McGill University
School of Computer Science
Sable Research Group

Adaptive Software Return Value Prediction

Sable Technical Report No. 2010-3

Christopher J.F. Pickett and Clark Verbrugge and Allan Kielstra
{cpicke,clump}@sable.mcgill.ca, kielstra@ca.ibm.com

April 21st, 2010

Adaptive Software Return Value Prediction

Abstract

Return value prediction (RVP) is a technique for guessing the return value from a function before it actually completes, enabling a number of program optimizations and analyses. However, despite the apparent usefulness, RVP and value prediction in general have seen limited uptake in practice. Hardware proposals have been successful in terms of speed and prediction accuracy, but the cost of dedicated circuitry is high, the available memory for prediction is low, and the flexibility is negligible. Software solutions are inherently much more flexible, but can only achieve high accuracies in exchange for reduced speed and increased memory consumption. In this work we first express many different existing prediction strategies in a unification framework, using it as the basis for a software implementation. We then explore an adaptive software RVP design that relies on simple object-orientation in a hybrid predictor. It allocates predictors on a per-callsite basis instead of globally, and frees the resources associated with unused hybrid sub-predictors after an initial warmup period. We find that these techniques dramatically improve speed and reduce memory consumption while maintaining high prediction accuracy.

1. Introduction

Return value prediction (RVP) is a runtime technique for guessing the result of a function, method, or procedure call. It is a specific case of value prediction in general, differentiated by the fact that functions may take arguments, and also by the fact that as the core building block of modularity, functions provide an extremely broad range of behaviour.

Value prediction is typically investigated in a hardware context, where the focus is on providing high accuracy with minimal circuit costs. Software designs are much less common, but can be supported on existing and off-the-shelf machines. Previous work in software value prediction has concentrated on mimicking hardware designs in software. We believe that software value prediction can be useful and is worth exploring in its own right, its relationship to hardware value prediction being analogous to that between software transactional memory and hardware transactional memory. In this work we seek to establish a software state of the art in value prediction by providing a fast, accurate, and memory efficient design and implementation for return value prediction.

The primary advantages of a software implementation are relatively unbounded memory resources, cheap development costs, and high level runtime information. A significant problem we encountered in reviewing the hardware literature was understanding exactly how the existing predictors worked, and how they were related to each other. To this end we developed a unification framework for organizing the various predictors, and created straightforward software implementations of them. We included both space-efficient computational predictors and space-inefficient table-based predictors in our design. We applied these predictors to standard Java benchmarks to measure their return value predictability, as well as the relative accuracy, speed, and memory consumption of individual predictor types.

We next needed a hybrid predictor design to bring together all of the predictors in our framework. Figure 1 shows what a typical implementation of hybrid RVP in hardware might look like. First to make a prediction, a callsite address is hashed to an entry in a primary hashtable. This entry contains the hybrid predictor state, which includes prediction accuracies for individual sub-predictors as well as stateful information they might need, such as a history of return values. The hybrid then selects the best performing sub-predictor to create a prediction. In-place sub-predictors compute a value based directly on the state, whereas table-based sub-predictors hash components of the state to a predicted value in a secondary hashtable. On each prediction, even though only one will be selected, all sub-predictors execute, which in hardware is easily parallelized. When the function returns from the call, sub-predictor correctness becomes known, and the hybrid state and all corresponding table-based predictor entries get updated. The most notable feature for our purposes is that due to hardware constraints, all data structures are fixed-size global singletons.

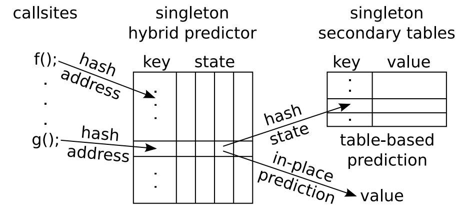

Figure 1. *Hybrid prediction in hardware.*

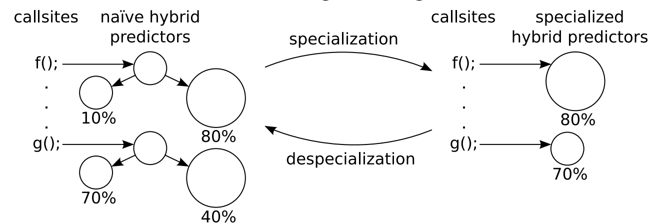

Figure 2. *Hybrid prediction in software (novel).*

Our hybrid design exploits its software context to provide adaptivity, as shown in Figure 2. The first major kind of adaptivity is that a single hybrid predictor instance is associated with each callsite, which allows for scaling according to program size and client usage. Each hybrid has some private state, and each sub-predictor has its own state as well. Importantly, there is no state sharing between sub-predictors. On prediction and update, the hybrids execute and update every sub-predictor. This design can be extended through sub-classing, avoids conflicts, achieves high accuracy, and allows for tables to grow as necessary. The primary disadvantages are that serialized sub-predictor execution leads to high overhead costs and that the memory consumption can be excessive. The second major kind of adaptivity is an attempt to optimize away these costs. After a warmup period, if the accuracy of an individual sub-predictor meets a certain threshold, the hybrid specializes. This frees all other predictor resources, such that prediction and update only access the individual sub-predictor. If accuracy ever drops below a certain threshold, the hybrid despecializes. Thus we maintain accuracy while reducing speed and memory overhead.

There are many potential applications for this technology. Return value prediction was originally conceived to support method level speculation, which executes a function continuation speculatively and in parallel with the function call. RVP significantly improves method level speculation performance in both hardware [11, 17, 30] and software [32] systems. Close to the original motivation of speculative execution, return value prediction could also enhance "safe" futures [45, 46], a source level continuation-based parallelization construct, by allowing for speculation past the consumption of the return value. Aside from certain predictors that take function arguments, there is nothing preventing our design from also being used for more general load value prediction, which has application to both software thread level speculation [29] and transactional memory implementations [31].

More broadly, any instruction that produces a value can be considered a function, and so the technique is readily extended to non-return values. A key analysis in JIT compilers is value profiling, which enables method body specialization according to expected values [10, 42]. Software (return) value prediction could be used to generalize value profiling to support multiple concurrent profiles and hence multiple specializations of a method. A second use of RVP-based profiling we are currently exploring is program understanding, where post-mortem analysis of specific predictor behaviours can provide insight into the run-time behaviour of individual programs and functions. A third use of RVP-based profiling is in software self-healing, which seeks to repair damage from network attacks [25]. Apart from profiling and speculative execution, value prediction can be used to prevent stalls due to memory latencies, both in distributed and multi-core systems [23], and to support prefetching [1]. Finally, outside of programming languages, our fast, accurate, and memory efficient software RVP design could apply throughout the field of machine learning.

1.1 Contributions

We make the following specific contributions:[1]

- A unification framework for specifying and relating predictors to each other based on the patterns they capture. This work clarifies the extant literature, exposes the potential for new predictors, and demonstrates how object-oriented composition can simplify understanding and implementation.
- Several new sub-predictors, including a 2-delta last value predictor, a table-based memoization predictor that hashes together function arguments, and memoization stride and memoization finite context method predictors derived from it.
- An adaptive software hybrid predictor composed of many sub-predictors that dynamically specializes to whichever sub-predictor performs best. Its object-oriented design and implementation enables two unique optimizations. First, it allocates one hybrid predictor instance per prediction point to eliminate conflicts and improve accuracy. Second, it identifies ideal sub-predictors at runtime and specializes at a prediction point granularity, bypassing the execution of unused sub-predictors and actually freeing their associated data structures. The end result is dramatic speed and memory consumption improvements that do not sacrifice high prediction accuracy.
- A software library implementation of return value prediction. This library is open source, portable, modular, and supported by unit tests. We use this library and its built-in profiling to obtain a comprehensive set of speed, memory, and accuracy prediction data for every method invocation over SPEC JVM98, a significant improvement to existing data.

In the next section, we present our predictor unification framework. Section 3 describes our experimental setup, and Section 4 provides an initial performance evaluation. We then develop and apply our adaptive hybrid design in Section 5 to optimize these results. Finally, Section 6 describes related work, followed by conclusions and future work.

[1] Note to reviewers: This paper is a completely rewritten and greatly extended version of a prior workshop paper, which is attached as supplemental material. We first refactored our JVM-based implementation into a software library with much cleaner, object-oriented code. This included removing the overlap between sub-predictor state, which in turn enabled our core hybrid specialization optimization. The unification framework we present is completely new and much more comprehensive with respect to approaches in the hardware value prediction literature, and it doubles the number of sub-predictors under consideration from six to twelve. We also now provide detailed speed, memory, and accuracy results for all predictors, as opposed to just limited memory and accuracy results for a non-specializing naïve hybrid and two table-based sub-predictors.

2. Predictor Unification Framework

A wide variety of value predictors have been proposed, making a basic organization and evaluation essential to our study. In designing a software solution, we abstracted the simplest implementation approach for each predictor, and so discovered many commonalities between predictors that are not immediately apparent in hardware designs. Based on this exploration, we developed a unification framework for value predictors to clarify their intended behaviour and implementation and relate them to each other. This framework also suggested several new predictors.

Tables 1–3 give a structured presentation of a variety of common predictors. These tables organize typical history-based predictor designs, extended predictors that also consume argument state, and composite predictors that contain sub-predictors. In each case we provide an idealized mathematical expression, an example if appropriate, and the stateful data and pseudo-code used to implement the actual predictor. The mathematical expressions illustrate predictor behaviour by showing how the current prediction (v_n) is derived from a history of actual return values ($v_{n-1}, v_{n-2}, \ldots$), as well as current and past function arguments ($args(n), args(n-1), \ldots$). Implementation details include fields for actual state and pseudo-code inside `predict()` and `update()` functions that provide a common predictor interface. `predict()` optionally takes function arguments and returns a new predicted value, while `update()` takes the actual return value and updates internal predictor state. For brevity we use several non-standard but self-explanatory functions in these descriptions. Our unification framework does not include predictors that are unsuitable for return value prediction, nor those that are substantially equivalent to the ones presented here. However, extensions are straightforward, and our experience suggests that all predictors benefit from expression in this form.

History-based predictors. Table 1 contains predictors based only the history of return values for the associated function. We used predictor names as reported in the literature, except for last N stride, which is a local version of the global gDiff predictor [47]. On the left of the table are predictors that derive their prediction from the value history directly, whereas on the right are predictors that use the differences or strides between values in the history. It is useful to think of the stride predictors as derivatives of the value based predictors; the word “differential” chosen by the creators of the differential finite context method predictor is expressing this relationship [16]. This organized division between primary and derivative forms suggests a new 2-delta last value predictor here. We used standard values of $N = 4$ and $C = 5$ in our experimental analysis of the last four predictors in this table.

Argument-based predictors. Return value prediction accuracy can be improved by taking into account function inputs instead of or as well as function outputs. Table 2 contains the predictors that exploit this information, again separated in terms of normal and derivative forms. In each of these cases the `predict()` function now receives the current function arguments as input. In our implementation we disable these predictors for methods that do not take any arguments.

The memoization predictor is a new predictor that behaves like the finite context method predictor but hashes together method arguments instead of a recent history of return values. The predictor name comes from the traditional functional programming technique known as memoization, alternatively function caching, that “skips” pure function execution when the arguments match previously recorded table entries. A key difference from traditional approaches is that memoization based predictions can be incorrect. This makes memoization applicable to all functions that take arguments instead of only the smaller subset of pure, side-effect free functions in a typical object-oriented program. The MS predictor is a simple stride derivative, and MFCM incorporates value history.

Last Value [15] – LV

$$v_n = v_{n-1}$$

Predicts using the last value.
example: $1, 2, 3 \rightarrow 3$
fields: last
predict():
```
  return last;
```
update(value_t rv):
```
  last = rv;
```

2-Delta Last Value (new) – 2DLV

$$v_n = v_{n-i}, \text{ where } i \text{ is the min } i \text{ s.t. } v_{n-i} = v_{n-i-1} \text{ or } v_{n-1} \text{ if no such } i \text{ exists}$$

LV that only updates if the last value is the same twice in a row.
example: $1, 1, 2, 3 \rightarrow 1$
fields: last1, last2
predict():
```
  return last2;
```
update(value_t rv):
```
  if (rv != last1) last1 = rv;
  else last2 = rv;
```

Last N Value [7, 22] – LNV

$$v_n = v_{n-i}, \text{ where } i \leq N \text{ is the min } i \text{ s.t. } v_{n-1} = v_{n-i-1} \text{ or } v_{n-1} \text{ if no such } i \text{ exists}$$

Predicts using the value at some position in the last N values.
example: $1, 2, 3, 1, 2, 3 \rightarrow 1$
fields: values[N], last_correct_pos
predict():
```
  return values[last_correct_pos];
```
update(value_t rv):
```
  last_correct_pos = contains (values, rv) ?
    index_of (rv, values) : 1;
  shift_into (values, rv);
```

Finite Context Method [38, 39] – FCM

$$v_n = v_{n-i}, \text{ where } i \text{ is the min } i \text{ s.t. } v_{n-c} = v_{n-i-c}, \text{ for all } c \leq C \text{ or } 0 \text{ if no such } i \text{ exists}$$

Captures value history patterns of length $C + 1$.
example: $1, 7, 2, 3, 8, 4, 7, 2 \rightarrow 3$ for $C = 2$
fields: key, context[C]
predict():
```
  key = hash (context);
  return lookup (key);
```
update(value_t rv):
```
  store (key, rv);
  shift_into (context, rv);
```

Stride [15] – S

$$v_n = v_{n-1} + (v_{n-1} - v_{n-2})$$

Predicts using the difference between the last two values.
example: $1, 2, 3 \rightarrow 4$
fields: last, stride
predict():
```
  return last + stride;
```
update(value_t rv):
```
  stride = rv - last;
  last = rv;
```

2-Delta Stride [39] – 2DS

$$v_n = v_{n-1} + v_{n-i} - v_{n-i-1}, \text{ where } i \text{ is the min } i \text{ s.t. } v_{n-i} - v_{n-i-1} = v_{n-i-1} - v_{n-i-2} \text{ or } v_{n-1} \text{ if no such } i \text{ exists}$$

S that only updates if the stride is the same twice in a row.
example: $1, 2, 3, 3 \rightarrow 4$
fields: last, stride1, stride2
predict():
```
  return last + stride2;
```
update(value_t rv):
```
  if (rv - last != stride1) stride1 = rv - last;
  else stride2 = rv - last;
  last = rv;
```

Last N Stride [47] – LNS

$$v_n = v_{n-1} + (v_{n-i} - v_{n-i-1}), \text{ where } i \leq N \text{ is the min } i \text{ s.t. } v_{n-1} - v_{n-2} = v_{n-i-1} - v_{n-i-2} \text{ or } v_{n-1} - v_{n-2} \text{ if no such } i \text{ exists}$$

Predicts using the stride at some position in the last N strides.
example: $1, 2, 4, 7, 8, 10, 13 \rightarrow 14$
fields: last, strides[N], last_correct_pos
predict():
```
  return last + strides[last_correct_pos];
```
update(value_t rv):
```
  last_correct_pos = contains (strides, rv - last) ?
    index_of (rv - last, strides) : 1;
  shift_into (values, rv - last);
```

Differential Finite Context Method [16] – DFCM

$$v_n = v_{n-1} + (v_{n-i} - v_{n-i-1}), \text{ where } i \text{ is the min } i \text{ s.t. } v_{n-c} - v_{n-c-1} = v_{n-i-c} - v_{n-i-c-1}, \text{ for all } c \leq C \text{ or } 0 \text{ if no such } i \text{ exists}$$

Captures stride history patterns of length $C + 1$.
example: $1, 6, 9, 11, 16, 19 \rightarrow 21$ for $C = 2$
fields: last, key, context[C]
predict():
```
  key = hash (context);
  return last + lookup (key);
```
update(value_t rv):
```
  store (key, rv);
  shift_into (context, rv - last);
```

Table 1. *History-based predictors.* Hashing and searching functions are not shown; the `shift_into` function adds a new value to an array, shifting all other elements down and removing the oldest element.

The parameter stride predictor identifies a constant difference between the return value and one parameter, and uses this to compute future predictions. A simple example of a function it captures is one that converts lowercase ASCII character codes to alphabet positions. Although the parameter stride predictor is in general subsumed by the memoization predictor, parameter stride is simpler in implementation, warms up very quickly, and requires only constant storage.

Composite predictors. Table 3 contains predictors that are composites of one or more sub-predictors. The hybrid predictor uses the other predictors directly, returning a prediction by the best performing sub-predictor over the last 64 return values, whereas composite stride is in fact a generalized implementation pattern. Each predictor on the right hand side of Table 1 as well as the memoization stride predictor in Table 2, for instance, can be alternatively constructed as a composite stride predictor containing the corresponding predictor on the left hand side. In our implementation we applied this pattern to implement all stride predictors, except the parameter stride predictor which does not follow this pattern because it predicts a constant difference between return value and one parameter. This object-oriented simplification was only realized once we expressed the predictors in this framework.

Memoization (new) – M

$$v_n = v_{n-i}, \text{ where } i \text{ is the min } i \text{ s.t. } \mathit{args}(n) = \mathit{args}(n-i), \text{ or 0 if no such } i \text{ exists}$$

Maps function arguments to return values.

example: $f(1,2,3) = 4,\ f(4,5,6) = 7,\ f(1,2,3) \rightarrow 4$

fields: key

predict(value_t args[]):
```
  key = hash (args);
  return lookup (key);
```
update(value_t rv):
```
  store (key, rv);
```

Memoization Stride (new) – MS

$$v_n = v_{n-1} + (v_{n-i} - v_{n-i-1}), \text{ where } i \text{ is the min } i \text{ s.t. } \mathit{args}(n) = \mathit{args}(n-i), \text{ or 0 if no such } i \text{ exists}$$

Maps function arguments to return value strides.

example: $f(1,2,3) = 4,\ f(1,2,3) = 7,\ f(1,2,3) \rightarrow 10$

fields: key, last

predict(value_t args[]):
```
  key = hash (args);
  return last + lookup (key);
```
update(value_t rv):
```
  store (key, rv);
  last = rv;
```

Memoization Finite Context Method (new) – MFCM

$$v_n = v_{n-i}, \text{ where } i \text{ is the min } i \text{ s.t. } v_{n-C} = v_{n-i-C}, \text{ for all } c \le C, \text{ and } \mathit{args}(n) = \mathit{args}(n-i), \text{ or 0 if no such } i \text{ exists}$$

Maps function arguments × value history to return values.

example: $f(1)=5,\ f(2)=6,\ f(3)=7,$ $f(3)=9,\ f(1)=5,\ f(5)=6,\ f(5)=8,$ $f(1)=5,\ f(2)=6,\ f(3) \rightarrow 7$ for $C = 2$

fields: key, context[C]

predict(value_t args[]):
```
  key = hash (concat (args, context));
  return lookup (key);
```
update(value_t rv):
```
  store (key, rv);
  shift_into (context, rv);
```

Parameter Stride [17] – PS

$$v_n = \mathit{args}(n)[a] + (v_{n-i} - \mathit{args}(n-i)[a]), \text{ where } i \text{ is the min } i \text{ s.t. } v_{n-i} - \mathit{args}(n-i)[a] = v_{n-i-1} - \mathit{args}(n-i-1)[a] \text{ for some argument index } a, \text{ or 0 if no such } i \text{ exists}$$

Identifies a constant offset between one parameter and the return value.

example: $f(\text{‘}r\text{’}) = 17,\ f(\text{‘}v\text{’}) = 21,\ f(\text{‘}p\text{’}) \rightarrow 15$

fields: a = A, old_args[A], strides[A]

predict(value_t args[]):
```
  copy_into (old_args, args);
  return a < A ? args[a] + strides[a] : 0;
```
update(value_t rv):
```
  for (i = A - 1; i >= 0; i--)
    if (rv - old_args[i] == strides[i]) a = i;
    strides[i] = rv - old_args[i];
```

Table 2. *Argument-based predictors.* A differential version of the memoization finite context method predictor would naturally follow from our framework; instead we investigate the parameter stride predictor.

Hybrid [9] (new design) – H

$$v_n = f(v_1, \ldots, v_{n-1}, \mathit{args}(n)), \text{ where } f \text{ is the best performing sub-predictor}$$

Combines many different sub-predictors and identifies the best one.

fields: predictors[], accuracies[], predictions[]

predict(value_t args[]):
```
  for (p = 0; p < P; p++)
    predictions[p] = predictors[p].predict (args);
  return predictions[max_index (accuracies)];
```
update(value_t rv):
```
  for (p = 0; p < P; p++)
    predictors[p].update (rv);
    accuracies[p] = (rv == predictions[p]) ?
      min (accuracies[p] + 1, 64) :
      max (accuracies[p] - 1, 0);
```

Composite Stride (new) – CS

$$s_{n-i} = v_{n-i} - v_{n-i-1},\ \forall\, 2 \le i < n$$
$$s_{n-1} = f(s_1, \ldots, s_{n-2}, \mathit{args}(n-1)), \text{ where } f \text{ is any sub-predictor}$$
$$v_n = v_{n-1} + s_{n-1}$$

Creates a stride derivative of any other predictor.

fields: last, f

predict():
```
  return last + f.predict ();
```
update(value_t rv):
```
  f.update (rv - last);
  last = rv;
```

Table 3. *Composite predictors.* Our software hybrid design is new, but conceptually similar to hardware hybrid designs. The composite stride predictor is a general implementation pattern for converting value predictions into stride predictions, rather than a specific predictor.

3. Experimental Setup

We modified a Java VM to communicate with an object-oriented C software library implementation of every predictor described in Section 2. This library is open source, portable, modular, and supported by unit tests that check for expected predicted behaviour. It currently runs on `x86_64` and `ppc64` architectures. It also includes profiling support, which we used to generate the raw data for our experimental results. At the library core is a map between physical callsite addresses and callsite probe objects. Each probe contains a hybrid predictor instance as well as callsite identification and profiling information. When the Java VM client allocates a non-void callsite during method preparation, it sends the callsite address, class, method, program counter, and target method descriptor to the library in exchange for a reference to a callsite probe object. This reference is used for all subsequent communication to avoid unnecessary table lookups.

We modified the VM to call `predict()` and `update()` RVP functions before and after non-void callsite execution respectively. The former takes method arguments, including any implicit `this` reference, and returns a predicted value, whereas the latter takes the actual return value and updates the predictors associated with the callsite. In the event of escaping exceptions, no update occurs. To minimize VM changes, the library parses arguments from the VM call stack using the target descriptor, zeroing out unused bytes and arranging the arguments contiguously in memory. Internally, the hybrid and all sub-predictors subclass a predictor class with `update()` and `predict()` methods. This design allows for easy composition and hybrid specialization, as described in Section 5.

Benchmarks. We used the SPEC JVM98 benchmarks with input set S100 for experimental evaluation [41]. These benchmarks are not as complex or memory-intensive as the more recent DaCapo benchmarks [2]. However, they are fast to execute, an important factor in performing a large number of experiments, and more than sufficient for a software RVP study as they use over 800 million non-void method calls in the absence of method inlining. Our choice of benchmark suite also directly extends previous work on

RVP for Java, which used the same benchmarks but alternatively ran only the tiny S1 dataset in a restricted hardware context that only considered boolean, int, and reference return types [17], ignored specific predictor behaviour [40], or focused on client application of the results [32].

benchmark	comp	db	jack	javac	jess	mpeg	mtrt
methods	670	714	936	1.51K	1.15K	838	863
callsites	2.48K	2.79K	4.56K	7.20K	4.32K	2.94K	3.71K
invokes (V)	93.4M	54.4M	35.0M	39.9M	23.3M	45.2M	28.4M
invokes (NV)	133M	116M	62.9M	82.3M	102M	65.8M	259M
escapes (V)	0	0	608K	0	0	0	0
escapes (NV)	0	0	68	41.5K	0	0	0
returns (V)	93.4M	54.4M	34.4M	39.9M	23.3M	45.2M	28.4M
returns (NV)	133M	116M	62.9M	82.3M	102M	65.8M	259M
booleans Z	6.70K	11.1M	17.3M	19.5M	35.8M	13.2M	3.07M
bytes B	0	0	580K	39.3K	0	0	0
chars C	8.85K	25.2K	8.53M	3.80M	24.4K	6.96K	20.8K
shorts S	0	0	0	73.0K	0	18.0M	0
ints I	133M	48.1M	17.9M	35.9M	20.7M	34.6M	4.54M
longs J	440	152K	1.23M	818K	100K	15.7K	2.07K
floats F	102	704	296K	104	1.04K	7.82K	162M
doubles D	0	0	0	160	1.77M	56	214K
references R	17.0K	56.2M	17.0M	22.2M	43.5M	24.3K	89.6M

Table 4. *Benchmark properties.* V: void; NV: non-void; escapes: escaping exceptions.

Table 4 presents relevant benchmark properties. The first section shows the number of methods and callsites in the dynamic call graph. In principle, we can associate predictors with methods, callsites, or the invocation edges that join them. We choose here to use callsites exclusively, mostly to limit the scope of our evaluation. Callsites seem like a reasonable choice because they capture the calling context without being type sensitive. In future work, it would be interesting to study how performance differs when methods or invocation edges are used instead.

The second section shows dynamic void and non-void invokes, escapes, and returns. An invoke is a method call, a return is normal method completion, and an escape is abnormal method termination due to an uncaught exception in the callee. We exclude void method calls from our analysis because they do not return values, but present them here for the sake of completeness. We make predictions on all non-void invokes, but only send updates on normal returns, because for escapes there is no return value and control does not return to the callsite. We thus report accuracy measures over the total number of non-void returns. As the data show, escaping exceptions are relatively rare, even for supposedly exception-heavy benchmarks such as `jack`, which means they do not have a large impact in any case.

The third section classifies non-void returns according to the nine Java primitive types. Return type information is interesting because some types are inherently more predictable than other types, suggesting specialization and compression strategies, and because it describes behaviour to some extent. We see that `mtrt` relies heavily on float methods, `mpegaudio` uses a surprising number of methods returning shorts, `compress` returns almost exclusively ints, and the remaining benchmarks use more or less equal mixes of int, boolean, and reference calls.

4. Initial Performance Evaluation

We used our software library implementation of the predictors in Section 2 to measure their accuracy, speed, and memory consumption performance over our benchmark suite. Knowing the performance characteristics of individual predictors can help when given a constrained resource budget. We expect the more complex predictors to have better accuracy but with higher speed and memory costs. The naïve hybrid predictor we study here does not specialize, visiting every sub-predictor on each call to `predict()` and `update()`. The next section contains a detailed exploration of adaptivity.

It is important to keep in mind while considering these results that a prediction is made for every single invocation in the program and that there is no inlining. We chose this approach to gather the most comprehensive set of data possible and to make our study generally useful, because different clients of RVP will invariably make different decisions about where to predict. Individual callsite prediction accuracies and overhead costs differ widely, which means that disabling prediction selectively can significantly affect the results. The actual runtime speed and memory costs in any practical scenario will scale with usage. This scaling effect is most notable in `mtrt`, which incurs significantly more overhead than the other benchmarks due to its high call density.

Accuracy. Figure 3 shows basic prediction accuracies for each predictor and for each benchmark. Accuracy is calculated as the number of correct predictions over the number of non-void calls that returned to their callsite. The benchmarks are clustered in alphabetical order and the predictors arranged in the order given by Tables 1–3. For comparison we have included as the first predictor a null predictor (N) that simply returns 0 for every prediction.

As expected, the hybrid beats individual predictor accuracies for every benchmark because it allows sub-predictors to complement each other. Accuracy otherwise scales roughly with complexity, at least for the non-memoization predictors. A basic last value predictor significantly improves on a null predictor, is in turn improved on by last N predictors, which themselves are overshadowed by context-based designs. Interestingly the stride versions of non-context predictors do not show significant differences from the last value predictors, suggesting that extending the predictors to higher level derivative forms does not necessarily improve accuracy. Including value history context has a significant impact. The finite context method and its differential form have the highest individual predictor accuracies, and even memoization is noticeably improved by adding value history. Argument based approaches are not as successful as history based approaches in isolation, but as we show later memoization can complement the FCM and DFCM predictors nicely in a hybrid.

Interesting differences also show up in terms of benchmark behaviour. `db`, `jack`, `javac`, and `jess` respond well overall, with even simple predictors reaching 40–60% accuracy levels. `mtrt` and `mpegaudio` are more resilient to prediction, due to their use of more irregular floating point types. `compress` improves dramatically with table-based prediction, indicating longer term patterns exist, even if `mpegaudio` and `compress` are naturally expected to be less predictable since they handle compressed data.

Speed. Figure 4 shows slowdowns due to predictor overhead for each predictor and for each benchmark. Slowdown is calculated as predictor performance relative to a null predictor, factoring out any overhead inherent in our experimental setup. The graph is structured similarly to Figure 3, although on a logarithmic scale. As expected, predictor speeds vary with complexity, with the table-based predictors being considerably slower than the fixed-space predictors. The table-based predictors are expensive for two reasons. First, hashing arguments or return value histories to table lookup keys is an expensive operation. Second, the memory requirements of the larger tables introduce performance penalties due to memory hierarchy latencies. The naïve hybrid is unsuprisingly very slow, incurring the summed cost of all sub-predictors.

Memory consumption. The memory consumption of each predictor for each benchmark is shown in Table 5. The memory requirements of the fixed-space predictors are calculated by summing the number of bytes used by each predictor and multiplying by the number of callsites. The table-based predictor memory requirements are calculated in the same manner for the fixed-space fields, and then the actual final sizes of the hashtables at individual callsites upon program completion are used to calculate the variable-sized fields. The main observation here is that the table-based predictors can consume large amounts of memory, and that

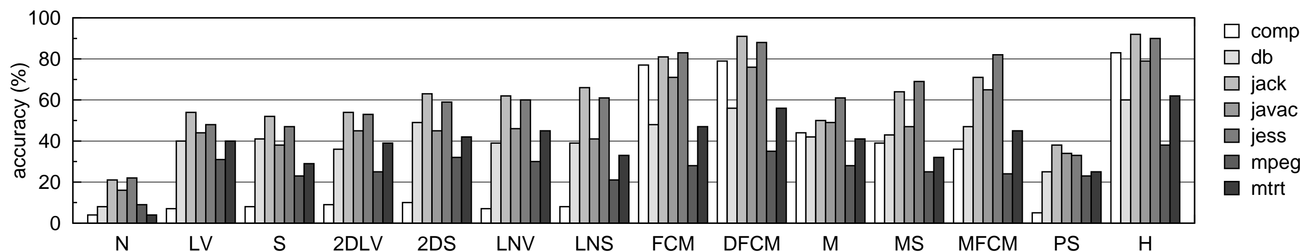

Figure 3. *Predictor accuracies for a null predictor (N) and all predictors in Tables 1–3.*

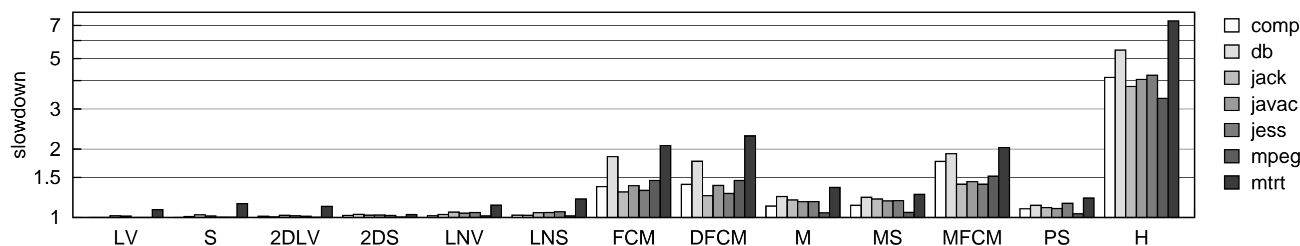

Figure 4. *Predictor slowdowns.*

predictor	comp	db	jack	javac	jess	mpeg	mtrt
N	4.67K	5.23K	10.5K	20.9K	10.1K	6.08K	11.0K
LV	9.34K	10.5K	21.0K	41.7K	20.2K	12.2K	21.9K
S	18.7K	20.9K	42.0K	83.4K	40.4K	24.3K	43.9K
2DLV	14.0K	15.7K	31.5K	62.6K	30.3K	18.2K	32.9K
2DS	23.4K	26.1K	52.5K	104K	50.5K	30.4K	54.8K
LNV	23.9K	26.8K	53.8K	107K	51.7K	31.2K	56.2K
LNS	33.3K	37.2K	74.8K	149K	71.9K	43.3K	78.2K
FCM	625M	0.97G	50.7M	205M	14.6M	1.61G	2.97G
DFCM	673M	784M	7.26M	197M	10.1M	1.60G	3.31G
M	6.81M	99M	7.75M	1.51M	4.03M	25.4M	7.19M
MS	6.82M	99M	7.77M	1.55M	4.05M	25.5M	7.21M
MFCM	31.1M	893M	16.6M	4.79M	13.4M	1.72G	80.6M
PS	12.4K	13.8K	29.5K	59.6K	26.9K	16.2K	28.0K
H	1.31G	2.80G	90.9M	411M	47.1M	4.98G	6.37G

Table 5. *Predictor memory consumption.*

this effect is compounded in the hybrid that has five table-based sub-predictors at each callsite. These data confirm that memory latencies are likely to contribute to predictor slowdowns for table-based prediction.

The data in Table 5 and Figures 3 and 4 assume hashtable sizes are unbounded, and so the tables grow as necessary to accommodate new values. This is obviously unrealistic, but if the sizes are bounded then new values overwrite old values once the maximum size is reached, which reduces overall accuracy if the old value is ever requested. Predictor accuracy as a function of maximum table size is thus shown in Figure 8. Here maximum table sizes are varied from 2^0 to 2^{25} entries, one power of 2 larger than the largest size any predictor was observed to expand to naturally, and accuracy examined for each table predictor and benchmark combination. In general, accuracy increases as table size increases, although only up to a point. After this point accuracy remains mostly constant, indicating no further impact from collisions, and in some cases may actually decrease due to the absence of lucky collisions that returned a correct value at smaller sizes.

Figure 8 also indicates that individual predictors can have complex interactions in a hybrid. For a given benchmark and table size, individual predictors often have noticeably different performance: memoization (stride) may work well in some instances whereas the (differential) finite context method works well in others. Interestingly, although the context predictors usually have the highest accuracies, the predictor complementation provided by the hybrid predictor can be seen in the shapes of the curves for individual benchmarks. The hybrid behaviour for `compress`, `jack`, `javac`, and `jess`, for example, combines the better accuracy of M(S) designs at low table sizes with the higher accuracy of (D)FCM at higher sizes.

5. Hybrid Adaptivity

The naïve hybrid design in Table 3 achieves very high accuracy. However, its speed suffers because it employs twelve different sub-predictors in series to make and update predictions, and its memory consumption suffers because it retains the memory for large table-based predictors even if they are never selected for prediction. We would like to maintain this high accuracy while optimizing for speed and memory consumption. We do this by specializing individual hybrid instances to particular sub-predictors and releasing the resources required by the other unused sub-predictors. This optimization relies on an important hypothesis: *for a given callsite, there is likely to be an ideal sub-predictor*.

We first tested this hypothesis with an offline profiling based experiment to identify ideal sub-predictors on a per-callsite basis. The ideal sub-predictor for a callsite is simply the one that performed best over the entire course of execution. If a subsequent run in which the hybrid immediately specializes to these predictors matches the accuracy of the naïve version, then it indicates that ideal sub-predictors are likely to exist. The performance of this offline hybrid can then provide an oracle for online optimization.

Offline specialization. We first ran each benchmark to completion using the naïve predictor, and processed the results to create a profile for offline specialization. Figure 9 shows the distribution of ideal predictors for each benchmark in terms of dynamically reached callsites and the number of dynamic calls. At the callsite level, most ideal predictors are null or last value predictors. In this analysis, cold callsites with one call are weighted equally with hot callsites that have 50 million calls, and they tend to specialize to simple predictors. Most of these cold callsites are found in initialization code, and there is simply no chance for sufficient history to develop such that the more complex predictors outperform the simple ones.

At the level of actual calls, the simple predictors still work well in many cases, particularly for methods returning constants or accessing static data structures. However, hot callsites tend to benefit from complex table-predictors predictors, indicating an important role for them in maximizing accuracy. This reconfirms the result in

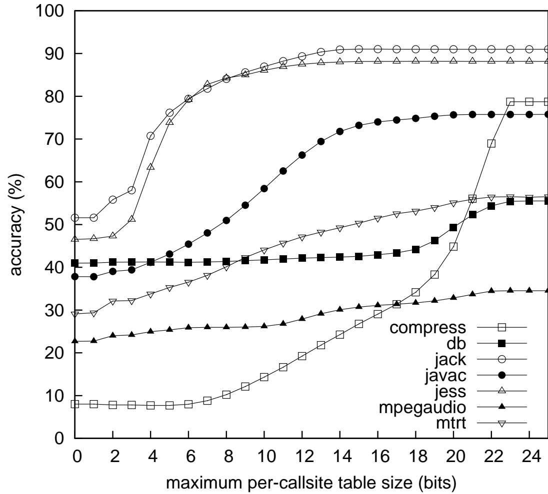

Figure 5. Differential finite context method (DFCM)

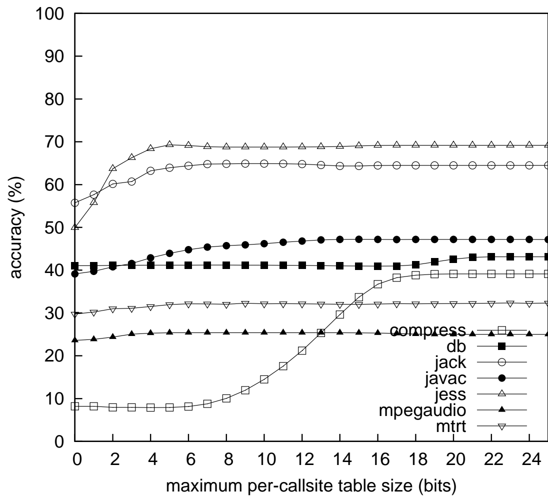

Figure 6. Memoization stride (MS)

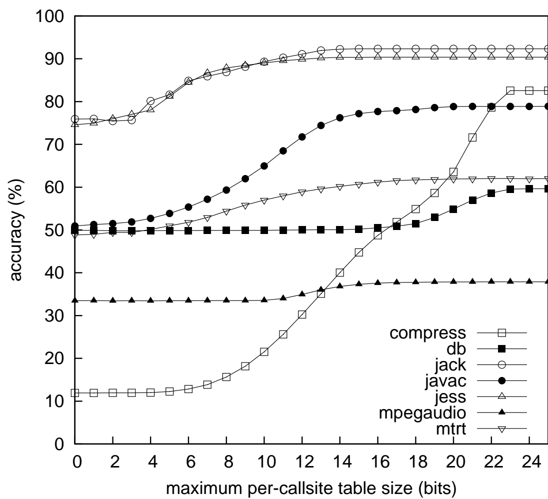

Figure 7. Hybrid (H)

Figure 8. *Predictor accuracy vs. maximum table size (Figures 5–7).* Results for the FCM, M, and MFCM predictors are omitted for space reasons. They generally perform worse than DFCM and MS.

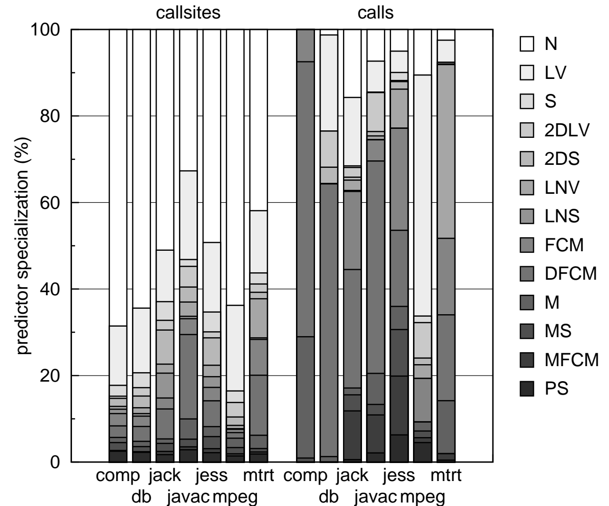

Figure 9. *Ideal predictor distributions.*

Figure 7, where a low cap on table size in the hybrid predictor can suppress accuracy significantly. `mpegaudio` provides a notable exception to the dominance of table predictors. It decodes an mp3 file, and so its return values are mostly random. It has very low overall predictability, and when there is repetition it is generally found in the last few values, meaning that simple predictors dominate.

Online specialization. We next attempted to determine ideal sub-predictors dynamically, without ahead-of-time profiling data. Online adaptivity is critical in dynamic compilation environments, where ahead-of-time techniques are not well accepted in practice. In this case online specialization can also accommodate callsites that exhibit phase-like behaviour, where the ideal sub-predictor is not constant throughout the program run.

There are three basic parameters we considered in constructing our online specializing hybrid. The first is a warmup period, w. The hybrid predictor will not specialize until $u \geq w$, where u is the number of predictor updates. The second is a confidence threshold for specialization, s. For the number of correct predictions c over the last n calls, if $c \geq s \wedge u \geq w$ then the hybrid specializes to the best performing sub-predictor, favouring cheaper predictors in the event of ties. We use a value of $n = 64$, the number of bits in a word on our machines. The third parameter is a confidence threshold for despecialization, d. If $c < d$ and the hybrid has already specialized, then it will despecialize again. We did not experiment with resetting the warmup period upon despecialization, although this could be a useful extension.

We performed a parameter sweep over w, s, d according to Figure 10. This generated 360 different experiments. For each, the average accuracy and slowdown were computed. The average accuracies were rounded to the nearest integer, and the minimum running time for each accuracy identified. These results are shown in Figure 11.

From these data, we selected the point at accuracy 67% with slowdown of 1.35x for use in future experiments. This choice is 5% worse than the optimal accuracy at 72% with slowdown of 2.40x. At this point, $\{W, S, D\} = \{3, 2, 0\}$, which corresponds to a warmup of $w = 512$ returns, specialization threshold of $s = 16$ correct predictions (25% accuracy), and a despecialization threshold of $d = 0$, meaning no despecialization will occur. The cheapest configuration of $\{-1, 0, 0\}$ is equivalent to the null predictor and only achieves an accuracy of 12%.

The data point at accuracy 61% with slowdown 1.74x also stands out. The corresponding configuration, $\{-1, 8, 0\}$, means that $w = 0$, $s = 64$, and $d = 0$. This predictor has no warmup, nor does it despecialize, and it is quite slow. It was selected by the optimization for that data point for two reasons. First, its high

specialization threshold did ultimately result in some good sub-predictor choices. Second, there were only three configurations to choose from at that accuracy level, because the distribution of experiments is not even along the x-axis and most experiments cluster in the upper accuracy range. Interestingly, in all but the top three most accurate and slowest cases, $d = 0$. We conclude that although slight accuracy benefits from despecialization may exist, they come with sharply increasing costs.

Performance comparisons. We finally compared the behaviour of our offline and online adaptive hybrids with the naïve non-adapative hybrid. We show predictor accuracies, slowdowns, and memory consumption for all three in Figures 12 and 13 and Table 14 respectively. We used a maximum table size of 2^{25} entries in these experiments to prevent memory constraints from interfering with accuracy results.

In terms of accuracy, we expected the naïve hybrid to act as an oracle with respect to the offline hybrid, behaving like the online hybrid but configured with an infinite warmup period. The data in Figure 12 show that offline specialization is quite effective, usually within a few percent of the naïve version. In some cases the accuracy is actually slightly better, because the constant availability of all predictors in the naïve version can lead to suboptimal choices. The close match between offline and naïve accuracies indicates two things. First, ideal sub-predictors do in fact exist for the vast majority of callsites. Second, for these benchmarks, significant program phases are either rare or non-critical with respect to adaptive RVP performance, because the offline hybrid uses a fixed set of sub-predictors over the entire program run.

Accuracy is not significantly compromised in the online hybrid, and is within 5–10% of offline accuracy for most benchmarks. However, `compress` performs significantly less well than the others. Investigation revealed that this difference is due to differences in the chosen sub-predictors for a few callsites, in particular the `getbyte()I` call in `Compress.compress()` which gets executed over 47 million times. Here the offline hybrid chooses a DFCM predictor with 79% accuracy, whereas the online hybrid specializes too early, selecting a null predictor that results in less than 10% accuracy overall. We could potentially remedy this problem by performing a more refined parameter sweep over `compress`.

Predictor slowdowns are dramatically reduced by both offline and online hybrids, as shown in Figure 13. Online is actually better than offline for most benchmarks, because the offline hybrid tends to choose accurate but expensive table-based predictors, while sub-optimal specialization in the online hybrid favours predictors with less state and thus less warmup. This effect can also be seen in the memory consumption data in Table 14. Both offline and online hybrids greatly reduce memory requirements, and in the case of offline `mpeg` by over 24 times. Online memory usage tends to be even smaller, with `db` providing an extreme example where the online hybrid is orders of magnitude cheaper. The bottom half of Table 14 shows the further memory reductions that straightforward elimination of wasteful memory use in our system would provide.

6. Related Work

Return value prediction is a kind of value prediction, a technique which has been researched for well over a decade, primarily in the context of novel hardware designs. A wide variety of value predictors have been proposed and examined, including simple computational predictors, more complex table-based predictors, machine learning based predictors, and hybrid implementations. Our work here extends existing investigations of RVP in a Java context [11, 17, 32, 40] with practical explorations of accuracy, speed, and memory consumption in an adaptive, dynamic software-only environment, and our unification framework brings together many known value predictors that are suitable for RVP.

Burtscher *et al.* provide a good overview of basic value prediction techniques [6]. As a general rule, accommodating more

for $W \leftarrow -1$ **to** 6 **do**
  for $S \leftarrow 0$ **to** 8 **do**
    for $D \leftarrow 0$ **to** S **do**
      if $W = -1$ **then** $w \leftarrow 0$ **else** $w \leftarrow 2^{3W}$
      $s \leftarrow 8S$
      $d \leftarrow 8D$
      measure (w, s, d)

Figure 10. *Online hybrid parameter sweep configuration.*

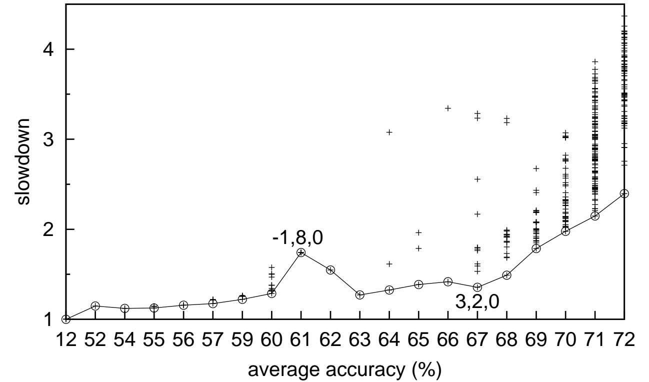

Figure 11. *Online hybrid parameter sweep.*

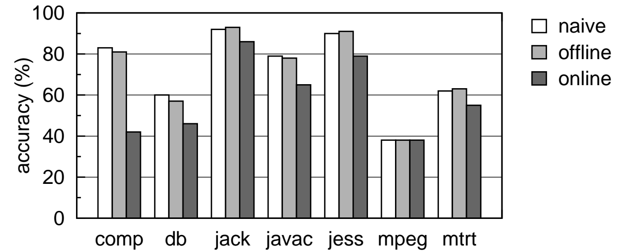

Figure 12. *Naïve vs. offline vs. online accuracies.*

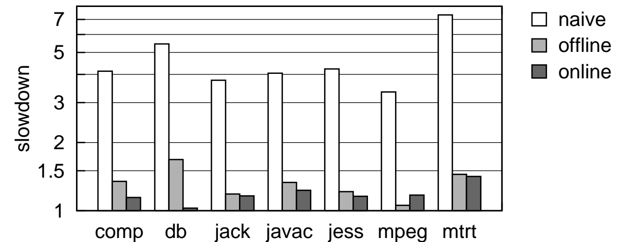

Figure 13. *Naïve vs. offline vs. online slowdowns.*

predictor	comp	db	jack	javac	jess	mpeg	mtrt
naïve	1.31G	2.80G	91.0M	412M	47.2M	4.98G	6.37G
offline	484M	771M	5.83M	190M	6.11M	206M	417M
online	197M	1.89M	5.56M	40.9M	5.23M	252M	252M
no logs	131M	1.41M	3.97M	27.6M	3.75M	168M	168M
32-bit keys	99M	1.22M	3.27M	21.1M	3.10M	127M	126M
type info	65.9M	1.00M	2.46M	14.5M	2.66M	84.7M	85.1M
perfect Z	65.9M	0.98M	2.42M	14.4M	2.63M	84.6M	85.1M

Figure 14. *Naïve vs. offline vs. online memory consumption.* The four additional rows indicate the cumulative memory consumption benefits due to removing a backing log from hash tables, using 32-bit table keys instead of 64-bit keys, using VM knowledge about type widths, and using perfect hashing for booleans in the context-based predictors. Perfect boolean hashing means that an order-5 context-based predictor only requires 5 bytes, 1 byte to hold the 5-bit context and 4 bytes to hold the $2^5 = 32$ possible values.

patterns and using more historical information can improve prediction accuracy, and generalizations of simple predictors, such as last N value prediction, have been studied by a number of groups [7, 22, 44]. Last N value prediction allows for short, repetitive sequences to be captured, and can yield good results; Burtscher and Zorn, for example, show a space-efficient last 4 value predictor can outperform other more complex designs [7]. Zhou *et al.* later provided the gDiff predictor, which is a global version of our last N stride predictor. Most predictors can be further improved by incorporating statistical measures such as formal confidence estimates, although this does add extra complexity [4].

Gabbay introduced the stride predictor and last value predictor, as well as several more specialized predictors, such as the sign-exponent-fraction (SEF) and register-file predictors [15]. Specialized predictor designs provide further ways to exploit value prediction where more general approaches work poorly. The SEF predictor, for instance, predicts the sign, exponent, and fraction parts of a floating point number separately. Although the sign and exponent are often highly predictable, the fraction is not, which usually results in poor prediction accuracy for floating point data. Tullsen and Seng extended Gabbay's register-file predictor to a more general register value predictor. It predicts whether the value to be loaded by an instruction into a register is already present in that register [43]. It may be worth considering a stack top predictor that is simply a register value predictor specialized for return values.

Pointer-specific prediction is also possible, an example being the address-value delta (AVD) prediction introduced by Mutlu *et al.* that predicts whether the difference between an address and the value at that address for a given pointer load instruction is stable [28]. Marcuello *et al.,* propose an increment-based value predictor [27] for value prediction within a speculative multithreading architecture. This predictor is like the 2-delta stride load value predictor, but is further differentiated by computing the storage location value stride between two different instruction address contexts.

Sazeides and Smith examine the predictability of data values produced by different instructions. They consider hardware implementations of last value, stride, and context predictors, showing the limits of predicability and the relative performance of context and computational predictors [39]. Subsequent work considers the practical impact of hardware resource (table size) constraints on predictability [38]. Goeman *et al.* proposed the *differential* finite context method predictor [16] as a way of further improving prediction accuracy. Burtscher later suggested an improved DFCM index or hash function that makes better use of the table structures [5]. We use Jenkins' fast hash to compute hash values because it is appropriate for software [19].

Hybrid designs allow predictors to be combined, complementing and in some cases reinforcing the behaviour of individual sub-predictors. Wang and Franklin show that a hybrid value predictor achieves higher accuracy than its component sub-predictors in isolation [44]. The interaction of sub-predictors can be complex, and Burtscher and Zorn show that resource sharing as well as the impact of how the hybrid selects the best sub-predictor can significantly affect performance [9]. Designs have thus been proposed to reduce hybrid storage requirements [8], and to use selection mechanisms that reduce inappropriate bias, such as cycling between sub-predictors [36], or the use of improved confidence estimators [18]. Sam and Burtscher argue that complex value predictors are not always necessary in optimal hybrid designs that maximize the efficiency of client applications [35].

Software value prediction, while less common, has also been investigated, usually in conjunction with a hardware design. For instance, Li *et al.,* use static program analysis to identify value dependencies that may affect speculative execution of loop bodies, and apply selective profiling to monitor the behaviour of these variables at runtime [20]. The resulting profile is used to customize predictor code generation for an optimized, subsequent execution [13, 21]. Liu *et al.* incorporated software value prediction in their POSH compiler for speculative multithreading and found a beneficial impact on performance [24]. The predictors are similar to those used by Li *et al.* [20], and handle return values, loop induction variables, and some loop variables. Hybrid approaches have also been proposed, combining software with simplified hardware components in order to reduce hardware costs [3, 14]. Performance can also be improved through software analysis, for example by statically estimating predictability [6]. Quiñones *et al.* developed the Mitosis compiler for speculative multithreading that relies on pre-computation slices for child threads, predicting thread inputs in software but performing the speculation in hardware [33].

Return value prediction is a basic component of method level speculation (MLS), and even simple last value and stride predictors have a large impact on speculative performance [11, 30]. Hu *et al.* introduced the parameter stride predictor in a hardware study that made a strong case for the importance of return value prediction in MLS [17]. Pickett and Verbrugge provided a software implementation of MLS that showed RVP had a beneficial impact on performance in a relative sense, but contributed to overall system slowdowns in an absolute sense [32]. Theoretical limits on RVP have also been considered: Singer and Brown applied information theory to analyse the predictability of return values in Java, independent of any specific predictor design [40]. Finally, several of our new predictor designs are based on memoization, particularly suitable for RVP. Memoization is obviously a well known technique, and has been used for both compiler and runtime optimizations [12].

Type information is another vector for optimizing performance. Sato and Arita show that data value widths can be exploited to reduce predictor size; by focusing on only smaller bit-width values accuracy is preserved at less cost [37]. Loh demonstrates both memory and power savings by using data width information [26], although the hardware context requires heuristic discovery of high level type knowledge. Sam and Burtscher later show that hardware type information can be efficiently used to reduce predictor size [34]. They also demonstrate that more complex and hence more accurate predictors have a worse energy-performance trade-off than simpler predictors and are thus unlikely to be implemented in hardware [35].

7. Conclusions and Future Work

The ideal choice of return value predictor varies widely, depending on dynamic benchmark and callsite properties. A flexible, software-based design for RVP thus has many advantages, permitting a wide variety of arbitrarily complex predictors and an adaptive mechanism for optimizing their application. The latter is especially important for software implementations, where a naïve design imposes memory and speed overheads that can easily outweigh any derived benefit. We found that using a variety of callsite-bound predictors that include complex, table-based predictors can result in very high accuracy. Our online adaptive hybrid is effective at maintaining this accuracy while reducing overhead costs to reasonable levels. It does so by identifying and specializing to ideal sub-predictors, which we found do generally exist at the callsite level. If the total runtime overhead of ubiquitous RVP in this study remains a concern, applications can easily tailor their usage to reduce it.

Our software-only focus played an important role in this work. The search for a simple hierarchical design led to the high level specialization optimization in our adaptive hybrid predictor, which suggests that clean design and object-orientation stand to benefit software analogues of hardware components in general. We found that after many years of research, history-based prediction studies covered the design space rather well, missing only the 2 delta last value predictor. This suggests that early attempts to formalize the design of runtime components may be beneficial. For example, our composite stride pattern makes it easy to create stride based derivatives of any predictor. Our software context allowed us to consider a large number of sub-predictors at low cost, and we found that they all had application at different points. Memoization

is particularly effective when applied to RVP, and complements existing predictors nicely in a hybrid.

For future work, we outlined the many potential applications of RVP in the introduction. With respect to performance, accuracy could be improved by identifying hot but unpredictable callsites and designing new predictors to accommodate them. Attaching predictors to methods and invocation edges instead of callsites may alternatively improve accuracy or reduce overhead. Various static analyses and program transformations to support prediction are also possible, building on previous work in this area [6]. Finally, generalized software value prediction using our framework may benefit from several predictors not suitable for return values.

In terms of implementation, a mixture of software and hardware support may be appropriate [3]. Our design could certainly accommodate hardware versions of specific sub-predictors when available. Furthermore, a general purpose hardware hash function could improve the performance of table-based predictors, and have broad applicability outside of value prediction. Finally, we think that a JIT compiler integration of RVP which weaves predictors into the generated code may be worthwhile. We are particularly interested in the impact of JIT compiler method inlining on predictor behaviour.

References

[1] Enhancing memory-level parallelism via recovery-free value prediction. *TC*, 54(7):897–912, July 2005. Huiyang Zhou and Thomas M. Conte.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, Oct. 2006.

[3] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *ASPLOS XIII*, pages 157–167, Mar. 2008.

[4] M. Burtscher. *Improving Context-Based Load Value Prediction*. PhD thesis, Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, USA, Apr. 2000.

[5] M. Burtscher. An improved index function for (D)FCM predictors. *Comp. Arch. News*, 30(3):19–24, June 2002.

[6] M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *PLDI'02*, pages 222–233, June 2002.

[7] M. Burtscher and B. G. Zorn. Exploring last n value prediction. In *PACT'99*, pages 66–77, Oct. 1999.

[8] M. Burtscher and B. G. Zorn. Hybridizing and coalescing load value predictors. In *ICCD'00*, pages 81–92, Sept. 2000.

[9] M. Burtscher and B. G. Zorn. Hybrid load-value predictors. *TC*, 51(7):759–774, July 2002.

[10] B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO 30*, pages 259–269, Dec. 1997.

[11] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98*, pages 176–184, Oct. 1998.

[12] Y. Ding and Z. Li. A compiler scheme for reusing intermediate computation results. In *CGO'04*, page 279. IEEE Computer Society, Mar. 2004.

[13] 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 *PLDI'04*, pages 71–81, June 2004.

[14] C.-Y. Fu. *Compiler-Driven Value Speculation Scheduling*. PhD thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, USA, May 2001.

[15] F. Gabbay. Speculative execution based on value prediction. Technical Report 1080, Electrical Engineering Department, Technion – Israel Institute of Technology, Haifa, Israel, Nov. 1996.

[16] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA'01*, pages 207–216, Jan. 2001.

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

[18] S. J. Jackson and M. Burtscher. Self optimizing finite state machines for confidence estimators. In *WISA'06*, Feb. 2006.

[19] B. Jenkins. A hash function for hash table lookup. *Dr. Dobb's Journal*, Sept. 1997.

[20] X.-F. Li, Z.-H. Du, Q. Zhao, , and T.-F. Ngai. Software value prediction for speculative parallel threaded computations. In *VPW1*, pages 18–25, San Diego, CA, June 2003.

[21] X.-F. Li, C. Yang, Z.-H. Du, and T.-F. Ngai. Exploiting thread-level speculative parallelism with software value prediction. In *ACSAC'05*, volume 3740 of *LNCS*, pages 367–388, Oct. 2005.

[22] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29*, pages 226–237, Dec. 1996.

[23] S. Liu and J.-L. Gaudiot. Potential impact of value prediction on communication in many-core architectures. *TC*, 58(6):759–769, June 2009.

[24] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS compiler that exploits program structure. In *PPoPP'06*, pages 158–167, Mar. 2006.

[25] M. E. Locasto, A. Stavrou, G. F. Cretu, A. D. Keromytis, and S. J. Stolfo. Return value predictability profiles for self-healing. In *IWSEC'08*, volume 5312 of *LNCS*, pages 152–166, Nov. 2008.

[26] G. H. Loh. Width-partitioned load value predictors. *JILP*, 5:1–23, Nov. 2003.

[27] P. Marcuello, A. González, and J. Tubella. Thread partitioning and value prediction for exploiting speculative thread-level parallelism. *TC*, 53(2):114–125, Feb. 2004.

[28] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: A hardware technique for efficiently parallelizing dependent cache misses. *TC*, 55(12):1491–1508, Dec. 2006.

[29] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE'08*, pages 23–32, May 2008.

[30] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT'99*, pages 303–313, Oct. 1999.

[31] S. M. Pant and G. T. Byrd. Extending concurrency of transactional memory programs by using value prediction. In *CF'09*, pages 11–20, May 2009.

[32] C. J. F. Pickett and C. Verbrugge. SableSpMT: A software framework for analysing speculative multithreading in Java. In *PASTE'05*, pages 59–66, Sept. 2005.

[33] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI'05*, pages 269–279, June 2005.

[34] N. B. Sam and M. Burtscher. Exploiting type information in load-value predictors. In *VPW2*, pages 32–39, Oct. 2004.

[35] N. B. Sam and M. Burtscher. Complex load-value predictors: Why we need not bother. In *WDDD'05:*, pages 16–24, June 2005.

[36] N. B. Sam and M. Burtscher. Improving memory system performance with energy-efficient value speculation. *CAN*, 33(4):121–127, Sept. 2005.

[37] T. Sato and I. Arita. Table size reduction for data value predictors by exploiting narrow width values. In *ICS'00*, pages 196–205, May 2000.

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

[39] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO 30*, pages 248–258, Dec. 1997.

[40] J. Singer and G. Brown. Return value prediction meets information theory. In *QAPL'06*, volume 164 of *ENTCS*, pages 137–151, Oct. 2006.

[41] Standard Performance Evaluation Corporation. SPEC JVM Client98 benchmark suite, June 1998. `http://www.spec.org/jvm98/`.

[42] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA'01*, pages 180–195, Oct. 2001.

[43] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *ISCA'99*, pages 270–279, May 1999.

[44] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *MICRO 30*, pages 281–290, Dec. 1997.

[45] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05*, pages 439–453, Oct. 2005.

[46] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in Java. In *PACT'07*, pages 130–139, Sept. 2007.

[47] H. Zhou, J. Flanagan, and T. M. Conte. Detecting global stride locality in value streams. In *ISCA'03*, pages 324–335, June 2003.