Journal of Instruction-Level Parallelism 5 (2003) 1-21 Submitted 09/03; published 11/03

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

Shiwen Hu HUSHIWEN@ECE.UTEXAS.EDU
Ravi Bhargava RAVIB@ECE.UTEXAS.EDU
Lizy Kurian John LJOHN@ECE.UTEXAS.EDU
Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin
1 University Station C0803
Austin, TX 78712-0240

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, allowing 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. For Java programs, we find that two-thirds of methods have a non-void return type, and perfect return value prediction improves performance by an average of 44% compared with a system with no return value prediction. However, the performance of realistic predictors is limited by poor prediction accuracy on integer return values and undesirable update characteristics due to the SMLP environment. A Parameter Stride (PS) return value predictor is proposed to address some of the deficiencies of the standard predictors by predicting based on method arguments. Combining the PS predictor with previous predictors results in 7% speedup on average versus a system with hybrid return value prediction, and 21% speedup versus a system with 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 limited by data and control dependencies. To further boost performance, different types of parallelism can be identified and exploited.

Speculative thread-level parallelism (STLP) is an approach where sequential programs are dynamically split into threads that execute simultaneously [1,7,12,14,25,26]. These threads are speculatively issued before dependencies with previous threads are resolved, providing a coarser granularity for applying parallel execution. One of the most popular points for spawning speculative threads is at method call boundaries.

In speculative method-level parallelism (SMLP) architectures [4, 20, 28], 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 and C++. 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. One goal of this work is to clarify the importance of return value prediction for a system with speculative method-level parallelism. In previous literature [4,19,20,28], 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 [20,28].

The initial goal of the paper is to further understand the importance of return value prediction on speculative method-level parallelism. The importance of return value prediction for SMLP is clarified by the runtime characteristics and performance results obtained in our experiments. For instance, perfect return value prediction reduces execution time by 44% versus a system with no return value for the presented SPEC JVM98 Java programs. Our other contributions and observations include:

- There are many opportunities for return value prediction to impact performance. Two-thirds of the dynamically encountered methods return values, of which 94% of the values are consumed within 10 instructions.
- The poor accuracy of current return value predictors on integer return values is partly responsible 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 execution environment degrades the prediction accuracy of current return value prediction schemes because updating a global return value predictor requires long delays, and can take place speculatively, out of order, or both.
- A new return value prediction scheme, Parameter Stride (PS) prediction, is proposed. Performing the PS prediction overcomes some of the update issues in the SMLP execution environment 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 a thread completes the execution of a 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 as shown in Figure 1b. 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.

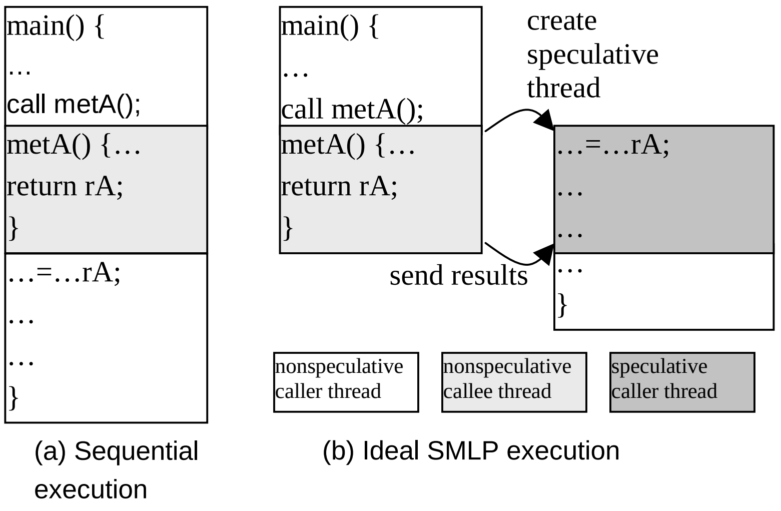

Figure 1. Comparison of sequential and SMLP execution models

Besides methods, loops are another source for speculative thread-level parallelism. In loop-based thread-level parallelism, multiple iterations of a loop body are transferred into multiple speculative threads. Speedup is achieved by overlapping execution of loop bodies on different processors. Since SMLP and loop-based thread-level parallelism exploit parallelism in different ways, a STLP system can make use of both to achieve high performance [20]. However, dynamically balancing the benefits of loop thread creation with the thread overheads can be tricky. Although loop boundaries for thread generation are easily identifiable, loop-carried data and control dependencies can limit the available loop-level parallelism.

2.2. Role of Return Value Prediction

Return value prediction is an important technique for exposing more method-level parallelism [4,19,20,28]. For non-void methods, the full benefits of method-level speculation can only be realized by accurate return value prediction. Typically, the original thread does not have enough time to produce the actual return value before the speculative thread encounters a use of the return value. Therefore, the common case is for a predicted value to be used instead (further explored in Section 4).

Figure 2a and 2b illustrate the difference between a return value that is incorrectly predicted and one that is correctly predicted. In both cases, the speculative thread executes using the predicted return value 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.

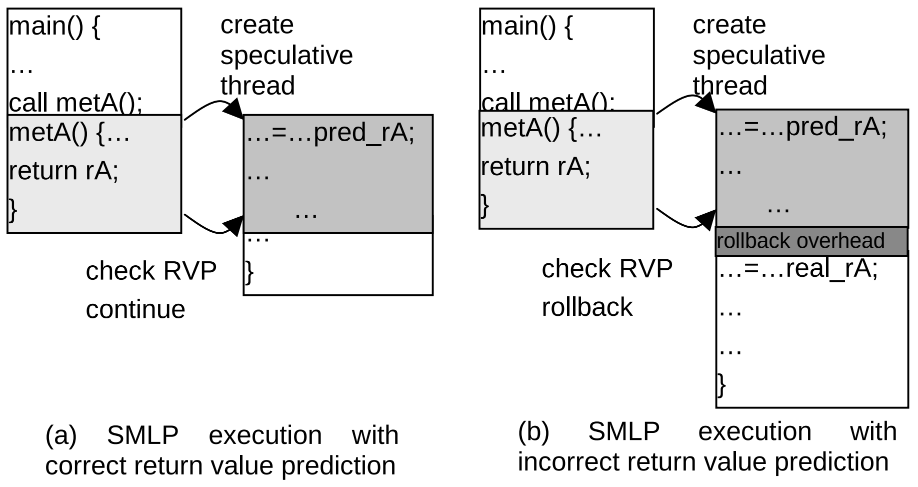

Figure 2. Comparison of SMLP execution with correct and incorrect return value predictions

2.3. Return Value Predictors

There are several mechanisms for value prediction, and all of them can be used for return value prediction with little change. Value prediction is typically used to predict the value of an instruction's operand. In an SMLP environment, values are associated with methods instead of a specific instruction. Figure 3 illustrates some of the value predictors that have been adapted for return value prediction. The address of the method's first instruction is the target of a method call and is therefore used to index the return value predictor tables. Other indices (e.g. method PC hashed with a local or global history) can also be used for the return value predictors. However, since Java workloads normally have only several hundreds methods (Table 1) and aliasings between methods rarely occur in large tables, this simple indexing scheme works pretty well and rarely hurts the accuracy of return value predictors.

The *last value predictor (LVP)* uses the most recent value produced as the predicted value [16,17]. A *stride value predictor (SVP)* assumes that the stride between two consecutive values is constant [9,10]. In this work, we use the two-delta version [8,22] of the stride predictor. A *context-based value predictor (CVP)* links a value to a context (an ordered sequence of recent values) and predicts this value when the same context occurs again [22,23]. It is also capable of capturing constant and stride patterns to some degree, but its learning phase is longer than last value predictor or stride predictor.

Hybrid predictors obtain good value prediction accuracy by combining multiple value prediction schemes that exploit different data value locality patterns [3,27]. The most commonly used prediction scheme in the literature is a hybrid scheme with a context-based predictor and two-delta stride predictor (HYBS), which we also analyze in this work. To predict a method's return value, an arbiter of the HYBS predictor first chooses one of the component predictors with higher accuracy on the method. The predicted value of the selected component predictor is used as the HYBS prediction. When the HYBS predictor is updated, both of its component predictors are updated. Then the arbiter is updated by the prediction accuracy of the two component predictors [3]. The HYBS predictor combines the abilities of both stride and context-based predictors. It can recognize repeated patterns without fixed pattern, and predict values with fixed strides as well.

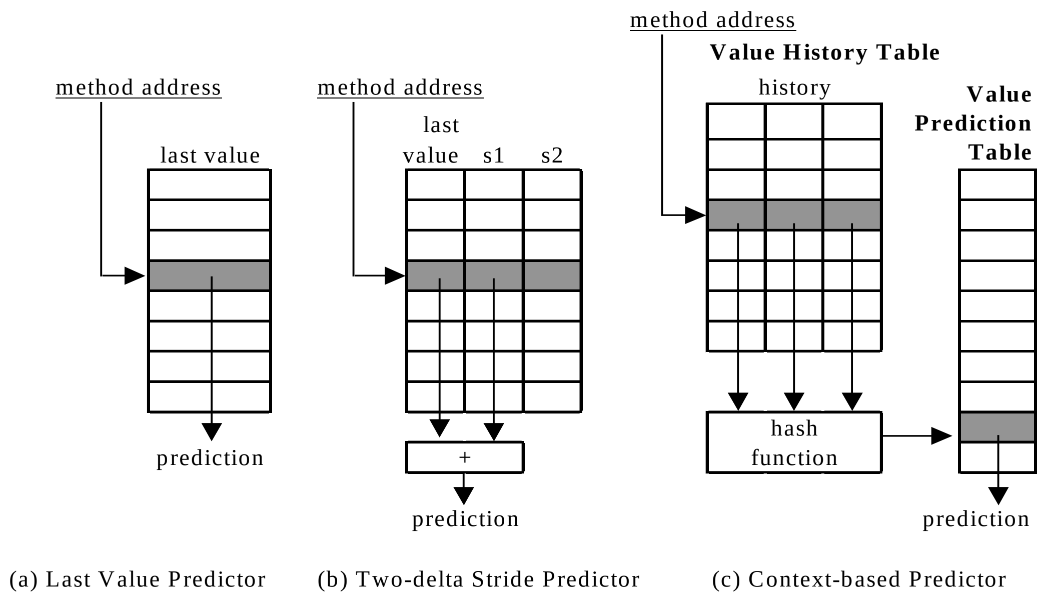

Figure 3. Return value predictors

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 [29] executes the Java programs using 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. This allows us to execute unmodified Java bytecodes in a real JVM. Compared with other schemes, such as off-line compilation and execution of Java programs [28], this simulation environment preserves realistic and typical execution features of a Java program.

The JVM is functionally executed using Sun's Shade analysis tool [6]. 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 so that the focus of the characterization is on the programs instead of the JVM. The Shade analyzer then feeds a summarized account of the program's execution to the SMLP simulator (described in Section 3.3).

3.2. Benchmark Programs

Table 1 presents the six evaluated benchmark programs from the SPEC JVM98 [30] benchmark suite, which contains a group of representative general-purpose Java applications. The input data sizes of the benchmarks are all set to S1 [30]. 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 attractive for this research. However, Java and C programs have been shown to have similar characteristics with regards to speculative method-level parallelism [28]. The program *jack* is omitted because it produces a large amount of JVM exceptions, which our annotation framework cannot currently handle. The SPEC JVM98 program *check* examines various JVM features to ensure the correct execution of SPEC JVM98 benchmarks. Since *check* is not used to test performance, it is omitted also in our experiments.

Name	Description
comp	Modified Lempel-Ziv method (LZW) to compress and decompress large files
db	Performs multiple database functions on a memory resident database
javac	The JDK 1.0.2 Java compiler compiling 225,000 lines of code
jess	Java expert shell system based on NASA’s CLIPS expert system
mpeg	Decoder to decompress MPEG audio file
mtrt	Dual-threaded ray tracer

Table 1. SPEC JVM98 workloads

3.3. SMLP Execution Model

In our model, speculative method-level parallelism is supported by a chip multiprocessor (CMP), which uses simple processor cores [7,12,14,25]. In the execution model, each operation, 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 [4,19,20,28] shows that such simplification does not compromise the accuracy of their study.

In a SMLP system, inter-method memory dependency violations may occur due to the method-level out-of-order execution nature of SMLP environment. For instance, a method may modify a global variable, which is then consumed by the second method. In a SMLP system, the second method may execute and consume the variable before the first method updates the variable, resulting in a memory dependence violation. Existing CMP proposals alleviate the problem by provide hardware support to detect and forward global values as soon as they are generated [12,14]. When the desired data are not available in other processors, they are predicted. The performance impact of inter-method data dependency is further discussed in Section 5.5.

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.
- 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 [8,22].
- 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 value or load value, the thread rolls back to where the value is first used. In addition, all threads created by this thread are terminated.

- A fixed 100-cycle overhead is applied to the following speculative thread management tasks: thread creation, thread completion, and rollbacks. These overheads are also used in [28].

Several return value predictors are used throughout the analysis. By default, the stride return value predictor is 1024 entries and uses the two-delta strategy [8,22]. The context return value predictor uses a 1024-entry value history table and a 4096-entry value prediction table [22,23]. The hybrid return value predictor consists of a stride and a context predictor [3,27]. Low-latency return value prediction is not a requirement for high performance since the prediction latency is overlapped with thread creation, which takes one hundred cycles.

4. Characterization of Return Values

This section analyzes the characteristics of Java methods and their return values, and studies the predictability of return values in a SMLP environment.

4.1. Runtime Method Characteristics

Table 2 shows the runtime method characteristics for the suite of Java programs. The table presents the number of dynamic instructions simulated, the number of static methods encountered, the number of dynamic method invocations, and the average number of dynamic instructions per invocation. The number of static methods 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.

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

Table 2. Runtime characteristics of SPEC JVM98 benchmarks

In method-level speculation, one of the pending problems is finding 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. LaTTe alleviates the problem by dynamically inlining suitable virtual methods. Therefore the average number of dynamic instructions per Java method is much larger than previously reported [28]. Inlining benefits SMLP architectures by reducing the number of short methods and increasing the granularity of the remaining methods. Thread management overheads become more tolerable under these circumstances.

4.2. Using Return Values

In this section, the presence and usefulness of return values are investigated. In a SMLP environment, not all methods need return value prediction. For example, void methods do not need return value prediction because they do not return a value. For non-void methods, if the return value is never used, then return value prediction is not necessary for these methods.

Figure 4 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. Looking at individual benchmarks, 44% of the methods in *db* return a used value while 90% of the methods in *mtrt* return a used value. This suggests that return value prediction's potential for improving performance is smaller for *db* than *mtrt*. Unused method returns are the smallest among the three categories, ranging from 0.1% for *mpeg* to 11% for *db*.

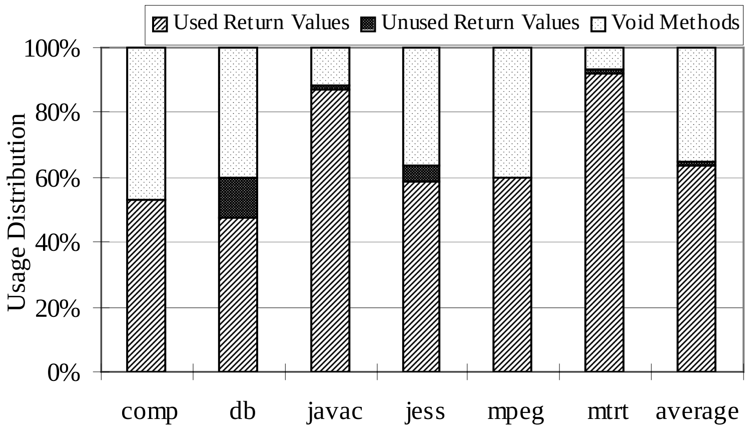

Figure 4. Usage distributions of return values (sequential execution)

A used return value does not always necessitate a return value prediction. If the time 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 by the method before the speculative thread uses it. Such a method does not need return value prediction. Figure 5 presents the percentages of used return values that occur within 10 and 100 instructions (which take one cycle each to execute) 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 2). This result confirms that return value prediction will be useful for most methods that return a value.

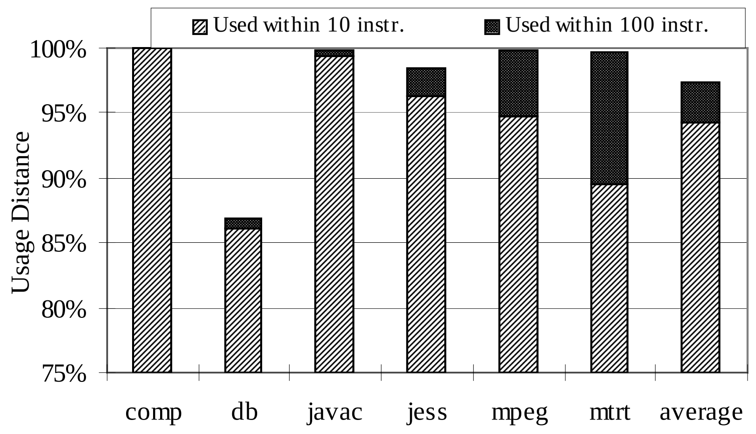

Figure 5. Percentages of used return values that occur within 10 and 100 instructions of the method invocations (sequential execution)

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. Reference return values are memory addresses that indicate object fields or array elements in the heap. As shown in Figure 6, on average, 49% of the return values are of type integer, and 34% of method invocations return no values. Reference and boolean return values are less frequent, and each represents less than 10% of all the return values.

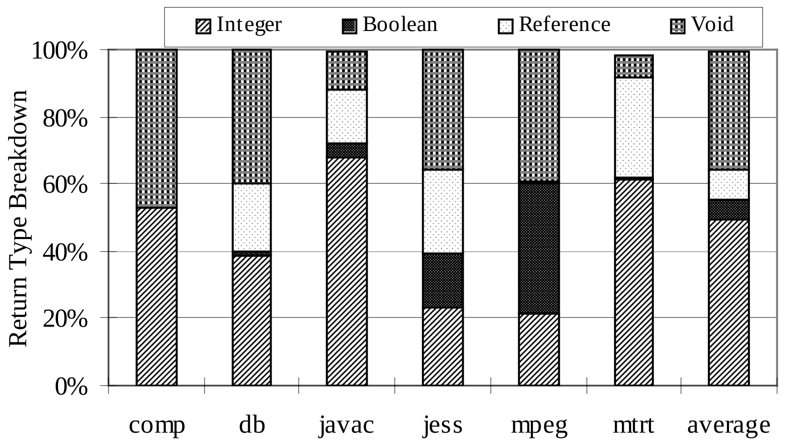

Figure 6. Return type breakdown (sequential execution)

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 coincides with the previous observation [24] that both programs have little heap activity. Finally, *jess* and *mpeg* have the largest percentage of boolean return values.

Figure 7 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 boolean return values have only two possible values. The program *mpeg* has near perfect prediction accuracy on boolean return values, which implies that those return values in *mpeg* are highly biased.

On reference return values, four out of six benchmarks have prediction accuracies above 60%, and the two benchmarks with very few reference return values (*comp* and *mpeg*) are among them. Most reference return values of those applications point to a few frequently accessed fields [24]. On the other hand, lack of hot fields in *javac* accounts for its poor prediction accuracy on reference return values [24]. 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.

Predicting integer return values proves to be most difficult. On average, 18% of integer return values are correctly predicted and only 30% at the most. This is unfortunate since integer return values are the predominant in four of the six Java programs. The prediction accuracy results obtained using other styles of value prediction exhibit similar trends to the stride return value predictor.

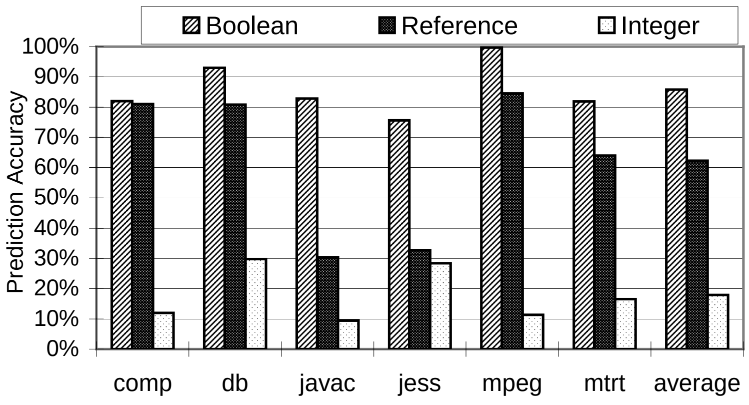

Figure 7. Return value prediction accuracies by return types (sequential execution; stride return value predictor)

4.4. Impact of SMLP Execution

Figure 8 illustrates the prediction accuracies of return value predictors for both sequential and SMLP execution. The configurations for the predictors are the same as the previous subsection. Each workload has two results, one for sequential execution and the other for SMLP execution. Although there is no need for return value prediction in sequential execution, these results are presented to show the effects that immediate, in-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, and the context-based predictor is most sensitive to the SMLP execution environment. The average prediction accuracy for the stride predictor is 36% during sequential execution, but falls to 30% during SMLP execution. For the context-based predictor, the average prediction accuracy is 50% for sequential execution, but only 33% for SMLP execution. The hybrid predictor has the highest prediction accuracy, and its average prediction accuracy drops from 62% for sequential execution to 53% for SMLP execution.

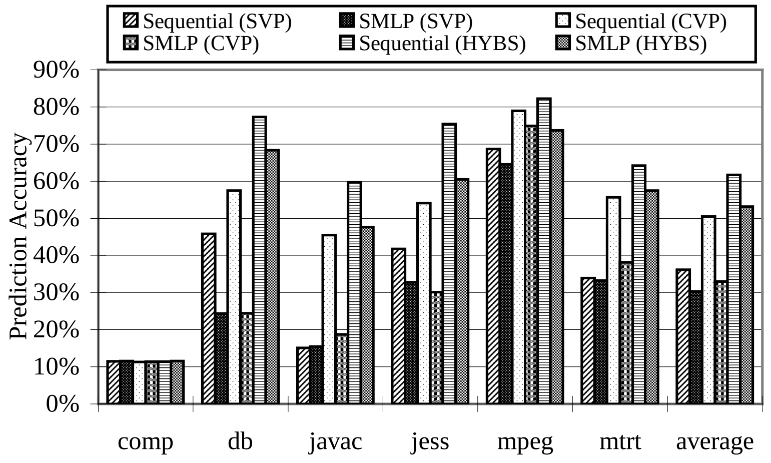

Figure 8. Prediction accuracy for sequential and SMLP environments (8-CPU SMLP; SVP is stride predictor, CVP is context predictor, HYBS is hybrid predictor)

Due to the method-level out-of-order execution of the SMLP environment, methods do not finish in the same order as in a sequential environment, hurting performance. Since return value predictors are updated when methods complete execution (instead of at commit) the predictors are update out of order. Furthermore, methods that may be squashed later are executed speculatively and therefore update the return value predictor speculatively, leading to differences in the return value predictor update patterns compared to a sequential environment.

In the SMLP environment, the change in the return value predictor update patterns affect the prediction accuracy. 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 observed by a predictor in a SMLP environment might not actually be indicative of future behavior. These SMLP predictors also suffer because their predictions are based on the return values currently stored in the predictors. If multiple dynamic instances of a method are invoked out of order, they may retrieve the wrong pattern-based return value prediction.

The results in Figure 8 demonstrate the effects of speculative, out-of-order return value predictor updates on prediction accuracy. 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.

5. Parameter Stride Prediction

The value prediction accuracy of the studied prediction strategies studied deteriorates in a SMLP execution environment. However, in return value prediction, an additional valuable input exists, the method parameters (i.e. arguments). This work proposes to improve prediction accuracy by leveraging the relationship between method parameters and the return values. This parameter stride (PS) relationship is not targeted or detected by any of the previous return value predictors discussed.

Two properties of the parameter-stride pattern make it useful for return value prediction. First, the PS pattern is 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.

5.1. The Parameter Stride Pattern

The parameter stride relationship exists when a method's return value equals the sum of one parameter value and a constant value (i.e. the stride). Table 3 presents some methods that show a constant parameter stride for each of the studied SPEC JVM98 programs. An example of both a zero stride and a non-zero stride are shown.

For example, the method *digit(char c, int radix)* of class *java/lang/Character* is used in *mtrt*. The method returns the numeric value of the character 'c' in the specified radix. Since the difference between the numeric value of the character and the number it represents is constant, this method has a parameter stride pattern. Another example is a method (such as the method *append* of the class *java/lang/StringBuffer*) 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.

Program	**Class**	**Method**	**Number of Parameters**	**Parameter Stride**	**Number of Invocations**
comp	spec/benchmarks/_201_compress/Input_Buffer	readbytes	2	0	103015
	sun/io/CharToByteISO8859_1	flush	3	-8192	68
db	java/lang/StringBuffer	append	1	0	6970
	sun/io/CharToByteISO8859_1	flush	3	-8192	986
javac	java/lang/StringBuffer	append	1	0	2789
	java/lang/String	indexOf	2	-1	89
jess	java/lang/StringBuffer	append	1	0	9001
	spec/benchmarks/_202_jess/jess/NodeTerm	CallNode	2	-2	289
mpeg	java/lang/StringBuffer	append	1	0	559
	spec/benchmarks/_222_mpegaudio/q	l	2	64	2808
mtrt	spec/benchmarks/_205_raytrace/Point	Add	1	0	46696
	java/lang/Character	digit	2	-48	1422

Table 3. Methods that possess the parameter stride pattern

Figure 9 analyzes the proportions of PS methods with zero strides. For each benchmark, two result bars are presented. The first bar indicates the percentage of distinct zero-stride methods over all PS methods, while the second bar indicates the percentage of invocations of those methods over all PS method invocations. For most programs, the proportions of dynamic invocations of zero-stride method surpass those of static methods. This indicates that on average, zero-stride methods are more frequently invoked than non-zero stride methods. This observation does not hold for two benchmarks, *mpeg* and *mtrt*. For *mpeg*, although 30% static PS methods have zero strides, they are rarely invoked. Hence, zero-stride methods account for less than 1% of the total PS method invocations in *mpeg*.

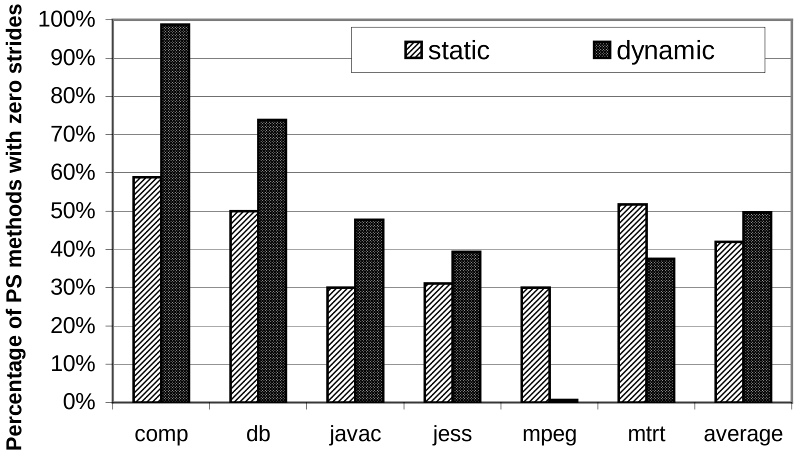

Figure 9. Proportion of the PS methods with zero strides (sequential execution)

5.2. Predictor Design

Figure 10 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.

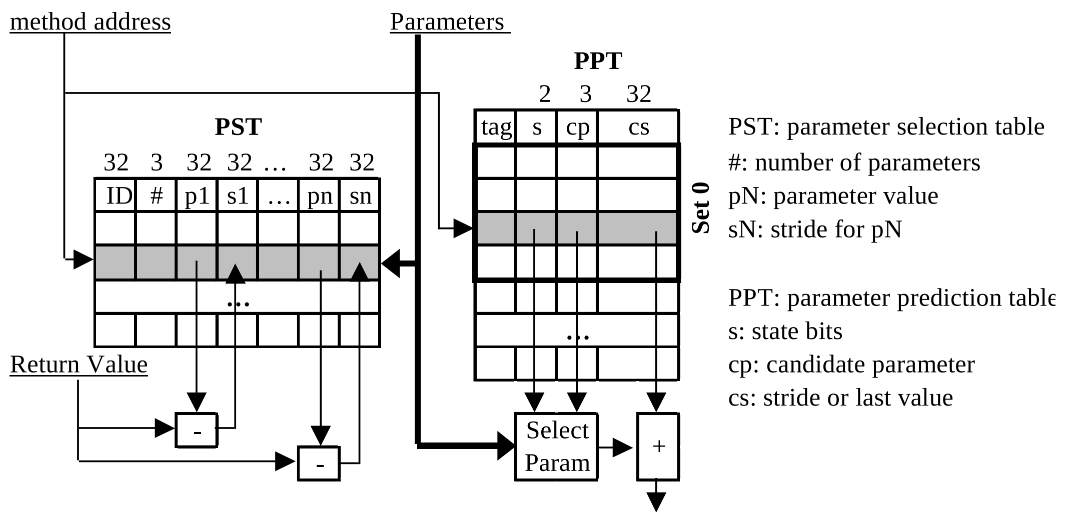

Figure 10. Organization of parameter stride predictor (Bold lines indicate multiple values)

The PST is a small, fully associative table. The ‘#’ field of the PST stores the number of parameters in the method, and a PST entry stores up to eight parameter and stride pairs. For the programs we evaluated, no method has more than eight parameters. Eight pairs of parameters and strides per PST entry are selected for simplicity, and a PST with narrower entries may achieve similar performance for some programs. 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 of a method (i.e. pattern detected or not). 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 replacement.

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 initialized to zero indicating that the method is in the detection state. A PST entry is also allocated and the method's parameters are stored in fields *p1* through *pn*. When the method's return value becomes available, the strides between the return value and parameters are computed and stored in the corresponding PST entry, 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, and 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. Other details of the predictor are presented in [13].

Once the parameter stride pattern is detected once for a method, its detected stride can be used in future PS predictions with an average accuracy of more than 99%. This is partly due to the strong dependence relation between the candidate parameters and return values of the PS methods. The results imply that a PS method's candidate parameter and stride rarely change. Therefore, once a PS method is detected, its PPT entry typically needs no further updates. This stability reduces the negative impact of speculative updates on the PS predictor.

5.3. Prediction coverage

Table 4 provides the percentages of static methods as well as dynamic return values 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. On average, about 13% of dynamically encountered return values exhibit a predictable parameter stride pattern, and 47% of these return values (6% of all return values) cannot be predicted by other types of return value predictors.

	comp	**db**	**javac**	**jess**	**mpeg**	**mtrt**	**average**
Static Methods with Correct PS Prediction	13.5%	12.0%	7.1%	7.0%	15.0%	12.5%	11.2%
Dynamic Methods with Correct PS Prediction	11.0%	24.5%	3.2%	10.2%	13.0%	14.6%	12.8%
Methods Predictable Only by PS Prediction	0.1%	14.7%	2.9%	8.2%	5.4%	4.5%	6.0%

Table 4. Percentage of PS methods (sequential execution)

Figure 11 compares the prediction accuracies of the HYBS and HYBS-PS predictors for an 8-CPU SMLP system. The HYBS predictor is a hybrid of a 1024-entry stride predictor and a context predictor with a 1024-entry value history table and a 4096-entry value prediction table. HYBS-PS is the hybrid of HYBS and a PS predictor with an 8-entry PST and a 512-entry PPT. Similar to the HYBS predictor (as described in Section 2.3), an arbiter in the HYBS-PS predictor chooses between the HYBS predictor and the PS predictor based on their previous accuracies on the predicted methods. When the HYBS-PS predictor is updated, both of its component predictors and the arbiter are updated. For each column, the upper portion of the bar indicates the increased accuracy due to the PS predictor.

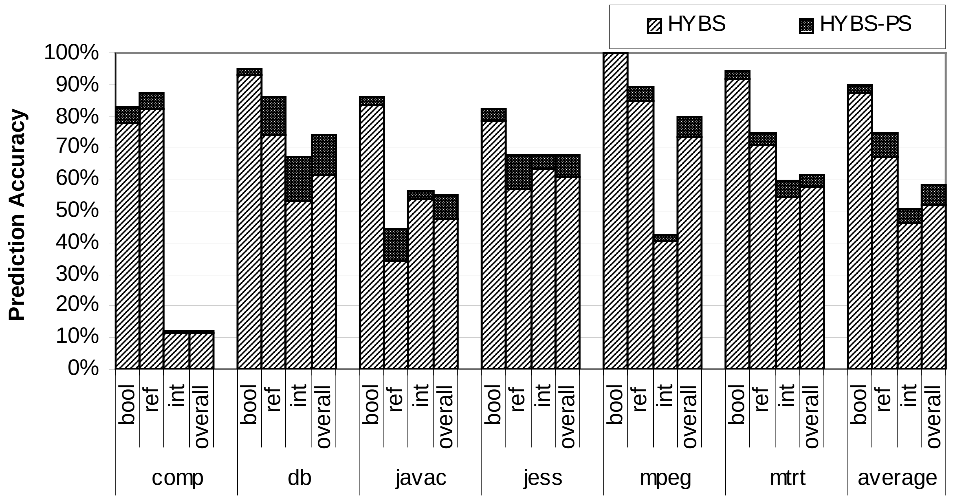

Figure 11. Prediction accuracies of HYBS and HYBS-PS predictor (8-CPU SMLP execution)

Using the PS predictor improves the prediction accuracies of all major return types. On average, the HYBS-PS predictor improves the prediction accuracy by 3% for boolean values, 11% for reference values, and 10% for integer values over the HYBS predictor. The PS predictor's accuracy improvement on boolean values is the smallest among all three return types because the HYBS predictor is accurate on predicting boolean values. The accuracy improvement on integer values due to the PS prediction varies among the programs. The accuracy of *comp*'s integer values improves little by the PS predictor. On the other hand, the PS predictor increases the accuracy by 26% on the integer return values of *db*.

5.4. Effect on Performance

Figure 12 shows the impact of return value prediction on speedup for an 8-CPU SMLP system compared with 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 described in Section 3.2), 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 from Figure 12 is that realistic return value prediction can improve the SMLP system performance. On average, the HYBS predictor improves performance by 14% over the baseline. The program *comp* shows the smallest improvement among all programs, which is mostly due to the poor prediction accuracy (Figure 8). Combining PS prediction with a hybrid prediction further improves performance in a SMLP environment. For the HYBS-PS predictor, the average speedup is 21% over the baseline system, and 7% over 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.

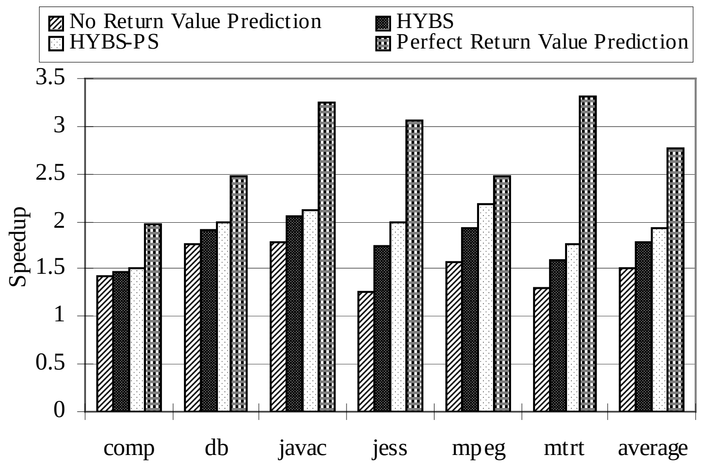

Figure 12. Normalized speedups for different return value prediction schemes (8-CPU SMLP execution)

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 among the programs.

5.5. Impact of Inter-Method Data Dependencies

In a SMLP system, inter-method data dependencies are critical in determining performance. The out-of-order execution nature of the SMLP system causes several data flow problems. First, a speculative thread should not affect the system status until it becomes non-speculative. Hence, several CMPs (e.g. Stanford HYDRA [12]) provide per-processor local buffers to temporarily store the data generated by speculative threads. A local buffer is invisible to other processes until the thread is successfully committed. If the speculative thread is squashed, the content in its local buffer is discarded without affecting the architectural state of the SMLP system.

Second, the data needed by a speculative thread may not be up-to-date. And the use of local buffers may deteriorate the situation since the required data may already reside in the local buffer of another thread. One commonly used technique to alleviate the problem is to forward the values to the consuming threads as soon as the values are generated. Consequently, the squash of a thread causes the rollbacks of the threads that consume the values forwarded by the squashed thread. Another technique is to employ value prediction when the data are not available. In this paper, we use the combination of both techniques. When the data are available in another processor, they are forwarded. Otherwise, the local data are predicted.

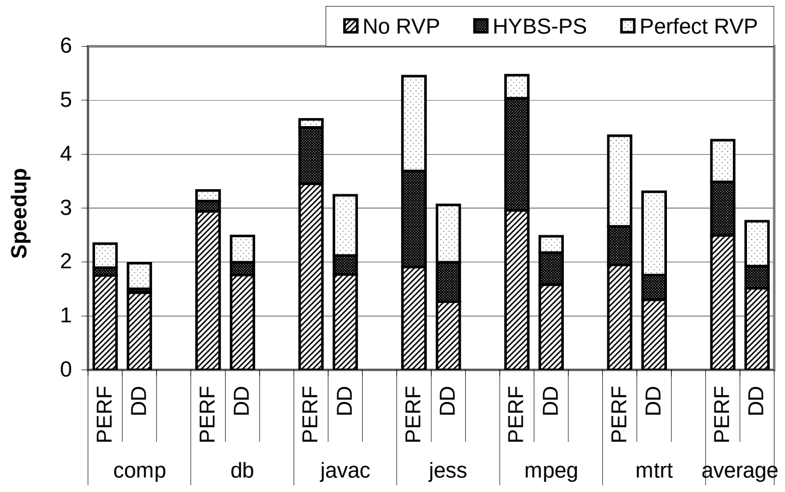

Figure 13. Normalized speedups for systems with and without inter-method data dependencies (PERF: 8-CPU SMLP system without inter-method data dependencies; DD: 8-CPU SMLP system with inter-method data dependencies)

While perfect resolution of inter-method data dependencies provides an upper bound on SMLP performance, it is useful to evaluate realistic return value predictors in the presence of realistic data dependency hardware. Figure 13 shows the performance impact of inter-method data dependencies on a SMLP system. For each benchmark, two sets of results are given. The first set of results is obtained assuming perfect data dependency hardware (i.e. no inter-method data dependency violations). The second set of results (also shown in Figure 12) detects inter-method data dependencies and rollbacks when violations occur (DD). Each result bar consists three columns: representing speedups achieved with no return value prediction, with HYBS-PS predictor, and with perfect return value prediction, respectively.

The impact of the inter-method data dependencies on achievable SMLP performance varies among the programs. With the perfect return value prediction, the performance gained by ignoring inter-method data dependencies varies from 18% (*comp*) to 121% (*mpeg*).

The existence of inter-method data dependencies also limits the performance improvement achievable using return value prediction because it causes extra rollbacks. For example, when a thread squashes, all threads that consume the values generated by the squashed thread are also rolled back, regardless the prediction correctness on those methods' return values. Using the HYBS-PS return value predictor, all workloads, except for *db*, show smaller speedups over no return value prediction without perfect inter-method data dependency handling. The average speedup achieved by using the HYBS-PS return value predictor is 41% for a SMLP system with perfect data dependency hardware, and 21% for a SMLP system with predicted inter-method data dependencies.

6. Related Work

Recent research has focused on method-level speculation, but provides limited discussion on the issue of return value prediction. Chen and Olukotun [4] 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. [19,20] 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 [28] 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. Therefore, 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 [18]. 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 that hold a method's parameters are always different than those that hold 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 [11]. They similarly observe that 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 [21]. 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. Previously, Hu et al. examined the relationship between parameters and return values and this is an extension of that work [13].

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 (SMLP). 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. The researchers are also partially supported by AMD, Intel, IBM, Tivoli and Microsoft Corporations.

References

[1] H. Akkary and M. Driscoll, “A Dynamic Multithreading Processor”, in *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 226-236, Nov. 1998.

[2] M. Burtscher, A. Diwan, and M. Hauswirth, “Static Load Classification for Improving the Value Predictibility of Data-Cache Misses”, in *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 222-233, June 2002.

[3] B. Calder, G. Reinman, and D. Tullsen, “Selective Value Prediction”, in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 64-74, May 1999.

[4] M. Chen and K. Olukotun, “Exploiting Method-Level Parallelism in Single-Threaded Java Programs”, in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 176–184, Oct. 1998.

[5] M. Cintra, and J. Torrellas, “Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors”, in *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pp. 36-47, Feb. 2002.

[6] R. Cmelik and D. Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling”, in *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, pp. 128-137, May 1994.

[7] L. Codrescu and D. Wills, “Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications”, *IEEE Transactions on Computers*, vol. 50, no. 1, pp. 67-82, Jan. 2001.

[8] R. J. Eickemeyer and S. Vassiliadis, “A Load Instruction Unit for Pipelined Processors”, *IBM Journal of Research and Development*, vol. 37, no. 4, pp. 547–564, July 1993.

[9] F. Gabbay and A. Mendelson, “Speculative Execution Based on Value Prediction”, Technion, Israel Institute of Technology, Technical Report 1080, 1996.

[10] J. Gonzalez and A. Gonzalez. “The potential of data value speculation to boost ILP”, in *Proceedings of the 12th International Conference on Supercomputers*, pp. 21-28, July 1998.

[11] J. Gummaraju, and M. Franklin, “Branch prediction in multi-threaded processors”, in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pp. 179-188, Oct. 2000.

[12] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, “The Stanford Hydra CMP”, *IEEE Micro*, vol. 20, no. 2, pp. 71-84, Mar. 2000.

[13] S. Hu, R. Bhargava and L. John, “The Role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism”, in *Proceedings of the First Value-Prediction Workshop*, June 2003.

[14] V. Krishnan, J. Torrellas, “A Chip-Multiprocessor Architecture with Speculative Multithreading”, *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 866–880, Sep. 1999.

[15] E. Larson, and T. Austin, “Compiler Controlled Value Prediction using Branch Predictor Based Confidence”, in *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 327-336, Dec. 2000.

[16] M. Lipasti, C. Wilkerson, and J. Shen, “Value Locality and Load Value Prediction”, in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, Oct. 1996.

[17] M. Lipasti and J. Shen, “Exceeding the Dataflow Limit via Value Prediction”, in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 226-237, Dec. 1996.

[18] P. Marcuello, J. Tubella, and A. González, “Value Prediction for Speculative Multithreaded Architectures”, in *Proceedings of the 32nd International Symposium on Microarchitecture*, pp. 230-236, Nov. 1999.

[19] J. Oplinger, D. Heine, S. Liao, B. Nayfeh, M. Lam, and K. Olukotun, “Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor”, Stanford University, Computer Systems Laboratory Technical Report CSL-TR-97-715, Feb. 1997.

[20] J. Oplinger, D. Heine, and M. Lam, “In Search of Speculative Thread-Level Parallelism”, in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pp. 303-313, Oct. 1999.

[21] B. Rychlik and J. P. Shen, “Characterization of Value Locality in Java Programs”, in *Proceedings of the 3rd IEEE Annual Workshop on Workload Characterization*, pp. 12-23, Nov. 2000.

[22] Y. Sazeides and J. E. Smith, “The Predictability of Data Values”, in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 248-258, Dec. 1997.

[23] Y. Sazeides and J. E. Smith, “Implementations of Context Based Value Predictors”, Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Technical Report ECE97-8, Dec. 1997.

[24] Y. Shuf, M. Serrano, M. Gupta, and J. Singh, “Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations”, in *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pp.194-205, June 2001.

[25] G. Sohi, S. Breach, and T. Vijaykumar, “Multiscalar Processors”, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414-425, June 1995.

[26] J. G. Steffan and T. Mowry, “The Potential for Using Thread-level Data Speculation to Facilitate Automatic Parallelization”, in *Proceedings of 4th International Symposium on High Performance Computer Architecture*, pp. 2-13, Feb. 1998.

[27] K. Wang and M. Franklin. “Highly Accurate Data Value Prediction Using Hybrid Predictors”, in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 281-290, Dec. 1997.

[28] F. Warg and P. Stenstrom, “Limits on Speculative Module-level Parallelism in Imperative and Objective-oriented Programs on CMP Platforms”, in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 221-230, Sep. 2001.

[29] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, “LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation”, in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pp. 128-138, Oct. 1999.

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