Return Value Prediction in a Java Virtual Machine

Christopher J.F. Pickett Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
{cpicke,clump}@sable.mcgill.ca

Abstract

We present the design and implementation of return value prediction in SableVM, a Java Virtual Machine. We give detailed results for the full SPEC JVM98 benchmark suite, and compare our results with previous, more limited data. At the performance limit of existing last value, stride, 2-delta stride, parameter stride, and context (FCM) sub-predictors in a hybrid, we achieve an average accuracy of 72%. We describe and characterize a new table-based memoization predictor that complements these predictors nicely, yielding an increased average hybrid accuracy of 81%. VM level information about data widths provides a 35% reduction in space, and dynamic allocation and expansion of per-callsite hashtables allows for highly accurate prediction with an average per-benchmark requirement of 119 MB for the context predictor and 43 MB for the memoization predictor. As far as we know, the is the first implementation of non-trace-based return value prediction within a JVM.

1 Introduction

Speculative multithreading (SpMT), otherwise known as thread-level speculation, is a dynamic parallelisation technique that splits sequential code into multiple threads, using out of order execution and buffering of main memory accesses to achieve speedup on multiprocessors. Although typically considered at a hardware level [3, 12, 17, 25], SpMT has shown potential to be effective in pure Java source code [14] and other software models [5, 18] as well.

More recently, speculative method-level parallelism (SMLP), an SpMT variant in which speculative threads are forked at method callsites (Figure 1), was shown to benefit significantly at the hardware level from accurate return value prediction for Java programs by Hu *et al.* [12]. Specifically, they ran six benchmarks from SPEC JVM98 on an 8-way multiprocessor in a realistic SMLP system, and reported the effect of return value prediction on speedups obtained (Table 1).

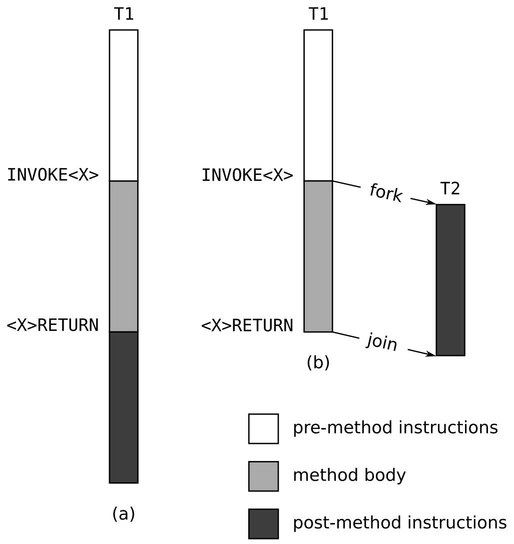

Figure 1: (a) *Sequential execution of Java bytecode.* The target method of an `INVOKE<X>` instruction executes before the instructions following the return point. (b) *Speculative execution of Java bytecode under SMLP.* Upon reaching a method callsite, the non-speculative parent thread `T1` forks a speculative child thread `T2`. If the method is non-void, a predicted return value is pushed on `T2`'s Java stack. `T2` then continues past the return point in parallel with the execution of the method body, buffering main memory accesses. When `T1` returns from the call, it joins `T2`. If the actual return value matches the predicted return value, and there are no dependence violations, `T2`'s buffered writes are committed and non-speculative execution jumps ahead to where `T2` left off, yielding speedup. If there are dependence violations or the prediction is incorrect, `T2` is simply aborted.

return value prediction strategy	average speedup
none	1.52
best hybrid	1.92
perfect	2.76

Table 1: *Effect of RVP on speedup under SMLP [12].*

On the basis of these data, we concluded that improvements beyond Hu *et al.*'s best hybrid were highly desirable, and set out to achieve high return value prediction accuracy in SableVM [10, 19], a Java Virtual Machine (JVM). Inspired by existing context and parameter stride predictors, we designed a new memoization-based predictor. Its resource requirements and performance are comparable to a context predictor, and a hybrid including memoization achieves excellent results.

By implementing our work directly within a mature JVM, we are able to analyse the complete language and complex benchmarks, taking exceptions, garbage collection, and native method calls into consideration. We provide detailed data for the full SPEC JVM98 benchmark suite, and expand on previous return value prediction results over SPEC in several ways.

In [12], the S1 (size 1) benchmark parameter setting that specifies minimal execution time was used. This results in the benchmarks spending most of their time in JVM startup code, and relatively little in application code [9], distorting conclusions about dynamic behaviour [8]. Our data were collected using S100, the value SPEC insists on for reportable results [23].

We also report prediction accuracy according to all nine primitive Java data types, which include floating point and 64-bit values. Previously, only boolean, int, and reference types were considered. Additionally, by explicitly accounting for exceptional control flow, we are able to analyse the `jack` benchmark, which was omitted from [12] due to its use of exceptions.

Lastly, our data include all runtime method calls, without inlining, whereas Hu *et al.*'s trace data were gathered with method inlining enabled. It can be argued that inlining is critical to efficient Java execution, and therefore an essential upstream property. However, inlining changes the set of methods available to predict, and unfortunately inlining decisions are not uniform across virtual machines. This makes it difficult to compare accuracy results between studies, and we give an example in Section 6 on the effect inlining has on apparent predictor accuracy. It may also be that the set of non-inlined callsites is not ideal as a set of potential fork points in SMLP, although Hu *et al.* did obtain good prediction accuracy and speedup in SMLP simulations over their subset of callsites, and noted that return value prediction in the absence of inlining appears to have little benefit [25].

In this respect, we pursue accuracy for its own sake, not simply as a means to an end, and hope to facilitate direct comparisons with our work. It seems likely that our improvements over the entire set of callsites will translate to most reasonable subsets, and that if our techniques were applied to Hu's simulation framework that improved accuracy and speedup would be obtained. We defer the treatment of a full SpMT implementation in SableVM to future work, and focus on exploiting the higher level techniques, resources, and information available to first push the performance limits of return value prediction, and then reduce memory requirements without sacrificing accuracy. We believe that our methodology provides a good basis for evaluation of return value prediction, and that an all-software SpMT implementation will ultimately benefit.

1.1 Contributions

1. An implementation of return value prediction in a JVM that includes previously reported last value, stride, 2-delta stride, context (FCM), parameter stride, and hybrid predictors. Predictions are made at runtime without reliance on trace data.

2. A comprehensive set of prediction data that includes every method invocation, for all of SPEC JVM 98, run with non-trivial settings. This significantly improves upon existing return value prediction data [12] for these benchmarks for the reasons stated above.

3. A new table-based *memoization* return value predictor, suitable for a JVM, and an investigation of its performance. Unlike context predictors, which compute lookup table indices from a recent history of return values, a memoization predictor computes an index from the method parameters. We find that it works well by itself, and complements existing predictors nicely in a hybrid.

4. An exploration of the performance limits of context, memoization, and hybrid predictors. In software, we can allocate predictor storage to the point where accuracy no longer improves, and reduce memory requirements by (a) expanding hashtables dynamically per callsite, and (b) exploiting VM level information about value widths.

2 Design

We implement a handful of previously described predictors. A last value predictor (L) simply predicts the last returned value. A stride predictor (S) computes the differential or stride between the last two values, and applies this differential to the last returned value to make a prediction. A 2-delta stride (2) predictor computes strides as per the stride predictor, but only updates when the last two strides are equal. A parameter stride (P) predictor captures return values that have a constant stride against one of the parameters [12]. Finally, an order-5 context predictor (C) computes an index into a hashtable using the previous 5 returned values [20, 21] (Figure 2a). We also present a new memoization (M) predictor that computes an index into a hashtable like the context predictor, but uses method parameters as hash function inputs instead (Figure 2b). All parameters are available as a sequence of 32-bit values on the Java stack [15], and we include the `this` parameter for non-static methods; zero is returned for static methods with no parameters. Finally, hybrid LS2PC and LS2PCM predictors keep a success history for each component sub-predictor over the last 32 return values and select the most accurate, choosing simpler strategies in the event of a tie.

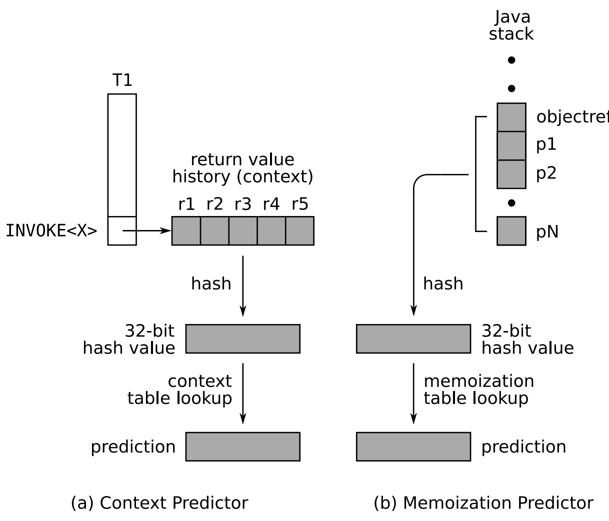

Figure 2: *Context and memoization predictor models. r1* through *r5* are the most through least recent return values, and *p1* through *pN* are method parameters.

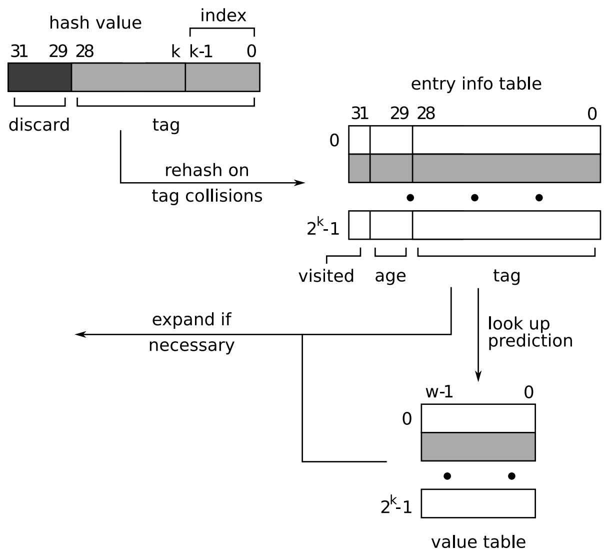

Figure 3: *Hashtable lookup and expansion model. k* is the table index width, and *w* is the actual value width. Data are aligned by separating values from entry info.

Other value prediction studies typically use global data structures for predictor storage, as the authors tend to be working under real-world hardware constraints. At the VM level we can afford to be more liberal with memory, and dynamically allocate and free arbitrary structures in general purpose RAM instead of expensive dedicated circuits. Predictor storage is allocated on a per-callsite basis, which has three important advantages: 1) if we need to synchronize predictor access, either to increase accuracy or to allow our data to move in memory, it means we do not need to acquire an expensive global lock; 2) it eliminates any chance of aliasing in a global table; 3) it allows us to allocate memory according the needs of individual callsites. We considered a per-method allocation strategy, and its advantages and disadvantages over a per-callsite strategy, but decided to defer an in-depth study to future work.

The L, S, 2, and P predictors require negligible amounts of memory per callsite. We use open addressing tables to store predictions for both C and M predictors [6]. These tables map easily to the hardware tables described in other value prediction designs, albeit under more stringent space requirements.

We initially experimented with Sazeides' standard context table index function [20]. A similar or identical function was used by Hu *et al.* when they studied return value prediction. We found an uneven distribution of hash values, and began looking for a better hash. Although an improvement to Sazeides' hash based on rotation was reported a few years later [1], we opted to use Jenkins' fast hash [13], on the basis of his detailed analysis and review. This hash has the nice property that every bit of the input affects every bit of the output, meaning that a single 32-bit or 64-bit hash value can be used reliably for multiple table sizes.

We find that for non-trivial programs, allocating large fixed size hashtables per callsite quickly exhausts available memory in the system. We overcome this limitation and realize the benefits of large tables by allocating initial minimum size 3-bit hashtables per callsite, and expanding them by 1 bit when the load exceeds 75%, up to a fixed maximum.

Each table entry has a 32-bit tag (Figure 3). Bits 0–28 hold a truncated Jenkins' hash value. They detect collisions, and allow us to rehash and expand up to a maximum size of 29 bits. Note that as the table size increases the ability to detect collisions decreases. Bits 29–30 are a two-bit saturating age counter that resets to zero upon returning an entry, and increments when an entry is skipped by rehashing. We find this is a better use of these bits than the standard two-bit saturating update counter. Bit 31 is a visited bit, and we use it to detect new entries, calculate load, and eliminate unnecessary rehashing during expansion.

Double hashing allows us to cycle through all table entries [6], and essentially eliminate collisions at non-maximum table sizes. These secondary hashes can be computed in a few cycles from an existing hash value:

```
h1 = hash_value & (size - 1);
h2 = h1 | 1;
rehash = (h1 + i * h2) & (size - 1);
```

where `i` is the rehash iteration, and `size` is the number of entries in the table. In order to avoid using modulo operations, `size` must be a power of two.

At the maximum size, upon detecting a collision, we rehash up to four times, at which point we replace the first-encountered oldest entry using the two-bit age counter. This yields up to a 5% increase in predictor accuracy depending on benchmark and maximum size.

3 Benchmark Properties

We present absolute numbers on the runtime properties of our benchmarks relevant to return value prediction in Table 2. These can be used to suggest specific areas of interest, convert percentages reported in future sections back to absolute numbers, and provide a general feel for dynamic behaviour and characteristics.

property	`comp`	`db`	`jack`	`javac`	`jess`	`mpeg`	`mtrt`
callsites	1.72K	1.89K	3.60K	5.12K	3.04K	2.17K	2.90K
forked	226M	170M	59.4M	127M	125M	111M	288M
aborted	36	18	608K	41.8K	290	114	62
void	93.4M	54.4M	24.4M	45.3M	23.3M	34.1M	20.5M
verified	133M	115M	34.4M	81.5M	102M	76.9M	267M
boolean `Z`	3.75K	11.1M	9.38M	17.5M	35.8M	24.3M	3.06M
byte `B`	0	0	580K	39.3K	0	0	0
char `C`	935	1.73K	1.55M	3.70M	6.65K	2.11K	9.84K
short `S`	0	0	0	73.3K	0	18.0M	0
int `I`	133M	48.0M	11.5M	36.5M	20.8M	34.6M	4.54M
long `J`	477	152K	1.23M	845K	101K	15.8K	2.13K
float `F`	0	0	0	96	280	7.81K	162M
double `D`	0	0	0	156	1.77M	56	188K
reference `R`	15.8K	56.2M	10.2M	22.9M	43.5M	32.7K	97.5M

Table 2: *SPEC JVM98 dynamic properties.* `raytrace` is omitted as `mtrt` is an equivalent multithreaded version. *callsites*: unique reached callsites at which one might fork a new speculative thread; *forked*: total number of method calls made over *callsites*; *aborted*: calls that never return to the callsite due to an uncaught exception in the callee; *void*: void calls that do return to the callsite; *verified*: non-void calls that return to the callsite, and at which we are able to verify whether our prediction is correct; this is *forked* − (*void* + *aborted*); *boolean* through *reference*: per-type verifiable predictions. Future prediction accuracies are always reported against total verifiable predictions.

Immediately, we note that our dataset is roughly 3 orders of magnitude larger than that presented in the study by Hu *et al.*, because we analyse S100 instead of S1 for SPEC JVM98, and because we do not inline any method calls. Second, we observe that by considering the complete set of data types, we expose the full diversity and relative proportions of method return types, and that this has the potential to significantly affect our assessment of prediction accuracy. For example, `mtrt` relies heavily on float methods, `mpeg` uses a surprising number of methods returning shorts, `comp` returns almost exclusively ints, and the remainder use more or less equal mixes of int, boolean, and reference calls. Third, there are large numbers of void calls in all cases, as previously reported, and if evenly distributed and also desirable as SMLP fork points, they might make our entire analysis moot. However, void methods are impure and completely side effect based by nature, and thus likely not the best candidates. Finally, the upper bound of 5.12K on the number of callsites translates to an upper bound on the memory required by context and memoization predictors at a given table size.

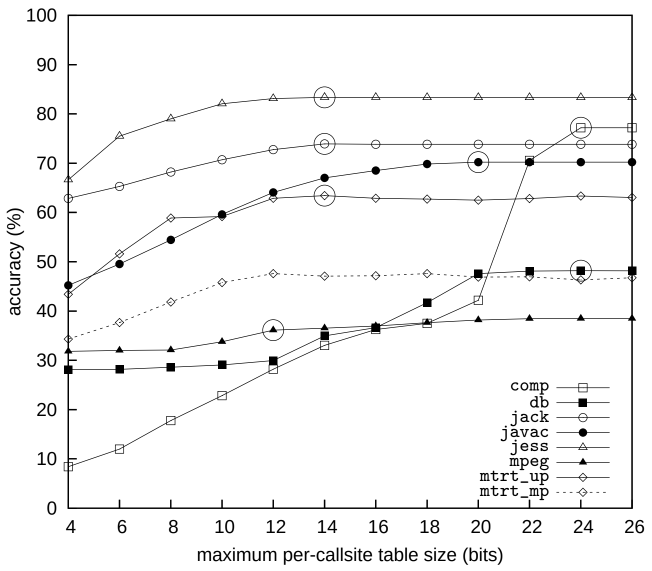

Figure 4: *Context size variation.*

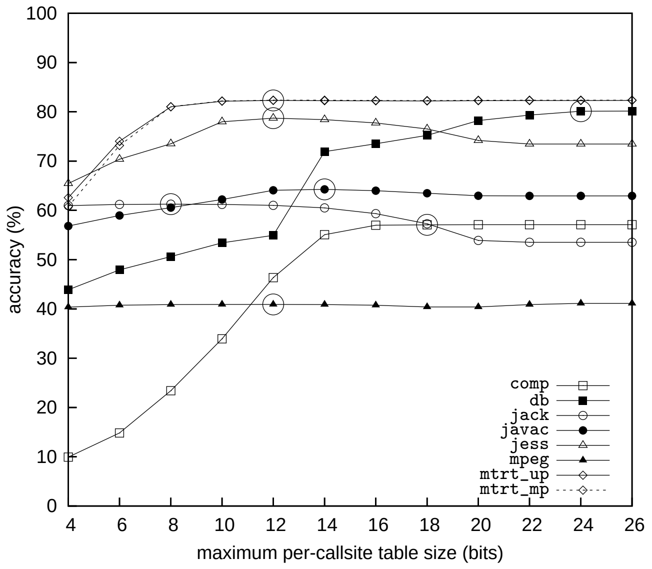

Figure 5: *Memoization size variation.*

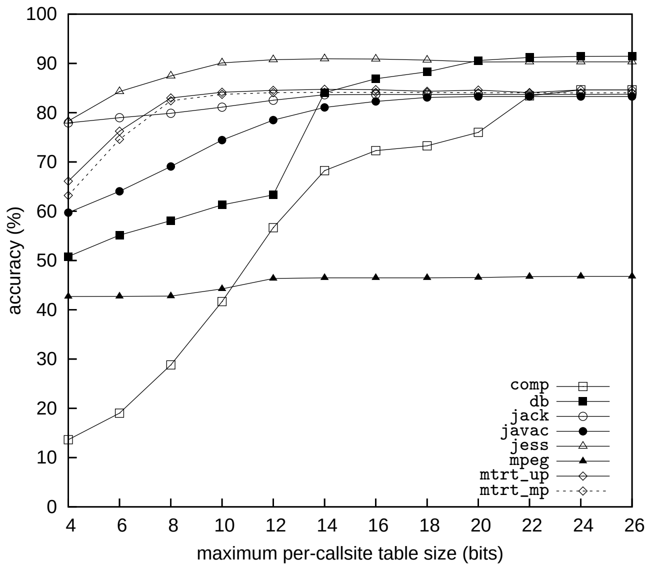

Figure 6: *Hybrid size variation.*

4 Size Variation

The performance of context (C), memoization (M), and hybrid LS2PCM predictors is graphed against varying maximum table size in Figures 4–6. C and M have the same maximum size in the hybrid graph. `jess` and `jack` perform better with C, whereas `db`, `mtrt`, and `mpeg` perform better with M. `comp` and `javac` perform better with M at small table sizes and C at large sizes. The mathematical compositions of these functions are neatly shown by the hybrid data. We discuss the poor performance of `mpeg` in Section 6.

`mtrt_up` and `mtrt_mp` refer to `mtrt` running on a uniprocessor and a multiprocessor respectively. There is a significant 15% performance hit when moving from `up` to `mp` for C, as the recent history of return values, or context, becomes rather volatile. This effect is not observed to the same extent for `up` because context-switching between threads is relatively infrequent. Interestingly, M is not history-based, and suffers few losses for `mp`. Furthermore, M is able to capture almost all of the `mp` predictions that C missed in the hybrid. As reported by Hu *et al.*, context prediction accuracy suffers badly in an SMLP environment, and these limited data suggest that memoization is able to compensate.

For `jack`, `javac`, and `jess` in M, we actually observe prediction accuracy start to decrease past a certain maximum. Based on tag information about collisions and visited entries, we attribute the peaks of these curves to different sets of parameters mapping to the same return value. We discuss future work to address this in Section 8.

We choose maximum table sizes per benchmark near to or at the performance peak for both C and M, indicated with an open circle around the point, and listed in Table 3. Note that a non-optimal size of 12 is chosen for `mpeg` in C as performance increases in the hybrid are negligible after 12 bits. We can afford to choose large table sizes for `comp` and `db` as only a handful of callsites ever need to expand that far. These sizes are used in all hybrid experiments following this section.

5 Memory Usage

Figures 7 and 8 show the final distributions of C and M tables when a maximum size of 26 bits is specified. There are slightly fewer tables in M than C, as static methods with zero parameters cannot be memoized. On average, we find that 70% and 65% of C and M tables respectively are never expanded past the initial size of 3 bits, and that 87% for each are never expanded beyond 8 bits. This indicates that a small number of methods in these benchmarks are responsible for the creation and processing of highly variable data, as table expansion at a given callsite is directly proportional to the variability of hash function inputs, namely return value history for C and method parameters for M.

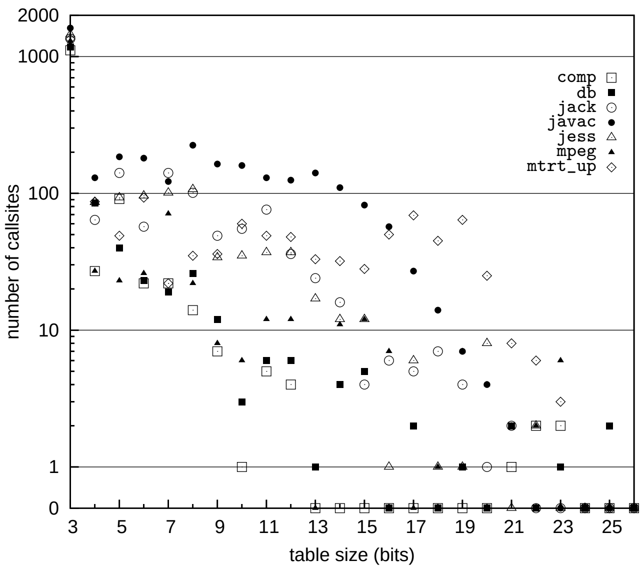

Figure 7: *Final context table distribution.*

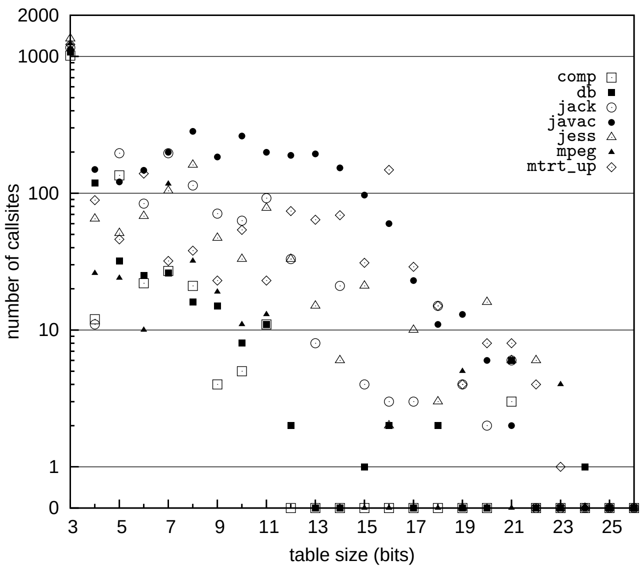

Figure 8: *Final memoization table distribution.*

In some cases (`comp`, `db`), it is important to capture this variability (as seen in Figures 4–6), which suggests greater reuse of intermediate results. In other cases, the variability is less important, and this allows us to conserve memory; for example, `mtrt` requires 2 GB of memory when fully expanded, yet shows no improvement in accuracy after 14 bits. The data for `mtrt_mp` are not shown, but differ only from `mtrt_up` by greater expansion in C between 8 and 22 bits, corresponding with `mtrt_mp`'s lower predictability (Figure 4).

Perhaps most importantly, these data indicate that a fixed size global table will be dominated by a small minority of callsites, evicting predictions belonging to those that are less variable. As less variable calls are by nature easier to predict, and furthermore appear in abundance, at least in the form of boolean return values, this provides good justification for our per-callsite allocation and expansion strategy.

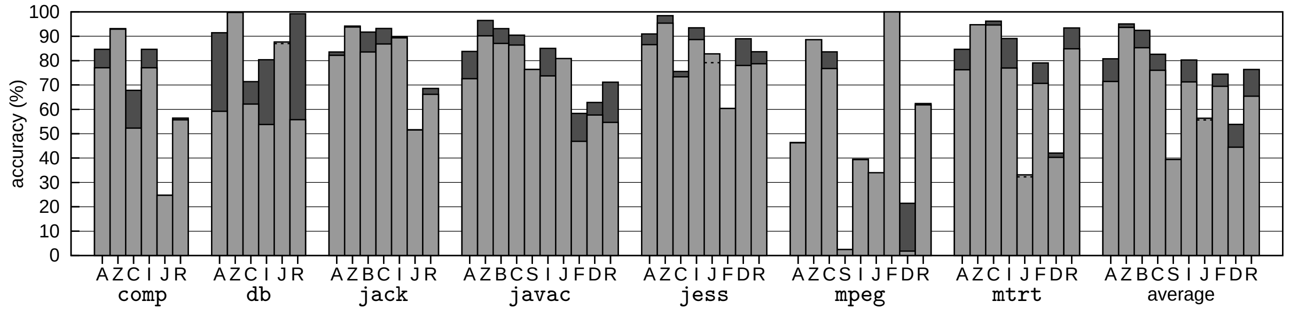

Figure 9: *Hybrid predictor performance.* The dark bars show the improvement of the LS2PCM hybrid over the LS2PC hybrid. In the three cases where accuracy is slightly worse (`jess(J)`, `db(J)`, and `mtrt(J)`), a dashed line is drawn. `A`: all verified predictions; `Z-D`: Java primitive types; `R`: object and array references. Types with zero calls are omitted for individual benchmarks and also excluded from the final average (refer to Table 2).

bench-	context			memoization		
mark	size	original	reduced	size	original	reduced
`comp`	24	313M	208M	18	9.60M	6.38M
`db`	24	541M	361M	24	345M	206M
`jack`	14	15.8M	10.7M	8	1.79M	1.11M
`javac`	20	291M	195M	14	103M	64.5M
`jess`	14	13.5M	9.59M	12	8.83M	5.62M
`mpeg`	12	3.72M	2.49M	12	1.46M	856K
`mtrt`	14	69.4M	46.4M	12	23.0M	15.3M
average	17	178M	119M	14	70M	43M

Table 3: *Context and memoization table memory.* The maximum table size in bits was chosen based on optimal points shown in Figures 4 and 5.

As was previously reported, taking value data widths into consideration can yield significant savings in table-based prediction strategies [16]. We allocate tables conservatively according to the widths of Java primitive types, which are 1, 8, 16, 16, 32, 64, 32, 64, and 32 bits for booleans, bytes, chars, shorts, ints, longs, floats, doubles, and references respectively. SableVM actually uses 8 bits to represent a boolean, and we have not packed these in our implementation. Data on the reduction achieved from data width considerations are given in Table 3.

On average, 33% and 37% memory reductions were achieved for context and memoization tables respectively. Furthermore, the average memory required by maximally-performing context predictors after reduction was 119 MB, whereas memoization required only 43 MB. This is seen in Figures 7 and 8, where the memoization data is shifted left, towards smaller sizes. This does not diminish the value of a context predictor, but strongly suggests that memoization will be effective in any environment where a context predictor is effective, provided information about which bytes in memory correspond to parameters is available.

Finally, we considered perfect hashing of booleans in a context predictor. Although 256^k entries are needed for perfect hashing of bytes in an order-k context table, if we treat all non-zero booleans as equivalent and only zero booleans as false, as would seem logical and as permitted by the JVM Specification [15], only 2^k entries are needed. This reduces memory usage if the initial table size is greater than k bits, and also improves speed, as the hash function is now trivial to compute. We found that accuracy was not significantly altered, and furthermore that space savings were marginal, as highly expanded boolean context tables are rare. We did not profile for speed.

6 Hybrid Performance

We construct an LS2PC hybrid out of previously reported predictors (Section 2). This is compared against an LS2PCM hybrid containing our new memoization predictor in Figure 9. Both C and M predictors in these hybrids benefit from benchmark-specific maximum table size selections described in Section 4, as well as rehashing and table expansion. LS2PC achieves an average accuracy of 72%, and LS2PCM 81%. Thus it appears that memoization complements existing predictors nicely. We evaluate the ability of our hybrid to capture correct sub-predictions as a percentage of the predictions that at least one sub-predictor got right, as the hybrid will always fail if none of the sub-predictors succeed, and find an average fitness of 96%.

Out of all the benchmarks, `mpeg` performs the worst at only 46%. Looking into the method call distribution, we observe that the majority of predictions are confined to `q.j(F)S` and `l.read()I`, and that these handle reading and processing of essentially random mp3 data. However, Hu *et al.* found `mpeg` to be highly predictable at 80% overall accuracy. `q.j(F)S` is a short leaf method, and was either excluded from their analysis, which only explicitly mentions booleans, ints, and references, or inlined by their JVM. `l.read()I` was possibly not inlined because it eventually invokes a native I/O operation and is not a leaf method; if so, this would explain their low int prediction accuracy of 42%, comparable to ours at 40%.

This provides a striking example of how reported prediction accuracies can differ depending on the subset of methods considered. The full set of methods needs to be analysed to allow meaningful comparisons, and to present data that include all classes of functions, even though a reduced set of fork-and-predict points is probably necessary in an effective SMLP environment.

7 Related Work

That software approaches to SpMT may also be viable was demonstrated by Kazi and Lilja on manually transformed Java programs [14], and comparable studies have been performed using C benchmarks [5, 18]. JVMs have also been advocated by others for SpMT related work; Chen and Olukotun, for example, give arguments for the value of VM level information [3], and have developed an extensive system in simulated hardware for dynamically parallelising Java programs [4].

Hu *et al.* analyse data from Java traces, and use simulated hardware to make a strong case for return value prediction in SMLP [12]. They present a parameter stride predictor and give prediction results for SPEC JVM98. Our work builds on their efforts by further including memoization in the analysis effort, and by extending the data collected.

Prediction strategies are of course numerous, and appear in various contexts. For value prediction, last value, context, and stride predictors of different forms have been introduced and examined by several researchers [1, 11, 17, 20, 21]. Limits on the possible success of such strategies have also been analysed [25], and many hybrid techniques have been proposed [2].

Although our work is the first to address memoization in a value prediction setting, memoization is obviously a well known technique, and has even been used to speed up speculative hardware simulations [22]. Effective memoization based compiler and runtime optimizations have also been described [7]. Note that unlike traditional memoization approaches, limitations due to conservative correctness are not necessary in our speculative environment.

Our use of data type information for memory optimization follows other work on using types in value prediction. Loh has demonstrated memory and power savings by using data width information [16], although in a hardware context, and with the additional need to heuristically discover high level type knowledge.

8 Conclusions and Future Work

We achieve high return value prediction accuracy in a JVM within reasonable software space constraints, implement and examine many previously described techniques for value prediction, and provide a powerful new table-based memoization predictor.

It is straightforward to extend our value prediction framework to include new kinds of predictors, for example a differential context (DFCM) predictor [11], and we encourage others to experiment with it. It would be particularly interesting to study prediction of reference types, as we have access to detailed object and heap layout information in the VM, and can also account explicitly for the effects of garbage collection. In an SMLP environment, we might also be concerned with predicting heap loads, and it should be possible to use the same codebase to develop general purpose predictors for `GETSTATIC`, `GETFIELD`, and `<X>ALOAD` instructions. Our work is available from the SableVM website [19] under a permissive LGPL license.

We have also developed some ahead-of-time compiler analyses in Soot [24] that are designed to improve the performance of return value prediction. A *parameter dependence* analysis that determines which parameters affect the return value can be used to minimize the set of inputs to our memoization predictor, decreasing table size while increasing sharing. A *return value use* analysis can be used to determine which return values are consumed, and eliminate the need to predict unconsumed values. That not all return values are consumed was previously reported, but the information was not apparently used during execution [12]. We can also substitute incorrect predictions for return values used only inside boolean and branch expressions, provided these expressions evaluate identically.

In Section 4, we showed that multithreaded programs may be more predictable by memoization than by a context predictor, as the history of return values at certain callsites is subject to concurrent and nondeterministic updates. Similar problems were observed with context predictors in full SMLP systems [12]. We would like to explore this phenomenon in greater detail on a larger set of multithreaded benchmarks and programs actually running under SMLP, and determine the extent to which memoization is able to compensate.

We have shown that a memoization predictor requires less resources than a context predictor, while offering a significant increase in accuracy, and believe that a hardware implementation is feasible. Other context predictors are implemented with a global prediction table [12, 20], whereas we used per callsite tables exclusively, and a comparison of the advantages and disadvantages of allocating tables at the global, method, and callsite levels is in order. Finally, rather than profile for optimum table sizes, we will implement dynamic adjustment of expansion parameters per callsite, based on prior prediction success and the importance of correctly predicting the return value in question.

We intend to finish our implementation of a working SMLP execution engine in SableVM, and hope to achieve speedup on existing multiprocessors. It will be interesting to examine the benefits and costs of return value prediction in this system.

Acknowledgements

We kindly thank Etienne M. Gagnon for discussions and implementation help with SableVM. This research was funded by FQRNT, and C. Pickett was additionally supported by a Richard H. Tomlinson Master's in Science Fellowship and an NSERC PGS A award.

References

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

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

[3] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 1998.

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

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

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[7] Y. Ding and Z. Li. A compiler scheme for reusing intermediate computation results. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, page 279. IEEE Computer Society, Mar. 2004.

[8] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168. ACM Press, Oct. 2003.

[9] L. Eeckhout, A. Georges, and K. de Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–186. ACM Press, Oct. 2003.

[10] E. M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montréal, Québec, Dec. 2002.

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

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

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

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

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

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

[17] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Oct. 1999.

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

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

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

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

[22] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 283–294. ACM Press, Oct. 1998.

[23] The SPEC JVM Client98 benchmark suite. `http://www.spec.org/jvm98/jvm98/`.

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

[25] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 221–230. IEEE, Sept. 2001.