This is an update of the paper that appears in the *Proceedings of the 5th Workshop on Multithreaded Execution, Architecture, and Compilation,* pages 23-34, Austin TX, December, 2001. It includes minor text changes and revised results.

The Predictability of Computations that Produce Unpredictable Outcomes

Tor Aamodt Andreas Moshovos Paul Chow

Department of Electrical and Computer Engineering
University of Toronto
{aamodt,moshovos,pc}@eecg.toronto.edu

Abstract

We study the dynamic stream of slices (i.e., slice traces) that lead to branches that foil an existing branch predictor and to loads that miss and measure whether these slices exhibit locality (i.e., repetition). We argue that this regularity can be used to dynamically extract slices for an operation-based predictor that speculatively pre-computes a load address or branch target (i.e., an outcome) rather than directly predicting the outcome based upon the history of outcomes. We study programs from the SPEC2000 suite and find they exhibit good slice locality for these problem loads and branches. Moreover, we study the performance of an idealized operation-based predictor (it can execute slices instantaneously). We find that it interacts favorably with an existing sophisticated outcome-based branch predictor, and that slice locality provides good insight into the fraction of all branch mispredictions it can potentially eliminate. Similar observations hold for operation-based prefetching of loads that miss. On average slice locality for branches and loads was found to be above 70% and 76% respectively when recording the 4 most recent unique slices per branch or load over a window of 64 committed instructions, and close to 68% and 71% for branches and loads respectively when we look at slices over a window of up to 128 committed instructions. The idealized operation predictor was found to correct approximately 74% of branch mispredictions or prefetch about 67% of loads that miss respectively (slices detected over a window of 64 instructions). At the same time, on average, the branch operation predictor mispredicts less than 0.8% of all branches that are correctly predicted by an existing branch predictor.

1 Introduction

Recently, the prospect of generalized *operation-prediction* has been raised as a way of boosting accuracy over existing *outcome-based* predictors. In operation prediction we guess a sequence of operations, or a computation *slice* that can be used to pre-compute a performance critical outcome (e.g., load address or branch target). This is in contrast to outcome-based predictors that directly predict outcomes exploiting regularities in the outcome stream. Since operation prediction does not require any regularity in the outcome stream, it has the potential of predicting outcomes that foil existing outcome-based predictors (in section 2, we provide an example that illustrates the potential of operation prediction).

Several recent proposals have shown that *slice-based precomputation* (the mechanism operation-prediction uses for predicting outcomes) can be used to successfully prefetch memory data, and may potentially be used to pre-compute hard to predict branches [4,9,15,16,10,11,12,17]. In this work, we study program behavior to understand *why operation-prediction works or may work* for predicting otherwise hard to predict program events.

We build on the experience with outcome-history-based dynamic prediction and study whether typical programs exhibit the behavior necessary for *operation history-based prediction* to be successful. We explain that, in a way that parallels outcome-based prediction, operation predictors can be built to exploit regularities in the *operation* (i.e., computation) stream. For example, previous work has shown that sufficient *locality*, or repetition exists in the value stream of many programs. This program characteristic is what facilitates outcome-based value prediction. In this work we study a set of programs from the SPEC2000 suite to determine whether sufficient repetition exists in the *slices* used to calculate performance critical outcomes that otherwise foil existing outcome-based predictors. This program characteristic is necessary (but not sufficient as we explain in section 2) if history-based operation prediction is to be successful. We restrict our attention to mispredicted branches and to loads that miss and study how much repetition, or *locality* exists in the operation streams that lead to them. To the best of our knowledge, no previous work on the dynamic locality characteristics of such slices exist. With few exceptions and as we explain in section 4, related proposals approach slice pre-execution as an alternate execution model, where the compiler orchestrates slice generation and pre-execution. While compiler directed slice pre-execution is an interesting and viable option, dynamic slice detection and execution can have its own advantages (e.g., binary compatibility). Accordingly, we believe it is an important alternative that deserves attention.

Our study provides the foundation necessary for understanding whether programs exhibit some of the behavior necessary for operation prediction. Moreover, our results provide insight on what kind of operation predictors we should be considering if we are to achieve a desired accuracy and coverage. For example, our study shows how successful a *last-operation* predictor can potentially be or whether *pattern-based operation* predictors may be necessary. A last-operation prediction would simply record the slice used to calculate a branch or load and use it the next time around to pre-calculate the branch or the load address. Such a predictor can be successful only if slices tend to repeat multiple times. Alternatively, a pattern-based operation predictor can exploit patterns in slice occurrence, e.g., slice S1 appears always after slice S2, and so on. While more complex, a pattern-based operation predictor could offer better accuracy and coverage over a last-operation one. However, in this work we restrict our attention to analyzing the potential of operation prediction. Specifically, the predictors we studied precompute their slices instantaneously. An actual predictor would require some time to execute through the predicted slice, hence it may not be able to pre-execute the slice early enough for prediction purposes. Further work is necessary to determine whether this is possible. Yet, in previous work we have shown that a simple predictor for loads that miss can successfully pre-execute loads that miss often for a set of pointer-intensive applications [9].

Our results indicate that performance critical slices exhibit high locality, more so for loads that miss. In particular, we find that average slice locality for branches and loads is above 70% and 76% when we record up the 4 most recent slices per branch or load respectively over a window of 64 committed instructions and close to 68% and 71% for branches and loads respectively when we look at slices over a window of up to 128 committed instructions. Our idealized operation predictor can correctly predict about 74% of mispredicted branches and accurately prefetch 67% of loads that miss (slices detected over a window of 64 instructions). At the same time, on the average the branch operation predictor mispredicts less than 0.8% of all branches that are correctly predicted by an existing branch predictor. Overall, we find that coverage (e.g., the fraction of branches that get a correct prediction from the operation predictor but an incorrect prediction from the existing outcome-based predictor) is highly correlated to the locality exhibited by the corresponding slices.

The rest of this paper is organized as follows. Section 2 reviews operation prediction, how it relates to outcome-based prediction, and the various choices existing when dynamically extracting slices. Section 3 presents our locality and accuracy results. In Section 4, we discuss related work explaining how operation prediction relates to other recently proposed slice-based execution models. Finally, Section 5 summarizes our findings and offers concluding remarks.

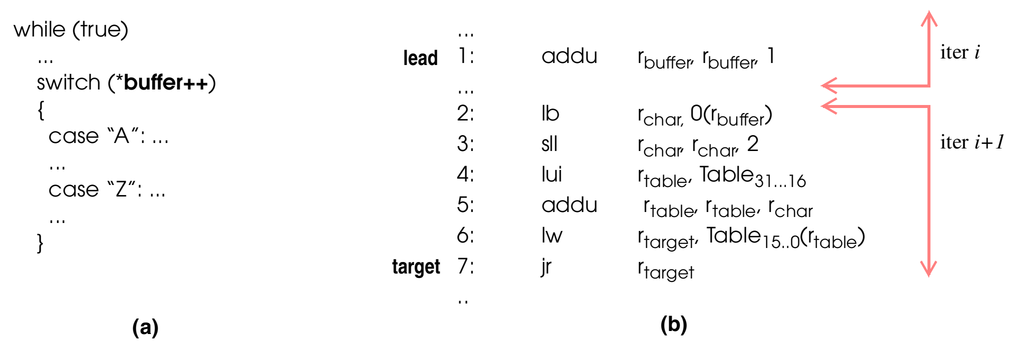
2 Operation Prediction Basics

In this section we review operation prediction, explain how it relates to existing outcome-based predictors, and discuss what requirements exist for operation prediction to be successful. In section 2.1, we discuss some of the choices that exist in dynamically extracting slices and explain the choices made for the purposes of our study.

Consider the example code fragment of figure 1(a). It is an infinite while loop containing a switch statement. What particular target the switch statement will follow depends on the value read from the uni-dimensional buffer. First, consider how an outcome-based predictor will attempt to predict the branch that implements the switch statement. Such a predictor will observe the outcome stream of this branch (and possibly of other branches also). That is, it will observe the various targets taken by the switch statement during successive iterations of the while loop. It will try to associate each target occurrence with an appropriate *target history,* that is a sequence of past targets that preceded the one in question. The hope is that next time the same target history appears, the same target will follow. For example, such a predictor may observe that when the targets for “A” and “B” appear, then with high probability the target for “C” appears. This predictor may then guess “C” every time “A” and “B” appear in sequence. Essentially, the outcome-based predictor builds a tabular, approximate representation of the program’s function by observing the values (outcomes) it generates. Outcome-based prediction is successful if the outcome-stream exhibits sufficient repetition, a property commonly referred to as *locality.* In our example code, repetition will exist only to the extent that the data stored in the buffer array follows some repeatable pattern. Operation prediction offers the potential of predicting outcomes that do not necessarily follow a repeatable pattern. Rather than trying to guess the program’s function based on the values it produces, it directly observes the computation stream, attempting to exploit any regularities found there. Returning to our switch statement example, let us now take a closer look at what happens during execution time. Figure 1(b) shows how the switch statement is implemented in pseudo-MIPS machine code. When the code of part (a) executes, the computation stream will contain repeated appearances of the computation *slice* shown in part (b). While the target computed by each slice may be different, we can observe that the actual slice remains constant. Operation prediction builds on this observation and attempts to *dynamically* identify such slices and use them to pre-compute outcomes that otherwise foil outcome-based predictors. As we explain in section 4, operation prediction has existed in restricted form for years. For example, stride-based prefetchers or value predictors are examples of specialized operation prediction where the actual slice or class of slices is built in the predictor design.

In this work we are concerned with generalized operation prediction where the slices are dynamically extracted and predicted. Following a generalization of the model proposed by Moshovos *et al.,* [9], an operation predictor for our example would identify the “jr” (instruction 7) as a problematic control flow instruction, or as a *target* instruction. At commit time, it will extract the computation slice that lead to the particular instance of the target instruction as shown in part (b). This slice, will contain only the instructions that contributed to the calculation of the actual target. Note that these instructions are not necessarily adjacent in the dynamic instruction trace (a mechanism for extracting such slices has been proposed [9]). This slice will be stored in a *slice cache* where it will be identified by the *lead* instruction (i.e., the oldest one, instruction 1 in our example). Next time the lead instruction appears in the decode stage, the slice will be executed as a separate *scout thread.* Provided that the scout thread completes before the appropriate instance of the target instruction appears, the processor may use its result to predict the target. The aforementioned steps for operation prediction parallel those for outcome-based prediction. In operation prediction the unit of prediction is a slice while in outcome-based prediction it is an outcome. Accordingly, detecting a slice and storing it in the slice cache is equivalent to observing an outcome and recording it in a prediction table. Executing a scout thread is equivalent to probing the prediction table.

The operation predictor described uses history-based prediction concepts. Such a predictor observes the slices of otherwise unpredictable results. If these slices tend to follow a repeatable pattern then it may be possible to use the past history of appearances to accurately predict the slices of future instances and hence pre-compute otherwise unpredictable outcomes. The same principle underlies many existing outcome-based predictors where instead of exploiting regularity in the slice stream we instead exploit regularity in the outcome stream (e.g., values, addresses and branch direc-



while (true)
...
switch (*buffer++)
{
case "A": ...
...
case "Z": ...
...
}
(a)
...
lead 1: addu r_buffer, r_buffer, 1
...
2: lb r_char, 0(r_buffer)
3: sll r_char, r_char, 2
4: lui r_table, Table31...16
5: addu r_table, r_table, r_char
6: lw r_target, Table15..0(r_table)
target 7: jr r_target
..
(b)
iter i
iter i+1

Figure 1: *A switch statement whose target behavior depends on the data stored within the buffer array. (b) The computation slice that calculates the target during run-time.*

tions). For history-based operation prediction to be successful it is necessary to have sufficient regularity in the computation, or slice stream of the instruction we want to predict. Moreover, the slices so identified must be able to execute and complete before the main thread needs the prediction itself. In this work we focus mainly on the first requirement. In particular, we study the slice locality characteristics of some SPEC2000 programs focusing on branches that are mispredicted by an outcome-based branch predictor and on loads that miss.

Before we present our results it is necessary to re-iterate why scout threads may be able to run-ahead of the main thread and to comment on how operation-prediction relates to outcome-based prediction. Scout threads may be able to pre-calculate a result because: (1) The main thread includes all other intervening instructions which need to be fetched, decoded and executed. (2) The main thread also may be stalled due to intervening control-flow miss-predictions. Since scout threads do not include any control flow, they may proceed undisturbed. Finally, while operation prediction may be able to predict outcomes that do not exhibit regularity, it does need to calculate these outcomes. Outcome-based prediction forgoes this calculation replacing it with a straightforward table lookup. Hence, whenever outcome regularity exists outcome-based prediction may be preferable over operation prediction.

2.1 Slices and Slice Locality

Before defining and measuring slice locality we must be clear about how we define a *slice*. Conceptually, a slice may include instructions that appear long in advance (e.g., thousands of instructions) of the target instruction. Moreover, a slice could be defined to contain arbitrary control-flow and memory dependences (to adhere to the static definition of a computation slice). With this definition, the slice for each instance of the "jr" instruction in figure 1 would include all preceding instances of instruction 1 (updates of the buffer pointer), plus all instructions that wrote the corresponding data element of the buffer array (this may include instructions past a system call). Such a definition is impractical for our purposes. Accordingly, our slice definition stems from a practical implementation of a slice detector [9] and of the sketch of how an operation predictor could work discussed earlier. In the rest of this section we explain the choices we made in defining and extracting slices, and then we present our definition of slice locality.

Slice Detection Window: In searching for instructions to construct a slice, we consider only those instructions that appear within a fixed distance from the target instruction. In particular, we extract slices using a fixed length *slice detection window* or *slicer*. The instructions in the slicer form a continuous chunk of the dynamic instruction trace. Only committed instructions enter the slicer. When a target instruction is committed, its slice is extracted using a backwards data-flow walk which eliminates all operations that do not directly contribute to the target outcome. Slicer size affects slice length and therefore it impacts slice locality and the ability to pre-execute slices early enough. While a shorter slicer may result in fewer shorter slices per target instruction and hence in higher repetition in the dynamic slice stream, the distance between the target and lead instructions in these slices could be small. Consequently, it may be harder for those slices to run-ahead of the main thread. For this reason we experimented with various slicers of 32, 64, 128, and 256 instructions.

Control-Flow: Besides how far back we look in the dynamic instruction trace, a second choice in detecting slices is whether we include intervening control-flow instructions. In this study we do not. Slice detection occurs over a chunk of the dynamic instruction trace. Since this is a trace, it only includes a specific control-flow path and does not contain the parts of the static slice that would appear on other control-flow paths. Accordingly, from a practical standpoint it is convenient to ignore any intervening control flow instructions. Later on we explain, that the *implied control flow path* (i.e., the directions of all intervening branches at detection time) can be used to select the appropriate slice for prediction.

Memory Dependences: Another choice regarding slices is whether we follow memory dependences including stores and their parents. Conceptually, the following tradeoffs exist: Including memory dependences may allow us to look further in the past, capturing a lead instruction that appears further away from the target. Moreover, including memory dependences may improve slice accuracy since, if a memory dependence exists, we will be waiting appropriately for the corresponding data. However, since memory dependences may be changing over time, including them could result in incorrect slices. As we found that the impact on locality was typically quite small, we restrict our attention to slices that follow memory dependencies.

Slice Size: Slices with only one instruction (the target), are always discarded in this study, as the practical implementation discussed earlier cannot use them to any benefit. We could also choose to restrict our attention to those slices that contain at most a fraction of all instructions in the slicer. While including more instructions may allow us to capture an earlier lead instruction, at the same time it has several, potentially negative implications: First, it reduces the chances of pre-executing the resulting slice in time. Second, it may increase slice detection latency and complexity. Finally, more space is required to store longer slices. At the extreme, we could include all instructions in the slice detection window, however, the chances of actually pre-executing such a slice are rather slim. We have experimented with two choices: Not restricting the number of instructions (e.g., up to 64 instructions may appear in a slice detected using the 64-entry slicer), and only considering those slices that contain as many instructions as the 1/4 of the slicer entries (i.e., 32, 16 and 8 for the 128-, 64- and 32-entry slicers). Restricting slice size results in fewer slices being detected.

Comparing Slices: Slices contain multiple instructions. For this reason and in contrast to outcomes, there are several ways in which two slices can be compared for the purposes of measuring locality. In this study, we consider two slices identical if they are *lexically identical*. That is, if they contain the same instruction sequence. With this definition two slices may be considered equivalent even if the PCs of individual instructions may differ. This definition is both practical and it accommodates identical slices that may appear on different control-flow paths. For the purposes of locality measurements we ignore the implied control flow in slices. So two slices that are lexically identical but appear on different control flow paths and have different implied control flow will be considered the same.

Slice Locality: For unrestricted slices (i.e. for any length, even slices containing just the target operation), we can now define *slice-locality(n)* of a target instruction as the relative frequency with which a detected slice was encountered within the last *n unique* slices detected by preceding executions of the same static instruction. *Slice-locality(1)* is the relative frequency that the same slice is encountered in two consecutive executions of a target instruction. A high value of *slice-locality(1)* suggests that a simple, "last slice encountered"-based predictor could be accurate. For values of *n* greater than 1, *slice-locality(n)* is a metric of the working set of slices per instruction. Formally, it is the relative frequency with which the same slice was detected within the last *n* unique slices detected for the specific instruction, assuming there is always a slice. When excluding slices due to the restrictions considered earlier, *slice-locality(n)* is the relative frequency that a given branch or loads's slice both meets the restriction criteria, and was seen in the last *n* unique slices that also matched the criteria. While a small working set does not imply regularity, we will later explain that it may be possible to execute all these slices in parallel and then select the appropriate one based on the implied control flow.

Detection Context: In practice, having identified a problem instruction, one might detect a slice and record it independent of the whether the underlying outcome based predictor was correct, or choose to record a slice only when a misprediction or cache miss actually occurs. The difference is that an outcome may only be hard for the outcome-based predictor to anticipate when following the implied control-flow of a small subset of all slices seen. We have measured the impact on locality as viewed from mispredicted branches and cache misses under both circumstances and conclude that statistically there is a benefit to waiting for a mispredicted target branch, or load that misses, when detecting slices for a particular static branch or load. Except where stated otherwise (i.e., in Section 3.2.3) all measurements reported in this paper are based upon the latter approach.

3 Measurements

We start by detailing our methodology in section 3.1. In Section 3.2, we report our slice locality analysis first for branches (Section 3.2.1) and then for loads (Section 3.2.2). Here we are interested in determining whether sufficient locality exists in the slice stream of mispredicted branches or of loads that miss. This is a property of the program (and of the underlying slice detection mechanism). In Section 3.2.3 we explore the impact of the detection context on slice-locality. In Section 3.3, we study how a specific operation predictor interacts with an existing outcome-based predictor for branches and how well it predicts the addresses of loads that miss. The operation predictors we studied execute slices *instantaneously* when a lead instruction is encountered. Our goal is to understand the potential of slice prediction. Further work is necessary to develop realistic predictors where slice execution takes some time. Our results provide the insight necessary to do so in a well educated manner.

3.1 Methodology

We have used the programs from the SPEC2000 suite shown in table 2. All programs were compiled with gcc (-O2 -funroll-loops -finline-functions) for the MIPS-like Simplescalar instruction set (PISA). We have used the test input data sets. To obtain reasonable simulation times, we skipped the initialization phase and warmed up the caches and the branch predictor for the next 25 million instructions. The actual number of instructions skipped (i.e., functionally simulated) is shown in table 2. Our measurements were made over the next 300 million instructions. In table 2, we also report the L1 data cache miss rates and the branch prediction accuracies (direction and target address). In the interest of space, we use the labels shown in table 2 in our graphs. To obtain our measurements we have modified the Simplescalar 3 simulator. Our base configuration is an 8-way dynamically-scheduled superscalar processor with the characteristics shown in table 1. Our base processor has a 12 cycle minimum pipeline.

3.2 Slice Locality

In this section we study the locality of slices first for branches and then for loads. For branches, we focus on those dynamic instances that are mispredicted by the underlying outcome-based predictor and study whether locality exists in their slice stream. This is necessary if history-based operation-prediction is going to be successful. For loads, we focus on those dynamic instances that miss in the data cache. In both cases we examine only the slices that lead to mispredictions, or cache misses, respectively (except in Section 3.2.3 were the impact of the detection context is more closely examined).

Measuring locality in the way we do here allows us to avoid any artifacts that a specific implementation of operation prediction may introduce. Later in section 3.3, we study models of specific operation predictors.

3.2.1 Branch Slice Locality

Figure 2 reports the weighted average of slice-locality(n) for those branches that are mispredicted by the underlying outcome-based branch predictor. To calculate *slice-locality(n),* the distributions for each static branch are weighted by the relative number of outcome-based mispredictions associated with that branch, and so the overall figure naturally emphasizes those static branches which are mispredicted most often. We report locality in the range of 1

Base Processor Configuration			
Branch Predictor	64K GShare + 64K bimodal with 64K selector, 64 entry RAS	**Fetch Unit**	Up to 8 instr. per cycle. 64-entry Fetch Buffer Non-blocking Fetch
Instruction Window Size	128 entries	**FU Latencies**	same as MIPS R10000
Issue/Decode/Commit BW	8 instructions / cycle	**Main Memory**	Infinite, 100 cycles
L1 - Instruction cache	64K, 2-way SA, 32-byte blocks, 3 cycle hit latency	**L1 - Data cache**	64K, 4-way SA, 32-byte blocks, 3 cycle hit latency
Unified L2	256K, 4-way SA, 64-byte blocks, 16 cycles hit latency	**Load/Store Queue**	64 entries, 4 loads or stores per cycle Perfect disambiguation

Table 1: *Base configuration details. We model an aggressive 8-way, dynamically-scheduled superscalar processor having a 128-entry scheduler and a 64-entry load/store queue.*

Benchmark	*Label*	*Inst. Skipped*	*MR*	*BPA*	*Benchmark*	*Label*	*Inst. Skipped*	*MR*	*BPA*
164.gzip	*gzp*	101 M	3.1%	92.3%	**183.equake**	*eqk*	359 M	2.7%	90.9%
175.vpr	*vpr*	33 M	2.6%	91.7%	**188.ammp**	*amp*	100 M	28.3%	99.1%
176.gcc	*gcc*	200 M	0.9%	91.1%	**197.parser**	*prs*	144 M	2.3%	90.9%
177.mesa	*msa*	101 M	0.7%	99.9%	**255.vortex**	*vor*	102 M	0.7%	98.5%
179.art	*art*	1,686 M	43.9%	98.5%	**256.bzip2**	*bzp*	100 M	3.9%	97.4%
181.mcf	*mcf*	50 M	5.3%	90.2%	**300.twolf**	*twf*	188 M	6.2%	84.9%

Table 2: *Programs used in our experimental evaluation. MR is the L1 data miss rate. BPA is the branch prediction accuracy (direction+target). We simulated 300 million committed instructions after skipping the initialization phase.*

(bottom bar) through 4 (top bar) and for a variety of slicer configurations. To identify the slicers we use an *"NSM"* naming scheme. *"N"* is the size of the slicer, i.e., 256, 128, 64 or 32. *"S"* can be either *"U"* (unrestricted) or *"R"* (restricted) and specifies whether we restrict slice size to up 1/4 of total number of instructions in the slicer or not. Finally, "M" signifies that slices include memory dependencies. For example, *64UM* corresponds to a slicer with 64 entries that can produce slices of up to 64 instructions and that is capable of following memory dependences. *32RM* is a slicer that has 32 entries and that can detect slices that include only up to 8 instructions and that can follow memory dependences. We have experimented with various slicer configurations. In the interest of space we report the following seven from left to right: *256RM, 128R*M, *64RM*, *32RM*, *64UM*, and *32UM.*

Before we present our results it is important to emphasize that while high locality is desirable, *any* locality may be useful for improving branch prediction accuracy. This is because we measure locality only for mispredicted branches. As we will show in section 3.3, even when little locality exists, it can positively impact overall branch prediction accuracy.

With unrestricted slices, in all cases but *gzip* and *mesa*, using a shorter slicer results in higher locality with the average locality going from 89.6% to 93.2% comparing 64UM to 32UM. With restricted slices and a short detection window (32RM) there is much lower locality compared to unrestricted slices (32UM), and furthermore, the locality increases going from a 32-entry slicer to a 64-entry slicer, on average from 65% to 71%. This result suggests that many slices have more than 8 instructions that are close to the target instruction. This result corroborates the observation by Zilles and Sohi that many operations that directly contribute to the computation of the target are clustered close to the target operation [15]. As we use a fixed ratio of 1/4 to restrict slices, a shorter slicer is penalized more heavily than a longer one. Indeed, for a 256-entry slicer (256RM) we see the dominant trend is again a decrease in locality for longer slices.

On the average, slice-locality(4) is about 61% with the 256RM slicer and rises to about 68%, and 71% for the 128RM and 64RM slicer, while falling back to 65% for the 32RM slicer. More importantly, most of the locality is captured even if we can record a single slice per instruction. In particular, slice-locality(1) is approximately 41%, 46%, 50% and 50% for the 256RM, 128RM, 64RM and 32RM slicers. This suggests that a last-slice-based predictor may be quite successful.

As we move towards larger slicers, locality usually drops. In the worst case of *vpr*, slice-locality(1) drops to about 10% with the 256RM slicer. For several programs the drop of locality with increased slicing windows is a lot less dramatic and slice-locality(1) remains well above 40% for 256RM for half the benchmarks. However, a larger slicer does not necessary result in lower locality. In particular, for *gzip* and *mesa* locality(1) *increases* as the slicer is increased from 32 to 64 entries, *even for unrestricted slices.* This anomaly has been studied and appears to occur for two reasons: The dominate effect is due to slices for return operations that are mispredicted by the baseline *return address stack* mechanism which does not repair itself when speculative calls and returns are squashed. These slices typically include only one or two instructions: the return operation, and occasionally the jump-and-link operation that stores the return address in register r31. As we only allow slices with more than one operation, shorter slices are penalized. We expect that using a more sophisticated RAS implementation (e.g., [19]) would largely eliminate this effect. However, a more subtle effect results from intervening control flow: A larger slicer allows us to look through more instructions when detecting a slice, and hence capture longer slices. Normally, this tends to strongly reduce slice locality because the number of implied control flow paths leading up to the target multiplies as additional basic blocks appear in the slicer. However, a longer slicer may also increase locality when a slice skips over a segment of instructions that fluctuates in length due to intervening control flow. With a short slicer, the earlier part of the slice may be evicted occasionally. With a longer

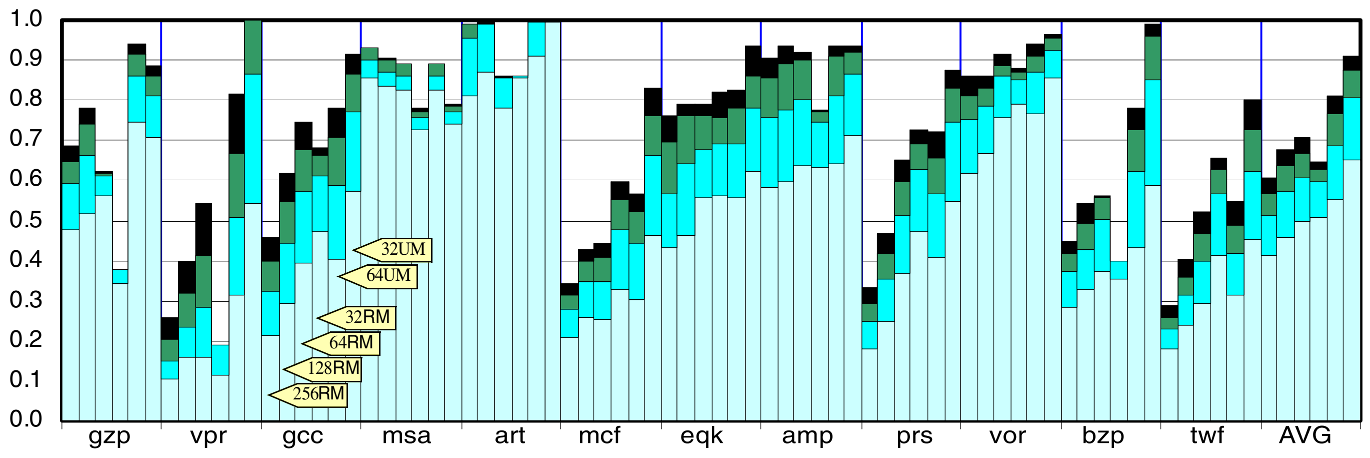


Figure 2: *Weighted average slice-locality distribution for mispredicted branches (see text for description of weighting procedure). Range shown is 1 (bottom) to 4 (top). We use a NSM scheme where N is the size of the detection window (256, 128, 64, or 32), S is "U" if no restrictions on slice size are placed and "R" if we restrict slices to 1/4 of the slice detection window, and finally, "M" indicates that we follow memory dependences in constructing slices.*

Program	***256-R-M***				***128-R-M***				***64-R-M***				***32-R-M***			
	Dist.	***Cnt.***	***LI***	***#St***	***Dist.***	***Cnt.***	***LI***	***#St***	***Dist.***	***Cnt.***	***LI***	***#St***	***Dist***	***Cnt.***	***LI***	***#St***
gzp	202.3	23.4	3.06	2.92	90.1	14.2	2.75	1.09	42.1	7.2	2.14	0.35	16.4	4.3	1.93	0.15
vpr	194.6	24.6	2.63	1.27	95.9	15.8	2.06	0.27	50.8	10.4	1.77	0.05	22.3	5.1	1.72	0.17
gcc	199.0	18.2	1.92	0.98	93.3	11.7	1.54	0.45	41.9	7.6	1.28	0.18	17.8	4.7	1.15	0.05
msa	164.3	8.0	1.29	0.01	77.2	5.7	1.33	0.00	43.0	4.6	1.36	0.00	16.2	3.6	1.31	0.00
art	237.4	13.3	2.03	0.00	109.6	9.1	2.03	0.00	46.0	5.4	1.87	0.00	15.3	4.1	2.00	0.00
mcf	208.4	28.0	1.90	0.29	100.0	16.8	1.76	0.12	43.3	8.5	1.66	0.04	17.2	5.2	1.59	0.00
eqk	121.3	6.8	0.89	0.02	64.0	6.0	1.01	0.00	35.3	4.5	1.14	0.00	14.7	3.2	1.17	0.00
amp	167.0	12.3	2.15	0.54	80.3	9.4	1.87	0.32	40.4	7.3	1.50	0.06	18.5	5.0	1.40	0.05
prs	213.6	23.5	1.76	1.04	101.2	12.2	1.52	0.48	47.7	7.4	1.28	0.18	19.8	4.6	1.13	0.07
vor	202.1	16.5	1.57	0.62	99.7	9.4	1.35	0.32	47.9	6.4	1.25	0.13	22.0	4.1	1.10	0.04
bzp	215.8	33.7	3.66	0.82	103.5	19.4	3.12	0.15	48.3	9.8	2.39	0.00	23.3	5.7	2.13	0.00
twf	131.7	16.4	2.00	0.21	74.3	13.2	1.93	0.10	44.5	9.0	1.88	0.06	20.3	4.8	1.73	0.03

Table 3: *Branch slice statistics: Weighted average instruction distance ("Dist."), instruction count ("Cnt."), register live-ins ("LI"), and number of stores ("#St") for various slice detection setups. Each slice weighted by the number of mispredictions potentially corrected.*

slicer, the whole slice may still appear in the slicer. In table 3 we report the average instruction distance between the lead and the target instructions and the average instruction count per slice. We define *instruction distance* as the number of intervening instructions (including the lead) in the original instruction trace. In the interest of space we restrict our attention to the 256RM, 128RM, 68RM and 32RM slicers. These two metrics provide an indication of whether the slices could potentially run-ahead of the main thread (of course, this can only be measured using an actual implementation of an operation predictor). Overall, slice instruction count is relatively small and remains small even when we move to longer slicers. Moreover, the lead to target instruction distances are on the average considerable, especially with the 256RM slicer.

Table 3 also reports the average number of stores and register live-ins per slice. The number of stores is a metric of the number of memory dependences in each slice, while the live-ins is a measure of the cost of spawning a speculative slice. The number of memory dependencies detected tends to grow with slicer size (similar to observations by Zilles and Sohi [15]), however, for the slicer sizes studied here, the number of dependencies detected was small. Furthermore, the average number of live-ins is always less than four.

These results are encouraging as they suggest that relatively high locality exists in the computation slices that lead to unpredictable branches. Moreover, slices tend to be small in size (on the average), spread over several tens of instructions of the original program. Having shown that programs exhibit the locality necessary for operation prediction of otherwise mispredicted branches, in Section 3.3.1 we measure how an approximate operation predictor interacts with the underlying outcome-based branch predictor.

3.2.2 Load Slice Locality

Figure 3 reports weighted average slice-locality(n) for those loads that miss in the L1 data cache. The weighting of the distribution for each static load is based upon the frequency of misses for that load. We report results for the same slicer configurations we presented in section 3.2.1. We observe trends similar to those for mispredicted branches but with locality being stronger. On the average slice-locality(1) is 62%, 62%, 55%, and 47% for the 32RM,

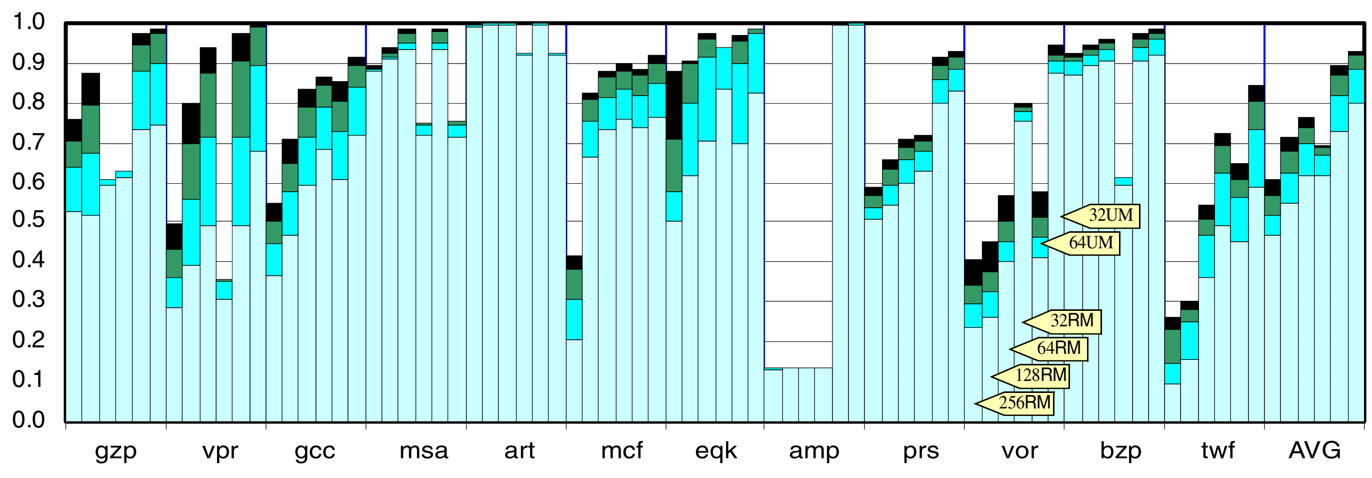


Figure 3: *Slice locality distribution for loads that miss in the L1 data cache. Range shown is 1 (bottom) to 4 (top). We report results for the same slicer configurations as in Figure 2.*

Program	***256-R-M***				***128-R-M***				***64-R-M***				***32-R-M***			
	Dist.	***Cnt.***	***LI***	***#S***	***Dist.***	***Cnt.***	***LI***	***#S***	***Dist***	***Cnt.***	***LI***	***#S***	***Dist.***	***Cnt.***	***LI***	***#S***
gzp	226.9	28.6	3.53	3.61	101.3	17.5	3.20	1.23	41.2	7.8	2.08	0.40	19.1	4.5	1.21	0.02
vpr	193.1	24.9	3.01	2.23	97.3	17.0	2.90	1.10	45.2	12.5	2.27	0.37	15.6	5.5	1.80	0.00
gcc	205.0	19.8	1.65	1.03	101.1	11.7	1.48	0.51	47.2	6.9	1.30	0.18	21.2	4.3	1.16	0.04
msa	224.3	8.0	1.03	0.00	109.9	4.9	1.05	0.00	44.8	2.8	1.04	0.00	18.1	2.0	1.01	0.00
art	236.9	10.6	1.17	0.00	109.5	6.2	1.17	0.00	46.1	3.9	1.18	0.00	17.4	3.0	1.19	0.00
mcf	230.1	29.6	1.44	0.01	109.1	14.1	1.78	0.01	47.2	7.9	1.78	0.00	14.8	4.9	1.69	0.00
eqk	183.1	9.7	1.97	0.11	80.8	7.2	1.72	0.01	33.4	5.3	1.62	0.00	17.3	4.1	1.50	0.00
amp	244.3	55.7	1.12	0.56	122.2	28.3	1.05	0.14	60.9	14.8	1.04	0.07	27.8	7.4	1.02	0.18
prs	240.6	28.3	1.90	0.65	116.4	15.7	1.83	0.33	55.6	9.3	1.76	0.17	22.7	5.8	1.62	0.03
vor	222.8	27.1	3.48	2.02	109.5	16.0	2.82	0.83	53.8	9.5	2.35	0.28	18.1	6.3	1.81	0.03
bzp	216.2	15.4	2.18	0.17	100.7	10.2	2.13	0.05	42.1	7.7	2.09	0.00	15.5	4.4	1.81	0.00
twf	123.7	12.3	1.84	0.05	61.8	10.1	1.81	0.01	40.6	7.5	1.65	0.08	22.4	5.0	1.47	0.04

Table 4: *Load slice statistics: Weighted average instruction distance ("Dist."), instruction count ("Cnt."), live-in registers ("LI"), and number of stores ("#St") for various slice detection setups. Each slice weighted by the number of L1 cache misses potentially prefetched.*

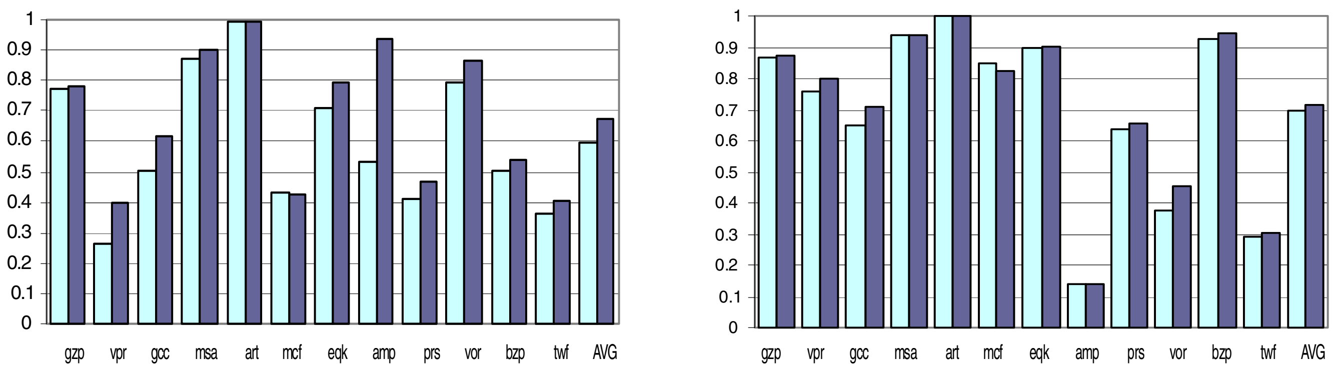


Figure 4: *The effect of detection context on slice locality for branches (left) or loads (right): Detecting slices specifically when a branch was mispredicted or load missed improves locality. Bars represent locality(4) for a 128-R-M detection mechanism. Darker bars are for detection only on mispredict, or on cache miss, for branch slices and load slices respectively. Lighter bars represent the locality for slices of mispredicted branches and loads that miss when slices are detected independent of whether there was a misprediction or cache miss.*

64RM, 128RM, and 256RM slicers. Recording up to 4 slices per instruction results in a locality of 70%, 76%, 71% and 61% respectively. For most programs, load slice locality is stronger than branch slice locality was. In table 4 we also report the average lead to target instruction distance, instruction count, register live-ins, and number of included stores for load slices. Slice instruction distance increases with the slicer size and is relatively large. Moreover, slice instruction count remains relatively small even with the larger slicers.

The results of this section are also encouraging as they show that high locality exists in the slice stream of the loads that miss in the L1 data cache. Overall, slicer size tends to play a dominant role in determining locality. While restricting slice size has an impact on locality for shorter slicers, its impact decreases with longer slice detection windows.

3.2.3 Effect of Detection Context on Locality

Figure 4 reports the change in observed locality using a 128RM detection mechanism when we allow slices to be added to the slice cache independent of whether the underlying outcome-based prediction mechanism was correct (in the case of branches), or there was a cache hit (in the case of loads). Note that both sets of measurements still represent only slices for branches that mispredict, or loads that miss, and are again weighted by the frequency of mispredictions and cache misses per static branch or load. We found that detecting slices only for those dynamic instances of the target instruction for which a misprediction, or cache miss event occurs (as done in Sections 3.2.1 and 3.2.2) improved locality on average by around 8% and 2% for branches and loads respectively. It only decreased locality for *mcf,* and then only by 1% and 2% for branches and loads respectively.

3.3 Accuracy and Interference with Outcome-Based Prediction

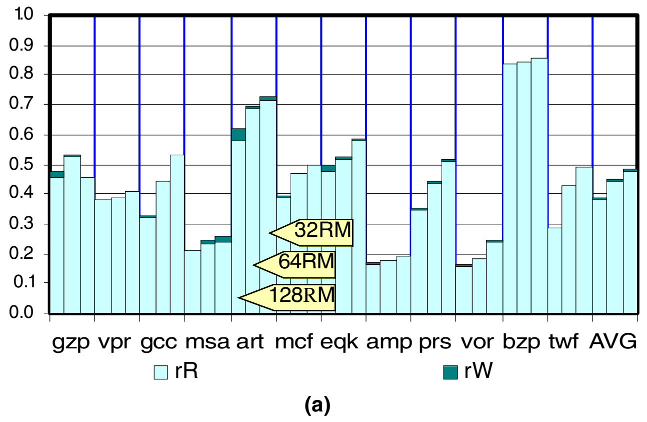
In this section, we model specific operation predictors and study their accuracy. We first explain how our branch operation predictor works. A slice is detected after each branch that was mispredicted. Detected slices are stored in an infinite slice cache where they are identified by the *lead* instruction. Only up to 4 slices per lead instruction can be present in the slice cache, however other than this there is no restriction on the total number of slices in the cache. Upon encountering a dynamic instance of the lead instruction we spawn *all* slices that are associated with it. Note that these slices may relate to the same, or different target operations. For the purposes of this study, we assume that the resulting scout threads complete *instantaneously*, however the outcomes of these threads are not used immediately. Also, all register and memory values from instructions before the lead are assumed to be available. The outcome from slice execution is saved while the slice is matched-up to the arriving flow of instructions. This matching is based upon matching instructions and register dependences. A more practical method would be to record the implied control flow of the slice when it was detected and compare this to the observed control flow after a slice has spawned, however, the latter technique does not readily allow control independence. On average we found that 47%, 58%, and 72% of all branch slice executions are discarded for the 32RM, 64RM, and 128RM slice detection mechanisms. When and if the target branch appears, if more than two slices have matched up to the instruction stream we select the first slice that spawned, or the most recently extracted slice if both spawned at the same time. Most of the time there is only a single prediction available to be consumed, if any (90%, 85%, and 80% for 32RM, 64RM, and 128RM respectively when executing branch slices).

3.3.1 Branches

To quantify the *potential* accuracy of our operation predictor for branches and how it interacts with the underlying outcome-based predictor we provide a breakdown of operation prediction for all dynamic branches. We break down branches based on whether the underlying outcome-based predictor correctly predicts the particular dynamic branch instance, on whether a prediction was available from the operation prediction and on whether the latter, if available, was correct. For ease of explanation we use a "vP" naming scheme. "v" can be *w*(rong) or *r(*ight) and signifies whether the outcome-based predictor correctly predicted the branch. "P" can be *N*(one), *W*(rong) or *R*(right) and signifies whether no prediction was available from the operation predictor, and if there was one, whether it was correct or not. For example, *rN* and *rR* correspond to branches that were correctly predicted by the outcome-based predictor and for which no prediction or a correct one was available from the operation predictor respectively. Category *rW* corresponds to *destructive interference* between operation and outcome-based prediction, while category *wR* corresponds to *constructive interference. "rN", "wN", "rR"* and *"wW"* do not impact the accuracy of the outcome-based predictor. In our results we report "rW" and "rR" as fractions measured over the total number of correctly predicted branches by the outcome predictor. We also report "wW" and "wR" as fractions measured over the total number of incorrectly predicted branches by the outcome predictor. Ideally, "rW", "wN" and "wW" would all be 0%, in which case "wR" would be 100% (all previously mispredicted branches are now correctly predicted by the operation-based predictor)

Figure 5 reports accuracy results for operation predictors that utilize, from left to right, a 128RM, 64RM or a 32RM slicer. Part (a) reports accuracy for correctly predicted branches (categories "rR" and "rW") while part (b) reports accuracy for mispredicted branches (categories "wR" and "wW"). Categories "rN" and "wN" are implied (missing upper part of the bars). In comparing the results of two graphs we must also take into account the relative fraction of correctly and incorrectly predicted branches (i.e., the accuracy of the underlying outcome-based predictor). We do so later in this section. In most cases, the operation predictors interact favorably with the underlying outcome predictor since "rW" is in most cases very small. In all programs, the operation predictor correctly predicts a large fraction of those branches that are mispredicted by the underlying outcome-based predictor as shown in part. (b) (category "wR") while it incorrectly predicts very few (category "wW").

On the average, ignoring timing considerations, the operation predictor offers correct predictions for about 72%, 74% and 65% of all mispredicted branches when the 128RM, 64RM or the 32RM slicers are used respectively. On the average, the operation prediction interferes destructively with the underlying outcome-based branch predictor in very few cases. We re-iterate that in interpreting he results of figure 5, one should also consider the relative fractions of correctly versus incorrectly predicted branches. We report the absolute change in prediction accuracy in addition to the outcome-based branch predictor in table 5 (the branch prediction accuracy of



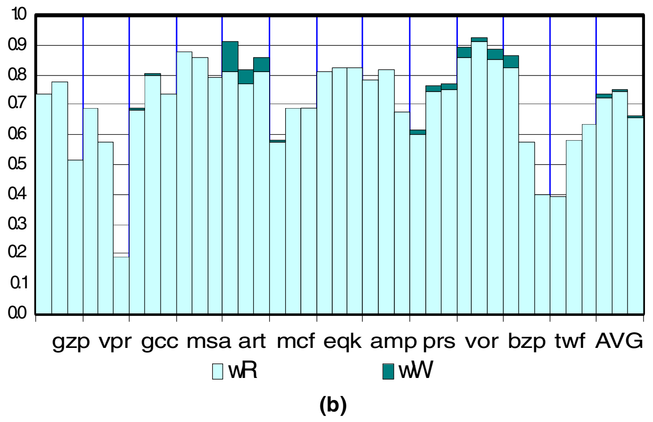


Figure 5: *Interaction of operation branch prediction and outcome-based branch prediction. We report results for the following slice detection mechanisms: 128-R-M, 64-R-M and 32-R-M. When outcome-based prediction: (a) Correctly predicted branches and (b) Mispredicted branches.*

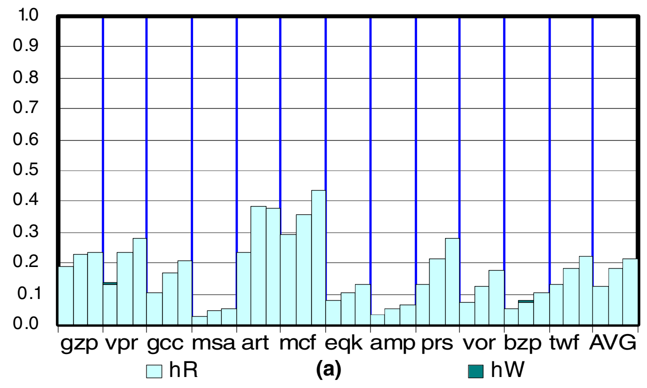
Program	*128-R-M*	*64-R-M*	*32-R-M*	*Program*	*128-R-M*	*64-R-M*	*32-R-M*
gzp	+3.3%	+4.9%	+3.9%	***eqk***	+5.5%	+7.0%	+7.0%
vpr	+5.9%	+5.3%	+1.7%	***amp***	+0.6%	+0.7%	+0.6%
gcc	+5.4%	+6.9%	+6.5%	***prs***	+4.8%	+6.2%	+6.3%
msa	+0.1%	-1.8%	-1.8%	***vor***	+1.1%	+1.3%	+1.2%
art	-1.7%	+0.2%	+0.2%	***bzp***	+1.9%	+1.5%	+1.1%
mcf	+4.7%	+6.7%	+6.7%	***twf***	+5.4%	+8.8%	+9.6%

Table 5: *Change in branch prediction accuracy with operation prediction over the base outcome-based predictor.*

the outcome based predictor was reported in table 2). We observe that in most programs the operation predictor helps the underlying outcome-based predictor resulting in higher overall accuracy. In some cases (e.g., mesa) where outcome-based prediction is very high, the operation predictor actually harms overall accuracy. Since in most cases, this destructive interference occurs for programs with high branch accuracy, it may be possible to use a confidence mechanism (e.g., a counter with every slice) to filter out those slices that lead to incorrect predictions very often or to simply disable operation prediction when outcome prediction is above a threshold. Such an investigation is beyond the scope of this paper. Overall the fraction of mispredicted branches that get a correct prediction from the operation predictor is greater than the locality we observed in section 3.2.1. Nevertheless, the trend in these two sets of values is similar. The main reason they differ in magnitude is that we restrict the number of slices to 4 per lead PC as opposed to 4 per target PC (this restriction was placed since we need to associate slices with the lead PC in this operation predictor). If the slices for a given target do not all share the same lead operation, the number of slices captured can exceed the four used when measuring locality. Indeed, the most striking difference is exhibited for *bzip* where coverage increases significantly for longer window size whereas locality(4) does not (compare Figures 2 and 5b). Although not shown, in this case we found the quantity *wR*+*wW* closely matched locality(*n*) in the limit as *n* grows large (i.e., for values of *n* much larger than 4).

3.3.2 Loads

Finally, we report accuracy for an operation predictor for load addresses. The results are shown in figure 6 for predictors based on the 128RM, 64RM and 32RM slicers. In part (a) we report results for those loads that *hit* in the data cache, while in part (b) we report results for the loads that *miss* in the data cache. The results of part (a) are provided for completeness. These loads hit in the data cache, so correctly predicting their addresses is not as important. We show two categories: *hR* are the loads whose addresses are correctly predicted while *hW* are the loads whose addresses are incorrectly predicted. In an actual implementation *hW* may translate into cache pollution. Overall, *hW* is negligible. In part (b) we report a breakdown of predictions for loads that miss in the data cache. Two categories are shown; *mR* includes the loads for whom the addresses are correctly predicted while *mW* includes those that are not. Ideally, mW would be 0% and mR would be 100%. In all cases, mW is barely noticeable. Moreover, mR tends to be higher for shorter slicers. We can observe that the accuracy of the operation predictor is extremely high (mR vs. mW). Moreover, the operation predictor offers correct predictions for many of the loads that miss in the data cache. Overall the idealized operation predictor could correctly prefetch 63%, 67% and 58% of the loads which otherwise miss for the 32RM, 64RM and 128RM mechanisms, respectively. This coverage is less than the locality we observed in section 3.2.2. Again, this relates to the restriction of 4 slices per lead PC as opposed to 4 per target PC. In many programs, the same lead PC appears in the slices for more than one target load. Accordingly, thrashing occurs and coverage suffers. For example, consider a linked list where each element is a structure with multiple fields. All loads that access each field may be missing at the same time. All these loads will be getting the base address of the element in question from the same load..Consequently, their slices could have the same lead instruction



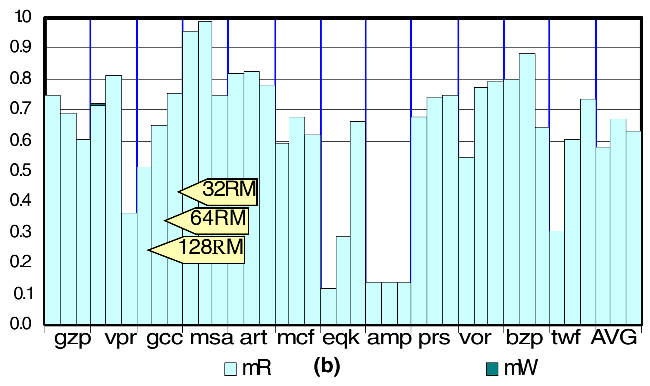


Figure 6: *Breakdown of load address operation prediction. We report results for the following slicers: 128RM, 64RM and 32RM. (a) Loads that hit in the data cache. (b) Loads that miss in the data cache.*

and hence they will cause thrashing in the lead PC's slice set. A potential solution to this problem could be to allow more slices per lead PC. Alternatively, we may opt for carefully selecting the loads for which we detect slices and apply operation prediction (e.g., first loads that misses per block as opposed to *all* loads that miss per block).

The tradeoffs in load address prediction are quite different than those for branch prediction. In load address prediction, an incorrect prediction does not necessarily impact performance negatively. It can only do so indirectly by increasing resource contention or by polluting the data cache. Also, while we may predict the exact address incorrectly, we may still predict the correct cache block address correctly. Moreover, while it is desirable to have a high *coverage (*that is to provide correct predictions for as many of the loads that miss as possible), higher coverage may not translate into higher performance for reasons that include the following: Two loads that miss may be accessing the same block, accordingly, we may actually prefetch the block even if we do not correctly predict both of them. Also, in some cases, performance may be limited by other loads, hence correctly predicting a load address may have a negligible impact on performance.

4 Related Work

Operation prediction shares similarities with a number of recently proposed multi-threaded models where a number of potentially speculative, *helper threads* are used to enhance an otherwise sequential, main thread. *Simultaneous subordinate micro-threading* and *assisted execution* are two such proposals [2,13]. In the example application of SSMT given in [2] the helper threads are implemented in microcode and are used to enhance branch prediction.

Zilles and Sohi suggested extracting slices at compile time and using them to pre-execute performance critical instructions [15,16]. Assuming compile-time extraction, they have demonstrated that such slices can greatly improve performance, especially if they are optimized. Farcy *et al.,* proposed an operation predictor for branches for a restricted class of branches [5]. Moshovos also suggested the possibility of generalized operation prediction [8]. Moshovos *et al.,* proposed slice processors, the first dynamic, autonomous and generalized operation predictor-based prefetcher and demonstrated that it can improve performance even when compared to an outcome-based predictor [9]. Collins *et al.,* demonstrated a software-driven slice-based prefetcher for an EPIC-like architecture [4]. In parallel with this work, Collins *et al.,* also proposed a dynamic slice pre-execution prefetcher where slices are optimized and can be chained [17]. Annavaram *et al.,* proposed a non-speculative slice-based prefetching scheme where slices are detected and pre-executed from the fetch buffer and demonstrated that it can effectively prefetch data for a 4-way OOO core with a 64-entry scheduler [1]. Luk described a software-controlled prefetching method based on slice pre-execution [6]. In the *Speculative Data-Driven Multithreading (SDDM)* execution model, proposed by Roth and Sohi, performance critical slices leading to branches or frequently missing loads are pre-executed [12]. A *register integration* mechanism is used to merge slice produced results into the main thread and to filter out any incorrectly calculated values. As proposed, SDDM relies on a profiling phase and the compiler to build slices and to orchestrate their execution.

Some operation outcome predictors exist. Stride predictors are an early example where the actual computation is built in the design. Roth *at al.,* proposed an operation predictor for recursive data structures [10], while Mehrotra *et al.,* proposed operation predictors for linked lists and arrays [7]. Roth *at al.,* proposed an operation predictor for indirect jumps [11]. In all aforementioned proposals, the class of predictable operations is fixed in the design. *Slipstream Processors* also use a helper thread to run-ahead of the main sequential thread in effect pre-executing instructions [14]. The helper thread is formed by removing predictable computations from the main sequential thread. They study the dynamic creation of *chaining slices* in which a slice can, in essence, re-spawn itself. A similar chaining mechanism was proposed by Zilles and Sohi in [16] based on hand-optimized slices. Finally, an *Instruction Path Coprocessor* could potentially be used to support dynamic extraction and execution of slices [3].

This study also appears in our recent technical report [18]. To the best of our knowledge, no other work on the locality characteristics of the slice stream of mispredicted branches or loads that miss exists. Moreover, in their majority, most aforementioned slice-based execution models rely on compile-time slice creation or manual selection.

5 Conclusion

In this study we were motivated by the recently proposed operation-based prediction. Existing outcome-based predictors rely on regularities in the outcome stream so that they can accurately predict a large fraction of the program's outcomes. However, some outcomes do not exhibit sufficient regularity. Operation prediction has the potential of successfully predicting some of these outcomes. Operation prediction looks for regularity in the computation stream that produces outcomes that do not exhibit sufficient regularity. It works by dynamically extracting the computation slices that lead to such outcomes and by attempting to pre-execute them next time around. For operation prediction to be successful, it is necessary that the computation stream of such outcomes exhibits regularity.

Several works have demonstrated that operation prediction methods work or may work for branches or loads, In this work we study program behavior and explain *why operation prediction may work*. In particular, we studied the locality of the computations that lead to otherwise unpredictable outcomes. We focused on loads and branches and studied locality under various realistic assumptions about slice detection. Moreover, we have studied models of operation predictors and how they interacted with an underlying outcome-based predictor. Our results demonstrate that high locality exists in the computation stream of unpredictable branches and of loads that miss in the data cache. Moreover, we have shown that the potential exists for operation prediction to boost accuracy over an existing outcome-based branch predictor and of accurately predicting the addresses of load references that would miss in the data cache. To the best of our knowledge no previous work on the locality of slices for mispredicted branches and loads exist.

While our results are promising we have not studied actual operation predictors taking timing into account. Nevertheless, we have seen that slices tend to spread over large region of the original instruction stream while they contain on the average few instructions. Moreover, our results remain valid and important as they demonstrate that programs do exhibit the behavior necessary for operation prediction to be successful. Further investigation is required in tuning operation predictors so that they make use of available resources effectively while being able to execute scout threads early enough for providing predictions for modern high-performance processors.

6 Acknowledgments

This research was supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) research grant and by research funds from the University of Toronto. Tor Aamodt was supported by an NSERC PGS ‘B’ scholarship. We gratefully acknowledge the comments and suggestions of the anonymous referees that helped improve this paper.

REFERENCES

[1] M. M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Pre-computation. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.

[2] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. 26th Intl. Symposium on Computer Architecture*, pages 186-195, May 1999.

[3] Y. Chou and J. Shen. Instruction path coprocessors. In *Proc. 27th Intl. Symposium on Computer Architecture*, pages 270-281, June 2000.

[4] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. Fong Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.

[5] A. Farcy, O. Temam, and R. Espasa. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st Annual International Symposium on Microarchitecture*, Dec. 1998.

[6] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.

[7] S. Mehrotra and L. Harrison. Examination of a memory access clasification scheme for pointer-intensive and numeric programs. In *Proc. 10th Intl. Conference on Supercomputing*, Sept. 1997.

[8] A. Moshovos. *Memory Dependence Prediction.* Ph.D. thesis, University of Wisconsin-Madison, Madison, WI, Dec. 1998.

[9] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice Processors: An Implementation of Operation-Based Prediction. In *Proc. International Conference on Supercomputing*, June 2001.

[10] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115-126, Oct. 1998.

[11] A. Roth, A. Moshovos, and G. S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *Proc. Intl. Conference on Supercomputing*, pages 356-364, June 1999.

[12] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proc. 7th International Symposium on High Performance Computer Architecture*, Jan 2001.

[13] Y. Song and M. Dubois. Assisted execution. Technical report, Technical Report CENG-98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[14] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[15] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, pages 172-181, June 2000.

[16] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proc. 28th International Symposium on Computer Architecture*, June-July 2001.

[17] J. D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, Dynamic Speculative Precomputation. To Appear *In Proc. 34th International Symposium on Microarchitecture*, Dec. 2001.

[18] Tor Aamodt, Andreas Moshovos, and Paul Chow, The Predictability of Computations that Produce Unpredictable Outcomes, Technical Report #TR-01-08-01, EECG, University of Toronto, August 2001.

[19] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark, Improving Prediction for Procedure Returns with Return Address-Stack Repair Mechanisms. In *Proc. 31st Intl. Symposium on Microarchitecture*, pages 259-271, Nov. 1998.