The Role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism

Shiwen Hu, Ravi Bhargava and Lizy Kurian John
Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin
{hushiwen, ravib, ljohn}@ece.utexas.edu

Abstract

This work studies the performance impact of return value prediction in a system that supports speculative method-level parallelism (SMLP). A SMLP system creates a speculative thread at each method call. This allows the method and the code from which it is called to be executed in parallel. To improve performance, the return values of methods are predicted in hardware so that no method has to wait for its sub-method to complete before continuing to execute. We find that two-thirds of return values need to be predicted, and perfect return value prediction improves performance by an average of 44% over no return value prediction. However, the performance of realistic predictors is limited by poor prediction accuracy on integer return values and unfavorable SMLP conditions. A new Parameter Stride (PS) predictor is proposed to overcome the deficiencies of the standard predictors by predicting based on method arguments. Combining the PS predictor with previous predictors results in an average 7% speedup versus a system with hybrid return value prediction and 21% speedup versus no return value prediction.

1. Introduction

Much of the recent performance improvement in microprocessors is the result of increasing and exploiting instruction-level parallelism (ILP). However, in general-purpose applications, the exploitable ILP is strongly limited by data and control dependencies. To further boost performance, different types of parallelism must be identified and exploited.

Speculative thread-level parallelism is one such promising approach where sequential programs are dynamically split into threads that execute simultaneously [5,13,14]. These threads are speculatively issued before the dependencies with previous threads are resolved, and provide a coarser granularity for applying parallel execution. One of the most popular points for spawning speculative threads is at methods.

In speculative method-level parallelism (SMLP) architectures [1,9,16], the original thread executes the called method while a new speculative thread is spawned to execute the code that follows the method's return. Inter-method data dependency violations are infrequent, especially in object-oriented programming languages such as Java. This property, as well as the lack of inter-method control dependencies, makes method-based thread generation a popular strategy for creating speculative thread-level parallelism.

A speculative thread often encounters a return value from a procedure that has not completed. To avoid a rollback, the return value can be predicted if it is not available at the time of use. In previous literature [1,9,16], method-level speculation is performed using simple prediction schemes, such as last value or stride prediction. However, there are conflicting observations on the importance of return value prediction [9,16].

The initial goal of the work is to further understand the importance of return value prediction on speculative method-level parallelism, which is clarified by the runtime characteristics and performance results obtained in this work. For instance, perfect return value prediction reduces execution time by 44% versus a system with no return value. Our other contributions and observations include:

- There are plentiful opportunities for return value prediction. Two-thirds of the dynamically encountered methods return values, of which 94% of the values are consumed within 10 instructions.
- Current predictors' poor accuracy on integer return values partly accounts for the performance gap between perfect and realistic return value predictors. Using a common value predictor, boolean return values are most predictable (86%), while the integer return values are the least predictable (18%).
- The SMLP environment also affects the prediction accuracy of current return value prediction schemes. Updating a global return value predictor in a SMLP environment can lead to long delays in predictor updates, which can be speculative and/or out of order.
- A new return value prediction scheme, the *Parameter Stride (PS)* prediction, is proposed. Performing the PS prediction overcomes some of the SMLP environment obstacles and achieves an average 7% speedup versus the best previous method for an 8-CPU system.

The rest of this paper is organized as follows. In Section 2, we provide background on speculative method-level parallelism and return value prediction. The simulation environment and the Java benchmarks are described in Section 3. Return values are characterized in Section 4. The Parameter Stride predictor is presented and analyzed in Section 5. Previous efforts are discussed in Section 6, and we conclude in Section 7.

2. SMLP and Return Value Prediction

In this section, we review the concept of speculative method-level parallelism and the role of return value prediction.

2.1. Speculative Method-Level Parallelism

In a sequential execution architecture, a method is executed by the same thread that calls it. After the thread finishes executing the method, it continues executing from the original call point. This process is illustrated in Figure 1a, in which method `metA` is invoked in method `main`. When a method call is encountered in a SMLP architecture, the original thread executes the method as before. At the same time, a new speculative thread is spawned to execute the code beyond the method call (Figure 1b and 1c). By introducing an additional post-method speculative thread that can be executed on a separate processor, a SMLP architecture can exploit parallelism that cannot be uncovered by typical superscalar microarchitectures.

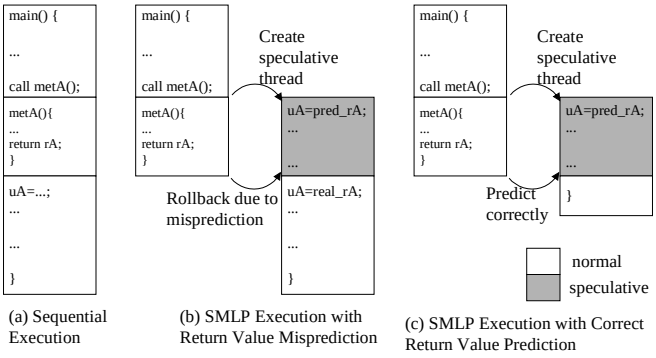

Figure 1. Comparison of Sequential and SMLP Execution Models

2.2. Role of Return Value Prediction

To expose more method-level parallelism, one of the most important techniques is return value prediction [1,9,16]. When a return value usage is encountered in a speculative thread, a predicted value is used if the real return value is not available. Typically, the speculative thread consumes the return value soon after the thread is produced by the original thread. Therefore the original thread will not have enough time to generate the return value for the speculative thread and the predicted return value is used in most cases (further explored in Section 4).

The speculative thread continues to execute while the original thread completes the method and computes the corresponding return value. When a method completes, the correct and predicted return values are compared. If the return value is mispredicted, the speculative thread must rollback to either the beginning of the thread or the position where the return value is first used. In either case, a mispredicted return value leads to wasted resources and increased computing time. Figure 1b and 1c illustrate the difference between a return value that is incorrectly predicted and one that is correctly predicted. For the non-void methods, the full benefits of method-level speculation can only be realized by accurate return value prediction.

3. Simulation Methodology

This section describes the simulation environment utilized to evaluate the effects of return value prediction on speculative method-level parallelism.

3.1. Simulation Environment

The LaTTe JVM [17] executes the Java programs by an advanced JIT compiler, which is modified to insert annotated code into the compiled native code of Java methods. Those markers indicate events such as method invocations, methods returns, parameter values, return values, and uses of the return values.

The JVM is functionally executed and simulated using Sun's Shade analysis tool [2]. When the annotated methods execute, the customizable Shade analyzer recognizes a method's execution by a pair of invocation/return markers and extracts instructions and other annotated events within the markers. JVM-specific operations, such as class loading and garbage collection, are excluded from our analysis. By doing so, we focus on the characteristics of programs, instead of one specific JVM. The Shade analyzer then feeds a summarized account of the program's execution to the SMLP simulator.

3.2. Benchmarks

Table 1 presents the evaluated benchmarks from SPEC JVM98 [18], which contains a group of representative general-purpose Java applications. While a method in a C program can return multiple values (e.g. a structure), a Java method returns at most one value. This is one of the features that make Java programs more attractive for our research. However, Java and C programs have been shown to have similar characteristics with regards to speculative method-level parallelism [16]. Hence, the results of the paper are not limited to Java programs.

Table 1. Benchmarks and Runtime Characteristics

Name	Dynamic Instructions	Static Methods	Dynamic Method Calls	Avg. Instr. / Method
compress	3310M	205	1.76M	1883
db	393M	216	46.4K	8461
javac	826M	495	172K	4790
jess	750M	559	273K	2743
mpegaudio	968M	304	343K	2826
mtrt	592M	264	638K	927

3.3. SMLP Execution Model

In our model, speculative method-level parallelism is supported by a chip multiprocessor (CMP), which uses simple scalar processor cores [5,13]. In the execution model, each memory access and inter-thread communication take one cycle to complete. This design choice places the focus on the interactions between inherent method-level parallelism and return value prediction. Previous literature on SMLP [1,9,16] shows that such simplification does not compromise the accuracy of their study. The other features of the considered SMLP execution model are:

- Each method invocation initiates a new speculative thread. When all processors are occupied by other tasks, the new tasks wait until there are free processors and are issued in sequential order.
- A fixed 100-cycle overhead is applied to speculative thread management tasks, such as thread creation, thread completion, and rollbacks.
- Inter-method memory dependencies are maintained and available values are forwarded to the consumer threads. A 4096-entry, two-delta stride load value predictor predicts unavailable load values [3,7].
- All threads must commit in sequential order, and data dependences are checked when a thread tries to commit. If a thread has used a mispredicted return or load value, the thread rolls back to where the value is first used. All threads created by this thread are terminated.

Various return value predictors are used throughout the analysis. By default, the stride return value predictor is 1024 entries and uses the two-delta strategy [3]. The context return value predictor uses a 1024-entry value history table and 4096-entry value prediction table [11]. The hybrid return value predictor consists of a stride and a context predictor [15]. For return value prediction, low-latency value prediction is not a big issue since the predicted result is not used until a speculative thread is created, which takes one hundred cycles.

4. Characterization of Return Values

In this section, the characteristics of Java methods and their return values are analyzed, and the predictability of return values in a SMLP environment is studied.

4.1. Runtime Method Characteristics

Table 1 shows the runtime method characteristics for the suite of Java programs. The number of static methods is interesting because it provides an estimate for the table size required for one-level predictors (e.g. a stride return value predictor). Also relevant to this study is the instructions per method invocation, which indicates the granularity of the thread size. On average, a dynamic method is called every 3600 instructions.

In method-level speculation, one of the pending problems is to find the balance between method-level parallelism and thread overhead. For example, frequent invocations of short methods may hurt performance due to the thread management overhead [16]. LaTTe alleviates the problem by dynamically inlining suitable virtual methods. Inlining benefits SMLP architectures by reducing the number of short methods. Thread management overheads become more tolerable under these circumstances.

4.2. Using Return Values

In a SMLP environment, not all methods need return value prediction. Void methods do not need return value prediction because they do not return a value. For a non-void method, if its return value is never used, then return value prediction is not necessary for the methods. Figure 2 sorts all dynamically invoked methods into three categories based on the return values: used, unused, and void return values. For most applications, at least half of the methods return a value that is used, and on average 66% of dynamic methods return a used value. These methods can potentially benefit from return value prediction. Unused method returns are the smallest among the three categories, ranging from 0.1% for `mpeg` to 11% for `db`.

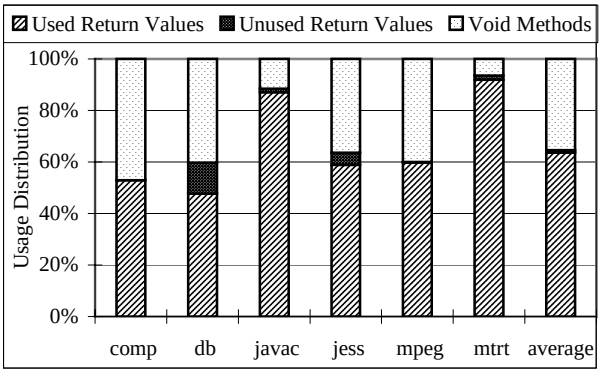

Figure 2. Usage Distributions of Return Values (sequential execution**)**

A used return value does not always necessitate a return value prediction. If the number of instructions between the method return and the first use of the return value is large enough, then it is possible that the return

value is produced before the speculative thread uses it. Such a method does not need return value prediction. Figure 3 presents the percentages of used return values that occur within 10 and 100 instructions of the method invocations. On average, 94% of return values are used within the first 10 instructions, and 98% are used within the first 100 instructions. These are very short distances considering the number of instructions per dynamic method invocation (presented in Table 1). This result confirms that return value prediction will be useful for most methods that return a value.

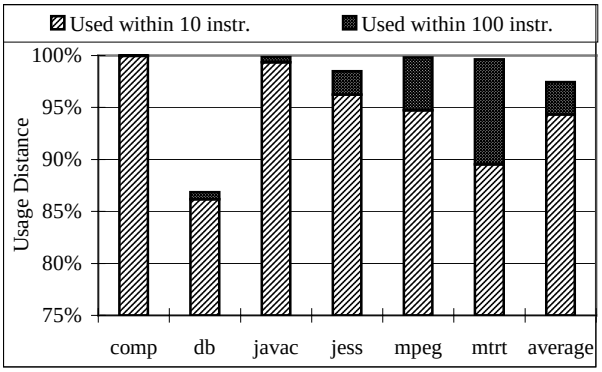

Figure 3. Percentages of Used Return Values that Occur Within 10 and 100 Instructions of the Method Invocations (sequential execution)

4.3. Return Type Breakdown

The Java language uses different bytecodes to indicate the return type of a method. Since the SPEC JVM98 benchmarks are integer programs, the most common return types are void, boolean, reference and integer (Figure 4). Reference return values are memory addresses that indicate object fields or array elements in the heap. On average, half of the return values are of type integer, and 34% method invocations return no values. Reference and boolean return values are less frequent, and both represent less than 10% of all the return values.

Individual programs display different return type characteristics. The program `javac` has the largest percentage of integer return values since it translates Java source code into bytecodes, which are represented by integer return values. In contrast with the other benchmarks, `comp` and `mpeg` have almost no reference return values, which confirms with the previous observation [12] that both programs have little heap activity. Finally, `jess` and `mpeg` have the largest percentage of boolean return values.

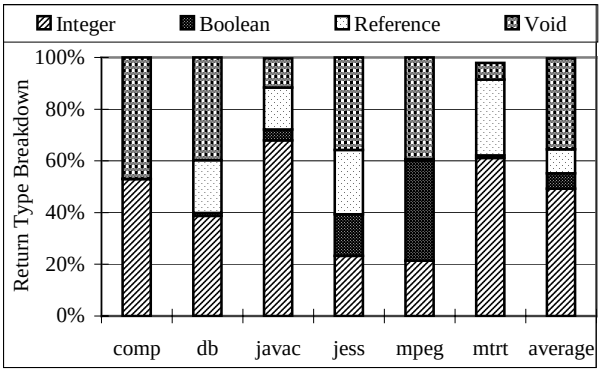

Figure 4. Return Type Breakdown (sequential execution)

Figure 5 shows the prediction accuracies categorized by the return types for a stride return value predictor during sequential execution. For all applications, boolean values are the easiest to predict, with an average prediction accuracy of 86%. This is primarily because they have only two possible values. On reference return values, four out of six benchmarks have prediction accuracies above 60%. Most reference return values of those applications point to a few frequently accessed fields [12]. On average, the prediction accuracy for reference return values is 61%, which is much higher than an average prediction accuracy of 18% for integer return values.

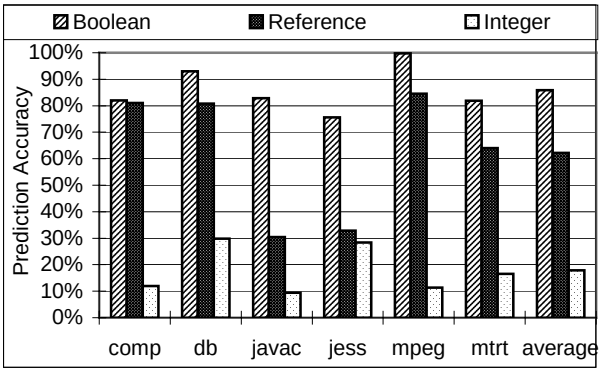

Figure 5. Return Value Prediction Accuracies by Return Types (sequential execution; stride return value predictor)

4.4. Impact of SMLP Execution

Figure 6 shows the prediction accuracies of return value predictors under both sequential and SMLP execution. These results are presented to show the effects that speculative, out-of-order return value predictor updates have on prediction accuracy.

When the simulation environment changes from sequential execution to SMLP execution, the accuracies of all predictors drop, but the context-based predictor is most sensitive to SMLP execution. The average prediction accuracy for the stride predictor is 36% for

sequential execution, but falls to 30% for SMLP execution. The average prediction accuracies of the context and the hybrid predictors drop 17% and 9% respectively.

Differences in the return value predictor update patterns account for the drop in accuracy. In the SMLP environment, stride and context-based predictors suffer for two reasons. The first reason is that the predictors depend on detecting a regular pattern. While a predictable pattern of values may exist in sequential execution, if the predictor updates occur out of order and speculatively, the patterns are less likely to exist. Likewise, patterns that are detected in a SMLP environment might not actually exist. The second reason these predictors suffer is because their prediction is based on the return values currently stored in the predictors. Even if the proper pattern is established for a method, if the most recently *captured* value is not the most recently *executed* value, then the prediction will be incorrect.

While it may seem counterproductive to update the return value predictor speculatively and out of order, our experiments show that performing in-order, non-speculative return value predictor updates during thread commitment actually leads to worse performance due to the long delays between updates which result in stale predictor values.

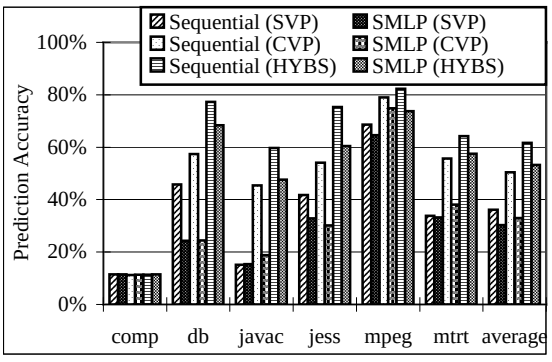

Figure 6. Prediction Accuracy For Sequential and SMLP Environments (8-CPU SMLP; SVP is stride predictor, CVP is context predictor, HYBS is hybrid predictor)

5. Parameter Stride Prediction

The value predictors studied thus far are skewed by SMLP execution. However, in return value prediction, we have an additional valuable input - method parameters (i.e. arguments). This work proposes to improve prediction accuracy by utilizing one relationship, the *parameter stride* (PS), between method parameters and return values. This relationship is not targeted or detected by any of the previous return value predictors discussed.

5.1. Detecting the Parameter Stride

The PS relationship exists when a method's return value equals the sum of one parameter value and a constant value. For example, a method may do some operations on the elements of an array, which is referenced as a parameter of the method. If the method ultimately returns the reference to the next array element, the return value will be the initial reference value plus the size of one array element. Another example is a method that operates on an object, with the object reference as one of the parameters and the return value. In such a case, the stride is zero.

5.2. Implementation

Figure 7 shows the organization of the parameter stride predictor. It is comprised of two tables: the parameter selection table (PST) and the parameter prediction table (PPT). The PST is used to detect the parameter stride pattern for new methods. Once a pattern is detected the method is placed in the PPT, which makes the return value predictions.

The PST is a small, fully associative table. To store all the parameters and strides, PST entries have more fields than PPT entries. The '#' field of the PST stores the number of parameters in the method, and a PST entry stores at most eight parameter and stride pairs.

The PPT is 4-way set associative. A PPT set is indexed by the lower bits of a method address, and then verified by comparing the higher bits of the method address with the tags of the chosen set. The state '`s`' of PPT indicates the current state, i.e. PS pattern detected or not, of a method. The '`cs`' field of PPT stores either the parameter stride or the last value, depending on the state. In the case that all entries are occupied, both tables use the LRU algorithm to choose which entry to be replaced by the incoming method.

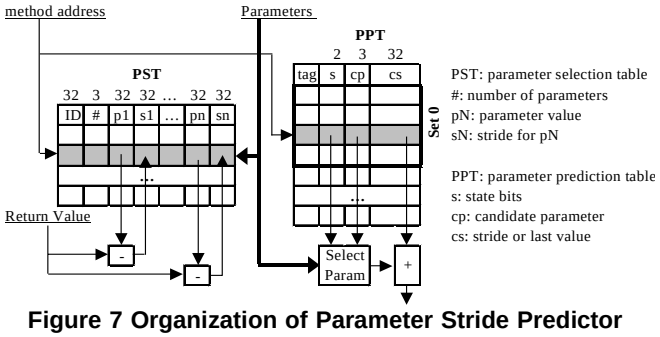

Figure 7 Organization of Parameter Stride Predictor (Bold lines indicate multiple values)

The predictor requires at least two invocations of the same method before it can provide a prediction. When a method is first encountered, a PPT entry is allocated for the method, and its state bits are set to indicate the detection state of the method. A PST entry is also allocated and the method's parameters are stored in

fields p1 through pn. When the method's return value is available, the strides between the return value and parameters are computed and stored in the same PST entry in fields s1 through sn.

When the method is invoked the second time, its current parameters overwrite the old ones in the PST entry. After the method computes its return value, the new strides are computed and compared with the old strides. If one stride pair has the same value, the method possesses the parameter stride pattern. When a parameter stride is detected, the corresponding parameter number and stride value are stored in the PPT entry, and the PST entry is freed. To predict, the appropriate parameter value is selected by the cp field of the PPT entry, then added to the stride value to obtain the predicted return value. If no parameter stride pattern is detected for a method, simple last value prediction is used instead. The last value is stored in the cs field.

Updating the PS predictor is fairly simple. Once the parameter stride pattern is detected for a method, its PPT entry needs no further updates since our experiments indicate that methods will always keep that pattern. This stability reduces the effect of speculative updates on the PS predictor. For those methods that do not possess the parameter stride pattern, their cs fields are updated by their most recent return values. Other details of the predictor are presented in [6].

5.3. Prediction coverage

Table 2 provides the percentages of static methods (Static) as well as dynamic return values (Dynamic) that are correctly predicted by the parameter stride pattern. In addition, the table shows the percentage of overall return values that are correctly predicted by the PS prediction but not by all other predictors (Uncovered). On average, about 13% of dynamically encountered return values exhibit a predictable parameter stride pattern, and 47% of these return values can not be predicted by other types of return value predictors.

Table 2. Percentage of Methods and Return Values that Possess the Parameter Stride Pattern (sequential execution)

	comp	db	javac	jess	mpeg	mtrt	average
Static	13.5%	12.0%	7.1%	7.0%	15.0%	12.5%	11.2%
Dynamic	11.0%	24.5%	3.2%	10.2%	13.0%	14.6%	12.8%
Uncovered	0.1%	14.7%	2.9%	8.2%	5.4%	4.5%	6.0%

Two properties of the parameter-stride pattern make it useful for return value prediction. First, the PS pattern is very stable. Our experiments show that once the PS relationship is established for a method, it rarely changes. Second, once the relationship is established, PS prediction does not rely on the update order. These attributes allow PS prediction to overcome some difficulties that SMLP imposes on other predictors.

5.4. Performance Impact

Figure 8 shows the impact of return value prediction on speedup for an 8-CPU SMLP system over the baseline 1-CPU sequential system. For each program, the normalized speedup is presented for four different return value prediction scenarios: no return value prediction, traditional hybrid prediction (HYBS), hybrid prediction with parameter stride prediction (HYBS-PS), and perfect return value predictor. HYBS-PS is the hybrid of HYBS and a PS predictor with 8-entry PST and 512-entry PPT.

The first observation is that realistic return value prediction improves the SMLP performance. On average, the HYBS predictor improves performance by 14% versus no return value prediction. The program comp shows the smallest improvement among all programs, which is mostly due to the poor prediction accuracy (Figure 6). Combining PS prediction with previously studied predictors improves the performance of the SMLP environment. For the HYBS-PS predictor, the average speedup versus no return value prediction is 21%, and 7% versus the HYBS predictor. For three of the benchmarks (jess, mpeg and mtrt), the HYBS-PS predictor provides more than 10% performance improvement over the hybrid predictor.

On average, perfect return value prediction achieves a 44% speedup over no return value prediction. The performance gap between perfect and realistic return value prediction is caused by realistic predictors' poor accuracy on integer return values and the unfavorable SMLP conditions. It also shows that SMLP execution can still benefit significantly from more accurate return value prediction.

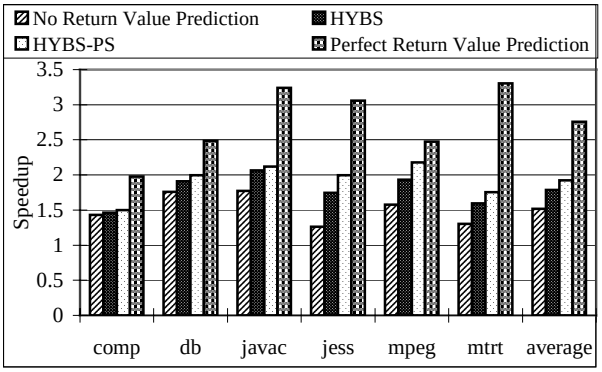

Figure 8. Normalized Speedups For Different Return Value Prediction Schemes (8-CPU SMLP execution**)**

Note that higher return value prediction accuracy may not mean higher performance. For instance, a correctly predicted speculative method can still be

rolled back if the method that calls it is rolled back due to a failed return value prediction. Furthermore, the performance improvement is affected by the available speculative method-level parallelism, which varies by programs.

6. Related Work

Recent research has focused on method-level speculation, but provide limited discussion on the issue of return value prediction. Chen and Olukotun [1] demonstrate that the Java virtual machine is an effective environment for exploiting speculative method-level parallelism. Although a simple return value prediction is incorporated in their simulator, there is no specific discussion about its effects. Oplinger et al [9] observe that employing return value prediction schemes, such as last value and stride prediction, leads to significant speedups over no return value prediction. Warg and Stenstrom [16] compare the speedups achieved under perfect, stride, and no return value prediction for a group of C and Java programs. The performance improvements gained by perfect return value predictions for the Java programs are generally very small. A notable difference in their simulation environment is that all Java programs are compiled to native code by a less sophisticated GCC Java compiler and executed without a JVM. As a result, the Java methods are compiled without inlining, and are much smaller than those in our simulation environment. Consequently, the abundance of thread management overhead in their simulation environment impairs the performance improved by accurate return value prediction.

Marcuello et al. analyze a value prediction technique specifically for architectures with thread-level parallelism [8]. They propose an *increment predictor* that predicts the thread output value of a register as the thread input value of the same register plus a fixed increment. Although the increment concept is similar to the parameter stride of this work, the PS pattern may exist between different registers and hence cannot be uncovered by the increment predictor. For instance, in SPARC processors, the registers held a method's parameters always differ with the register held the method's return value. Hence, the increment predictor cannot be used for PS prediction in SPARC machines.

Gumaraju and Franklin study the effects of a single-program, multi-threaded environment on branch prediction [4]. They similarly observe that the multithreading affects the branch history and decreases the branch prediction accuracy. However, the thread-correlation branch prediction scheme that they propose will not help return value prediction in a SMLP environment.

Rychlik and Shen briefly discuss the locality of method return values [10]. They observe the difference in argument and return values between successive invocations of methods. Instead of looking for relationships between arguments and return values, they are searching for repetition of values.

7. Conclusion

In this work, we characterize method return values in Java programs and discuss the role of return value prediction in a system that supports speculative method-level parallelism. The study is done using general-purpose Java programs running on an advanced Java virtual machine with an aggressive JIT compiler.

We find that return value prediction has the potential to greatly improve performance, and we identify possible characteristics that can be exploited by return value predictors. Two-thirds of dynamically encountered methods return a value. Of those, boolean return values are the most predictable (prediction accuracy of 86%). Integer return values are the least predictable (18%) and prove to be a big challenge for SMLP systems.

Further analysis into the behavior of the return value predictors reveals that the SMLP environment creates performance-related problems. Predictor updates are done speculatively and potentially out of sequential order. Stride and context-based strategies are inherently sensitive to this change in update behavior, hurting their performance in a SMLP environment. Therefore, we propose a Parameter Stride return value predictor designed specifically to cope with the SMLP behavior. The PS predictor makes predictions based on method argument values and a fixed stride that, once computed, rarely changes. The PS predictor complements the previous predictors, increasing performance by 7% versus hybrid return value prediction and by 21% over a system with no return value prediction.

Acknowledgements

We would like to thank the anonymous reviewers for their suggestions that provided more depth to this work. This research is partially supported by the National Science Foundation under grant number 0113105, and by AMD, Intel, IBM, Tivoli and Microsoft Corporations.

References

[1] M. Chen and K. Olukotun, "Exploiting Method-Level Parallelism in Single-Threaded Java Programs", in *Proc. PACT 1998*, 1998, pp. 176–184.

[2] R. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", in Proc.SIGMETRIC 1994, 1994, pp. 128–137.

[3] F. Gabbay and A. Mendelson, “Speculative execution based on value prediction”, Technical Report 1080, Technion – Israel Institute of Technology, 1997.

[4] J. Gummaraju, and M. Franklin, “Branch prediction in multi-threaded processors”, in Proc. PACT 2000, 2000, pp. 179-188.

[5] L. Hammond, B. Hubbert, etc, “The Stanford Hydra CMP”, in IEEE Micro, V.20 No.2, Mar. 2000, pp. 71–84.

[6] S. Hu, R. Bhargava and L. John, “The Role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism”, TR-020822-02, University of Texas at Austin, 2002.

[7] M. Lipasti, C. Wilkerson and J. Shen, “Value Locality and Load Value Prediction”, in Proc. ASPLOS’7, 1996, pp. 138–147.

[8] P. Marcuello, J. Tubella, and A. González, “Value Prediction for Speculative Multithreaded Architectures”, in Proc. MICRO’32, 1999, pp. 230–236.

[9] J. Oplinger, D. Heine, and M. Lam, “In Search of Speculative Thread-Level Parallelism”, in Proc. PACT 1999, 1999, pp. 303–313.

[10] B. Rychlik and J. P. Shen, “Characterization of Value Locality in Java Programs”, in IEEE/WWC’3, 2000, pp. 12–23.

[11] Y. Sazeides and J. E. Smith, “The Predictability of Data Values”, in Proc. MICRO’30, 1997, pp. 248–258.

[12] Y. Shuf, M. Serrano, etc, “Characterizing the Memory Behavior of Java Workloads: A Structured View and opportunities for Optimizations”, In Proc. SIGMETRIC 2001, 2001, pp.194 - 205

[13] G. Sohi, S. Breach and T. Vijaykumar, “Multiscalar Processors”, in Proc. ISCA’22, 1995, pp. 414–425.

[14] J. Steffan and T. Mowry, “The Potential for Using Thread-level Data Speculation to Facilitate Automatic Parallelization”, in Proc.HPCA’4, 1998, pp. 2–13.

[15] K. Wang and M. Franklin. “Highly Accurate Data Value Prediction Using Hybrid Predictors”, in Proc. ISCA’30, 1997, pp. 281–290.

[16] F. Warg and P. Stenstrom, “Limits on Speculative Module-level Parallelism in Imperative and Objective-oriented Programs on CMP Platforms”, in Proc. PACT 2001, 2001, pp. 221–230.

[17] B. Yang, S. Moon, etc, “LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation”, in Proc. PACT 1999, 1999, pp. 128–138.

[18] SPEC JVM98 Benchmarks, at http://www.spec.org/osg/jvm98.