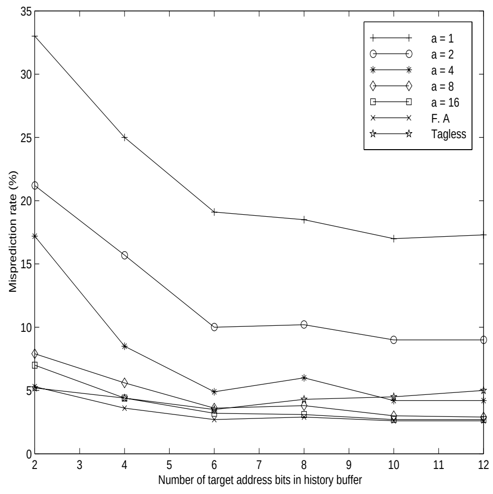


Fig 13: Comparison of tagged and tagless target buffers using *javac*. The size of the tagged and tagless target buffers were 2K and 4K respectively. The XOR hashing scheme with global path history were used for all cases.

Figures 15 and 16 show the variation in misprediction rates for the various buffer sizes. A tagged target buffer requires additional area overhead for maintaining the tags as compared to a tagless target buffer. Hence, a tagless target buffer can have more number of entries corresponding to the same implementation cost. It can be observed that an associative tagged target buffer with 8 or more entries per set outperforms the tagless buffer. It can also be observed that the increase in buffer size reduces the conflict misses significantly for the tagless and tagged buffers with associativity less than or equal to 4. The tagged buffers with associativity greater than 4 do not benefit much from increase in buffer size since the higher number of entries per set already takes care of most of the conflict misses. Due to the increase in access and cycle times associated with the higher associativities [21], the area overhead of the tags in the tagged buffers and the small difference in the misprediction rates between tagless and tagged buffers with large associativities ($a > 4$), the tagless target buffer may be a better choice in many cases.

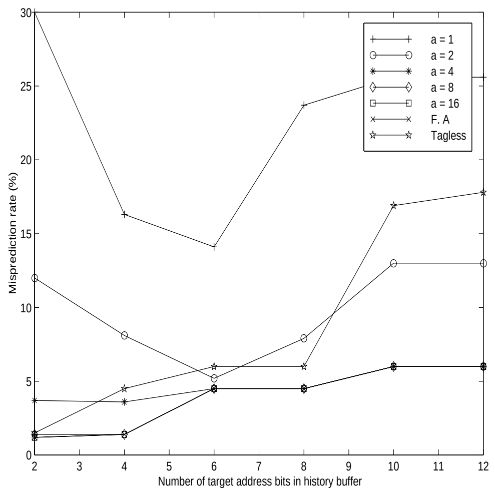


Fig 14: Comparison of tagged and tagless target buffers using *richards*. The size of the tagged and tagless target buffers were 2K and 4K respectively. The XOR hashing scheme with global path history were used for all cases.

6 Conclusion

The effectiveness of using path history to predict the target addresses of indirect branches due to virtual method invocations used in Java applications was investigated. The influence of the various parameters such as number of history buffers, path length, hashing function and the structure of the target buffers on the misprediction rates was investigated. The XOR hashing scheme with a global path history and a 2-bit update pol-

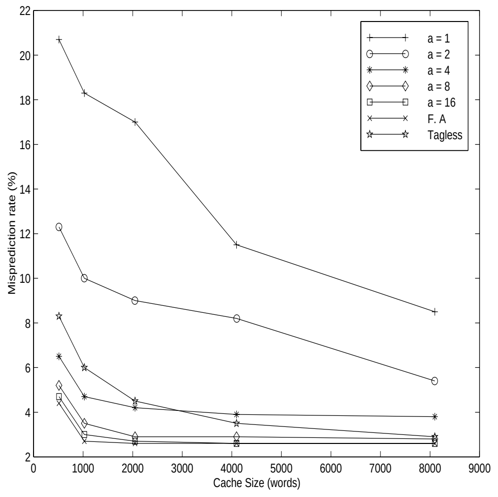


Fig 15: Comparison of tagged and tagless target buffers using *javac* with variation in target buffer size. A path length of 2 was used.

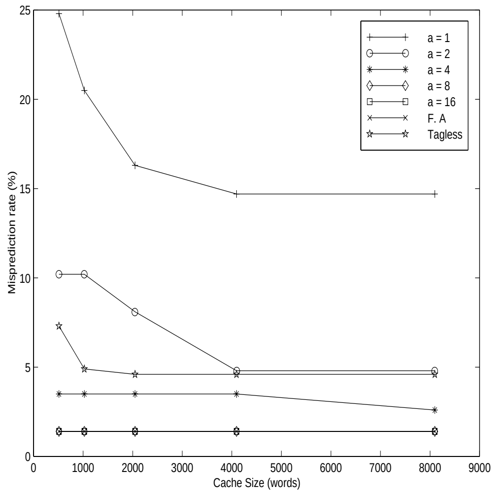


Fig 16: Comparison of tagged and tagless target buffers using *richards* with variation in target buffer size. A path length of 4 was used.

icy performed the best for almost all configurations. Also, it was found that the tagless target buffers achieve a prediction rate as good as the tagged buffers without suffering from the area overhead for tags and the increased access times associated with the associative buffers. Using the branch target buffer based predictor with an 8K buffer, misprediction rates of 4.9% and 23.4% were obtained for the *javac* and *richards* benchmarks respectively. The misprediction rates reduce to 3.6% and 2.4% for the two benchmarks using the proposed path history based predictor. The results show that the design of microarchitectural features such as the branch predictor will influence the execution speed of Java code.

References

[1] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.

[2] A. Krall and R. Grafl, "CACAO - a 64 bit JavaVM just-in-time compiler", Concurrency: Practice and Experience, 9(11):1017-1030, 1997.

[3] B. Venners, "Under the hood: Java's garbage-collected heap", http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html.

[4] M. Wolckzo, "Benchmarking Java with Richards and DeltaBlue", Sun Microsystems. http://www.sunlabs.com/people/mario/java_benchmarking/index.html

[5] U. Holzle, C. Chambers, and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches" Proceedings of ECOOP '91.

[6] J. E. Smith, "A study of branch prediction strategies", Proc. 8th Annual Intl Symposium on Computer Architecture, pp. 135-148, 1981.

[7] T. Yeh and Y. N. Patt, " Two-level adaptive branch prediction", Proc. of the 24th ACM/IEEE Intl Symposium on Microarchitecture, pp 51-61, 1991.

[8] R. Nair, "Dynamic path-based branch correlation", Proc. of the

28th ACM/IEEE Intl Symposium on Microarchitecture, pp 15-23, 1995.

[9] C. A. Hsieh et. al., " A study of cache and branch performance issues with running Java on current hardware platforms", Proc. of COMPCON, Feb 1997, pp. 211-216.

[10] T. H. Romer et. al., "The Structure and Performance of Interpreters", Proceedings of ASPLOS VII, 1996, pp. 150-159.

[11] D. Griswold, "Breaking the speed barrier: the future of Java performance", JavaOne Worldwide Java Developer Conference, 1997.

[12] T. R. Halfhill, Intel's P6, Byte Magazine, April 1995. http://www.byte.com/art/9504/sec7/art1.htm

[13] N. Vijaykrishnan, N. Ranganathan and R. Gadekarla, "Object-Oriented architectural support for a Java processor architecture", Proc. of the 12th European Conference on Object-Oriented Programming, July 1998.

[14] J. A. Dean, Whole-Program optimization of object-oriented languages, Ph.D Thesis, University of Washington, 1996.

[15] J. Vitek, "Compact dispatch tables for dynamically typed programming languages", Object Applications, ed. D. Tsichitzis, University of Geneva, Centre Universitaire d'Informatique, Aug. 1996.

[16] C. Young, N. Gloy and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction", Proc. of the 22nd Annual Intl Symposium on Computer Architecture, June 1995.

[17] P. Y. Chang, E. Hao and Y. Patt, "Target prediction for indirect jumps", Proc. of the 24th Annual Intl Symposium on Computer Architecture, 1997, pp. 274-283.

[18] B. Calder and D. Grunwald, "Reducing indirect function call overhead in C++ programs", Proc. of the 6th Intl Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[19] J. Emer and N. Gloy, "A language for describing predictors and its application to automatic synthesis", Proc. of the 24th Annual Intl Symposium on Computer Architecture, July 1997.

[20] K. Dreisen and U. Holzle, "Accurate indirect branch prediction", Proc. of the 25th Annual Intl Symposium on Computer Architecture, pp. 167-178, June 1998.

[21] N. P. Jouppi and S. J. E. Wilton, "An enhanced access and cycle time model for on-chip caches", DEC- WRL Technical Report, 93.5, July 1994.