Compiler Analyses for Improved Return Value Prediction

Sable Technical Report No. 2004-6

Christopher J.F. Pickett and Clark Verbrugge
{cpicke,clump}@sable.mcgill.ca

October 21, 2004

Abstract

Speculative method-level parallelism has been shown to benefit from *return value prediction*. In this paper we propose and analyse two compiler analyses designed to improve the cost and performance of a hybrid return value predictor in a Java virtual machine setting. A *return value use analysis* determines which return values are consumed, and enables us to eliminate 2.6% of all non-void dynamic method calls in the SPEC JVM98 benchmarks as prediction candidates. An extension of this analysis detects return values used only inside boolean and branch expressions, and allows us to safely substitute incorrect predictions for 14.1% of return values at runtime, provided all future use expressions are satisfied correctly. We find an average 3.2% reduction in memory and up to 7% increase in accuracy. A second, interprocedural *parameter dependence analysis* reduces the number of inputs to a memoization-based sub-predictor for 10.2% of all method calls, and the combination of both analyses reduces overall memory requirements by 5.3%.

1 Introduction

Speculative parallelization is a relatively new performance optimization technique that has shown promising speedups both in hardware [7] and software [5, 18] designs, and including Java environments [4, 10]. Speculative execution can often proceed further and with less rollbacks if unknown data values can be correctly predicted. In the case of *Speculative Method-Level Parallelism* (SMLP) [3], predicting the value returned from a method call can be critical to the success of speculative execution past the method invocation point, and *return value prediction* (RVP) has been thus shown to significantly increase the potential performance benefit of SMLP [9].

For RVP, hybrid predictors incorporating both simple prediction strategies and more complex *context* and *memoization* approaches have been shown to be very effective, even if memory hungry. Speedups of 26% or more over the base SMLP speedup can be achieved, with an empirical upper bound of 86% improvement given perfect prediction [9]. There is a price to pay for high accuracy, however, and we found 10s to 100s of megabytes in storage were required before performance limits were reached [16].

In this paper we propose and analyse two compiler analyses designed to improve the cost and performance of a hybrid return value predictor in a Java virtual machine setting. A *return value use analysis* determines which return values are consumed, and enables us to eliminate 2.6% of all non-void dynamic method calls in the SPEC JVM98 benchmarks as prediction candidates. An extension of this analysis detects return values used only inside boolean and branch expressions, and allows us to safely substitute incorrect predictions for 14.1% of return values at runtime, provided all future use expressions are satisfied correctly. We find an average 3.2% reduction in memory and up to 7% increase in accuracy. A second, interprocedural *parameter dependence analysis* reduces the number of inputs to a memoization-based sub-predictor predictor for 10.2% of all method calls, and the combination of both analyses reduces overall memory requirements by 5.3%.

1.1 Contributions

Specific contributions of this paper include:

1. The design, implementation, and analysis of a new return value use analysis (RVU), suitable for enhancing the effectiveness of speculative method level parallelism. This analysis eliminates the need for context and memoization tables at callsites corresponding to unconsumed return values, reducing memory requirements. It also identifies relaxed accuracy requirements on the prediction of certain used return values, reducing the need for costly rollbacks of speculative execution.

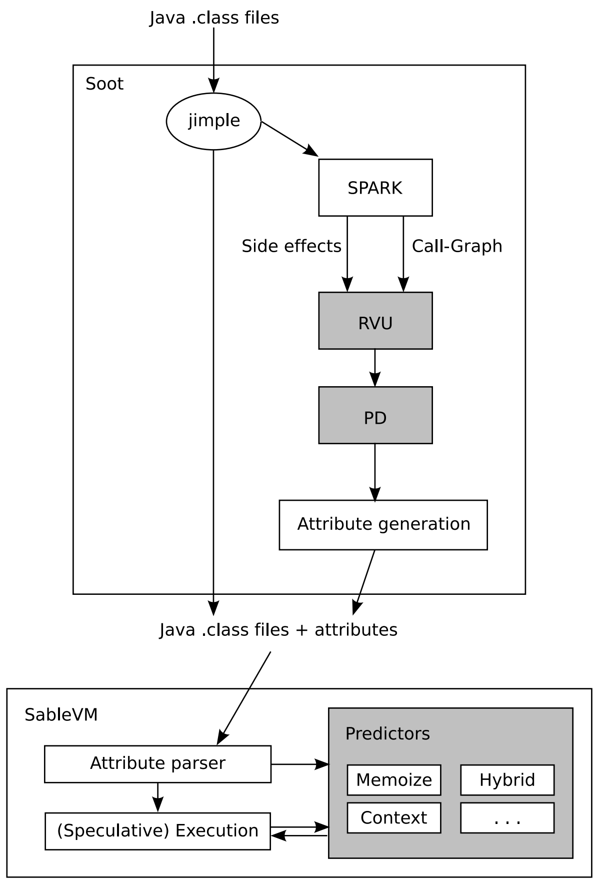


Figure 1: *Basic implementation framework.*

2. The design, implementation, and analysis of a parameter dependence analysis (PD). Parameters not essential to prediction accuracy can be ignored, and this results in better hashtable utilization, again reducing memory requirements.

3. An experimental investigation of the effect of simple, high-level compiler analysis information on value prediction. We are able to achieve a 0–7% increase in accuracy, along with an overall 5.3% decrease in memory requirements.

In the following section we briefly describe the implementation environment we used for our experimentation. This is followed in Section 3 by a description of the return value use analysis, and its corresponding improvements. Section 4 describes the design and experimental data for the parameter dependence analysis. These two approaches are then combined in Section 5. Finally, we discuss related work in Section 6, and then conclude and describe future work in Section 7.

2 Framework

Our analyses are implemented in Soot, an offline Java bytecode optimization framework [25], and communicate analysis data to SableVM [19] using attributes. An overview can be seen in Figure 1.

Input Java .class files are converted to *Jimple,* Soot's stackless, 3-address intermediate representation of

Java bytecode. This is analyzed by Spark [11], a points-to analysis, from which can be derived side-effect information as well as an accurate call graph. Our return value use (RVU) and parameter dependence (PD) analyses then use the resulting information, and generate a classfile *attribute* to be attached to the output class file [17].

In SableVM, the attribute is parsed and the information is used to optimize the various predictors. Predictor data is subsequently gathered and used during actual execution. Note that our current implementation is sufficient to gather return value prediction data, but that actual speculative execution is the subject of future work.

Previously, in SableVM we implemented a handful of simple, well-known predictors, as well as the more complex context and hybrid predictors. The order-5 context predictor [21] we used associates a hashtable with each callsite, and records the observed histories of the last 5 returned values. These histories are converted to a hash key and the hashtable consulted to predict the next value. Accuracy increases with hashtable size, with very high accuracies achievable using storage in the 100s of megabytes.

Memoization also associates hashtables with callsites. In this case though the arguments to the called method are hashed together, and the hash values are associated with specific method return values. Future method calls then can predict the return value based on a hash of the argument values. This also can result in large tables, depending on the variation in method arguments, and hashtables are necessary to keep space requirements manageable. Note that both context and memoization hashtables dynamically expand, and so hashtable usage is reasonably efficient. A complete description of each predictor can be found in [16].

Context, memoization and a few other simpler predictors are then aggregated into a hybrid predictor, which employs all strategies simultaneously and chooses the best one for prediction on a per-callsite basis. Although effective, such a hybrid inherits the memory requirements of its subpredictors, and so performance and memory limits and tradeoffs are especially important to consider. These issues inspired the design and use of the two analyses we now define.

3 Return Value Use Analysis

In Hu *et al.*'s study on return value prediction [9], it was reported that not all return values are consumed in Java programs, but the information was apparently not employed to improve prediction accuracy or reduce memory requirements. This was likely due to the lack of compiler support in their trace-based annotation and simulation framework. If we know that a return value is *unconsumed*, we do not need to allocate predictor storage, and can simply push an arbitrary value onto the stack. In turn, we eliminate predictor misses at unconsumed callsites altogether.

There are also other situations where it is safe to substitute an incorrect prediction. In the following code, any prediction for `r` that results in correct control flow is acceptable.

```
r = foo (a, b, c);
if (r > 10)
  {
    ...      // r == 11, 12, 13, ...
  }
else
  {
    ...      // r == 10, 9, 8, ...
  }
```

In general, if the return value r at a callsite appears only inside boolean and branch *use expressions* of the form (r op v) or (v op r), our safety constraint is relaxed from simple identity

$$r_{predicted} = r_{actual} \tag{1}$$

to identity between between use expression evaluations such that

$$(r_{predicted} \text{ op } v) = (r_{actual} \text{ op } v) \tag{2}$$

or

$$(v \text{ op } r_{predicted}) = (v \text{ op } r_{actual}) \tag{3}$$

for all use expressions. We collect use expressions statically in Jimple, and evaluate them at runtime in SableVM, after returning from a consumed method call. If both predicted and actual return values satisfy all use expressions identically, it is safe to substitute an incorrect prediction. We say that these return values are consumed but *inaccurate*.

Although we could consider more complex analyses such as algebraic simplification [15] to relax constraints on the return value further, preliminary results indicate that the vast majority of suitable use expressions contain only r and a constant[1].

An intraprocedural analysis over all methods in the Spark callgraph provides return value use information. Pseudocode for this analysis is given in Figure 2. It is trivial to detect unconsumed return values as non-void method calls assigned to dead variables, and we exploit Soot's dead assignment eliminator to do so. To find consumed but inaccurate return values, we rely on a simple reachability analysis, and on local definitions and local uses analyses that provide us with UD/DU chains.

Having found a consumed return value at a callsite c, we add use expressions at conditional branches (`if` and `switch`), and at assignment statements whose rvalues are boolean expressions that evaluate to -1, 0, or 1. If any use expression u_i contains a local, we further assert that no definitions d of the local occur such that there exists a path from c to d and from d to u_i. We allow for integer, floating point, and null constants inside use expressions, but not string constants, as comparison of two strings is slow.

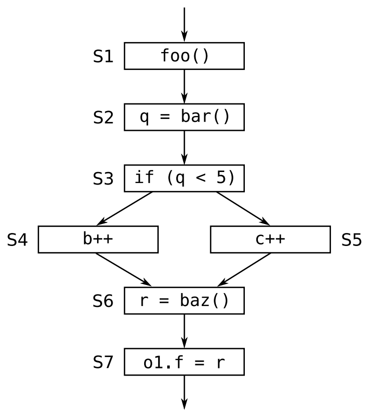
An example return value use analysis is shown in Figure 3. There are three callsites in the control flow graph, at `S1`, `S2`, and `S6`. The call to `foo()` in `S1` has had its assignment to a dead variable eliminated, and we mark the return value as unconsumed and inaccurate. The call to `bar()` in `S2` is assigned to `q`, which is subsequently used only inside the branch expression `(q < 5)` in `S3`. Therefore we mark `S2` as consumed but inaccurate, and record the use expression. Finally, the call to `baz()` at `S6` is assigned to `r`, which is then written into `o1.f` on the heap. Without accurate (and expensive) *must points-to* information, we are forced to mark `S6` as consumed and accurate.

The table of entries in Figure 3 captures the salient points of the final structure generated by Soot's attribute generation framework. There is one entry in the attribute per callsite, with both consumption and accuracy information recorded. In the case of inaccurate consumed values, there are extra columns for operator kind, operand positions, operand values, operand sizes, and whether v is a constant, stack value, parameter local, or non-parameter local. The attributes are named `org.sablevm.ReturnValueUseTable` as per the JVM Specification [13].

[1]We do consider comparisons with locals in boolean and branch expressions, but due to a temporary limitation in our analysis framework we are unable to provide results including locals at this time.

```
for each callsite c with method call m assigned to return value r

  if (r is a dead variable)
    consumed = false
    accurate = false

  else
    consumed = true
    accurate = false

    for each use statement in DU chain of r

      /* get simple conditionals */
      if (use instanceof IfStmt)
        add branch expression to use expressions

      /* create expressions for switch-based branches */
      else if (use instanceof TableSwitchStmt ||
               use instanceof LookupSwitchStmt)
        for all indices
          add "(r == index)" to use expressions

      /* get assigned boolean expressions */
      else if (use instanceof AssignStmt)
        if (eval(rvalue) belongs to {-1,0,1})
          add rvalue to use expressions
        else
          accurate = true
      else
        accurate = true

    /* assert that locals are always defined before call */
    if (!accurate)
      for each use in use expressions
        let (r op v) or (v op r) be the use expression

        if (v instanceof Local)
          for each reaching def d in UD chain of v
            if there exists a path from c to d
              accurate = true
```

Figure 2: *Pseudocode for return value use analysis.*



	consumed	accurate	uses
S1	no	no	———
S2	yes	no	q < 5
S6	yes	yes	———

Figure 3: *Intraprocedural Return Value Use Analysis.* Callsites `S1`, `S2`, and `S6` are marked with respect to being consumed and needing to be accurate, and the use `q < 5` is recorded for `S2` as it is consumed but not accurate. This information is conveyed to the virtual machine via classfile attributes.

Table 1: *Static Return Value Uses. unconsumed* return values are never used, *inaccurate* return values must only satisfy a set of use expressions, and *accurate* return values must be predicted correctly. Class libraries are included in the analysis and only callsites with non-void return values are considered

bench-mark	non-void callsite return values			
	total	unconsumed	inaccurate	accurate
`comp`	7156	23.1%	9.4%	67.5%
`db`	7322	22.7%	9.6%	67.7%
`jack`	8090	21.2%	10.0%	68.8%
`javac`	10503	17.3%	12.8%	69.9%
`jess`	9531	18.8%	9.4%	71.8%
`mpeg`	7586	22.4%	9.9%	67.7%
`mtrt`	8029	21.3%	8.7%	70.0%
average	8317	20.7%	10.1%	69.2%

Table 2: *Dynamic Return Value Uses. unconsumed* return values are never used, *inaccurate* return values must only satisfy a set of use expressions, and *accurate* return values must be predicted correctly. Class libraries are included in the analysis and only callsites with non-void return values are considered

bench-mark	non-void callsite return values			
	total	unconsumed	inaccurate	accurate
`comp`	133M	0.0%	0.0%	100.0%
`db`	115M	0.0%	27.9%	72.1%
`jack`	34M	0.9%	24.5%	74.6%
`javac`	82M	1.5%	20.7%	77.8%
`jess`	102M	0.1%	40.9%	59.0%
`mpeg`	77M	14.4%	10.3%	75.3%
`mtrt`	267M	3.0%	2.8%	94.2%
average	116M	2.6%	14.1%	83.3%

Static analysis results over SPEC JVM98 (size 100) are given in Table 1. On average, 20.7% of callsites are unconsumed, 10.1% are consumed but may be inaccurate provided use expressions are satisfied, and 69.2% must be accurate. Large numbers of dynamically unreached callsites are included in the static analysis, which is derived from Spark's *may points-to* information, and these exist largely in class library code. This leads to a small standard deviation and relatively similar results for all benchmarks. We reported the total number of unique callsites reached at runtime in [16].

Dynamic results from the use of analysis information at runtime are given in Table 2. On the whole there is a much stronger bias towards consumed but inaccurate callsites with an average of *14.1%* of predicted return values not needing to be accurate, and only *2.6%* being unconsumed. This is comparable with the dynamic results for unconsumed return values reported in [9]. `comp`, `mpeg`, and `mtrt` reach particularly low percentages of inaccurate callsites, following from the fact that they are essentially numerical benchmarks, `comp` performing numerical compression, `mpeg` decompressing random mp3 data, and `mtrt` computing a 3D scene using a raytracer. The other more object-oriented benchmarks exhibit fairly high percentages of inaccurate return values, with `jess` being the highest at 40.9%.

In Figure 4, the performance of the context predictor is graphed against maximum per-callsite table size, before and after application of the RVU analysis. Performance increases are relatively constant at all table

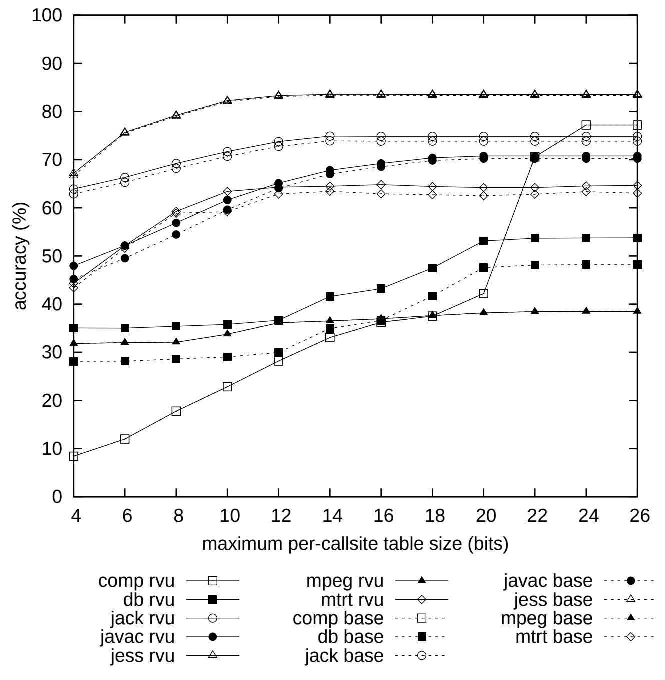


Figure 4: *Context Size Variation after Return Value Use Analysis.*

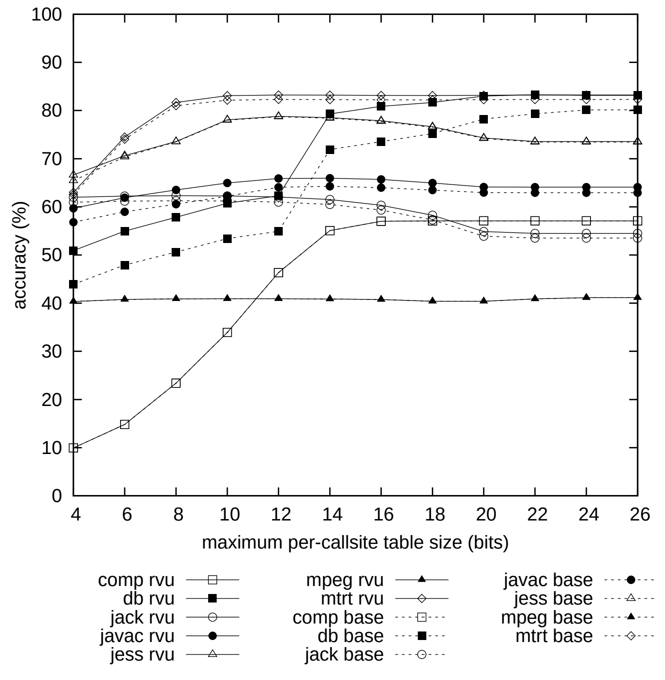


Figure 5: *Memoization Size Variation after Return Value Use Analysis.*

sizes, with `jack`, `javac`, and `mtrt` displaying slight increases and `db` showing a nice 5–7% improvement. The differences are somewhat larger at small table sizes, as more collisions occur and predictions are more likely to be incorrect but still accurate enough to satisfy use expressions. `mpeg` performs poorly despite elimination of 14.4% of all predictions, which indicates that these values were predicted correctly before

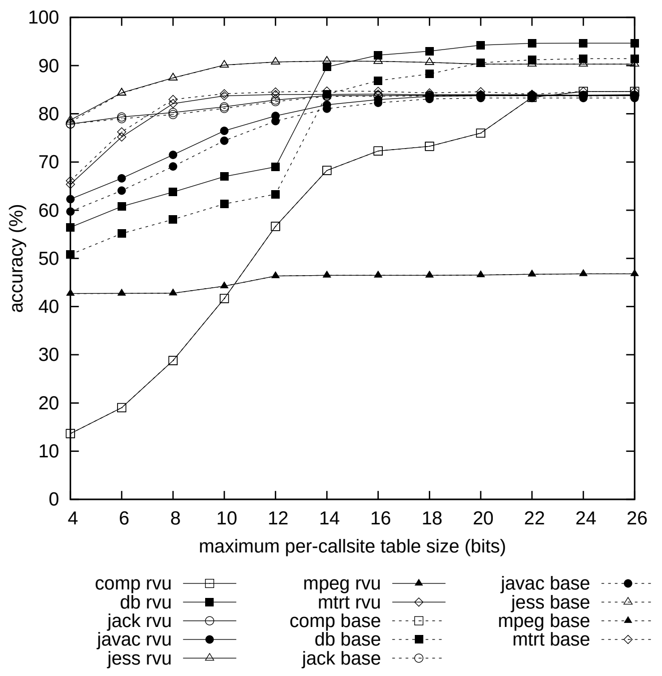


Figure 6: *Hybrid Size Variation after Return Value Use Analysis.*

elimination; nevertheless, there is a slight 2.2% memory reduction available (Table 5). There are comparable changes in the memoization predictor (Figure 5), and when context and memoization predictors are combined in a hybrid (Figure 6), they complement each other nicely as before [16]; `db` maintains a 5–7% accuracy increase, reaching as high as 95% at extreme sizes.

4 Parameter Dependence

Memoization is a well-known technique for caching function results in computer science, and we previously introduced the first use of memoization in a speculative environment [16]. One of the nice properties of our memoization predictor is that conservative correctness, although generally beneficial, is not required, and this provides opportunities for aggressive optimistic analyses. In Figure 7, a memoization table is shown that contains previously computed values for three distinct calls to `foo()`.

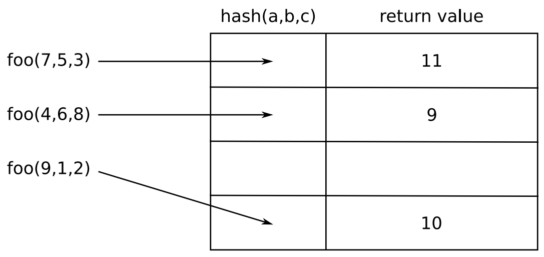


Figure 7: *Memoization.* Three different calls to `foo()` hash to three different slots in the memoization lookup table, and distinct return values are stored for each set of parameters.

When we studied the performance of the memoization predictor at varying maximum table sizes ([16], also Figures 5 and 14), we observed that for `jack`, `javac`, and `jess`, performance actually starts to decrease as the maximum table sizes increase past 8, 14, and 12 bits respectively. Tag information indicated that distinct sets of parameters were mapping to the same return value, and that the peaks of these curves actually arose from beneficial collisions.

Our hypothesis is that such redundancy might occur if the return value of the method in question is not dependent on some or all of the parameters. For example, in method `foo()`, the return value only depends on the first and third arguments:

```
static int foo(int a, int b, int c) {
    int k = 0;
    if (a > 5)
        System.out.println(b);
    else
        k = 7;
    return (c + (k = k + 1));
}
```

and in Figure 8 we show three different calls to `foo()` where the first and third parameters do not vary but the second parameter does, leading to redundant entries. In general, redundant entries in memoization tables will exist for each set of method invocations in which only unimportant parameters vary.

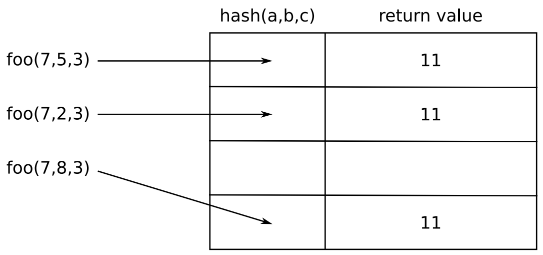


Figure 8: *Memoization with Redundancy.* Three different calls to `foo()` hash to three different slots, but the same return value of 11 is stored for each set of parameters. In this case, parameter `b` does not affect the return value and redundancy is introduced.

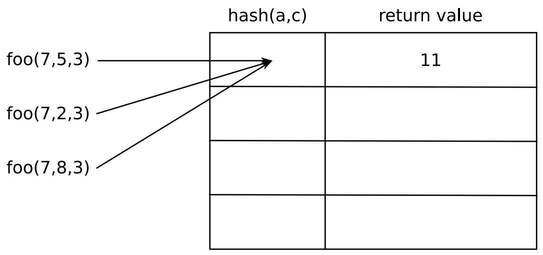


Figure 9: *Memoization with Sharing.* Three different calls to `foo()` hash to one slot, with a return value of 11 for each set of parameters. This is possible after a parameter dependence analysis eliminates inputs to the memoization hash function that do not affect the return value.

If we can remove these parameters from the set of inputs to the hash function, then we should be able to increase positive sharing of table entries, as shown in Figure 9, leading to improved prediction accuracy and reduced memory requirements. Furthermore, if the return value is dependent on zero parameters, we can avoid using memoization altogether, and let the hybrid pick the best out of the remaining predictors at runtime.

```
for each statement s considered in backwards flow analysis
  copy out(s) to in(s)
  add s to in(s)

  branch_independent =
    intersection of all in(t) over all t in succ(s)

  branch_dependent =
    out(s) - branch_independent

  h.put(s, branch_dependent)
```

Figure 10: *Branch dependence helper analysis.*

To compute parameter dependencies, we rely on a simple pre-analysis that finds branch dependent units in a control flow graph, i.e. those units that are reached depending on a particular branch being taken (Figure 10). This is a backwards may analysis with the value for *END* and the initial approximation for all nodes being the empty set, and the merge operator being union. We put branch dependent units of each statement *s* in the CFG into a hashtable *h* for future retrieval. This analysis may also be expressed in terms of post-dominance and reachability, but here we opt for the simpler explanation.

Intraprocedurally, our parameter dependence analysis computes which values at *START* affect the return value. If we take the intersection of this set with the set of formal parameters, which includes `this` for non-static methods, we get parameter dependencies. Our analysis performs a kind of optimistic slicing; for a review of conservative slicing techniques see [24].

Pseudocode for the analysis is given in Figure 11. This is also a backwards may analysis, with the value for *END* and the initial approximation for all nodes being the empty set, and the merge operator being union. We add value uses to *in(s)* as appropriate, and propagate them backward through the control flow graph until a least fixed point is reached. Constants and caught exception references are not propagated, minimizing the size of flow sets.

We use the side effect information provided by Spark [11] to determine if any statement *s* may write to a value in *out(s)*, and if so add *uses(s)* to *in(s)*. We also record that the statement may write to a value of *out(s)* for future use. Furthermore, if *s* must write to a value in *out(s)*, we kill that value. Spark does not provide must points-to information, and therefore we cannot kill fields and elements of objects and arrays on the heap. In fact, we did perform some experiments where we unsafely assumed that Spark does provide a must points-to analysis, but these did not increase the accuracy of our results.

If we encounter a return statement, we add the returned value to *in(s)*, and if we encounter any *s* with multiple successors, we add *uses(s)* to *in(s)* if the return value depends on a particular branch being taken. This dependency between the branch condition and *r* exists if any branch-dependent unit of *s* can write to *out(s)*, as previously computed.

However, we only add *uses(s)* if *s* is an explicit branch statement. Aside from an explicit `throw`, many instructions in Java and therefore Jimple can throw exceptions, and this leads to many units in the CFG technically having multiple successors. In such cases, even though there may be no units on the exception handler branch that affect the return value, *s* will have all units in the normal path of execution marked as branch-dependent. In a safe analysis we should also be adding the uses of these exception-throwing statements. However, we previously observed [16] that for SPEC JVM98, exceptions are extremely unlikely to be thrown at run-time, with the exception-heavy `jack` having only 1% of all method calls throw an exception. Ignoring implicit exception handler edges is unsafe, but as noted we do not require safety in our speculative environment; this optimization slightly improves the accuracy of our results.

```
let w be the set of statements writing to out(s)

for each statement s considered in backwards flow analysis

  copy out(s) to in(s)

  for each value v in out(s)
    if (s may write to v)
      add s to w
      if (s must write to v)
        remove v from in(s)
      add uses(s) to in(s)

  if (s instanceof ReturnStmt)
    add uses(s) to in(s)

  for each branch dependent statement d of s
    if (w contains d || d instanceof ReturnStmt)
      if (s instanceof IfStmt ||
          s instanceof LookupSwitchStmt ||
          s instanceof TableSwitchStmt ||
          s instanceof ThrowStmt)
        add uses(s) to in(s)

  remove constants and caught exception references from in(s)
```

Figure 11: *Pseudocode for parameter dependence analysis.*

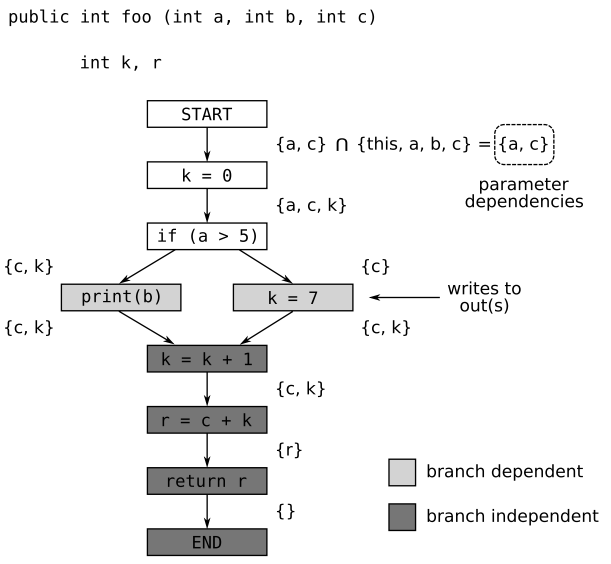


Figure 12: *Intraprocedural parameter dependence analysis.* This analysis captures the dependencies between the return value r and method parameters; in the figure, these are $r \rightarrow a$ and $r \rightarrow c$.

An example intraprocedural parameter dependence analysis over `foo()` is shown in Figure 12. Starting at *END*, we add uses and kill definitions as appropriate, and the resultant flow sets are shown in between nodes. At the merge point `if (a > 5)`, we must add *a* because one of the if statement's branch dependent units, `k = 7`, writes to its *out(s)*. At *START* the intersection of *out(s)* with the set of formal parameters yields parameter dependencies, and `this` is included in that set because `foo()` is non-static.

Having finished our intraprocedural analysis, we decided to build on it with a more aggressive interprocedural analysis. When we reach a callsite whose return value is assigned to a member of *out(s)* and therefore affects the current method's return value, we only want to add as uses the parameter dependencies of the

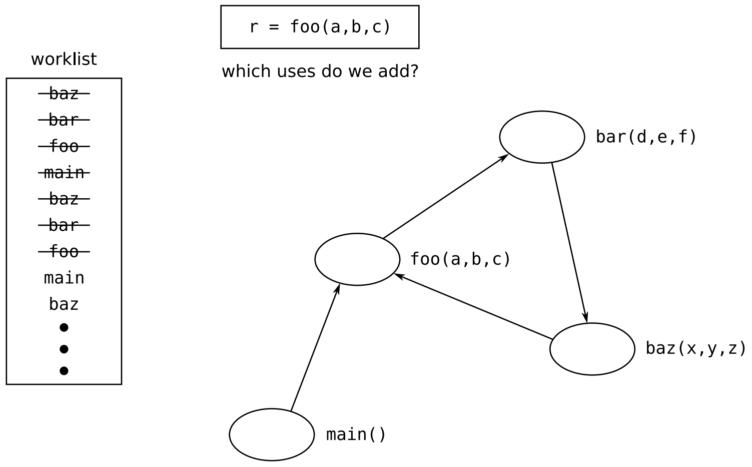


Figure 13: *Interprocedural Parameter Dependence Analysis.* Call graph nodes `main()`, `foo()`, `bar()` and `baz()` are placed onto a worklist, and the intraprocedural parameter dependence calculated with the initial assumption that there are zero dependencies at each callsite. If a node changes, all of its callers are placed onto the worklist, and this continues until a least fixed point is reached.

target, iterating over all targets if the callsite is polymorphic. We treat abstract targets as having zero dependencies, and native targets as having full dependencies. If any target is itself able to write to *out(s)*, as detected via Spark's side-effect information, we add all uses.

This is a relatively simple worklist-based least fixed point analysis. All nodes in the callgraph are placed onto a worklist in pseudo-topological order, with the initial assumption being that zero parameters affect the return value. Nodes are processed intraprocedurally, and if the computed parameter dependencies change, all nodes with edges into the current node are placed back onto the worklist. This continues until a least fixed point is reached. In Figure 13 an example analysis is shown for a simple callgraph with a `main()` method and a cycle between `foo()`, `bar()`, and `baz()`.

Final parameter dependence information for each callsite is computed as the union of parameter dependencies over all of its targets, and is again conveyed to SableVM via classfile attributes. At runtime these dependencies are mapped to method argument locations on the Java stack, which in turn form the inputs to the memoization predictor hash function. It might actually be simpler to attach attributes per method instead of per callsite, but again we defer a full investigation of the advantages and disadvantages of per callsite vs. per method return value prediction to future work. We do find that our interprocedural analysis as given doubles the precision of our results.

Table 3 provides information on parameter dependencies at statically reachable callsites. We only consider consumed callsites with more than one parameter, and find an average 24.8% having zero dependencies, 23.1% having partial dependencies, and 52.1% having full dependencies. As per the static return value use results given in Table 1, the standard deviation is small, as the majority of callsites considered reside in common library classes.

Dynamic parameter dependence results in Table 4 are strikingly different, as was the case for dynamic return value use results. On average far lower percentages of consumed method calls with zero or partial parameter dependencies occur, with 7.1% having zero dependencies and only 3.1% having partial dependencies. However, the standard deviation in dynamic results is again much larger, with `mpeg` having zero or partial parameter dependencies for roughly half of its method calls, and `comp` consuming an overwhelming majority of fully dependent return values.

Table 3: *Static Parameter Dependencies.* *n* is the number of parameters and *d* is the number of dependencies, or parameters affecting the return value. Class libraries are included in the analysis and only callsites with consumed return values having $n > 0$ are considered here

bench-mark	consumed callsites with $n > 0$			
	total	$d = 0$	$d > 0 \wedge d < n$	$d = n$
`comp`	5294	25.4%	24.3%	50.3%
`db`	5446	25.4%	24.1%	50.5%
`jack`	6159	27.0%	24.9%	48.1%
`javac`	8460	24.7%	20.6%	54.7%
`jess`	7476	23.2%	21.4%	55.4%
`mpeg`	5671	25.7%	24.8%	49.5%
`mtrt`	6100	22.6%	21.2%	56.2%
average	6372	24.8%	23.1%	52.1%

Table 4: *Dynamic Parameter Dependencies.* *n* is the number of parameters and *d* is the number of dependencies, or parameters affecting the return value. Class libraries are included in the analysis and only callsites with consumed return values having $n > 0$ are considered here

bench-mark	consumed method calls with $n > 0$			
	total	$d = 0$	$d > 0 \wedge d < n$	$d = n$
`comp`	133M	0.0%	0.9%	99.1%
`db`	115M	3.8%	0.0%	96.2%
`jack`	33M	7.4%	1.9%	90.7%
`javac`	80M	12.8%	4.2%	83.0%
`jess`	102M	20.3%	0.0%	79.7%
`mpeg`	66M	20.0%	28.0%	52.0%
`mtrt`	259M	2.0%	0.3%	97.7%
average	112M	7.1%	3.1%	89.8%

Performance in the context predictor remains constant after application of the PD analysis, as it only has the potential to affect the memoization predictor, and therefore in turn the hybrid predictor. This analysis reduces the total number of predictions made, as we do not predict if there are zero dependencies. Thus in Figure 14 we look at the memoization accuracy over memoized predictions. We still observe the prior decrease in performance that motivated this analysis for `jack`, `javac`, and `jess`, and thus put forward a new hypothesis that even for benchmarks with partial or full parameter dependencies, beneficial collisions may occur in hashtables at smaller sizes if external factors such as heap side-effects also global variables also affect the return value.

In any case, we report a maximal 13% increase in accuracy for `jess`, with comparably large increases for `jack` and `javac`; this correlates with these benchmarks relatively high percentages of unconsumed return values. At first it seems odd that there is a decrease in the performance of `mpeg`, but this simply means that the untaken predictions were correct before elimination by the PD analysis; again this indicates that values other than method parameters are likely affecting return values. In the hybrid predictor (Figure 15), all benchmarks perform as well as before, save for a slight reduction in `javac`, and extra memory savings due to the PD analysis are given in Table 5.

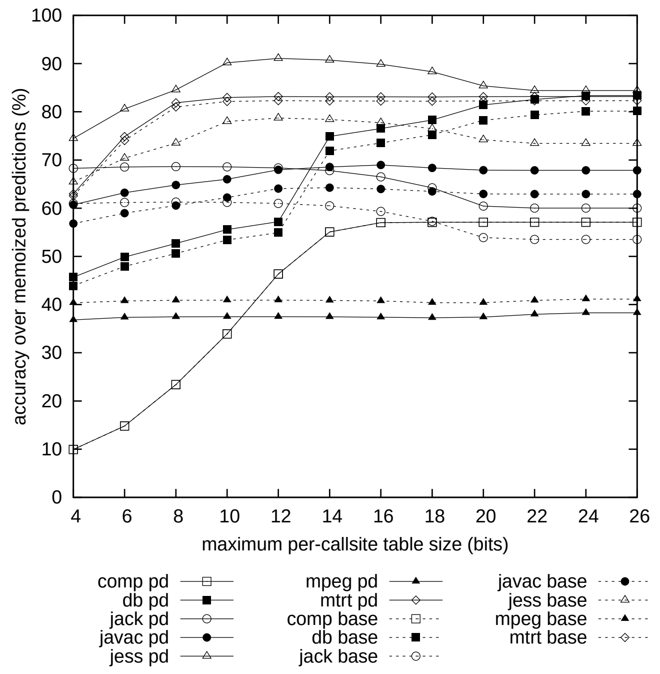


Figure 14: *Memoization Size Variation after Parameter Dependence Analysis.*

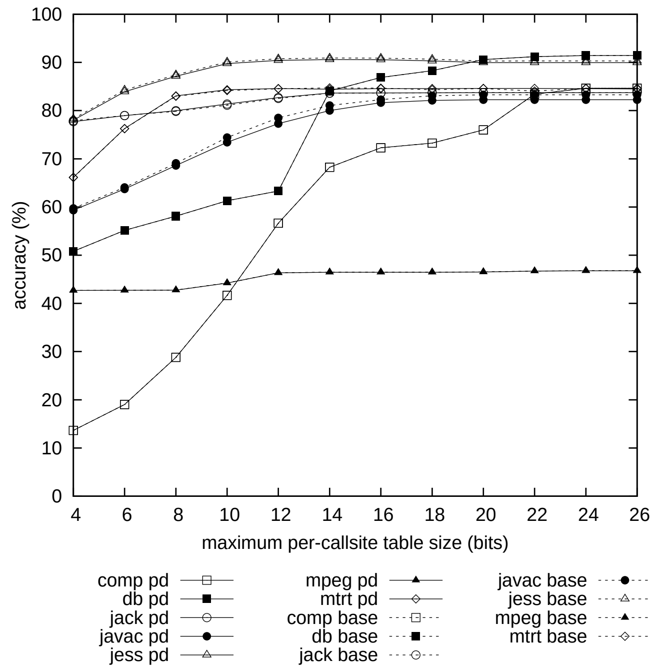


Figure 15: *Hybrid Size Variation after Parameter Dependence Analysis.*

5 Combined Analyses

Lastly we look at the combination of both return value use and parameter dependence analyses. The performance of the hybrid predictor with both analyses applied is shown in Figure 16, and we see the same performance increases as per the hybrid with only the return value use analysis applied (Figure 6).

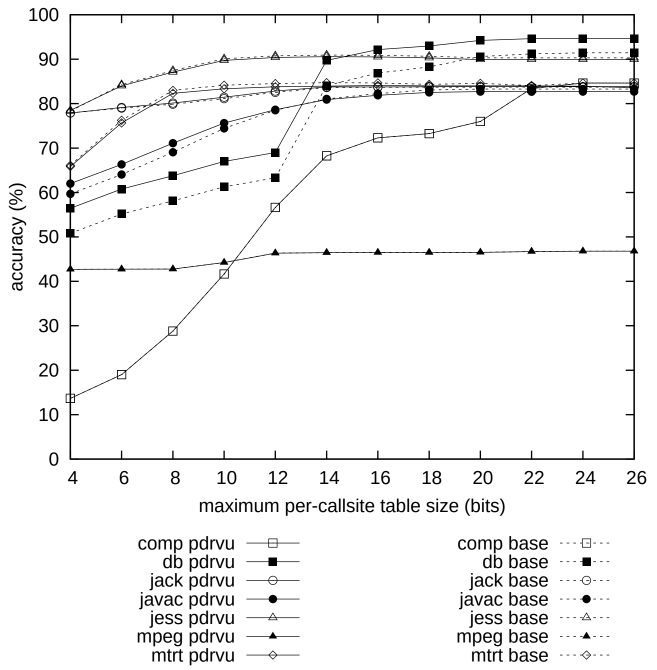


Figure 16: *Hybrid Size Variation after Combined Analyses.*

Table 5: *Context and memoization table memory.* The maximum table size in bits was chosen as the highest point on the baseline size variation curve for each benchmark [16]

bench-mark	context				memoization					
	size	base	rvu	savings	size	base	rvu	pd	pdrvu	savings
`comp`	24	208M	208M	0.0%	18	6.30M	6.24M	6.28M	6.22M	1.3%
`db`	24	361M	361M	0.0%	24	206M	205M	195M	195M	5.3%
`jack`	14	10.5M	10.3M	1.9%	8	939K	874K	718K	651K	30.7%
`javac`	20	211M	203M	3.8%	14	65.7M	61.2M	51.8M	48.5M	26.2%
`jess`	14	9.15M	8.76M	4.3%	12	5.01M	4.52M	4.41M	3.92M	21.8%
`mpeg`	12	2.27M	2.22M	2.2%	12	589K	523K	544K	477K	19.0%
`mtrt`	14	46.1M	43.8M	5.0%	12	15.1M	14.3M	14.8M	13.9M	7.9%
average	17	121M	120M	2.5%	14	42.8M	41.8M	39.1M	38.4M	16.0%

Previously we reported hybrid accuracies and memory consumption for profiled predictor sizes [16], choosing aggressively accurate data points at the performance limit of context and memoization predictors, with maximum table size as small as possible without sacrificing this accuracy. We report memory consumption results at these same maximum table sizes in Table 5, noting that dramatic decreases in memory with only marginal decreases in accuracy can be obtained by choosing more realistic table sizes. We observe an average 2.5% reduction in the context predictor obtained from the RVU analysis, and 16.0% reduction in the memoization predictor from the combination of RVU and PD analyses. When the total memory required by a hybrid predictor is considered, we observe a 3.2% reduction in memory after application of the RVU analysis, and 5.3% reduction after application of the PD analysis.

6 Related Work

Speculative multithreading is typically examined in the context of hardware designs [7], even for Java [3, 4]. Software approaches at the assembly or C language level however have also been proposed and shown to have potentially good speedup [5, 18]. For Java specifically, the feasibility of a software implementation was demonstrated by Kazi and Lilja through manual transformations [10] of Java source code.

Value prediction itself is a well-known technique for allowing speculative execution of various forms to proceed beyond normal execution limits, and a number of prediction strategies have been defined and analysed. These extend from relatively simple last value predictors [14] to more complex finite context (FCM) [21, 22] and differential context (DFCM) [8] value predictors. These are also typically designed with hardware constraints in mind, though again software techniques have been examined; Li *et al.*, for example, develop a software prediction scheme based on profiling and inserting recovery code following a cost-driven model [12]. The utility of return value prediction for method level speculation in Java was shown by Hu *et al.* [9], with further prediction accuracy investigated by the authors [16].

Our approach here is to develop compiler analyses that assist or improve value prediction. Others have also investigated software help for prediction, though with very different analyses. Burtscher *et al.* analyse program traces to divide load instructions into *classes*, with different groupings having logically distinct predictability properties [2]. Du *et al.* use a software loop unrolling transformation to improve speculation efficiency, but also evaluate likely prediction candidates from trace data using a software cost estimation [6]. Code scheduling approaches that identify and move interthread dependencies so as to minimize the chance of a misprediction have been developed by Zhai *et al.* [26]. A more general consideration of compiler optimization is given by Sato *et al.*, who analyzed the effect of unrelated optimizations on predictability and found that typical compiler optimizations do not in general limit predictability [20].

We have implemented our return value use analysis such that it only considers cases involving simple comparisons of the predicted variable with constants and locals. Obviously, other uses could be accommodated in this scheme; most candidates for algebraic simplification [15], for instance, could easily allow for a variety of inaccurate, predicted values. This in fact has recently been considered in hardware approaches to speculation [1], but statically we do not expect a large number of trivial operations, which should be removed by prior compiler optimizations.

The parameter dependence analysis we develop is similar to static, interprocedural program slicing techniques [24], which compute the data and control dependencies of a given variable use. Slicing is a relatively expensive task, although performance optimizations to slicing exist [27]. In our case we need only discover parameter dependencies, not all dependencies, and moreover can exploit the speculative nature of the problem to simplify the algorithm.

7 Conclusions and Future Work

We have described the design and implementation of two compiler analyses aimed at improving the feasibility of software return value prediction. A return value use removes the need for prediction data to be gathered or stored for methods that do not have their return value consumed. A further refinement reduces prediction accuracy requirements on methods that have limited subsequent uses. Our second analysis computes a simple parameter dependence relation, establishing which function input parameters may contribute to the final value of a returned variable. This allows us to reduce false sharing in memoization hashtables, and thus reduce cold start prediction misses and also decrease memory requirements.

We have implemented our analyses in Soot, and applied them in SableVM to context, memoization and hybrid predictors. These associate prediction storage tables at each callsite, and so memory usage can be a concern if optimal accuracy is desired. Our return value use analysis statically identifies 30.8% of callsites in the SPEC JVM98 benchmarks as needing zero or reduced prediction accuracy, although this reduces to 16.7% of runtime callsites. Parameter dependence follows a similar pattern, with 47.0% of callsites statically having zero or only partial dependence on parameter values, dropping to just 10.2% of dynamic callsites.

Given the reduced number of dynamic opportunities, accuracy improvements are correspondingly small, although db does well with a 5–7% improvement in the final, combined analysis. Space savings, however, are still significant, with context memory reducing by an average of 2.5% and memoization costs reducing by 16%, at the highest accuracy and thus highest memory costs.

We aim to continue our work in a number of ways. More complex expressions for inaccurate return value uses could be allowed, further improving predictor accuracy, although in a practical sense this would have to be carefully balanced with the cost of runtime verification. Also, the return value use analysis could be extended, allowing for inaccurate predictions to be substituted in general purpose load value predictors. A *purity* analysis [23] would identify methods which do not generate side effects, and so would be good candidates for perfect memoization in the traditional conservative sense.

We are actively working on a Java software implementation of SMLP. This would enable us to measure the actual performance improvement provided by return value prediction in a real speculative system running on common multiprocessor hardware, and determine how to balance memory access costs, verification costs, accuracy, and other factors affecting optimal speculative performance.

Acknowledgements

We kindly thank Patrick Lam for discussions and implementation help with Soot. This research was funded by Le Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and the Natural Sciences and Engineering Research Council of Canada (NSERC). Chris Pickett was additionally supported by a Richard H. Tomlinson Master's in Science Fellowship and an NSERC PGS A award.

References

[1] E. Atoofian, A. Baniasadi, and N. Dimopoulos. Improving performance by speculating trivializing operands in trivial instructions. In *Proceedings of the Second Value-Prediction and Value-Based Optimization Workshop (VPW2)*, pages 26–31, Oct. 2004.

[2] M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 222–233. ACM Press, June 2002.

[3] M. K. 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 (PACT)*, Oct. 1998.

[4] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th annual International Symposium on Computer Architecture (ISCA)*, pages 434–446. ACM Press, June 2003.

[5] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–24. ACM Press, June 2003.

[6] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation (PLDI)*, pages 71–81. ACM Press, June 2004.

[7] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin–Madison, 1993.

[8] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 207–216. IEEE Computer Society, Jan. 2001.

[9] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 5:1–21, Nov. 2003.

[10] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 559–564. IEEE, May 2000.

[11] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, Montréal, Québec, Dec. 2002.

[12] X.-F. Li, Z.-H. Du, Q. Zhao, and T.-F. Ngai. Software value prediction for speculative parallel threaded computations. In *Proceedings of the First Value-Prediction Workshop (VPW1)*, pages 18–25, June 2003.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.

[14] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 138–147. ACM Press, Oct. 1996.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *Proceedings of the Second Value-Prediction and Value-Based Optimization Workshop (VPW2)*, pages 40–47, Oct. 2004.

[17] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the 10th International Conference on Compiler Construction (CC)*, pages 334–554, Apr. 2001.

[18] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3:1–28, Oct. 2001.

[19] SableVM. `http://www.sablevm.org/`.

[20] T. Sato, A. Hamano, K. Sugitani, and I. Arita. Influence of compiler optimizations on value prediction. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking (HPCN)*, pages 312–321. Springer-Verlag, June 2001.

[21] Y. Sazeides and J. E. Smith. Implementations of context-based value predictors. Technical Report TR ECE-97-8, University of Wisconsin–Madison, Dec. 1997.

[22] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 248–258, Dec. 1997.

[23] A. Sălcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, Massachusetts Institute of Technology, May 2004.

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[25] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Montréal, Québec, July 2000.

[26] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[27] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 94–106. ACM Press, June 2004.