Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation

Amir Roth, Andreas Moshovos and Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W Dayton Street, Madison, WI 53706
{amir, moshovos, sohi}@cs.wisc.edu

Abstract

We introduce dependence-based pre-computation as a complement to history-based target prediction schemes. We present pre-computation in the context of virtual function calls (v-calls), a class of control transfers that is becoming increasingly important and has resisted conventional prediction. Our proposed technique dynamically identifies the sequence of operations that computes a v-call's target. When the first instruction in such a sequence is encountered, a small execution engine speculatively and aggressively pre-executes the rest. The pre-computed target is stored and subsequently used when a prediction needs to be made. We show that a common v-call instruction sequence can be exploited to implement pre-computation using a previously proposed prefetching mechanism and minimal additional hardware. In a suite of C++ programs, dependence-based pre-computation eliminates 46% of the mispredictions incurred by a simple BTB and 24% of those associated with a path-based two-level predictor.

1 Introduction

Accurate branch and target prediction is an important factor in maintaining a continuous supply of useful instructions for processing. Traditional prediction is done dynamically and works by capturing patterns in the outcome (direction or target) history of the branch stream. While these techniques handle many of the control transfers found in programs, some continue to resist. In this paper, we present a complementary approach in which branch targets are correlated with the instructions that compute them rather than with previous targets. Our mechanism pre-computes targets rather than guessing at them statistically.

In this initial work, we concentrate on pre-computing *virtual function call* (*v-call*) targets. The v-call mechanism enables code reuse by allowing a single static function call to transparently invoke one of multiple function implementations based on run-time type information. As shown in Table 1, v-calls are currently used in object-oriented programs with frequencies ranging from 1 in 200 to 1000 instructions (4 to 20 times fewer than direct calls and 30 to 150 times fewer than conditional branches). V-call importance will increase as object-oriented languages and methodologies gain popularity. For now, v-calls make an ideal illustration vehicle for pre-computation techniques since their targets are both difficult to predict and easy to pre-compute.

Conventional target prediction techniques are branch target buffers (BTB), which store a single target per static instruction, and path-based two-level predictors [3, 6], which associate targets with static instruction/target-history combinations. BTBs cannot effectively handle v-calls, which feature multiple targets per static instruction. On the other hand, since v-call targets are determined by object type, *path-history* based schemes can exploit the correlation between multiple v-calls to the same *object reference* to predict the targets of the subsequent v-calls. However, they frequently mispredict the initial v-call in such a sequence and must contend with aliasing that results from the incorporation of irrelevant information into the target history.

A *pre-computation* approach, in which targets are pre-computed using "recipes" extracted from the executing program, can avoid these problems. Pre-computation's program-based nature gives it many advantages. Mimicking program actions directly rather than extracting statistical correlations from address streams translates into *improved prediction accuracy.* A program based representation is also *more compact* than a history based one and *avoids the aliasing problems* that plague the latter. Finally, pre-computation has a *shorter learning period* and *works even in the absence of statistical correlation*. Although it is a general purpose technique that can be applied to any type of instruction, v-call pre-computation has a particularly simple hardware implementation that leverages a previously proposed mechanism for prefetching linked data structures [15]. As a

complement to conventional prediction, pre-computation reduces mispredictions by 24%.

In the rest of the paper, Section 2 presents a more detailed overview of the problem and our proposed solution. We describe an implementation in Section 3 and evaluate it in section 4. The last sections discuss related work and our conclusions.

2 Target Pre-Computation

The purpose of this section is two-fold. We begin with a more detailed discussion on the effectiveness of conventional, history-based techniques in predicting v-call targets. Then we describe a common implementation of v-calls and show how we can exploit this implementation to create a *pre-computation* solution that avoids some of these difficulties. We support our qualitative arguments with data and examples from the OOCSB C++ benchmark suite, a collection of programs that has been used to study indirect call target prediction [6], and Coral [14], a deductive database developed at the University of Wisconsin. We compiled the programs for the MIPS-I architecture using the GNU GCC 2.6.3 compiler with flags `-O2` and `-finline-functions`. Table 1 summarizes the benchmarks, their input parameters and dynamic instruction counts. It also shows the total number of static and dynamic v-calls for each benchmark.

2.1 Conventional Target Prediction

History-based schemes for predicting v-call targets are the branch target buffer (BTB) and the proposed, though as yet unimplemented, path-based two-level predictor [3, 6]. The simple BTB records the last target of all control transfer instructions, including v-calls. BTBs are effective for control transfers that have only one possible taken target, like direct calls and conditional branches. They do not effectively predict v-calls, which feature multiple targets per static instruction, unless individual static v-calls exhibit high levels of temporal target locality. Path-based two-level predictors overcome this problem by maintaining a history of recent indirect targets and associating targets with static instruction/history pairs. Additional data in Table 1 shows v-call target misprediction rates for two predictor configurations. The BTB configuration uses a 4-way associative, 2K-entry BTB. PATH adds a direct-mapped, untagged 2K-entry path-based predictor that uses a three target (4 bits per target) history.

Although PATH eliminates 68% of the mispredictions incurred by BTB, there are still cases that it cannot handle. To understand these hard to predict cases, we look to the programs themselves. A high rate of v-calls implies iteration over a collection of objects with potentially multiple v-calls per object reference. Since v-call targets are determined by object type, it is not surprising that v-calls using the same object reference are highly correlated. As a result, once the first v-call target is known, the second and subsequent same-reference v-calls can usually be predicted by history-based techniques. Predicting the first v-call's target is more difficult. Here, history-based schemes may succeed if objects are statistically ordered by type, however this is not always the case. Objects may be ordered randomly, as in *richards*, or in an input-dependent way, as in *ixx* and *porky.* History based schemes must also contend with *aliasing*, which occurs when irrelevant targets are incorporated into the history. Aliasing, often due to interspersed calls to multiple objects, occurs frequently in *eqn* and *troff*.

Program	Description	Input Parameters	Dyn Inst Count	V-Calls		BTB		PATH	
				Stat	Dyn	Misp	90%	Misp	90%
coral	*deductive database*	4 joins, 110 relations	170M	54	1350K	11.0%	1	0.1%	1
deltablue	*constraint solver*	10,000 constraints	179M	16	4250K	1.5%	2	0.0%	1
eqn	*equation formatter*	eqn.input.all	71M	111	100K	28.1%	18	21.7%	24
idl	*IDL compiler*	all.idl.i	81M	516	1756K	0.1%	4	0.1%	16
ixx	*IDL parser*	Som_PlusFresco.idl	49M	157	102K	7.2%	10	3.7%	9
lcom	*VHDL compiler*	circuit3.l	192M	321	1103K	2.2%	31	1.6%	47
porky	*SUIF middleware*	g2shp.snt	323M	270	2636K	24.7%	5	6.4%	12
richards	*OS simulator*	50 processes	372M	7	3290K	54.0%	1	20.1%	1
troff	*text formatter*	gcc.1	98M	116	827K	20.3%	7	10.7%	8

Table 1. OOCSB benchmarks. *Static and dynamic v-call counts. V-call misprediction rates on a 4-way associative, 2K-entry BTB and a 2K-entry direct-mapped path-based two-level predictor with a history depth of three targets. Number of static calls that account for 90% of all BTB and PATH mispredictions.*

2.2 Our Solution

We introduce dependence-based pre-computation as a complement to history-based prediction. Pre-computation is a program-based technique that captures the target generation process and mimics it to obtain predictions. By considering information relevant to a particular v-call in isolation, pre-computation avoids the aliasing problems that plague history based schemes schemes. Pre-computation can also supply predictions in the absence of statistical target correlation.

Pre-computation is a general technique that can be applied to any type of instruction. However, v-call target pre-computation can be implemented as a minimal addition to a previously proposed prefetching mechanism [15]. The simple implementation exploits a common v-call implementation: the virtual function table (*vtbl*) shown in Figure 1. Part (a) shows an example of a class Base and a class Derived that implement methods (class functions) Valid and Print. The v-call mechanism allows the programmer to treat an array of objects of mixed types Base and Derived uniformly and use a v-call to invoke the correct function implementations. A *vtbl* contains the addresses (at predetermined offsets) of all the methods accessible by a given object class. Every object is initialized with a pointer to the vtbl corresponding to its class and accesses its functions via this pointer as shown in figure 1(b). The characteristic v-call implementation, which we call *OVFC*, accesses the vtbl using a sequence of three dependent loads and an indirect call. The initial load accesses the base of the Object, the second uses the object's base address to access its Vtbl, and the third retrieves the Function address which is used by the indirect Call. The OVFC sequences corresponding to the Valid and Print calls are shown boldfaced in figure 1(c). Note that both VFC chains use the same object reference O. Likewise, a single VFC chain may be dynamically attached to multiple static Os due to the use of conditionals.

Target pre-computation is a simple process. For a given v-call site, we isolate the OVFC sequence that calculates the target address. Whenever an instance of the first instruction in the sequence (O) completes, we quickly pre-execute the rest and calculate a target in anticipation of having to make a prediction. A calculation that completes before the actual v-call (C) is encountered can be used to supply a prediction. We show the pre-computation process in the abstract in Figure 1(d) scenario 1 (shaded). The left and right most parts of the figure show the execute and fetch schedules, respectively, for the loop from part (a) of the figure. Per our proposed solution, as soon as the instruction 3 (O) completes, we consult an internal representation and pre-execute the remainder of the dependence chain (VFC). The pre-computed target is buffered and used to predict the next fetched instance of instruction 6. We call these kinds of pre-computations *simple pre-computations*.

Figure 1(d) contains one critical abstraction, namely the misrepresented distance between the launch of the pre-computation (completion of instruction 3) and the prediction that uses its result (fetch of the instruction that succeeds 6). On an aggressive processor, this distance

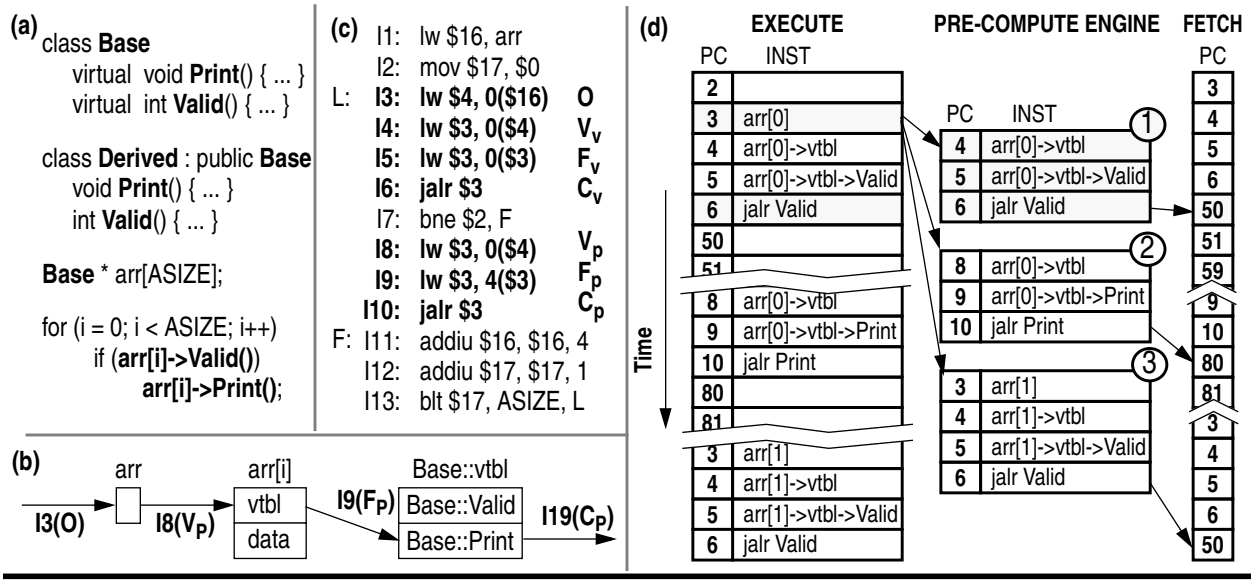

FIGURE 1. V-call mechanism and pre-computation. *(a) Classes Base and Derived both implement functions Valid and Print. A collection of these objects may be treated uniformly as a collection of Base objects. (b,c) Runtime v-call target disambiguation is performed with code that traverses a per class-type function table through a special pointer found in each object. (d) The v-call disambiguation mechanism can be pre-computed to achieve a target prediction effect.*

may not exist at all. Pre-computation relies on this separation to provide a timely predictions. A distance of only a few dynamic instructions implies that we may not be able to complete the pre-computation in time.

Fortunately, two common programming constructs provide us with the necessary dynamic instruction spacing. First, many object references are used to make multiple v-calls, each of which will be increasingly distant from the object reference. In our example, each reference to the object in arr[i] is used to make calls to Valid and Print. While the call to Valid is probably too close to the object reference arr[i] to receive a timely prediction, the subsequent call to Print is probably distant enough as shown in Figure 1(d) scenario 2. Second, many objects are typically kept in data structures, like arrays or lists, that are amenable to address prediction [2, 13]. While referencing one object in the data structure we can predict the address for the next object and use this predicted address to launch pre-computations. For instance, in scenario 3 of Figure 1(d) we launch a pre-computation for arr[1]->Valid using the address arr[0] and stride information. Pre-computations launched with the help of object address predictions are called *n-lookahead pre-computations* with *n* being the object reference distance between the current object and the address predicted object (n=1 in our example). An implementation of simple and lookahead pre-computation is described in the next section.

3 Mechanism

Our implementation of dependence-based pre-computation comprises three main components. The first is responsible for detecting data dependences among loads and between loads and indirect calls and representing these dependences internally. The second is a simple dataflow engine that uses this internal representation to aggressively execute dependent chains that terminate in indirect calls. The final piece, and the focus of our presentation, collects the pre-computation results, orders them and orchestrates their use by the processor's main target prediction mechanism. We describe an implementation of the basic solution, then show how it can be extended to include lookahead pre-computations.

3.1 Performing Pre-Computations

We borrow the first two components, for dependence detection and speculative dataflow pre-execution, from the previously proposed dependence-based prefetching (DBP) mechanism [15]. Initially designed for prefetching linked data structures, these components can be used virtually unchanged to capture and pre-execute OVFC sequences, which are essentially chains of pointer dereferences. The dependence predictor is a cache in which each entry represents a true data dependence between a load that produces (loads from memory) an address and a subsequent load which consumes (dereferences) that address. For target pre-computation we modify the predictor to recognize a dependence between a load and an indirect call. Loads that complete can access the predictor to determine which other loads can be speculatively issued using the just-loaded value as an input address. A separate execution engine services these loads and sends those that themselves produce addresses back to the predictor to potentially launch other loads. The actions performed by the pre-execution engine have no architecturally visible effect and are scheduled to minimize interference with the main program.

3.2 Matching Pre-Computations to Predictions

The component that concerns us most in this paper is the pre-computation/prediction interface. For our technique to succeed, pre-computations must be paired with their intended predictions. We call this process *correspondence* since it amounts to maintaining a one-to-one correspondence between pre-computations and predictions. Correspondence has two parts. First, a pre-computation must figure out which *static* v-call it belongs to. This is easy given our dependence framework which names the v-call explicitly as part of the OVFC chain representation. The more difficult part is matching a pre-computation with the *dynamic* instance of that v-call. A wrong dynamic pairing can often result in an incorrect prediction, and there are many ways to produce wrong pairings. A pre-computation may arrive late in which case it must discarded rather than used spuriously to predict a future v-call instance. Similarly, an early pre-computation must be buffered until it is needed.

To solve the problem of dynamic correspondence we exploit *non-interleaving*, a common property of the dynamic execution of programs. Consider a pair of static program instructions which are dynamically data dependent, like the O and C endpoints of an OVFC chain. Non-interleaving says that a dependent O/C pair does not occur in a dynamic execution interleaved with another dependent O/C pair. In other words, O and C sequences appear in programs as O_1, C_1, O_2, C_2 and never as O_1, O_2, C_1, C_2. If O_2 were to intercede between O_1 and C_1, it would overwrite the value written by O_1 and convert C_1 to C_2.

Non-interleaving helps us to enforce correspondence by ensuring that every object reference O_i is associated with at most one dynamic instance C_{Xi} of any dependent static v-call C_X and that this C_{Xi} appears before O_{i+1}. As a result, if a simple pre-computation corresponding to C_{Xi} then it must have been initiated by O_i. Similarly, a correct n-lookahead pre-computation for C_{Xi} must have been launched by O_{i-n}. To exploit non-interleaving, pre-computations must be ordered using their launch sequence (the O_i) rather than their completion order.

In the next section, we present a correspondence implementation for handling simple pre-computations. Before we proceed, we note that non-interleaving is not a universal property. However, when it does occur, it is the result of an explicitly interleaving optimization like software pipelining and is not random, but rather highly structured. Although we do not discuss it in this paper, it is simple to extend our dependence framework to capture these cases and correct for them.

3.3 Implementing Simple Correspondence

To exploit non-interleaving, a mechanism is required to *order* pre-computations, or rather their launching object references, with respect to predictions. Since pre-computations are launched at load completion time and retrieved during fetch, this is accomplished by assigning to each fetched instruction a monotonically increasing *Fetch Sequence Number* (FSN). The FSN travels with an instruction until it completes. A pre-computation is tagged with the FSN of the object reference that launched it.

With pre-computations and predictions sequenced using FSNs, we implement correspondence using two small tables: the *Pre-computation Buffering Table* (PBT) and the *Pre-computation Ordering Table* (POT). The PBT stores the latest completed pre-computation for each static v-call while the POT stores the FSN of the last instance of each static object reference. At prediction time, information in the POT is used to determine whether or not the pre-computation buffered in the PBT is the correct one.

The PBT is indexed by *static v-call* and records the predicted target (TARG), the PC of the launching object reference (O) and the FSN of any pre-computation intended for that static v-call as it completes. The POT is indexed by *static object reference* and records the FSNs of object references as they are fetched. In order for the POT to distinguish between object references and other loads, POT entries are *allocated* on pre-computation completion (using the pre-computation O field) and only *updated* at fetch time.

At prediction time, a v-call instance accesses the PBT and retrieves a pre-computation corresponding to its *static* instruction. The next step is to decide whether it was intended for the current *dynamic* v-call instance. Recall, the correspondence criterion for a simple pre-computation is that it was launched by the most recent instance of a static object reference. To verify this condition, we first obtain the static object reference using the O field in the PBT. We determine the last *dynamic* instance of this object reference by indexing the POT using O. If the FSN found in the POT matches the FSN attached to the pre-computation in the PBT, then the pre-computation was indeed launched by a most recent object reference and can be used to make a prediction.

We work through an example in Figure 2. In part (a), as arr[1] is fetched, the POT entry corresponding to arr[i] is updated with the latest FSN. In part (b) arr[1] completes and launches a pre-computation for arr[1]->Print, the pre-computation is tagged with the launching instruction's PC (O) and FSN. On completion, the pre-computation is deposited in the PBT. In part (c), we predict a target for arr[1]->Print. A pre-computation is retrieved from the PBT using the arr[i]->Print PC (action 1, circled). To determine whether or not the pre-computation corresponds we use its object identifier (O) to access the POT (action 2). Next we compare the pre-computation FSN with the most recent object reference FSN (action 3) Since they match, we know that the pre-computation was launched by the most recent arr[i] (arr[1]) and was intended for the current instance of arr[i]->Print (arr[1]->Print). The precomputation is forwarded to the prediction unit (action 4). Finally, in part (d) arr[2] is fetched and the POT is again updated. This action effectively invalidating the pre-computation in the PBT as the

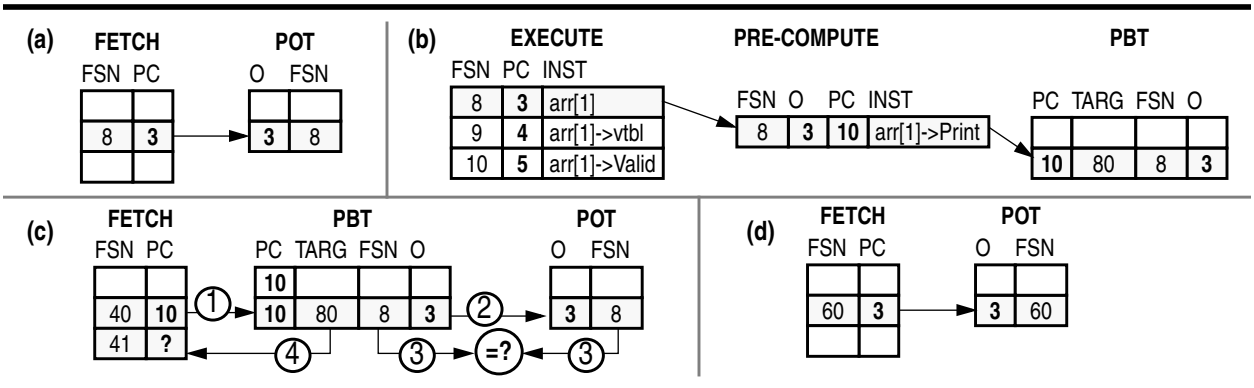

FIGURE 2. Simple Correspondence Working Example. *(a) arr[1] is fetched and updates the POT. (b) arr[1] completes and launches a pre-computation for arr[1]->Print which is entered into the PBT on completion. (c) When a prediction for arr[1]->Print is needed, we verify that the object reference that launched the pre-computation in the PBT is the most dynamic instance of that instruction. Verification succeeds, and the pre-computation can be used. (d) When the next instance of arr[i] is fetched, the POT update invalidates the pre-computation.*

object reference that launched it is no longer the most recent one.

3.4 Adding One Instance Lookahead

In a lookahead scheme, object references that exhibit either recurrent behavior (pointer chasing) or arithmetically regular input (stride) can launch multiple pre-computations. A pre-computation scheme is referred to as *n*-lookahead if an object reference O_i is used to predict the address of and launch pre-computations for object reference O_{i+n}. Implementing a general n-lookahead pre-computation scheme is easy, but potentially expensive and not extremely useful. We have found a 1-lookahead solution to be both cheap and effective.

Implementing 1-lookahead correspondence is straightforward. Recall, a simple pre-computation can be used as a prediction if it was launched by a most recent object reference instance. Similarly, a 1-lookahead pre-computation is valid if it was initiated by the object reference instance prior to the most recent one. To be able to check this condition, we extend the POT to track the last *two FSNs* per static object reference. We also expand the PBT to buffer *two pre-computations* per static v-call (the most recent 1-lookahead pre-computation and the most recent simple pre-computation). Finally, each pre-computation is tagged with lookahead bit. At prediction time, both pre-computations are checked in parallel against the same POT entry. The simple pre-computation FSN must match the most recent object reference FSN, while the lookahead pre-computation must reference the prior one. In case of multiple matches, priority is given to the simple pre-computation since, not relying on address prediction, it is more likely to be accurate.

4 Evaluation

In this section, we evaluate the effectiveness of target pre-computation. We present the reduction in v-call misprediction rates observed when pre-computation is added to a history-based predictor. We do not display direct performance data for the following reason. The frequency with which our benchmark suite executes v-calls, while relatively high, is low in absolute terms. Consequently, even a complete elimination of target mispredictions is not expressed in speedups of over 3%.

Although timing data is not shown, cycle-level simulation is still used to ensure that unreasonable assumptions about the completion of pre-computations are not made. Our results were obtained using the SimpleScalar [2] simulator. We simulate a conventional 5-stage, 4-wide superscalar pipeline with a maximum of 64 in-flight instructions. We model speculative, out-of-order issue and require loads to wait until all previous store address are known. Our memory hierarchy consists of 32KB, 32B-line, 2-way associative first-level instruction and data caches with 1 cycle access latency and a unified 512KB, 64B-line, 4-way associative second level-cache with a 12 cycle latency. Latency to memory is 70 cycles. We allow a maximum of 4 simultaneously outstanding data cache misses, and model contention throughout the memory system. Conditional branches are predicted using an 8K-entry combined predictor with 10-bit history gshare on one side and 2-bit counters on the other. Again, we simulate two target prediction schemes: BTB uses a 2K-entry, 4-way associative BTB, PATH adds a 2K-entry direct mapped 2-level predictor with a history length of three targets to handle v-calls. Our pre-computation machinery consists of a 256-entry dependence predictor, and 64-entry POT and PBT struc-

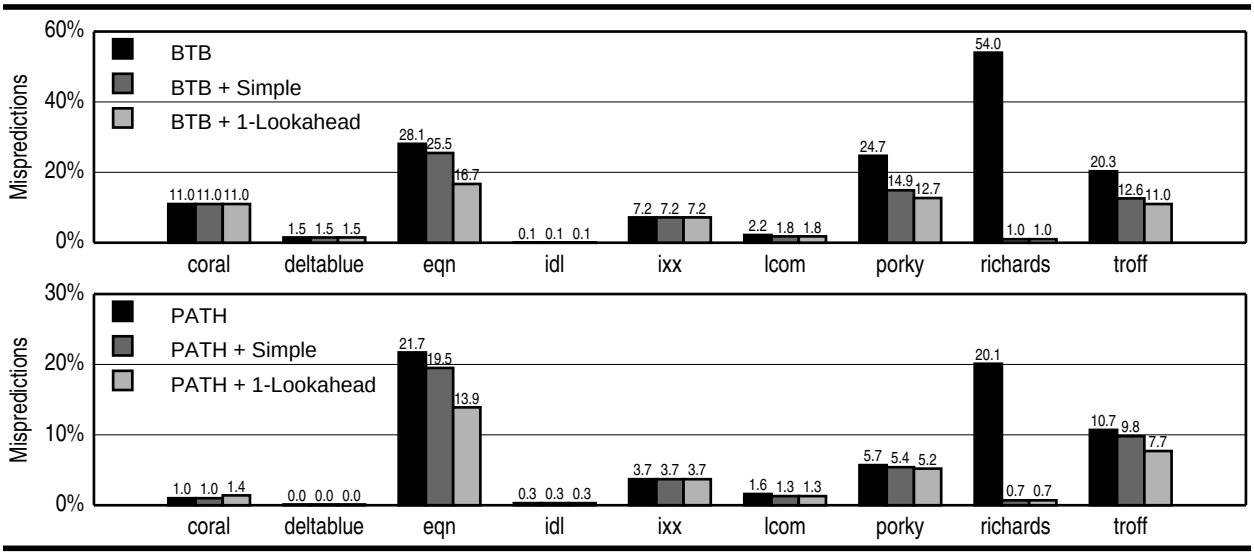

FIGURE 3. Misprediction Rate Impact. *V-call target mispredict rates for both (a) BTB and (b) path-based two-level predictor base configurations. Each base configuration was also augmented with a simple pre-computation scheme and a one instance lookahead scheme.*

tures. Speculative dataflow pre-execution uses two dedicated address generation units and the processor's data cache ports, although only when they are idle.

4.1 Impact on Misprediction Rates

We report the reduction in v-call mispredictions observed when dependence-based pre-computation is used to complement both our BTB and PATH predictor configurations. For each base mechanism we evaluate two pre-computation schemes: a simple scheme and one that adds 1-lookahead pre-computations. Figure 3 shows these results. Overall, simple pre-computation reduces v-call mispredictions by 42% over BTB and 21% over PATH. These numbers grow to 46% and 24% respectively when we add 1-lookahead pre-computation. As we predicted, there is some synergy between path-based prediction and pre-computation, especially for the 1-lookahead scheme which attacks mispredictions that are not easily captured by statistical correlation.

The greatest improvement using the simple scheme is observed for *richards*, an operating system scheduling simulator. *Richards* has a single, atrociously unpredictable static v-call. Fortunately, the associated object's address is available well in advance of the call itself, giving simple pre-computations ample time to complete and supply useful predictions. Significant improvements are also observed for *eqn*, *lcom*, *porky* and *troff*. These benchmarks contain many uses of single object reference/multiple v-call sequences and iteration over object data structures with one or multiple v-calls per iteration. These are the programming constructs we initially identified as being suitable for simple pre-computation and lookahead pre-computation, respectively.

As for the other benchmarks, *idl* and *deltablue* are highly predictable even with a simple BTB and provide little room for improvement. In *coral*, a single static v-call contributes over 90% of the mispredictions. However, the underlying object structure is a database tree that is not address predictable at its outer levels. Although object types are statistically correlated, accounting for the effectiveness of the path-based predictor, object addresses cannot be correctly predicted by our 1-lookahead mechanism. Although *ixx* uses v-calls, it also employs C++ features that make the detection of OVFC chains impossible using our simple dependence detection scheme. We do not investigate a more general dependence detector in this work.

4.2 A Closer Look at Pre-Computation

To provide further insight and support our earlier assertions, we break down the positive and negative contributions of our technique. On the positive side we are interested in the fraction of would-be mispredictions that were *corrected* by pre-computations. On the negative side, we count the number of mispredictions *introduced* by pre-computations. These are would-be correct predictions that were overturned.

The graph in Figure 4 breaks down corrected mispredictions for the 1-lookahead scheme. The left bar in each group shows results for the BTB based configuration, the bar on the right for the PATH based configuration. Each bar shows the fraction of would-be mispredictions that were either corrected by a simple pre-computation (black portion, bottom), corrected by a 1-lookahead pre-computation (gray portion, middle) or received an incorrect pre-computation (white portion, top). Pre-computations are not always available due to either insufficient time or an inability to correctly predict the next object's address, as indicated by the missing portion of each bar.

We make three observations from this plot. First, most v-call mispredictions are sufficiently distant from their associated object references, giving even simple pre-computations time to complete. Timely pre-computations are available for most mispredictions, at least for benchmarks with significant *a priori* misprediction rates like *eqn*, *porky*, *richards*, and *troff*. Second, when a pre-computation is available, it is usually accurate (at least 75% of the time and completely accurate in some cases). Finally, both availability and accuracy are higher in the BTB context than in PATH with most of this discrepancy is due to a reduction in relative effectiveness of the simple scheme. This is not a surprise since both path-based predictors and simple pre-computation attack the same kinds of v-call mispredictions, namely second and

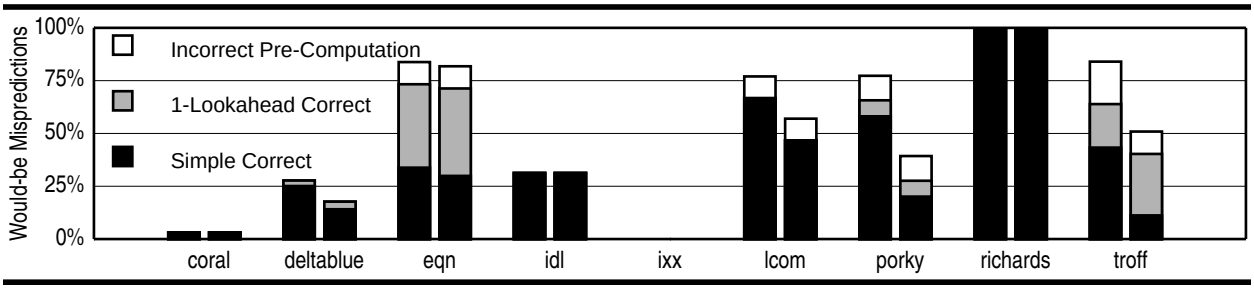

FIGURE 4. Breaking Down Pre-Computation Effects *Would-be mispredictions broken into incorrect predictions, correct predictions supplied by simple pre-computation and correct predictions supplied by lookahead pre-computation. The missing portion of each bar indicates that no pre-computation was available.*

subsequent v-calls to the same object reference.

The numbers in Figure 4 do not match up precisely with the misprediction rates shown previously. The reason for the mismatch is that, although it mimics program actions, pre-computation occasionally provides incorrect targets, mainly due to conditional execution and lookahead address misprediction. Introduced mispredictions occur when incorrect pre-computations override would-be correct BTB or PATH predictions. The fraction of correct predictions turned into mispredictions varies from 0% to 4% in the BTB configuration and from 0% to 7% in the PATH configuration with the difference attributed to the fact that PATH simply produces more correct predictions. In those rare cases where more mispredictions are introduced than corrected, for instance in *coral*, a confidence scheme can be used to selectively disable pre-computation. Our experiments found that the addition of confidence mostly eliminates introduced mispredictions, but also slightly reduces the observed benefit for *eqn*, *troff*, *porky* and *richards* as some impending mispredictions attain unjustifiably high confidence levels.

In addition to eliminating introduced mispredictions, a confidence mechanisms can also be used to reduce the number of *unnecessary* pre-computations. These are pre-computations that typically arrive late, are unsuccessful at correcting mispredictions or attack v-calls that are likely to be predicted correctly anyway. We note, however, that while unnecessary pre-computations may be not be ideal, they do not directly disturb the executing program. Pre-computations rarely miss in the cache and even when they do, they often provide a beneficial prefetching effect. The only adverse effect of unnecessary pre-computations is contention with other, potentially useful, pre-computations.

5 Related Work

Indirect jump target prediction in general, and v-call target prediction in particular, have been the subject of some recent investigation. A number of software methods have been proposed that convert v-calls into cheaper, and more predictable, direct calls [1, 5, 9, 10]. Where applicable, these techniques are preferable to hardware solutions since they reduce the cost of the call itself *and* improve its target predictability. They also expand the scope of compiler analysis and enable further optimizations. However, these methods have drawbacks as well: they may be overly conservative, incur software misprediction detection and recovery overhead, replace unpredictable v-calls with equally unpredictable branches, or duplicate code. One emerging possibility is to duplicate our pre-computation process in either software, using architected branch speculation constructs like the PlayDoh architecture [12] branch target register file and prepare-to-branch instructions, or microcode [4]. In both cases, multithreading may be used to hide the cost of the pre-computation.

A similar volume of work has been done in the area of hardware prediction mechanisms although most target indirect calls in general rather than the somewhat narrower class of v-calls. All of these [3, 6, 7, 11] use some form of history (branch or path based) BTB indexing. The advantage of these techniques is arguable hardware simplicity and leverage of existing control prediction structures and techniques. These techniques improve prediction accuracy for a larger class of indirect control transfers, but do not perform as well on v-calls in particular. The main reason for this deficiency is that relevant pieces in the history are both difficult to isolate and vary on a call-by-call basis; recent work [11] has begun to attack this problem.

The use of pre-execution techniques for branch prediction is a still more recent development [8, 16]. The *branch flow window* approach [8] uses static instruction tagging to copy instruction sequences that compute branch outcomes into a separate buffer. To produce predictions, the computation is executed without side effects using predicted values as inputs. This technique has been shown to apply to branches in general, but can handle only a single loop-resident, input-predictable branch at a time. Our pre-execution method uses *load-value dependences* to drive a decoupled execution engine. Load value dependences and the components that capture and manipulate them were initially proposed for use in prefetching linked data structures [15].

6 Summary and Future Directions

We introduce dependence-based pre-computation as a complement to history-based methods for v-call target prediction. Our mechanism leverages previously proposed prefetching hardware to capture the characteristic instruction sequence that computes the v-call target. Given the appropriate input, it then pre-executes the sequence to supply a target prediction. Dependence-based pre-computation reduces v-call target mispredictions 46% over a BTB and 24% over a path-based two-level predictor. We make the following contributions:

- We introduce pre-computation as a complementary method for supplying target predictions for virtual function calls. We argue that history based schemes are fundamentally limited by target history aliasing and a lack of correlation across objects, and show that pre-computation has the potential for overcoming these limitations.
- We show that a common v-call implementation allows us to leverage a previously proposed dependence-based prefetching mechanism to capture and perform the appropriate pre-computation.

- We show that correspondence, the problem of matching pre-computations with predictions, can be solved by exploiting a common property of dynamic program execution. We devise a compact scheme to implement correspondence reliably.

The work presented in this paper is an initial foray into the area of dependence-based target pre-computation, and the proposed implementation simply demonstrates the potential power of this technique. Many other implementations are possible, and several are likely to be more effective, efficient, and practical. In addition, the design space we outlined has not been fully explored and the interaction between different points along each of its dimensions is unclear. How would two-instance lookahead perform? Is there a benefit to adopting different lookahead policies on a static v-call basis? These and many other questions are open. Along other fronts, work has already begun on using pre-computation variants to attack chronically mispredicted conditional branches and pre-computation is gaining popularity as a general purpose technique with many applications.

Acknowledgements

The authors thank Craig Zilles for his comments on several drafts of this paper, and the anonymous referees for their suggestions. This work was supported in part by NSF grant MIP-9505853 and by an equipment donation from Intel. Amir Roth is supported by a Cooperative Graduate Fellowship from IBM.

References

[1] G. Aigner and U. Hoelzle. Eliminating Virtual Function Calls in C++ Programs. In *Proc. 10th European Conference on Object Oriented Programming*, Jun. 1996.

[2] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[3] P-Y. Chang, E. Hao, and Y.N. Patt. Target Prediction for Indirect Jumps. In *Proc. 24th Annual International Symposium on Computer Architecture*, pages 274–283, Jun 1997.

[4] R.S. Chappell, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. 26th Annual International Symposium on Computer Architecture*, May 1999.

[5] G. DeFouw, D. Grove, and C. Chambers. Fast Interprocedural Class Analysis. In *Proc. Annual Conference on Principles of Programming Languages*, pages 222–236, Jan 1998.

[6] K. Driesen and U. Hoelzle. Accurate Indirect Branch Prediction. In *Proc. 25th International Symposium on Computer Architecture*, pages 167–178, Jun. 1998.

[7] K. Driesen and U. Hoelzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Proc. 31st International Symposium on Microarchitecture*, pages 249–258, Dec. 1998.

[8] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st International Symposium on Microarchitecture*, pages 59–68, Dec. 1998.

[9] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *Proc. 10th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–128, Oct 1995.

[10] U. Hoelzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Technical report, Stanford University, 1994.

[11] J. Kalamatianos and D.R. Kaeli. Predicting Indirect Branches via Data Compression. In *Proc. 31st International Symposium on Microarchitecture*, pages 272–281, Dec. 1998.

[12] V. Kathail, M. Schlansker, and B.R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, HP Laboratories, Feb. 1994.

[13] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Program. In *Proc. 10th International Conference on Supercomputing*, pages 133–139, May 1996.

[14] R. Ramakrishnan, W.G. Roth, P. Seshadri, D. Srivastava, and S. Sudarshan. The CORAL Deductive Database System. In *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, pages 544–545, 1993.

[15] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[16] A. Roth and G.S. Sohi. New Methods for Exploiting Program Structure and Behavior in Computer Architecture. In *Proc. 2nd International Workshop on Innovative Architecture*, pages 24–28, Oct. 1998.