McGill University
School of Computer Science
Sable Research Group

Adaptive Software Return Value Prediction

Sable Technical Report No. 2009-1

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

June 5th, 2009

www.sable.mcgill.ca

Adaptive Software Return Value Prediction

Christopher J. F. Pickett
Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada
{cpicke,clump}@sable.mcgill.ca

Allan Kielstra
IBM Toronto Lab
Markham, Ontario, Canada
kielstra@ca.ibm.com

Abstract

Return value prediction (RVP) is a useful technique that enables a number of program optimizations and analyses. Potentially high overhead, however, as well as a dependence on novel hardware support remain significant barriers to method level speculation and other applications that depend on low cost, high accuracy RVP. Here we investigate the feasibility of software-based RVP through a comprehensive software study of RVP behaviour. We develop a structured framework for RVP design and use it to experimentally analyze existing and novel predictors in the context of non-trivial Java programs. As well as measuring accuracy, time, and memory overhead, we show that an object-oriented adaptive hybrid predictor model can significantly improve performance while maintaining high accuracy levels. We consider the impact on speculative parallelism, and show further application of RVP to program understanding. Our results suggest that software return value prediction can play a practical role in further program optimization and analysis.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Procedures, functions, and subroutines; D.2.8 [*Software Engineering*]: Metrics—Complexity measures; Performance measures; D.3.4 [*Programming Languages*]: Processors—Code generation; Compilers; Interpreters; Optimization; Run-time environments; D.4.1 [*Operating Systems*]: Process Management—Concurrency; Threads

General Terms Design, Experimentation, Languages, Measurement, Performance

Keywords adaptive optimization, memoization, method level speculation, program understanding, return value prediction

Submitted to OOPSLA'09 on March 23rd, 2009. Rejected on May 10th, 2009. Please note that our experimental setup treated unconsumed callsites identified by the return value use analysis described in SABLE-TR-2004-6 [32] as void for the purposes of return value prediction. This was unintentional and it perturbed the results slightly.

1. Introduction

Return value prediction, (RVP) and the more general purpose *value prediction,* are techniques which allow a program to guess the result of a method call or computation prior to actual execution. A variety of speculative optimizations are thus enabled, with their success and practical value depending on the accuracy and relative overhead of the prediction system. Many approaches to (R)VP of varying complexity and accuracy have been proposed; the majority, however, focus on hardware designs as the primary means of reducing overhead costs, limiting application of RVP to novel architectures. Software approaches to value prediction remove this barrier, allowing greater and more portable use of value prediction data in optimization and analysis, but require careful optimization and understanding of predictor performance in order to ensure practical efficiency.

In this paper we perform a comprehensive study of software RVP in a Java context. We first consider various kinds of basic predictors presented in the literature, presenting an organizational framework that suggests several further novel predictor designs. This includes both simple, resource-limited designs and more complex, table-based predictors that have significant resource requirements. The behaviour of these individual predictors gives a core understanding of the (return value) predictability of non-trivial Java programs, as well as the relative effectiveness, cost and accuracy of predictor types.

We use this data to justify the design of a *hybrid* predictor [10], a composite predictor that adaptively applies and selects individual sub-predictors for performance and accuracy. Either through offline profiles, or from actual runtime adaptation, a hybrid can optimize RVP performance, dramatically reducing overhead while maintaining a very high accuracy. We show that accuracies of up to 80% can feasibly be achieved on standard benchmarks, at a small fraction of the cost of a non-adaptive hybrid model.

A deeper measure of success is in terms of the effect on applications of RVP. Such applications include exposing further speculation opportunities with concurrency constructs such as “safe” futures [48, 50], increasing the ac-

curacy of run-ahead prefetching [1], and aiding software self-healing [26]. RVP may also contribute to profile oriented non-speculative compiler optimizations. We consider the effects on method level speculation and program understanding. Method level speculation is perhaps the most well-known consumer of RVP data, and benefits from greater parallelism exposed by the longer thread lengths enabled by successful prediction. For the latter application, predictor selection correlates with program behaviour, with the distribution of ideal sub-predictors within an adapting hybrid suggesting specific behaviours.

Our study takes advantage of and considers its software context at several levels. The higher level of abstraction provided by a software approach simplifies the design of easily composable, modular predictors, and this was in fact essential to designing an effective hybrid, as well as recognizing the potential for our proposed, new predictors. Several of our predictors also make use of the relatively abundant memory resources available at the software level, improving accuracy by reducing interference otherwise caused by resource sharing. Although effective, resource-intensive predictors have an obvious cost, and a deep understanding of experimental behaviour in terms of cost and accuracy allows us to determine optimized trade-offs.

1.1 Contributions

We make the following specific contributions:

- A unification framework for relating predictors to each other based on the patterns they capture, their mathematical expression as functions of inputs and value sequences, and their implementation. This framework shows how some predictors can be composed of others in an object-oriented sense, which in turn further simplifies understanding and implementation. We provide several new predictors suggested by the unification framework. The 2-delta last value and last N stride predictors are logical counterparts of the previously reported 2-delta stride and last N value predictors. We also provide a new table-based memoization predictor that hashes together function arguments, as well as memoization–stride and memoization–finite context method predictors based on it.
- An adaptive software hybrid predictor composed of many sub-predictors that dynamically specializes to whichever sub-predictor performs best. It is fast, accurate, memory-efficient, and suitable for a dynamic compilation environment. Although hardware hybrid predictors are well studied, our object-oriented design and implementation enables two unique optimizations. First, we allocate one hybrid predictor object per prediction point, and therefore enable specialization at prediction point granularity, eliminating false sharing and improving accuracy. Second, we bypass the execution of unused sub-predictor strategies and actually free their associated data structures during specialization, improving speed and reducing memory consumption.
- A software library implementation of return value prediction. This library is open source, portable, modular, and supported by unit tests. We use this library to perform a set of experimental analyses with a range of benchmarks that reveal the memory consumption, speed, and accuracy characteristics of individual sub-predictors as well as the hybrid predictor. We then demonstrate application of return value prediction to both method level speculation and program understanding.

In the next section we present our main experimental system design. Section 3 describes the basic predictors we investigate and gives experimental data on their behaviour. Our hybrid design and experimentation is shown in Section 4, and in Sections 5 and 6 we discuss results with respect to method level speculation and understanding applications. Finally, Section 7 describes related work, followed by conclusions and future work.

2. System Design and Environment

Our approach is based on a library implementation of return value prediction. The library includes a variety of predictors, as well as a clean interface to a client virtual machine. This modular design allows for relatively easy experimentation, and we explore RVP behaviour using an interpreter-based research Java VM client. Below we describe the basic system structure and client configuration, followed by initial analysis of our benchmarks and overhead costs.

2.1 System

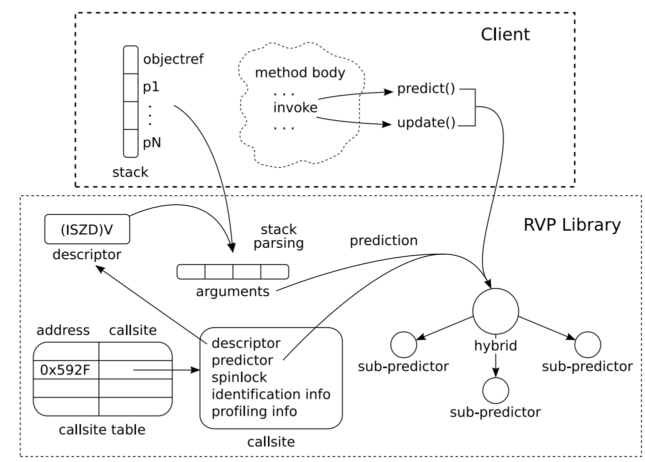

Figure 1. *Client–library communication.*

Figure 1 gives an overview of the general RVP library structure and client–library communication process. At the library's core is a table of callsite objects, each of which has a (hybrid) predictor, a spinlock to protect that predictor, a method descriptor, identification information, and profiling information. When a non-void callsite is first encountered during code generation/preprocessing, the client registers the

callsite identification with the library, using the callsite address in VM memory, class, method, program counter, and target method descriptor. The library maintains a hashtable of callsite objects, and either creates or returns a new object representing the callsite to the client on registration. The client will then use this object as context during runtime execution of the associated callsite. The association of unique predictor objects with each callsite eliminates false sharing and simplifies predictor specialization decisions.

When non-void method calls are encountered during execution, the client transfers control and context information to the RVP library code. This is generically performed through bracketing library calls to two functions: a `predict()` function that performs the actual prediction, and an `update()` function that updates predictor state after the actual return value is known. Note that for performance, in neither case is the Java Native Interface (JNI) call mechanism used—we bypassed the JNI mechanism after carefully verifing that our code does not violate any assumptions made by the client JVM used for testing. More general and conservative approaches are possible, *e.g.,* through JNI-inlining [45].

Several predictors rely on method argument data for the prediction and update steps. This includes explicit method arguments as well as any implicit object reference for non-static methods, the precise arrangement of which can vary between clients. The library thus contains an argument parsing module that reads arguments from the interpreter call stack, zeroes out unused bytes, and arranges the arguments contiguously in memory, based on the argument descriptor. Although this is not the most efficient implementation, it is the most portable, and we control for it in our experiments.

Finally, each actual predictor in our system is generically contained within the hybrid predictor interface. The hybrid design delegates to one or more sub-predictors as required, and so this acts as a general wrapper for all of our predictors. Once a prediction is made, the value is returned to the client, and a subsequent call updates the predictor state. Hybrid specialization to specific sub-predictor strategies is also performed, as described in Section 4.

The return value prediction framework is implemented as part of libspmt, a library for speculative multithreading [36]. libspmt is written in an object-oriented fashion in the common subset of C and C++, compilable with GCC and XL C/C++, and runnable under Linux and AIX on 64-bit Intel and POWER machines. We use modern 64-bit Intel dual-core machines in our experiments with 8GB of RAM.

2.2 VM Client

Our interpreter client is SableSpMT, a complete implementation of method level speculation in a Java virtual machine [34]. SableSpMT is an extension of SableVM, a portable Java bytecode interpreter designed as a research framework [16], and was chosen to allow us to also evaluate application of return value prediction to method level speculation.

2.3 Initial Analysis

The basic framework allows for exploration of program behaviour in a variety of ways that do not necessarily depend on specific predictor designs. Below we describe our benchmarks, their important structural properties, and general system overhead. Sections 3 and 4 provide more detailed experimentation on the individual and hybrid predictors respectively.

Benchmarks. We use the SPEC JVM98 benchmarks with input set S100 for experimental evaluation [44]. Although these benchmarks are not as complex or memory-intensive as the more recent DaCapo benchmarks [2], they are faster to execute, more cooperative with SableVM, and more than sufficient to explore our contributions to software return value prediction. These results greatly extend previous, hardware-specific work in the area, which used datasets several orders of magnitude smaller in a more restricted context [18].

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 1. *Benchmark properties.* `V`: void; `NV`: non-void; escapes: escaping exceptions.

Table 1 presents various benchmark properties of interest to our investigation. In the first section, the numbers of methods and callsites in the dynamic call graph are presented. Predictors may in principle be associated with methods, callsites, or the invocation edges that join them. We choose to associate predictors with callsites to limit the scope of our experimental evaluation. An interesting area for future exploration would involve associating predictors with methods or invocation edges instead and studying the differences.

In the second section, void and non-void invokes, escapes, and returns are presented. Void methods do not return values and are excluded from our analysis, but we present these data for the sake of completeness. Method invocation can in general complete normally, updating the predictor as described earlier, or abnormally through exceptional control

flow. In the latter case case there is no return value and the predictor cannot be updated. Our experiments make predictions on all non-void invocations, but only update the predictor on normal returns. Accuracy measures are thus only reported over the total number of non-void returns. As the data show, exceptions are relatively rare, even for supposedly exception-heavy benchmarks such as `jack`, which means that exceptions do not have a large impact in any case.

In the third section, the non-void returns are classified according to the nine Java primitive types. Type information is interesting because some types are inherently more predictable than other types, suggesting specialization and compression strategies, and because it describes the behaviour of the benchmarks to some extent. `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.

Communication overhead. Our design emphasizes modularity and ease of experimentation over performance. The use of an external library, multiple calls, portable argument parsing, and so forth has an obvious performance impact, much of which could be ameliorated by incorporating the RVP code directly into the VM, interleaving RVP code in generated code, and generally optimizing its performance along with other VM activities. We thus performed basic experiments to isolate and measure the overhead of our framework.

Figure 2 shows the slowdown due to communication overhead with the predictor module of the library. These data are gathered using a "null" predictor that simply returns zero for every prediction, and performs no actual update computation. In future experiments we control for this overhead by using the null predictor performance results as a baseline. The comparatively large slowdown for `mtrt` is mainly due to excessive contention in our simple predictor locking strategy, combined with a relatively high number of method calls. Improved lock-based or even lock-free designs would alleviate this problem and in general multithreaded predictor interactions are an interesting vector for future work. We observed that `raytrace`, the single-threaded version of `mtrt`, not included in SPEC JVM98, itself has a slowdown of 4.25. In general, overhead scales with the number of calls, and we expect downstream applications to tailor their use of RVP to the locations where it can actually be useful.

3. Predictors

A wide variety of predictors have been proposed, and a basic organization and evaluation are essential to our study. It is also the case that many predictors described in the literature are presented as hardware implementations, often using circuit diagrams. This clearly expresses the design in terms of feasibility, power and space efficiency, but many of these

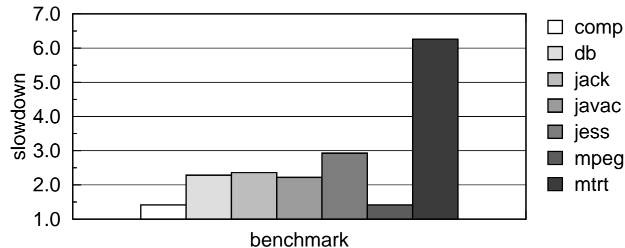

Figure 2. *Null predictor slowdowns.*

details can also obscure the intended algorithmic behaviour of the predictor. In designing a software solution, we abstracted the simplest implementation approach for each predictor, and so discovered many commonalities between predictors that we had not previously considered. 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 2, 3, and 4 give a structural presentation of a variety of common predictors. These are organized as core 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 data and pseudo-code used to implement the actual predictor. Mathematical expression illustrates the predictor behaviour by showing how the current prediction (v_n) is derived from the previous history of actual return values ($v_{n-1}, v_{n-2}, \ldots$). Implementation details include actual state, with pseudo-code divided into `predict()` and `update()` methods according to our predictor interface. The former may receive current method arguments, and returns a new predicted value, while the latter receives the actual return value in order to update its internal state. Note that for brevity we have made use of several non-standard, but self-explanatory functions in these descriptions.

In the following subsections we describe our logic in constructing Tables 2, 3, and 4 and give further detail on the individual predictors. We follow this with an experimental examination using our RVP framework, exploring the relative accuracy of different predictor designs, as well as their memory and time costs.

Note that we do not include in the unification framework predictors that are unsuitable for return value prediction, nor predictors that are substantially equivalent to the ones presented here.

3.1 History-based predictors

Table 2 contains the predictors that are based on only the history of return values for the associated function. Predictor names follow the names reported in the literature. On the left of the table are predictors that derive their prediction from the values 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 authors of the differential finite context method predictor study is expressing this relationship [17]. An organized division between base and derivative forms suggests two new predictors here, the 2-delta last value and last N stride.

Last value. The last value predictor is perhaps the simplest useful predictor. It merely predicts that the return value v_n will be the same as the last value v_{n-1} returned by the function. It has a single field `last` that gets returned when making a prediction and assigned to when the actual return value is known. In the example, after seeing the sequence 1,2,3, a last value predictor would simply predict 3 as the next output.

Stride. A stride predictor can be seen as a derivative of the last value predictor, computing a prediction based on their sum of the last difference and last value. For instance, seeing 1,2, then 3, it would predict 4 as the last value plus the difference between 2 and 3. While not completely comparable, this captures most of the same patterns as the last value predictor as well as new ones. In particular, many loop indices and other incrementing/decrementing sequences are easily recognized. Disadvantages are that it takes an extra prediction to warm up, the update and predict operations are somewhat slower, and there is an extra field of storage.

2-delta stride. The 2-delta stride predictor is similar to the stride predictor, imposing the extra constraint that the stride must be the same twice in a row before the predictor updates the stride used to make the prediction. In the example, the stride of 1 detected early in the history is still used to predict 4 even after seeing 3 twice, whereas a simple stride predictor would predict 3 based on the last stride. This design reduces mispredictions by being able to ignore single abberations in a sequence, as can occur in the context of nested loop iterations. In the hardware literature the 2-delta stride predictor has an extra “hysteresis” bit to control this.

2-delta last value. The 2-delta last value predictor is a new predictor that was suggested by the lack of a corresponding, non-derivative form of the 2-delta stride predictor. A last value approach is used, but the value stored is only updated if the last value is the same twice in a row. Given a sequence such as 1,1,2,3 for instance, the stored last value is not updated during periods of change, and the predicted value will be 1 until the return value again repeats.

In a general sense, the 2-delta pattern can be generalized to arbitrary C-delta predictors, for arbitrary predictors and constant or bound C. Increasing C improves robustness, at a cost of increased warm-up time and larger state.

Last N value. The last N value predictor maintains an N-length history of the most recent return values, and uses that list to search for matches to the previous return value. A match results in a prediction of the next value in the history. This allows the last N value predictor to identify short repeating sequences, capturing simple alternations such as 0,1,0,1,. . ., or more complex patterns such as 1,2,3,1,2,3,. . ., neither of which are ideally predicted by the last value or stride predictors. Our example illustrates the latter case, where assuming $N \geq 3$, a 1 is predicted based on the previous value of 3, and a history where in the past a 3 was followed by a 1.

Last N value is a generalization of the last value predictor, which may also be expressed as a last 1 value predictor. In their analyses Burtscher and Zorn found that $N = 4$ was a reasonable tradeoff of accuracy against predictor complexity [7], and so we use this configuration in our experiments.

Last N stride. The last N stride predictor is a new predictor suggested by the lack of a corresponding predictor for the last N value predictor. Structurally identical, it records strides rather than value history, but also generates a prediction by searching the history for previous repetition to guide the prediction. It generalizes and subsumes the stride predictor, which can then be considered a last 1 stride predictor.

The example shows a sequence with strides repeating in the pattern 1,2,3. Given the last value of 13, the last stride was 3, which historically was followed by a stride of 1. Adding that to the last value gives the current prediction of 14. This example is obviously contrived, but repeating stride patterns can occur in several ways, such as by accessing field addresses that have identical offsets in in multiple objects.

Finite context method. To capture more complex historical patterns, the finite context method predictor hashes together a context, or recent history of return values of length C. The hashed value is used as hashtable key to index the corresponding return value. This allows for multiple, different patterns to coexist, trading hashing and storage costs for improved accuracy; in the example the pattern 2,3 is detected as recurrent and used for the next prediction, despite the existence of other output behaviour before and since. In our suggested implementation the key is stored as a predictor field so that later updates do not have to recompute the hash value, improving performance, although also potentially reducing accuracy.

Hashtable management is a non-trivial concern here, and as well as good hashing functions table size and growth must be controlled. We allow hashtables to dynamically expand up to a maximum table size, and use this design to assess how accuracy and memory requirements interact. For a context length we use $C = 5$ in our experiments, which Sazeides and Smith also favoured in their study of finite context method predictors [41].

Differential finite context method. Analogous to the finite context method, the differential finite context method predictor hashes together a recent history of strides rather than

values. This is used to look up a stride in a hashtable for adding to the last value in order to make its prediction. The example shows a sequence containing the stride pattern 3,2, which is recognized and used to predict the next value of 21. DFCM has the potential to be more space efficient and faster to warm up than the finite context method predictor.

3.2 Argument-based predictors

Accuracy can be also improved by taking into account method inputs instead or as well as previous outputs when making a prediction. Table 3 contains the predictors that exploit this information, again separated in terms of normal and derivative forms. In each of these cases the predict function is expanded to receive the current method arguments as input. In our implementation these predictors are all disabled for methods that do not take any arguments.

Memoization. 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 or function caching that "skips" pure function execution when the arguments match previous recorded (arguments, return value) table entries. In our example, the argument pattern of 1,2,3 is hashed together and the key found existing in the hashtable, resulting in a prediction of 4 for the third invocation of f. Note that a key difference from traditional memoization 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.

Memoization stride. A similar memoization approach can be applied to stride values. Memoization stride stores a stride between return values in its hashtable instead of an actual value, much like the differential finite context method predictor, and adds this value to the last value to make a prediction. The example shows a stride of 3 associated with arguments 1,2,3, resulting in a new prediction of 7 based on the previous value of 4 and the stride found for that argument pattern. Unlike the differential finite context method predictor, it is not necessarily more space efficient than its non-derivative form, since the set of values used to compute a hashtable key remains the same.

Memoization finite context method. The memoization finite context method predictor is a direct combination of the memoization and finite context method predictors. It concatenates the recent history of return values with the function arguments and uses the result to compute a hash value for table lookup. This is significantly more expensive than either memoization or finite context method predictors, but has the potential to capture complicated patterns that depend on both historical output and current input. The example shows a context of length 2, recognizing the output sequence 5,6 followed by an argument of 3, and so predicting the previously seen value of 7. In comparison, a pure memoization predictor would predict 9 here from the prior argument/return pair $f(3) = 8$, and a pure FCM predictor would return 8 due to the preceding output sequence of 5,6,8.

Parameter stride. 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.

3.3 Composite predictors

Table 4 contains predictors that are composites of one or more sub-predictors. The hybrid predictor uses the other predictors directly, whereas composite stride is in fact a generalized pattern for creating other predictors.

Hybrid. The hybrid predictor is composed of one of each kind of sub-predictor. To make a prediction, it first obtains a prediction from each sub-predictor and records this value. It then returns the prediction of the predictor with the highest accuracy. In our implementation we keep track of accuracy over the last n values, where n is the number of bits in a word. This allows sub-predictors with locally good but globally poor accuracies to be chosen by the hybrid. To update the hybrid, for each such sub-predictor update() is called, the actual return value is compared against the predicted return value, and the accuracy histories are updated accordingly.

Composite stride. The composite stride predictor is not an individual predictor but rather a pattern for constructing stride predictors. A composite stride simply contains another predictor that it will use to predict a stride value, and adds that to the previous return value. Each predictor on the right hand side of Table 2 as well as the memoization stride predictor in Table 3, 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.

3.4 Runtime Behaviour

The above predictors were implemented within our framework, and experimentally examined for individual accuracy and efficiency. More complex predictors are expected to improve accuracy, but at a cost of memory and computation,

Last Value [15]

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

e.g.: $1, 2, 3 \rightarrow 3$
fields: last
predict():

```
return last;
```

update(value_t rv):

```
last = rv;
```

Stride [15]

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

e.g.: $1, 2, 3 \rightarrow 4$
fields: last, stride
predict():

```
return last + stride;
```

update(value_t rv):

```
stride = rv - last;
last = rv;
```

2-Delta Last Value (new)

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

e.g.: $1, 1, 2, 3 \rightarrow 1$
fields: last_1, last_2
predict():

```
return last2;
```

update(value_t rv):

```
if(rv != last1) last1 = rv :
else last2 = rv;
```

2-Delta Stride [42]

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

e.g.: $1, 2, 3, 3 \rightarrow 4$
fields: last, stride_1, stride_2
predict():

```
return last + stride2;
```

update(value_t rv):

```
if(rv - last != stride1)
   stride1 = rv - last;
else stride2 = rv - last;
last = rv;
```

Last N Value [7, 24]

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

e.g.: $1, 2, 3, 1, 2, 3 \rightarrow 1$
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;
push(values,rv);
```

Last N Stride (new)

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

e.g.: $1, 2, 4, 7, 8, 10, 13 \rightarrow 14$
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;
push(values,rv - last);
```

Finite Context Method [41, 42]

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

e.g.: $1, 7, 2, 3, 8, 4, 7, 2 \rightarrow 3$ for $C = 2$
fields: key, context[C]
predict():

```
key = hash (context[]);
return lookup (key);
```

update(value_t rv):

```
store (key, rv);
push(context,rv);
```

Differential Finite Context Method [17]

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

e.g.: $1, 6, 9, 11, 16, 19 \rightarrow 21$ for $C = 2$
fields: last, key, context[C]
predict():

```
key = hash (context[]);
return last + lookup (key);
```

update(value_t rv):

```
store (key, rv);
push(context,rv - last);
```

Table 2. *History-based predictors.* Hashing and searching functions are not shown; the `push` function adds a new value to an array, shifting all other elements down and removing the oldest element.

Memoization (new)

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

e.g.: $f(1,2,3)=4,\ f(4,5,6)=7,\ f(1,2,3)\rightarrow 4$
fields: key
predict`(value_t args[]):`

```
key = hash (args[]);
return lookup (key);
```

update`(value_t rv):`

```
store (key, rv);
```

Memoization Stride (new)

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

e.g.: $f(1,2,3)=4,\ f(1,2,3)=7,\ f(1,2,3)\rightarrow 10$
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)

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

e.g.: $f(1)=5,\ f(2)=6,\ f(3)=7,$
$f(3)=9,\ f(1)=5,\ f(5)=6,\ f(5)=8,$
$f(1)=5,\ f(2)=6,\ f(3)\rightarrow 7$ 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);
push(context,rv);
```

Parameter Stride [18]

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

e.g.: $f(‘r’)=17,\ f(‘v’)=21,\ f(‘p’)\rightarrow 15$
fields: a = A, old_args[A], strides[A]
predict`(value_t args[]):`

```
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 3. *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 [10]

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

fields: predictors[], accuracies[], predictions[]
predict`(value_t args[]):`

```
predictions[] = predictors[].predict(args);
return predictions[indexOfMax (accuracies[])];
```

update`(value_t rv):`

```
for (p = 0; p < P; p++)
   predictors[p].update(rv);
   if (rv == predictions[p])
      accuracies[p]++;
   else accuracies[p]--;
```

Composite Stride (new)

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

fields: last, f
predict`():`

```
return last + f.predict();
```

update`(value_t rv):`

```
f.update (rv - last);
last = rv;
```

Table 4. *Composite predictors.* Our particular hybrid design is new, although many structurally similar hardware designs have been proposed. The composite stride predictor is a general implementation pattern for converting value predictions into stride predictions, rather than a specific predictor.

and so it is important to determine which predictors are most effective for a given time/memory budget.

Accuracy. Figure 3 shows basic prediction accuracies for each predictor and for each benchmark. The benchmarks are clustered in alphabetical order from left to right for each predictor. The predictors are arranged in left-to-right top-to-bottom order presented in Tables 2 and 3, with the hybrid predictor last. This version of the hybrid uses every sub-predictor on every call to predict() and update() and is very expensive. For comparison we have also included as the first predictor a null predictor that returns 0 for every prediction. Accuracy is calculated as the number of correct predictions over the number of calls that returned to the callsite.

The hybrid beats all sub-predictor accuracies combined for every benchmark, as expected, because it allows sub-predictors to complement each other. Predictor accuracy otherwise roughly scales 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 improve accuracy. Context clearly has a significant impact on accuracy. The finite context method and its differential form have the highest individual predictor accuracies, and even memoization is noticeably improved by context. Method argument approaches are not as successful as context in general, although as we show later memoization behaviour does sometimes complement context forms.

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, likely due to their use of more irregular floating point types. `compress` improves dramatically with greater context (input or output), 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. The graph is structured similarly to Figure 3, although on a logarithmic scale and without the null predictor. Slowdown is calculated as the ratio of the program running with the particular predictor enabled to the program running with the null predictor enabled, per the control experiment in Figure 2. 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 a somewhat expensive operation. Second, the memory requirements of the larger tables introduce performance penalties due to memory hierarchy latencies. The hybrid is unsuprisingly very slow, roughly representing the summed cost of running all 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 this effect is compounded in the hybrid that has five table-based sub-predictors at each callsite. These data also indicate a further cause of individual benchmark and 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. *Memory consumption.*

The data in Table 5 and Figures 3 and 4 assume hashtable sizes are unbounded, and so the tables grow as necessary to accomodate new values. This is obviously unrealistic, but if the sizes are bounded then new values overwrite old values once the maximum size is reached, and so may reduce overall accuracy if the old value is ever requested. Predictor accuracy as a function of maximum table size is thus shown in Figure 11. Here maximum table sizes are varied from 2^0 to 2^{25} entries, one power of 2 larger than the largest 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 may be constant, indicating no further impact from collisions, or in some cases may actually decrease. We hypothesize that the reductions are due to an interaction with garbage collection at large heap sizes. Since many tables hold object references that will be invalid after garbage collection, to prevent invalid references from causing unnecessary hashtable pollution, tables are are freed and reallocated at size 0 on each collection. This simple solution is acceptable when garbage

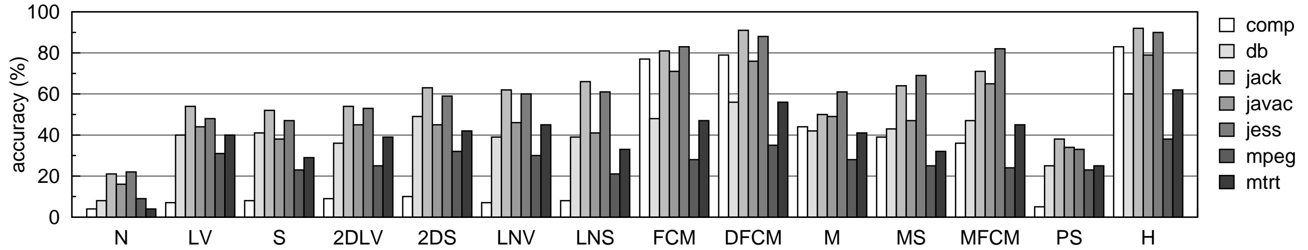

Figure 3. *Predictor accuracies for the null predictor and all predictors in Tables 2, 3, and 4*

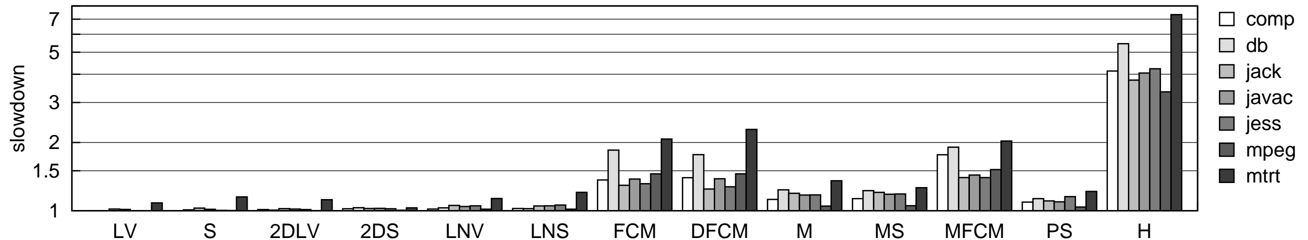

Figure 4. *Predictor slowdowns relative to the null predictor.*

collection is infrequent or with small tables that accumulate less history, but at large table sizes collection frequency tends increases, tables are more frequently rebuilt, and accuracy thus sometimes reduced. Improved interaction of RVP structures with garbage collection is part of our future work.

Figure 11 also shows the complex interaction of individual predictors. For a given benchmark and table size individual predictors often have different performance; memoization may work well in some instances whereas the 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` and `jess`, for example, combines the better accuracy of memoization designs at low table sizes with the higher accuracy of (D)FCM at higher sizes.

4. Hybrid Adaptivity

Although the hybrid as presented so far achieves very high accuracy, it consists of many sub-predictors, some of which are quite complicated, and the overhead is simply too high. Improvements are possible, however, by adaptively changing the hybrid so that each instance specializes to the best predictor for the associated callsite. The intention is to maintain the high accuracy of the naïve hybrid predictor in Table 4 while optimizing for speed and memory consumption.

Here we investigate two approaches to specialization, an oracle-based solution using ahead of time profile data and an actual online adaptive model. The former allows us to determine rough upper bounds on specialization performance, while the latter provides a more practical design that does not require pre-profiling runs. Comparison of these results with our naïve, non-adaptive hybrid then shows the potential speed and memory improvements offered by adaptivity, and the extent to which accuracy is a trade-off.

A variety of parameters of course influence hybrid design, and in particular the mechanism by which sub-predictors are chosen. Our adaptive hybrid relies on a warm-up period to prime predictors, and then selects the best performing sub-predictor over the last n predictions, favouring cheaper predictors in the case of ties. The length of the warm-up period as well as accuracy confidence thresholds for specialization and despecialization have the potential to change hybrid performance and we thus assess a range of parameterizations.

Offline specialization. In our offline experiment the program is run to completion, and the results analysed to determine the "ideal" predictors—sub-predictors which performed best at individual callsites over the course of the entire program execution. In a second program run, these data are passed via a configuration file to the library runtime, and when the client registers a new callsite, the hybrid associated with it immediately specializes to the predictor given in the profile. No state for unused predictors is created, and space and time overhead is minimized.

Figure 12 shows the distribution of ideal predictors for each benchmark in terms of both reached callsites at runtime and number of dynamic calls. From the perspective of callsites the majority of specializations are to the null predictor or last value predictors. This is partially due to initialization code, where infrequently executed methods are more easily specialized to simple predictors that depend on limited history—the null predictor is the cheapest and fastest to specialize on a method returning only 0. It is also the case that while many methods may be captured by simple predictors, they are not the ones heavily exercised at runtime, and maximizing accuracy in general depends on advanced, complex

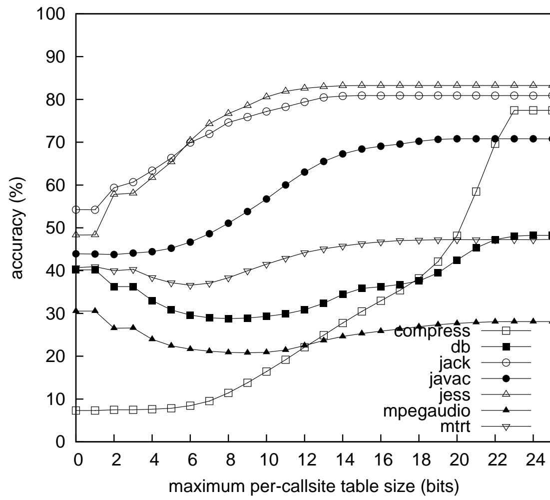

Figure 5. Finite context method

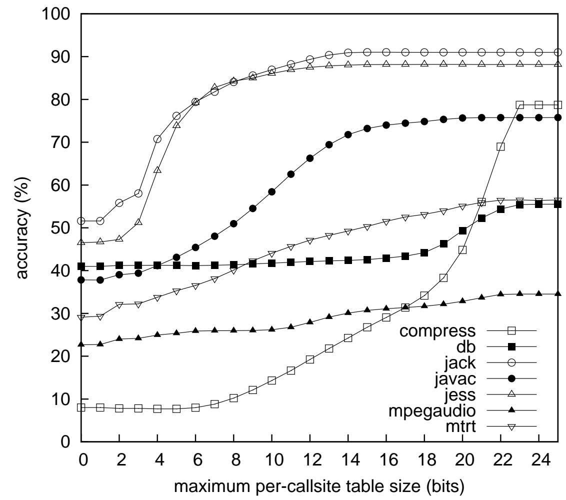

Figure 6. Differential finite context method

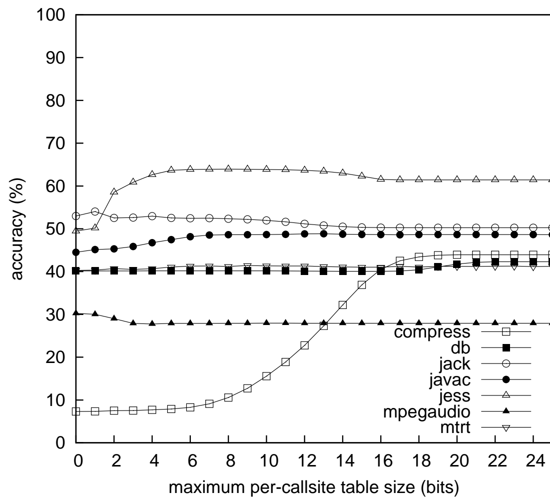

Figure 7. Memoization

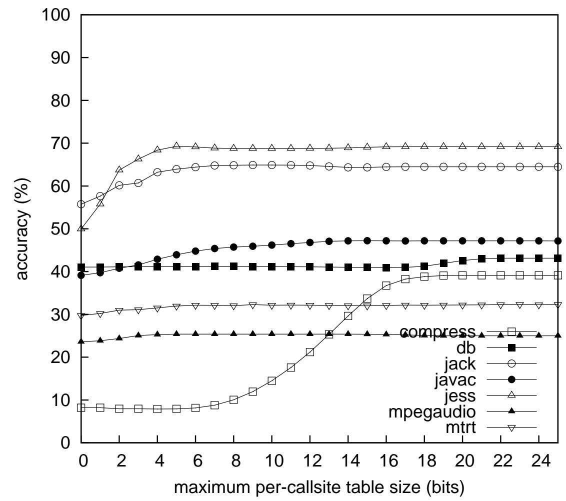

Figure 8. Memoization stride

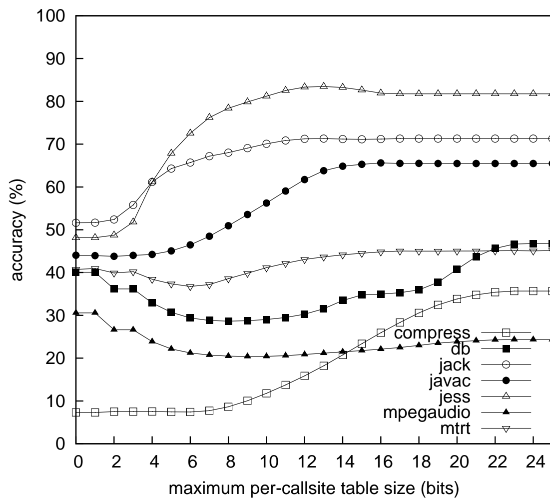

Figure 9. Memoization finite context method

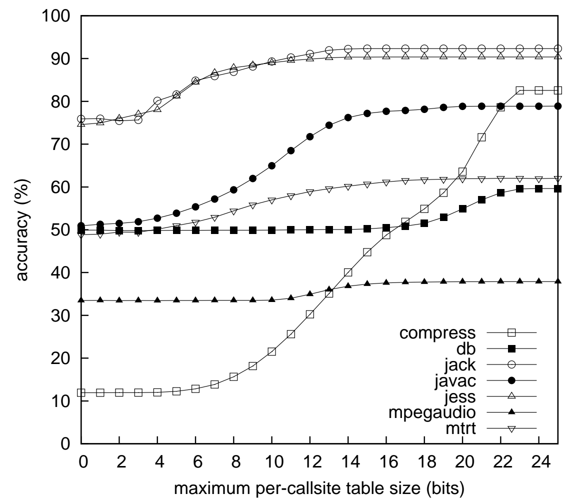

Figure 10. Hybrid

Figure 11. *Predictor accuracy vs. table size*

predictors. Table-based predictors clearly dominate in terms of actual calls. `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, and so these predictors dominate.

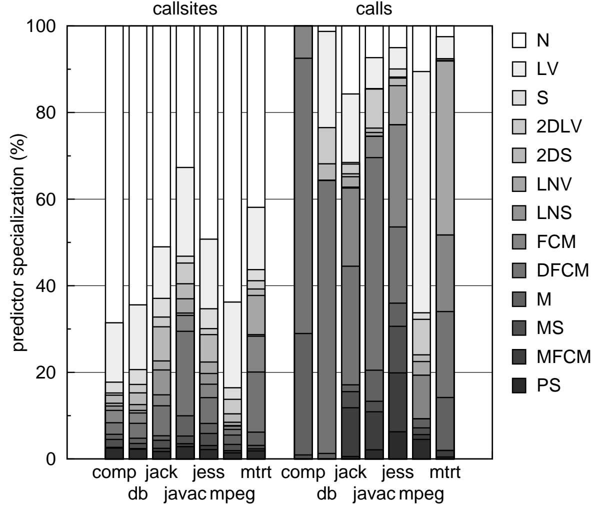

Figure 12. *Ideal predictor distributions.*

Online specialization. In online experiments, the system attempts to determine ideal predictors dynamically, without ahead of time profiling data. This simplifies practical application of RVP, and may also better accommodate programs in the case of phase-like behaviour, where the optimal choice of predictor may not be 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. A hybrid predictor will not specialize until $u \geq w$, where u is the number of predictor updates. The second is a "disable" confidence threshold for specialization, d. For the number of correct predictions c over the last n calls, if $c \geq d \wedge u \geq w$ then the hybrid specializes to the best performing sub-predictor. We use a value of $n = 64$, the number of bits in a word on our machines. The third parameter is an "enable" confidence threshold for despecialization, e. If $c < e$ 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 technique.

```
for W in -1 .. 6
  for D in 0 .. 8
    for E in 0 .. D
      w = (W == -1) ? 0 : 2^(W * 3)
      d = D * 8
      e = E * 8
```

Figure 13. *Configuration of online hybrid parameter sweep.*

A parameter sweep over w, d, e was performed with these parameters varying according to Figure 13. This generated 360 different experiments. For each, the average accuracy and total running time 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 14.

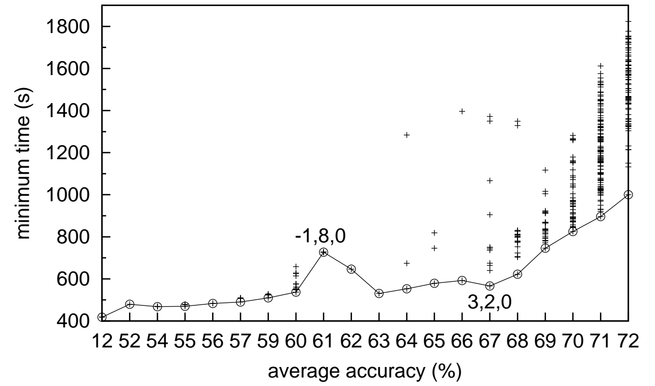

Figure 14. *Minimum time vs. average accuracy for online hybrid parameter sweep.*

From these data, we selected the point at accuracy 67% with running time 566 seconds for use in future experiments. This choice is 5% worse than the optimal accuracy of 72%. At this point, $\{W, D, E\} = \{3, 2, 0\}$, which corresponds to a warmup of $w = 512$ returns, specialization threshold of $d = 16$ correct predictions (25% accuracy), and a despecialization threshold of $e = 0$, meaning no despecialization will occur. This configuration is only 35% slower than the cheapest configuration $\{-1, 0, 0\}$ which only achieves an accuracy of 12%, effectively specializing immediately to the null predictor. The data point at accuracy 61% with running time 727 seconds also stands out. The corresponding configuration, $\{-1, 8, 0\}$, means that $w = 0$, $d = 64$, and $e = 0$. This predictor has no warmup, nor does it despecialize, and is quite slow. It was selected by the optimization for that data point partially because its high specialization threshold did ultimately result in some good sub-predictor choices, but also due to a lack of better performing configurations at that accuracy level in our plot—the distribution of experiments is not even along the abscissa, for most experiments cluster near the upper range. Interestingly, in all but the top 3 most accurate and slowest cases, $e = 0$. Accuracy benefits from despecialization may exist, but certainly come with sharply increasing costs.

Comparative performance. To determine the relative improvement offered by an adaptive hybrid we compare the behaviour of offline and online designs with the naïve, non-adapative model. Predictor accuracies, slowdowns with respect to a null predictor baseline, and memory consumption for all three are shown in Figures 15 and 16 and Table 6 re-

spectively. These experiments use a maximum table size of 25, testing hybrid behaviour at maximal accuracy.

In terms of accuracy, the naïve hybrid predictor without specialization should act as an oracle. It behaves like the online hybrid with an infinite warmup period. Data in Figure 15 shows that the offline adaptation is quite effective, usually within a few percent of the naïve version. In some cases accuracy slightly increases over the oracle, presumably because the continuing availability of all predictors in the naïve version occasionally allows suboptimal choices to be made. The close match of offline and naïve accuracy also indicates that significant program phases are either rare or at least not critical for RVP performance—the fixed choices of the offline model clearly do not overly affect accuracy.

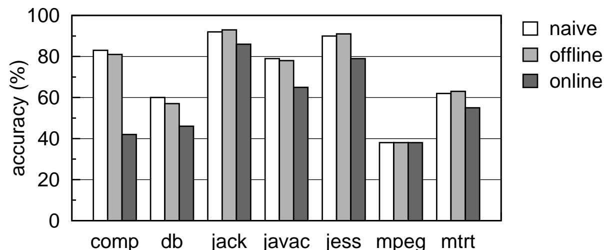

Figure 15. *Naïve vs. offline vs. online accuracies.*

Accuracy is not significantly compromised in the online version, and is within ≈5-10% of offline for most benchmarks. `compress`, however performs significantly less well than the others. Deeper analysis shows that this difference is due to different prediction strategies for a few callsites, and in particular the `getbyte()I` call in the `Compress.compress()` method, exercised over 47 million times. The offline version chooses a DFCM predictor with 79% accuracy, whereas the online version specializes too early, selecting a null predictor that results in less than 10% accuracy overall.

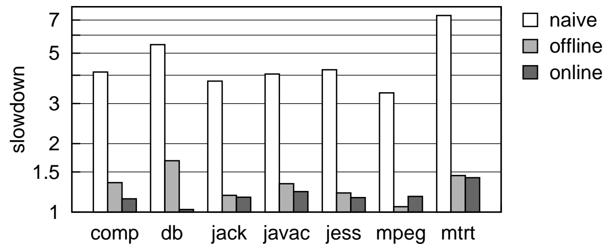

Figure 16. *Naïve vs. offline vs. online slowdowns.*

Time overhead is dramatically reduced by both offline and online designs, as shown in Figure 16. Online improvement actually exceeds offline for most benchmarks; the offline tends to choose the more accurate but expensive table-based predictors, while sub-optimal specialization in the online version favours predictors with less state and thus less warmup. This effect can also be seen in the memory consumption data, shown in Table 6. Both offline and online modes greatly reduce memory requirements, in the case of offline `mpeg` by over a factor 24. Online memory usage tends to be smaller in general, `db` providing an extreme example where online is orders of magnitude cheaper. The bottom half of Table 6 shows further memory reductions possible by eliminating common sources of potentially unnecessary memory use in our experimental system.

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

Table 6. *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.

5. Method Level Speculation

To better determine the value of RVP information and accuracy, we investigate the impact of return value prediction on an RVP consumer, method level speculation (MLS). Accurate RVP is quite important for MLS, allowing us to get a more application driven view of accuracy levels and hybrid performance. Below we give more detail on MLS and describe our experimental system, followed by measurements of MLS performance changes due to inclusion of both naïve and online hybrid implementations.

Speculation model. Method level speculation is a runtime optimization technique for automatic parallelization. At various points in the execution speculative threads are spawned to execute portions of a program in parallel with the current execution. Speculative threads execute in a safe and isolated fashion, and must be validated before having a visible effect. In the case of MLS, speculation is performed at method calls, creating speculative execution of a method continuation concurrent with non-speculative execution of the method itself. When the parent, non-speculative thread returns from the method call it joins the speculative child, validates the child state, and either commits and makes visible the child state or aborts the speculative execution accordingly. Since method continuations often make immediate use of method return values, and incorrect assumptions will result in a child failing to validate, RVP has a direct impact on MLS performance measures.

Impact of RVP on MLS In automatic parallelization, performance is improved and speedup is achieved when the useful parallelism exposed outweighs the overhead incurred. A

number of measurements of impact due to RVP are thus possible. The lengths of speculative threads in terms of clock cycles gives a good indication of speculative success, where longer thread lengths enabled by better return prediction correlate with more opportunity for parallel execution. A speculative coverage measure for the percentage of the original non-speculative sequential program that could be successfully executed in parallel and committed also increases with RVP support. We previously explored the impact of RVP on these measures [34, 35]. At that time, the biggest concern with respect to RVP was excessive overhead due to the cost of naïve hybrid predictor updates, and this was in fact the primary motivation for this study.

There are many possible configurations for the method level speculation system, and our full system model is described elsewhere [34, 35]. In these experiments, we allow for one child thread to be created at every non-speculative method invocation, void or non-void. However, we do not allow speculative child threads to create even more speculative child threads of their own. Our system supports this, but it significantly complicates the understanding of the impact of return value prediction, and it is better investigated in a full study of child thread creation. Threads run for as long possible, until joined by the parent returning to the call. Speculative threads can enter and exit methods, allocate objects, and read from and write to the heap via a dependence buffer that is a kind of software transactional memory [21].

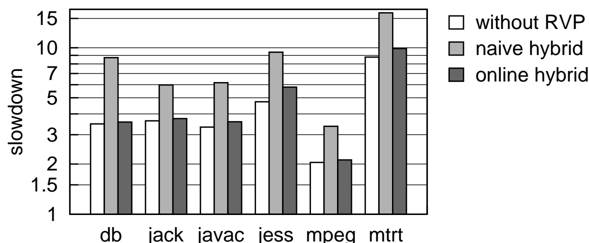

Figure 17. *Method level speculation slowdowns.* `compress` is excluded due to last-minute technical difficulties

Our speculation environment does not yet incorporate fork heuristics for applying speculation judiciously, and instead performs speculation at every opportunity. This extremal strategy provides an abundance of data for analysis, but also results in slowdowns in terms of absolute performance, as can be seen in Figure 17. The addition of a hybrid RVP component increases this load further, particularly in the case of the full naïve version. The online hybrid adaptation, however, eliminates most of the extra overhead provided by RVP, and in many cases matches the performance without RVP. Several benchmarks still experience significant slowdowns, suggesting that further individual tuning of the accuracy / performance trade-off may be worthwhile, but the large overhead reductions provided by the online adaptive system are encouraging, and indicate that overhead is not an inherent limit on application of software RVP.

6. Program Understanding

Another interesting application of return value prediction is in understanding the behaviour of programs. The relative success and failure of prediction in general and different prediction designs reveals various aspects of predictability for a given callsite, exposing a variety of properties according to the prediction strategy under consideration. Here we discuss general insights inspired by close analysis of RVP performance. This includes table-based input/output characterization, identification of simple behaviours, and higher level application to program understanding with an example based on reverse engineering.

Table-based input/output characterization. Table size is important for accuracy in advanced predictors, and heuristically correlates with predictability. If the most accurate predictions require large tables, then data is necessarily more diverse. This can be further divided according to whether the predictor focuses on input or output data. A large memoization hashtable means that the callsite consumes highly variable data, whereas a large finite context method hashtable means that the callsite produces highly variable data. In Figures 18 and 19 the final distributions of predictor hashtables according to size are shown for the differential finite context method and memoization predictors. These sizes are gathered from experiments where specialization does not occur.

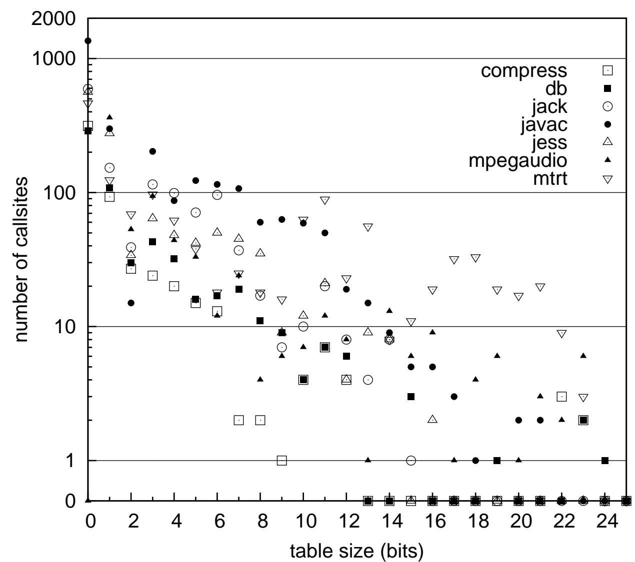

Figure 18. *Differential finite context method size distribution.*

Most hashtables are very small, with many not expanding beyond the initial size. Even when expansion does occur, approximately 90% of tables never expand beyond 8 bits. Tables that reach large sizes thus indicate significant variability in method input or output state. If predictors retain accuracy this further indicates a pattern in the return value data, characteristic of calculations that are not easily captured by simple predictors, but which in practice return bounded or repetitive values. In our benchmark suite the largest and most suc-

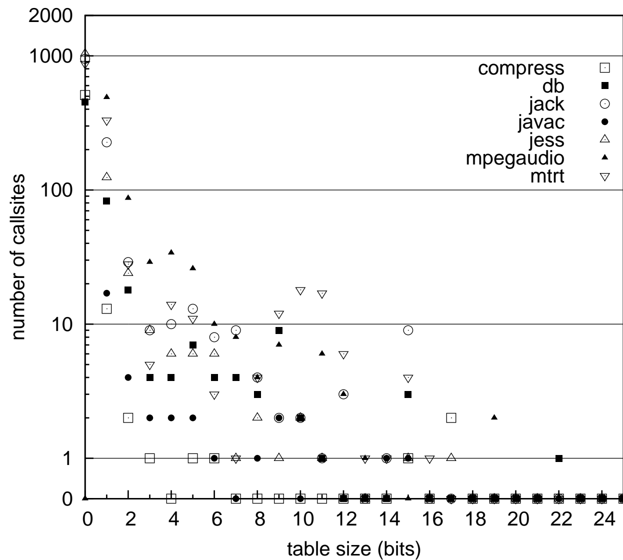

Figure 19. *Memoization size distribution.*

cessful (>75% accuracy) table predictors are DFCM predictors attached to hash table lookups in the `compress` benchmark and Vector component getter methods at certain points in `mtrt`. In the former case high accuracy in the DFCM predictor and not in the memoization predictor derives from a compact hashtable with a limited set of possible content. In the latter, the behaviour indicates a sparsely populated and more regular data space at that particular state of computation, unsurprising in a simple raytracing application.

Calls where memoization is best applied correspond to situations where strong input/output relations exist, but output otherwise have little pattern. Non-trivial use of `equals (Object,Object)Z`, for instance, is specialized to memoization in `db`, as are some of the other Vector component getters in `mtrt`, and size/field getters in `jess`. Overall, and despite the flexibility of memoization in a speculative context, use of non-input state in a method call tends to overly perturb the input/output mapping, and the hybrid specializes to memoization primarily for pure, side-effect free methods. This does, however, allow easy identification of experimentally side-effect free methods. Preliminary experiments indicate that GC plays a significant role in the accuracy of memoization predictors, many of which take an implicit `this` object reference as input.

Simple behaviours. The presence of simpler predictors also provides useful understanding. A callsite that is well-predicted by the null predictor, for example, reveals code where runtime constants exist and control flow is constant. This tends to identify callsites with error return codes, which are typically 0, but also places with invariant boolean returns, such as calls to `isReadOnly` in `ByteBuffer.-hasArray()` in `jess`, many calls to `isType` in `Node.-convert()` in `javac`, and so forth. Invariant computations represent useful opportunities for many code optimizations, such as basic block straightening, specialization, and dead code elimination. The call to `GetChild()I` in the `OctNode.Copy(OctNode)` method of `mtrt` shows a particularly useful application. This code copies one node in a tree into another, including a loop to individually copy each of 8 child nodes in an array. A 100% accuracy for the null predictor at the method call to `GetChild()I` implies that the target node in the copy operation is invariably a leaf node, with no children. Replication of the child array contents could thus be simplified or eliminated altogether.

Reverse engineering. More complex behaviours can also be exposed by analyzing RVP data. `mpegaudio`, for instance, is a good example for program understanding because it is an obfuscated program with poor prediction accuracy. Presumably this is because it decodes an mp3 file and the floating point data are highly irregular. We looked for callsites with many calls and poor predictability and found that there are 4 callsites within `q.l([SI)I` that call `j(F)S` approximately 4 million times each and that the predictor accuracy is only 3%. The float to short conversion of `j(F)S` indicates some kind of data processing, and the fact that the callsites are in close proximity, are each invoked the same number of times, and are frequently invoked (hot) indicates that they are either part of a larger data conversion within the same loop or perhaps are the result of manual loop unrolling. Given the knowledge about what mpegaudio does at a high level, we can combine all of these data to make an educated guess that these callsites are in the loop where the mp3 file gets decoded. RVP is not intended as a decompilation tool of course, but it is interesting that prediction success can be correlated with fundamental algorithm properties, and so help reveal the behaviour of even intentionally obfuscated programs.

7. Related Work

Return value prediction is a form of the more general problem of value prediction, which has been researched for well over a decade, although primarily in the context of hardware designs and constraints. A wide variety of value predictors have been proposed and examined, including simple computational predictors, more complex table-based predictors, machine learning techniques, and hybrid implementations. Our work here extends existing investigations of RVP in a Java context [18, 33], considering practical accuracy and performance in an adaptive, dynamic software-only environment.

Burtscher *et al.* provides a good overview of basic value prediction techniques [6]. As a general rule, accommodating more 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, 24, 47]. 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]. Most predictors can be further improved by incorporating statistical measures such as formal confidence estimates into the prediction process, although this also adds extra complexity [4, 8].

Gabbay introduced the stride predictor and last value predictor, as well as several more specialized predictors, such as the sign-exponent-fraction (SEF) predictor, and register-file predictor [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, usually resulting in poor performance of basic approaches to 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 [46]. For our purposes it may be worth considering a simple stack top predictor that is a register value predictor specialized for return values. Pointer-specific prediction is also possible; Mutlu *et al.* introduced address-value delta (AVD) prediction. It predicts whether for a given pointer load instruction the difference between the address and the value at that address is stable [30]. Unfortunately, this predictor is not useful for return value prediction in an object-oriented context. Marcuello *et al.,* propose an increment-based value predictor [28,29] for value prediction within a speculative multithreading architecture. The increment 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 [42]. Subsequent work considers the practical impact of hardware resource (table-size) constraints on predictability [41]. Goeman *et al.* proposed the *differential* finite context method predictor [17] 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 [20].

Hybrid designs allow predictors to be combined, complementing and in some cases reinforcing the behaviour of individual sub-predictors. Wang and Franklin use a MIPS-based simulation to show that a hybrid value predictor achieves higher accuracy than its component sub-predictors in isolation [47]. The interaction of sub-predictors can be complex, though, 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 hybrid performance [10]. Designs have thus been proposed to reduce hybrid storage requirements [9], and to use selection mechanisms that reduce inappropriate bias, such as cycling between sub-predictors [39], or use of improved confidence estimators [19]. Optimal hybrid design of course maximizes the efficiency of the applications using the predictions, and Sam and Burtscher argue that complex value predictors are not always neccessary [38].

Software value prediction, while less common, has also been investigated. Li *et al.,* for instance, 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 [22]. The resulting profile is used to customize predictor code generation for an optimized, subsequent execution [13, 23]. Liu *et al.* incorporated software value prediction in their POSH compiler for speculative multithreading and found a beneficial impact on performance [25]. The predictors are similar to those used by Li *et al.* [22], 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, such as by statically estimating predictability [6].

Return value prediction is a basic component of method level speculation, and even simple value and stride predictors have a large impact on speculative performance [11,31]. Hu *et al.* introduced the parameter stride predictor as part of their study of Java traces, and use simulated hardware to make a strong case for the importance of return value prediction in MLS [18]. Our own work here is largely inspired by the RVP requirements of software-based method level speculation [34, 36], and this study extends an earlier workshop paper that gave preliminary data on RVP behaviour [33]. Theoretical limits on RVP have also been considered: Singer and Brown applied information theory to analyse the predicatabilty of return values, independent of any specific predictor design [43].

Our inclusion of data type information in considering RVP behaviour follows existing work on using types in value prediction. 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 [40]. Loh demonstrates both memory and power savings by using data width information [27], although in a hardware context, and with the additional need to heuristically discover high level type knowledge. Sam and Burtscher later showed that hardware type information can be efficiently used to reduce predictor size [37]. They also demonstrated that more complex and hence more accurate predictors have a worse energy-performance tradeoff than simpler predictors and are thus unlikely to be implemented in hardware [38].

Several of the new predictor designs here are based on memoization. Memoization is obviously a well known technique, and effective memoization based compiler and runtime optimizations have been described [12]. Our interest in memoization approaches in RVP is partly based on their ability to be applied to the large proportion of "impure" methods found in an object-oriented language [49].

8. Conclusions and Future Work

Return value prediction is useful in a variety of contexts, and software approaches to RVP have the great advantage of immediacy and flexibility. Overhead is non-trivial, however, and a good understanding of overhead and accuracy tradeoffs, as well as optimized predictor designs allows RVP to be applied appropriately. We have shown that an adaptive hybrid predictor can be efficiently implemented, maintaining accuracy levels comparable to non-adaptive designs at a fraction of the cost. This has a direct impact on consumer applications such as method level speculation, and also reveals interesting program understanding information. Our object-oriented, software interpretation of hardware predictor implementations also exposed the potential for several new predictor designs, including different stride and value variants as well as memoization approaches. This further fills out the space of predictors, and allows us to investigate a much more extensive hybrid design than is typically possible in hardware.

Our study suggests several possible routes to further improving RVP, both in terms of accuracy and overhead. At its simplest, performance can certainly be improved through tighter integration of predictor and VM environments. Our library separation facilitates experimentation, but callouts to the RVP library are expensive, and most predictor actions could be easily inlined with normal execution. We are currently investigating JIT integration where predictor code can be woven into the internal intermediate program representation with the rest of program code, allowing the full gamut of JIT optimizations to be applied. Further benefit would also be realized by making GC more aware of RVP data, allowing predictor tables to persist through GC.

Other authors have used proposed static program analyses to improve RVP, such as by estimating the better candidates for prediction [6]. There are many opportunities for analysis design, and we are developing several analyses that should reduce overhead and increase accuracy. A parameter dependence analysis, for example, determines which parameters actually affect the return value. This aids argument-based predictors by eliminating unnecessary state, improving memory consumption, hashtable sharing, and speed. A similar approach can be applied to method outputs. Not all return values are consumed, nor do they all need precise values, and we are investigating a return value use analysis to find return values that do not need prediction, or for which prediction can be less precise.

We have focused on software prediction for current availability and general flexibility. Hybrid software/hardware designs, however, are an obvious strategy for reducing overhead. Predictor virtualization, for instance, while intended more to reduce hardware predictor complexity, exposes some elements of hardware predictor state to software [3]. With suitable mechanisms for controlling hardware predictor activity, the benefits of software selection and high level choices can be combined with hardware speed. Hardware hashing components are part of many predictor designs [5, 42], but even a general hardware hash function instruction that combined a message of arbitrary length into a single word, would greatly reduce a significant source of RVP overhead.

Finally, in exploring the application of return value prediction, there are likely many other predictors that could be beneficial. These are easy to implement in our software framework, and they can improve the generality of our predictor unification framework. As explored in this paper, specific program location predictor accuracies, state, and behaviour can help with understanding program behaviour. More variety in predictor designs may reveal even further interesting program properties.

Acknowledgments

This research was supported by the IBM Toronto Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada.

References

[1] Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Transactions on Computers*, 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: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[3] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 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. *ACM SIGARCH Computer Architecture 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: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 222–233, June 2002.

[7] M. Burtscher and B. G. Zorn. Exploring last n value prediction. In *PACT'99: Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques*, pages 66–77, Oct. 1999.

[8] M. Burtscher and B. G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *JILP: Journal of Instruction-Level Parallelism*, 1:1–25, May 1999.

[9] M. Burtscher and B. G. Zorn. Hybridizing and coalescing load value predictors. In *ICCD'00: Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 81–92, Sept. 2000.

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

[11] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98: Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184, Oct. 1998.

[12] Y. Ding and Z. Li. A compiler scheme for reusing intermediate computation results. In *CGO'04: Proceedings of the International Symposium on Code Generation and Optimization*, 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: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 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] E. M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, Dec. 2002. `http://sablevm.org`.

[17] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA'01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 207–216, Jan. 2001.

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

[19] S. J. Jackson and M. Burtscher. Self optimizing finite state machines for confidence estimators. In *WISA'06: Proceedings of First Workshop on Introspective Architecture*, Feb. 2006.

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

[21] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, Dec. 2006.

[22] X.-F. Li, Z.-H. Du, Q. Zhao, , and T.-F. Ngai. Software value prediction for speculative parallel threaded computations. In *VPW1: Proceedings of the 1st Value-Prediction Workshop*, pages 18–25, San Diego, CA, June 2003.

[23] X.-F. Li, C. Yang, Z.-H. Du, and T.-F. Ngai. Exploiting thread-level speculative parallelism with software value prediction. In *ACSAC'05: Proceedings of the 10th Asia-Pacific Computer Systems Architecture Conference*, volume 3740 of *LNCS: Lecture Notes in Computer Science*, pages 367–388, Oct. 2005.

[24] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, Dec. 1996.

[25] 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: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, Mar. 2006.

[26] 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: Advances in Information and Computer Security: Third International Workshop on Security*, volume 5312 of *LNCS: Lecture Notes in Computer Science*, pages 152–166, Nov. 2008.

[27] G. H. Loh. Width-partitioned load value predictors. *JILP: Journal of Instruction-Level Parallelism*, 5:1–23, Nov. 2003.

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

[29] P. Marcuello, J. Tubella, and A. González. Value prediction for speculative multithreaded architectures. In *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 230–236, Nov. 1999.

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

[31] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT'99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, Oct. 1999.

[32] C. J. F. Pickett and C. Verbrugge. Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, Sable Research Group, School of Computer Science, McGill University, Oct. 2004.

[33] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *VPW2: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, pages 40–47, Oct. 2004.

[34] C. J. F. Pickett and C. Verbrugge. SableSpMT: A software framework for analysing speculative multithreading in Java. In *PASTE'05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 59–66, Sept. 2005.

[35] C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of *LNCS: Lecture Notes in Computer Science*, pages 304–318, Oct. 2005.

[36] C. J. F. Pickett, C. Verbrugge, and A. Kielstra. libspmt: A library for speculative multithreading. Technical Report SABLE-TR-2007-1, Sable Research Group, School of Computer Science, McGill University, Mar. 2007.

[37] N. B. Sam and M. Burtscher. Exploiting type information in load-value predictors. In *VPW2: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, pages 32–39, Oct. 2004.

[38] N. B. Sam and M. Burtscher. Complex load-value predictors: Why we need not bother. In *WDDD'05: Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, pages 16–24, June 2005.

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

[40] T. Sato and I. Arita. Table size reduction for data value predictors by exploiting narrow width values. In *ICS'00: Proceedings of the 14th International Conference on Supercomputing*, pages 196–205, May 2000.

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

[42] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, Dec. 1997.

[43] J. Singer and G. Brown. Return value prediction meets information theory. In *QAPL'06: Proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages*, volume 164 of *ENTCS: Electronic Notes in Theoretical Computer Science*, pages 137–151, Oct. 2006.

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

[45] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining Java native calls at runtime. In *VEE'05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 121–131, June 2005.

[46] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *ISCA'99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.

[47] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 281–290, Dec. 1997.

[48] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 439–453, Oct. 2005.

[49] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *PASTE'07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 75–82, June 2007.

[50] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in Java. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 130–139, Sept. 2007.