COMP591 Research Paper

Analysing Branch Prediction Algorithms using OpenPAT

The University of Waikato

Neil Bradley

Proforma

Author:	**Neil Bradley**
Project Title:	**Analysing Branch Prediction Algorithms using OpenPAT**
Supervisor:	**Dr Simon Spacey**
Word Count:	**8100**
Version:	17 February 2015
Contact:	ncb9 AT students.waikato.ac.nz

Contents

List of Figures

List of Tables

Acknowledgements

I am grateful to my supervisor, Dr. Simon Spacey, whose help and guidance was invaluable in the completion of this project. To Brad Cowie and the Waikato University Computer Science department, for the provision of dedicated workspace. To Assoc. Prof. David Bainbridge, who provided tips and hints along the way. And finally, to my patient (ie, long-suffering) wife Sophie - life is just simply better with you.

Chapter 1

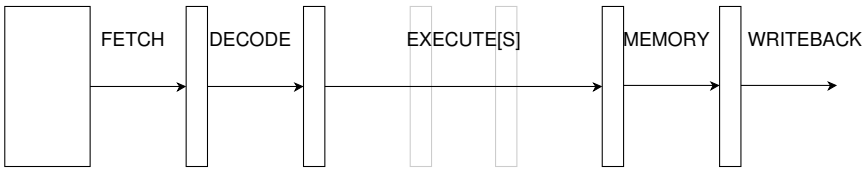
Introduction

Humans are obsessed with a need for speed. This can be seen in the popularity of all sorts of races, but nowhere is it more evident than in the field of computer science. Since the first digital computer ENIAC in 1946 [2], scientists, mathematicians and engineers have produced ever-increasing speeds of calculation, always pushing the bounds of what is physically possible for the production techniques of the day. From an initial speed of 5kHz, over the last seven decades processor speed has developed to over 4GHz in production processors, with CPUs over 10GHz in development [1, 2].

A number of technological advances made this progress possible: First, computers went from valve powered to transistor to integrated circuits. As integrated circuits matured, die sizes became ever smaller, allowing more processing power and faster transmission times between components. At the same time, processors were partitioned into task-units, or *pipelined*.

Processing an instruction is a multi-step process - instructions need to be fetched, decoded, executed, and then the results written back to either memory or the next instruction. In a pipelined processor, each of these tasks, and/or their sub-tasks, is assigned to a single unit, running for one clock cycle and passing its state on to the next unit for the next processor cycle. Overall, this results in each instruction taking longer to process (as it now takes multiple cycles), but allows multiple instructions to be processed concurrently, each using a different task-unit. Because each task-unit is now physically much smaller than the whole processor was before, this partitioning also allows processors to operate at much higher clock speeds than before. But it also brings with it a significant problem, in the form of data dependencies.

A data dependency is where one instruction relies on the result of a previous instruction or instructions in order to make its calculation. But if the instruction depended on is still in the pipeline, unresolved, the dependent instruction is



		CPU Cycle and Pipeline Stages Present										
$PC	$IR	1	2	3	4	5	6	7	8	9	10	11
0000	-	F	D	E	M	W				F	D	E
0001	-		F	D	E	M	W				F	D
0010	B0101			F	D	E						F
0011	-				F	D	x					
0100	-					F	x					
0101	B0000						F	D	E			
0110	-							F	D	x		
0111	-								F	x		
1000	-											

Figure 1.1: Pipelined CPU Stages and Execution

unable to proceed. For most instructions, say, calculations, this is readily solved by both forwarding unresolved instructions and looking ahead deeper into the pipeline to get results early. However, there is one class of instruction for which these options are not enough - the conditional jump, or branch, instruction.

The problem posed by branch instructions is that they alter the processors fetch cycle. Where the processor's program counter or instruction register is merely incremented for a normal instruction, a branch instruction alters the very value of the register, depending on either the condition being tested or the value of a register. But in a pipelined processor, this value may not yet be available, meaning that the processor is unable to process the branch instruction in the current fetch cycle. This leaves the processor with two different options: first, it could *stall* the processor, that is wait doing nothing until the condition is resolved (this could also be achieved by the insertion of `nop` instructions). Or, the processor could attempt a guess at the branch outcome - a technique known as *branch prediction*.

Each of these options is not without it's detriment, however. Inserting `nop`'s or otherwise stalling the pipeline reduces the processors efficiency, by reducing the amount of instructions it can process in parallel. And as branches can make up to an estimated thirty percent of processed instructions, even a shallow pipeline will suffer a significant performance penalty [22]. The other alternative,

branch prediction, suffers no penalty when the prediction is correct. However, if incorrect, it now suffers the same performance degradation of stalling, with the added overhead of having to flush the pipeline - *squashing* the invalid instructions and ensuring they do not corrupt state [22]. Figure 1.2 illustrates such a processor, and a fictional program experiencing these stalls. Careful examination shows this fictional program squashes two out of every five instructions, and therefore only 60% of the work the CPU is doing is actually useful.

A stalling pipeline is much easier to implement than one which utilises a branch predictor. But as we have already seen, a stalling pipeline guarantees reduced performance, whereas a branch predictor promises the potential for increased performance, depending on the accuracy of the predictor utilised. In our quest for ever-increasing speed, therefore, it is desirable to have pipelined processors with highly accurate branch predictors.

Over the last four decades, many predictor designs have been proposed to tackle this need-for-speed. This research project will investigate some of their performances, and methods by which their performance may be improved. Section 2 investigates the history of branch predictor designs, including componentry, algorithms, and performance testing. Section 3 explains the method used for the testing of a selection of branch predictors, some modifications investigated, benchmark and performance analysis configuration, and the rationale behind choices made. Section 4 presents the results of the investigation, with Section 5 discussing relevant findings and limitations of the study, before conclusions and recommendations are presented in Section 6.

Chapter 2

Background

2.1 Prediction Mechanisms

Branch predictors can only make predictions based on the information available to them at the time of execution. There are six main sources of information that can be used: rules, branch target address, program counter, histories, profile information, and the `OP_CODE` itself. What makes branch predictors more successful or not is the algorithms each uses to combine these sources of information into a prediction. In pipelined processors, the mis-prediction penalty is usually directly related to pipeline depth. Hwu *et al* note that, if the penalty for an incorrect prediction is high enough, even a small increase in the accuracy of a branch predictor has a large effect on the performance of the processor [18].

2.1.1 Static Prediction

Early methods of branch prediction made simple guesses about generic branch behaviours based on perceived programmer habits. Depending on how a programmer for a processor was recommended to order branches, the options for the predictor were `ALWAYS_TAKEN`, `NEVER_TAKEN`, `BACKWARDS_TAKEN`, or `FORWARDS_TAKEN`. The latter two options used the branch target address to determine the action to be taken, whilst the former used no information at all. `ALWAYS_TAKEN` is beneficial in applications that involve a large amount of loops, for example, numerical processing, as loops are generally taken far more than not [22]. `NEVER_TAKEN` is beneficial in applications involving large amounts of `switch`-type instructions, as most of the branches will not be taken. Good examples of this kind of behaviour are regular expression evaluators and text sorts. `BACKWARDS_TAKEN` and `FORWARDS_TAKEN` married these two behaviours together, as both behaviours are usually present in the same application. Predictions made

in this way have maximum benefit if the program is written/compiled for them. Programmer habits such as writing most likely outcome in the corresponding branch entry stem from these behaviours. Emer and Clark report a 67% accuracy for an `ALWAYS_TAKEN` predictor on the VAX 11/780 [14].

2.1.2 Profiled Information

But what if the compiler or programmer doesn't program in the way the predictor expects? Or there are branches that, due to certain conditions, go the opposite way? In such cases, the predictor is now more likely to be wrong than right, and the processor suffers performance degradation.

One method implemented to counter this involves the use of extra instruction bits to indicate the branches behaviour [24,25]. In 1986, McFarling and Hennessey suggested that predictor performance could be further increased if the compiler knew which branches are usually not taken. They suggest using a 'squashing direction' bit to convey this information to the branch predictor [22]. Hwu *et al* referred in 1989 to a 'prediction bit' in the branch instruction format, which is set by profiling. They call this method the *Forward Semantic*, and suggest a two-stage compilation with the first stage using nodes inserted at block entries to gather data on branch behaviour [18]. Wall found static prediction based on a profile to perform almost as well as a simple dynamic predictor (bimodal) [32].

These *hint* bits are generated using a two-stage compilation [10]. The first stage compiles the program and runs it with representative data. From this run, the program is then tagged, with the bias bits either stored in the branch instruction, or the programs data section, depending on the architecture. The advantage of this scheme is that if necessary, the program can be re-profiled and the executable re-tagged without a recompilation. The architecture can even be designed in such a way as to re-profile each execution. This is particularly useful for programs where branch behaviour is heavily data dependent.

Predictors that use either profiled information or the `OP_CODE` are typically classified as static predictors [10, 33]. Smith first proposed such a predictor in 1981 to save the calculation time of testing branch direction. Several variants have been proposed since, with several authors proposing "rules" that should be followed - the most notable example of which is the paper "Branch Prediction for Free" by Ball and Larus [4].

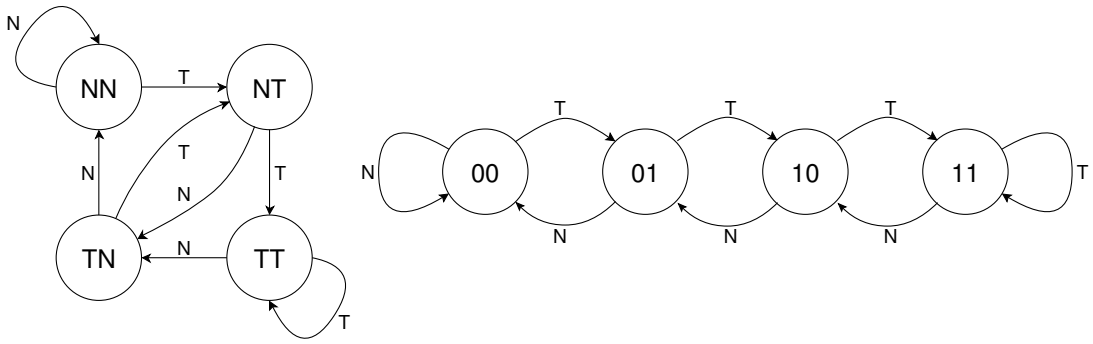


Figure 2.1: Alternative Saturating Counter State Machines

2.1.3 Saturating Counters

Of course, it is not always possible to know when a branch will be taken. Many branches' behaviour is data-dependent, for which a static prediction may not be suitable. Therefore a second class of predictor, *dynamic* predictors which make predictions based on run-time data were proposed. At the heart of most such predictors lies a simple predictor - the saturating counter [29]. Patented by J. Smith in 1979, saturating counters are a means by which a processor may be 'self-profiling' [29]. The most common version uses two bits to store information about branch direction. If the last branch was taken, the counter is incremented, without overflow. If the last branch was not taken, the counter is decremented, without underflow. A prediction can be made from the higher order bit of the counter - if it is a `1`, the branch is predicted `TAKEN`, otherwise it is predicted `NOT_TAKEN`. Two variations of this are shown in Figure 2.2. The middle two modes, 1 and 2, are known as `WEAKLY_NOT_TAKEN` and `WEAKLY_TAKEN` respectively. Nair concluded that out of 5248 possible variations of the 2-bit saturating counter, the simple up-down variant performs the best overall [25]. 1-bit and 3-bit counters are also sometimes used/recommended, but the 2-bit is the most prevalent [29].

2.1.4 Pattern History Tables

By itself, a single saturating counter doesn't make much of a predictor. Therefore, it is common practice to have a set of saturating counters, known as a *Pattern History Table* (PHT), which is indexed via an algorithm. PHTs typically make up the bulk of the silicon of a branch predictor, with larger tables increasing predictor accuracy. Predictors often have multiple PHTs, and the subtlety of each predictor is in the algorithm used to select which saturating counter to use to make the prediction with. Early predictors, such as the *bimodal* predictor, used the branch instruction address to index into the table [22, 29]. Each byte

of a PHT typically contains four 2-bit saturating counters. Because it is not feasible for a PHT to be large enough to store every address, early proposed implementations of PHTs recommended either LRU or FIFO algorithms be used as index values [29]. Many authors have proposed hashing algorithms since, such as Yeh and Patt's Per-address History, utilising a long history and a LRU algorithm on the PHT [33].

2.1.5 History Registers

Whilst it is easy to develop a predictor that uses either the branch instruction or branch target addresses, such predictors are susceptible to variation caused by data patterns. The primary weakness of a saturating counter, and thus anything based on it, such as PHTs, is that it will oscillate when the data oscillates.

For example, say there is a program matching the letter '`a`' from the string "`abababababababababababab`". At first glance, it looks like the predictor will be right 50% of the time, but this is not necessarily the case. The accuracy of the saturating counter to predict `TAKEN` or `NOT_TAKEN` actually depends on the initial state of the counter. It just so happens that, if the counter is at `WEAKLY_NOT_TAKEN` when the first '`a`' is encountered, it will always be wrong. This is because the counter will adjust in the opposite direction whenever it is wrong, but by starting at the `WEAKLY` opposite, it always changes away from what would be the next correct prediction. Thus, in the scenario given, a saturating counter will be correct 50% of the time *at most*, and may possibly even be 100% wrong.

One way of getting around this tendency of a saturating counter to oscillate is to keep track of the branch behaviour. There are multiple approaches to this, but the most common is to use a shift register that records a *history* of previous branch behaviour. This history can be either *global*, with a single register for all branches, or *local*, with nominally per-branch shift registers. Instead of using the branch instruction address, the PHT can now be indexed using a history register. This approach helps eliminate PHT conflicts caused by data dependency.

Consider the example again, this time using a history register to index the saturating counter. As the program progresses, the local history looks something like "`10101010101010101`". If even only two bits of history register are used, whenever the program is about to process an '`a`', the register contains the value '`10`', but when it's about to process a '`b`', the register contains '`01`'. It can quickly be calculated that the saturating counter indexed by '`10`' will quickly become `TAKEN`, and the counter indexed by '`01`' becomes `NOT_TAKEN`. Barring the

few initial mis-predictions based on relative counter state, the predictors are now always correct, and the previous 50% maximum accuracy no longer holds.

History registers also have the advantage that they are simple to implement, use very little hardware, and the next prediction can be known well in advance [5, 26]. Nair suggests using the `BRANCH_TARGET` to generate a history instead of `TAKEN` / `NOT_TAKEN` [24]. Programs with fewer active branch sites, which are more difficult to predict due to higher data dependencies, benefit more from longer histories than from increased PHT capacity [6].

2.1.6 Branch Target Buffers

Waiting for a predictor to make a prediction takes time. In order to alleviate this, and thus speed up the processor, many processors use a cache known as a *Branch Target Buffer* (BTB) to speed up selection of the target [5]. A branch target buffer stores the most likely target, hashed against the branch address, for quick evaluation upon fetch. Using this BTB, it is possible to reduce branch fetches to a single cycle [29]. Indirect branches cannot be so easily predicted as conditional jumps, so they are predicted using BTBs instead [32]. In order to facilitate faster predictions, Yeh and Patt suggest also storing the next prediction for a given branch address in the BTB when the branch predictor is updated with the correct prediction status [33, 34]. Calder and Grunwald argue against BTBs, arguing that profiled predictors consistently outperform any form of dynamic predictor [5].

2.1.7 Two-level Predictors

In the previous example, a local history was used. In order for such a local history to exist, however, the predictor must have the capacity to hold multiple branch histories, necessitating the use of multiple tables. In 1992, Yeh and Patt proposed using two levels of tables in order to generate a prediction: the first was a *Branch History Table* (BHT) which held local histories, and was indexed by the program counter; the second was a Pattern History Table, and was indexed by the Branch History Table (see Figure 2.3). Yeh and Patt showed this particular design (later called a *local* predictor [21]) to saturate around 97% accuracy, though they used averages of the SPEC benchmarks to claim this [33]. Most predictors since then have been multiple levels.

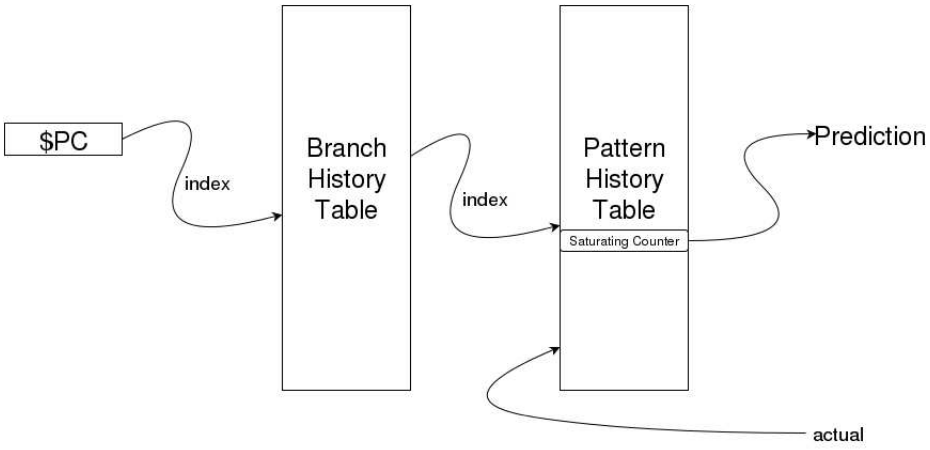


Figure 2.2: A Simple Two-Level '*local*' Predictor

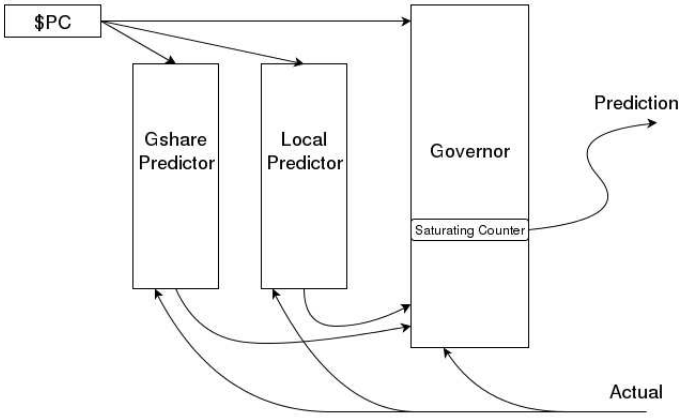


Figure 2.3: McFarling's local/gshare combined predictor

2.1.8 Combined Predictors

In 1993, Scott McFarling of DEC proposed another way to improve predictor accuracy - *combining* predictors. This involves taking two (or more) smaller predictors and combining them with a selector mechanism in order to choose which predictor to use for the current branch. McFarling's design centered around using a table of saturating counters for each branch, thus keeping track of which predictor was most accurate for the branch selected (see Figure 2.4). He proposed and tested two designs: the *local/gshare* and *local/bimode*, and claimed a saturated accuracy of 98.1% [21].

Other methods have also been proposed for combining predictors, such as the *Cascaded* predictor and static methods. In his 2000 PhD paper 'Improving Branch Prediction by Understanding Branch Behaviour', Marius Evers argued

against static methods for predictor combination, stating that "the best predictor for a branch changes between data sets. This indicate[s] that is is probably best to use a dynamic selection mechanism to select which component predictor to use for each branch" [15]. Whilst some authors have argued against combined predictors, most predictors since McFarling's work have been combined predictors [13].

Evers *et al* proposed a combined predictor that used a selection of accuracy counters to choose which predictor to use for a branch. In their predictor, all sub-predictors generate predictions, and the accuracy counters utilise a priority encoder to select which predictor makes final selection. They claimed 97.13% accuracy for their benchmark suite at 64KB [16].

Grunwald *et al* propose using a profiled hint bit to select the best predictor to use, instead of a dynamic selector [17]. Young *et al* suggest using profiling to assign dynamic predictors to difficult-to-predict branches ('per-stream'), and leaving easy-to-predict branches to a static predictor [36].

2.1.9 Predictor Sizes

One of the most commonly used ways to improve predictor performance has been to increase the size of the tables underlying the algorithm. Larger PHTs mean more saturating counters available to be assigned to pattern predictors, and larger BHTs mean more varied histories available to be assigned. The reason increasing each of these capabilities is important is to reduce *collisions* in predictor selection. However, larger predictors don't always result in more accuracy. Yeh and Patt noted that increasing history length rather than PHT size resulted in better performance. Their view may be coloured however, by their inclusion of floating point benchmarks with very few branch sites [35]. Chang *et al* discussed this in relation to McFarling's *gshare* predictor, where they showed that, despite the large silicon area assigned to the predictor, most of it remained unused for their benchmark suite, due to the high number of collisions involved [7].

Larger predictors also consume more power and execute slower. Most early predictor designs were optimistic in the amount of future space allocable for their predictors. Chang *et al* proposed 32K predictors in 1994, for example, and Evers tested predictors up to 384KB in size [8, 15]. Processor speeds were still increasing rapidly, advancements in manufacturing techniques were allowing more silicon space (in accordance with Moore's Law), and power was relatively cheap. Approximately 5% of a modern CPU is dedicated to branch prediction [11]. Since the mid-2000's, however, there has been a trend to smaller, more power-efficient predictors (and processors). Therefore, an important trade-off to consider in predictor design is the size of the predictors.

2.2 Benchmarking Methods

Over the years, the methods of testing branch predictor designs has varied. When evaluating an author's claimed accuracy, it is necessary to take into account the benchmark they used to test this. Because of the variance in methods used, it is hard to compare published results directly - rather most authors include previously published predictors in their implementations to facilitate comparison. There is, however, an alarming trend to ignore more recent predictors; most authors compare their work with McFarling's *gshare* predictor (which `XOR`s the `BRANCH_ADDRESS` and `GLOBAL_HISTORY` to index its PHT) only, despite it being superseded in his own work. Early implementers also used to restrict the running time of their benchmarks to a certain number of instructions - understandable given the cost of computing power at the time [21].

2.2.1 Benchmark Suites

When branch prediction was first introduced by researchers, they typically used small Unix programs, such as `wc` and `gzip` to test predictor performance. Later, the SPEC benchmark suite became popular [4, 5, 8, 21]. Early researchers often used the full suite, however, since 1994 it has been the trend to use a reduced set, as certain benchmarks were shown to skew the results due to high predictability. The primary culprits here were floating point benchmarks, and certain others that had highly used loops which were not data dependant [24]. Five of Yeh and Patts original ten benchmarks, for example, were floating point, and conversely, exhibited less increase in accuracy as predictor sizes were increased - though were very accurate initially [33, 35]. A more recent benchmark often used is the IBS Ultrix traces, which includes OS calls, library calls and context switches, and substantially more branches than SPEC [10]. Sechrest *et al* note that more recent benchmark suites tend to have a higher percentage of difficult branches [28].

2.2.2 Profiling Tools

Various methods have also been used to get the information required for testing. Most authors run their tests on profiled information, either gathered themselves or obtained from a third party source. Dynamic testing has also been done - that is, the calculations made during execution. One such program is `pixie`, used by Smith *et al*: '`pixie` is a program that allows you to trace, profile, or generate dynamic statistics for any program that runs on a MIPS processor. `pixie` is fast and convenient since it annotates executable object code with additional

instructions that collect the dynamic information during run time. Your program is run directly on the machine, no emulation is done nor are additional system calls made.' [30] Calder `et al` used ATOM, a two-staged tool that profiled then executed their benchmarks [5]. Sprangle *et al* used SoSS, a virtual system with a profiling API, which ran at 1/30th of a normal CPU [31]. And some used QPT on DEC machines, which also generates traces for system libraries [4].

Predictors algorithms themselves have typically been implemented as either stand-alone `C` programs, or plug-ins for these tools.

2.3 Performance Metrics

As well as using different benchmarks to test their predictors, predictor performances are usually reported against predictor size. However, there is also large variance in the way authors report their findings. The main methods of reporting are `success%`, `failure%`, and *contiguous instruction count*. Some authors have used more esoteric methods (such as the ratio between profiled accuracy and actual accuracy [27]), but these are usually individual papers. What is most noticeable is that authors report methods which show their predictor is more accurate than whatever they have chosen as baseline. When combined with the variance in benchmarks used, this makes direct comparison problematic. Young *et al* go so far as to say such comparison is pointless [36].

2.3.1 Success Rate

Success rate is the most basic performance indicator - how many branches were predicted correctly? This is useful for calculating processor efficiency. Pan, So and Rahmeh used prediction accuracy as their chosen metric [26]. However, as predictors get more accurate, the gains are smaller and so this statistic seemingly fell out of favour.

2.3.2 Failure Rate

As predictor designs became more accurate, it became more commonplace for authors to report their performance based on *mis-predictions*. This metric enables people to make claims such as "this predictor has 20% less mis-predictions than *gshare*", making it look like their predictor makes significantly better improvements than it does. However, this is smoke-and-mirrors, as the reality is these accuracy increases result in very minor speed increases. Combined with the tendency to ignore more recent predictors, this metric makes it difficult to

give credence to the reported results. Yeh and Patt used this method, claiming mis-prediction was the most relevant metric to use as it gives the squash rate [33]. It is interesting to note, however, that in later work they reverted to success rate as the relevant metric [35].

2.3.3 Contiguous Execution

Often there is also the option of reporting *average contiguous execution* - how many instructions are executed without a mis-prediction. There are two problems with this metric, however. First, it is easy to skew with benchmarks that have a large amount of loops. Second, unless the author specifically states branches only, it is easy to skew with benchmarks that have low branch density. As such, this metric does not provide an accurate depiction of predictor performance.

2.3.4 Average vs Total vs Best

There is also variance in how predictor performance is reported in relation to the benchmark suite run. It is common practice to report the *average* performance of the predictor across the benchmark suite, however, this is not a fair representation of how the predictor would actually run in reality. The reason for this is due to the varied complexity and length of the benchmarks used. Small, easily predictable benchmarks have an undue influence on the averages reported. Usually, authors report the individual results for benchmarks, allowing the reader to gauge the accuracy of the predictor for themselves.

Chapter 3

Implementation

In this project, branch predictors were tested using OpenPAT and MiBench. First, each predictor was tested for each size, ranging from 1B through to 1MB, with all possible configurations tested, in order to find the most accurate predictor. Second, the best settings were used to generate per-branch tables for the benchmarks. Third, a profiled(*biased*) predictor was investigated using these tables. And finally, a *Tagged* Branch Predictor(TBP) was investigated using these tables. The research was carried out on `LUNACY`, a 4GHz Intel i7 with 32GB of RAM and an SSD RAID, running Debian Wheezy.

3.1 Predictor Implementation

A predictor consists of a single `C` file, with a consistent API. A separate program was used to feed this API with information it would have available at run-time, sourced from trace files generated using a new OpenPAT tool, `branch_trace`. The information made available to the predictors was `BRANCH_ADDRESS`, `BRANCH_TARGET_ADDRESS`, `BRANCH_BIAS` and `BRANCH_TAKEN`. `BRANCH_TAKEN` allows the predictor to update itself after making its prediction. It should be noted that the use of trace files was merely to speed up processing time - OpenPAT can be configured to run the branch predictors directly during profiling. An original tool, `branch_prediction` was used in this way to test the functionality of predictors. A sample predictor (`gshare`) can be found in Appendix A.

Yeh and Patt propose a complex rule for specifying the associated hardware cost of a branch predictor [34]. However, in the same way that the overhead associated with a CPU cache is not specified, I do not include control hardware part of the hardware cost for comparative purposes. Also, like Ball and Larus, amongst others, I do not consider indirect branches [4].

3.1.1 Configuration Testing

Each predictor is initially configured in such a way that it tests all variants possible for the given predictor size. 17 different predictors were tested in this project: 4 static, 10 dynamic and 3 combined; with thousands of different configurations tested. Table 3.1 shows these predictors, and their respective features.

Table 3.1: Branch Predictors Tested

Predictor	Static	Saturating Counter	PHT	History	Combined
all [29]	✓				
backwards [29]	✓				
forwards [29]	✓				
bias [18]	✓				
saturating [29]		✓			
agree [31]	✓	✓	✓	✓	
bimode [20]		✓	✓	✓	
markov [10, 23]		✓	✓	✓	
bimodal [21, 29]		✓	✓		
global [21]		✓	✓	✓	
local [21, 34]		✓	✓	✓	
gselect [21]		✓	✓	✓	
gshare [21]		✓	✓	✓	
correlated [24]		✓	✓		
local/gshare [12, 21]		✓	✓	✓	✓
cascaded (all/gshare) [12, 21]		✓	✓	✓	✓
cascaded (all/gselect) [12, 21]		✓	✓	✓	✓

3.1.2 Optimizing

Once all predictor settings had been investigated, the best performing settings for each predictor were filtered out and set into a different per-branch-per-predictor table, which was then used to generate comparative results. These results can be seen in Section 4.1.

3.1.3 Biased Predictors

From the per-branch-per-predictor results, it is possible to calculate how a 1-bit *biased* predictor would behave for the same benchmark. The biased predictor proposed assumes a profiled executable, where one bit from the branch target address is used to indicate whether the branch predictor is accurate or not. If the branch predictor is less than 50% accurate, the bit is set to `1`, otherwise `0`. The underlying branch predictor's prediction is then muxed with it's bias, allowing for a profiled executable to account for predictor inaccuracies. This can be simulated per-branch as follows:

```
if (correct[$PC] < (total[$PC] + 1) / 2) {
        correct[$PC] = total[$PC] - correct[$PC];
}
```

It is essential to note that this process is invisible to the predictors - predictors that update their state based on whether they were correct or not are thus not affected by the bias implementation. Jacobsen *et al* suggested such a predictor in 1996 [19].

3.1.4 Combinations

It is also possible to calculate from the per-branch tables how a *Tagged Branch Predictor*(TBP) would perform. A Tagged Branch Predictor uses a set of predictors which always update every branch, but the predictor used to make the prediction is selected by a *tag bit* in the branch instruction. Once again, this process is invisible to the predictors - they are not effected by the TBP implementation.

The TBP is very similar to Grunwald *et al*'s proposed *Static Hybrid Predictor*, or Chang and Banerjee's *Profile-guided Multi-heuristic Predictor*, except that it allows for more sub-predictors, and every predictor is updated regardless of it's selection or not [9, 17]. Evers also considered such a predictor, however he wrote off a profiled version without actually testing it, claiming data variances were too great [15]. This goes against many earlier investigations which showed that in the vast majority of cases, data variance had negligible effect on a profiled predictor, provided the profile data was sufficiently representative of actual use [37]. Chang *et al* also proposed multiple component predictors, however it utilised a single underlying PHT into which all predictors indexed [8]. In the TBP, each predictor has its own inviolable PHT.

A TBP can use as many predictors as bit-space in the instruction allows; for example, a 32-bit instruction with 20-bits of address space could be modified to include two tag bits, with only 18 bits left for address space. It is theorised that this minor trade-off will have next to no effect on software - the largest branch found in preliminary testing used only 14 bits.

In order to find the best combination per size allowance, the following algorithm was used:

```
// Calculating best predictor sets per size
foreach (size) {
    best = {"", "", total=0};
    foreach (subset) {
        foreach (size_combination) {
            total = 0;
            foreach (branch) {
                total += max (subset);
            }
            report (size, subset, size_combination, total);
            if (total > best.total) {
                best = (subset, size_combination, total);
            }
        }
    }
    report (size, best);
}
```

It should be noted this is a simplified version of the algorithm used to calculate combinations. The process is complicated by the various different size combinations possible; static predictors, for example, were calculated as having no space, which allowed more space to be allocated to other predictors in the set. It was also complicated by the fact various predictors already used their own 'tag-bit'; namely the Agree and other biased predictors. This meant sometimes the subset has less predictors than it's bit-space would suggest. Preliminary testing also showed that a 3-bit TBP (eight predictors) had insignificant gains over a 2-bit TBP (four predictors) for the selected predictor set, so experimentation was confined to 2-bit TBPs. Across all sizes and optimisation levels, 175,654 different TBPs were tested.

3.2 OpenPAT Configuration

Traces for the benchmarks were gathered using OpenPAT. OpenPAT is an open-source profiling suite developed at Imperial College in conjunction with Cambridge, and is available upon registration from `www.openpat.org`. OpenPAT uses customised tools in order to generate profiled information - for example, the included `hotspot` tool was used to count the number of `IR`s in a given trace execution. For this project, results were generated using two custom tools - `branch_predictor` and `branch_trace`. These tools require modifications to the OpenPAT framework to work properly, however.

3.2.1 Target Address

Some of the predictors tested required information not immediately available in OpenPAT's data structures. Therefore, OpenPAT was modified to include in it's pre-parsing phase a java program which changed its `.label` field to hold the `TARGET_ADDRESS` for each branch. Due to framework limitations, these are the assembly file line numbers. This could mean that the results may not be identical to the benchmark running on hardware due to changes in the interference pattern. However, as these values are consistent across all benchmark/size tests, their usefulness for comparative analysis is not considered diminished.

Due to the way OpenPAT handles multiple input files, it was also necessary to implement a hashing scheme for both the branch address and branch target address.

3.2.2 Bias Bits

Sprangle claims that using the first encountered direction for the Agree predictor is 85% as accurate as using a profiled version [31]. However, sixteen years earlier, Smith had stated that '[it is possible] to use only the first execution of the branch instruction as a guide; a strategy of this type, although accurate, has been found to be slightly less accurate than other dynamic strategies' [29]. This was found to also be the case for the Agree predictor, so in order to garner best results, my implementation used a profiled bias-bit as its initial indicator.

Yeh and Patt also used the same data set as profiled to benchmark, allowing for perfect testing of the branch predictor[s] [33]. As I am testing optimal settings, it was therefore necessary, for the predictors that use a bias bit to be pre-loaded with a per-branch bias bit. In order to generate these bits, before either the bias or agree predictors were run, a `branch_bias` tool was run first, which determined

the exact bias of each branch. These biases were saved in a separate configuration file, which the respective predictors then loaded during their setup phase.

3.3 MiBench Configuration

MiBench is an benchmark suite designed for testing embedded systems [3]. Its use for profiling was suggested by Dr S. Spacey, author of OpenPAT. Like SPEC, several of the benchmarks contained in the suite were not considered useful for testing branch predictors, as they were far too easy to predict. Benchmarks were run at the highest achievable optimisation level - Table 3.2 records the benchmarks used, their respective optimisation level, and the number of branches encountered in each. In total, the chosen suite contained 485M branches from 4498 sites.

Table 3.2: Benchmarks Used

Benchmark	Optimisation	Branches
adpcm	O3	293M
dijsktra	O3	48M
fft	O3	3.6M
ispell	O1	85M
jpeg	O1	15M
patricia	O3	2.9M
qsort	O3	1.9M
rijndael	O3	2.4M
stringsearch	O3	377k
susan	O1	33M

3.4 Branch Densities

In order to calculate estimated processor performance, it was necessary to calculate the branch density, or *branches per instruction* (BpI). For this, the branch density per benchmark was found by dividing the total number of instructions run (found using OpenPAT's `hotspot` tool) by the number of branches encountered by `branch_trace`. The results per predictor can be found in Section 4.3.

3.5 Branch Offset Distribution

It was also necessary to calculate the number of bits typically required for a branch instruction, in order to ascertain the feasibility of implementing a TBP for current software. Branch offset was measured on `LUNACY` by disassembling `/bin/*`, and parsing the result with grep to test branch distance:

```
objdump -d /bin/* | grep 'j' > dump.csv
```

Minimum bits required for jump (including sign bit) were then calculated and collated using LibreOffice Calc. As this is for indicative purposes only, the small file sizes typical of `/bin/` were not considered a major issue regarding jump distances. Results can be found in Section 4.4.

Chapter 4

Results

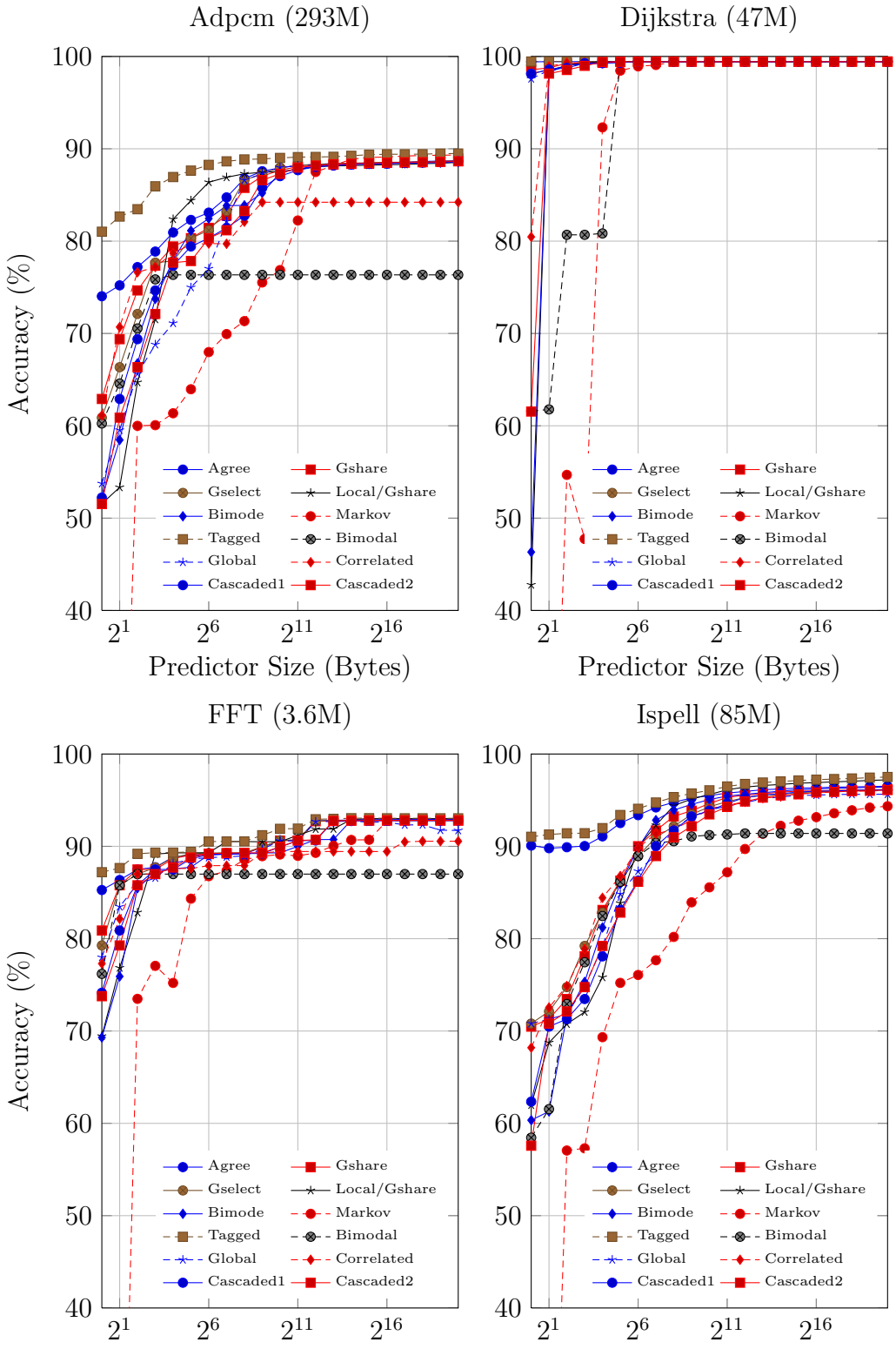
Using the trace files generated by `branch_trace`, two sets of results were garnered for each predictor. The first set, covered in the section Best Predictors per Benchmark, uses the 'locally' best settings - that is, the results presented are the best settings for the predictor for the benchmark in question. This means performance between benchmarks is not directly comparable, as the representative predictor may not be the same per benchmark.

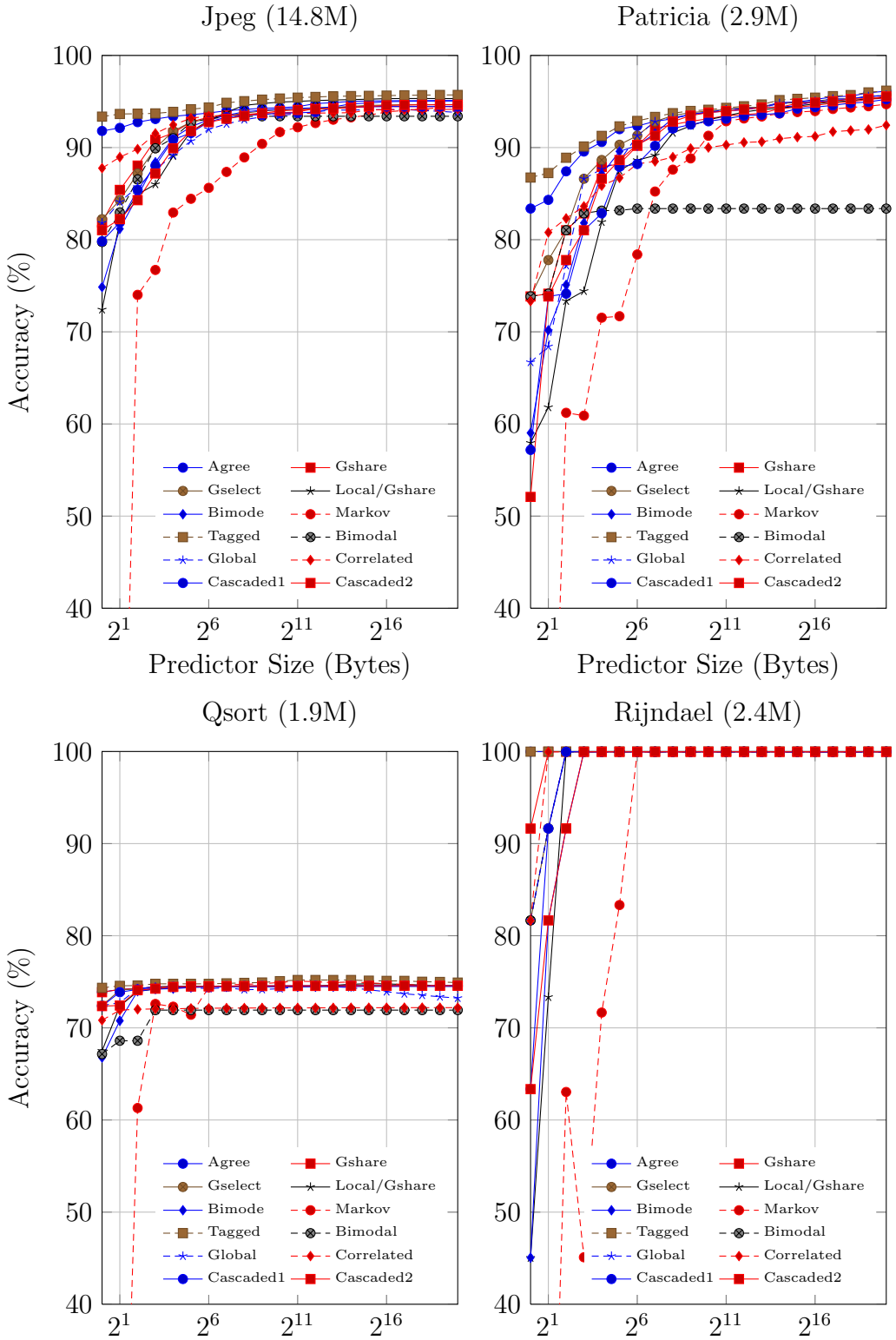
The second set of results contains the overall best performers, both global and local. The 'global' best settings are defined as the configurations for each predictor that performed the best overall, according to:

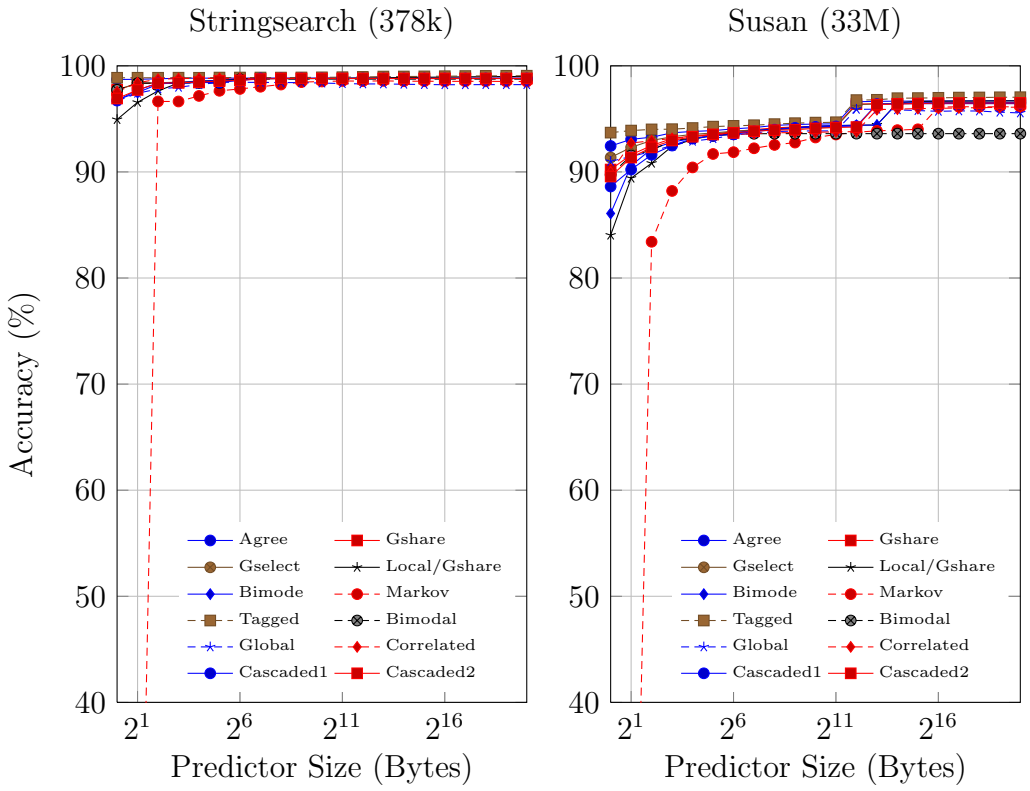
$$\frac{\text{branches\_correctly\_predicted}}{\text{branches\_encountered}}$$

The local best settings uses the best local configurations for each predictor, and takes an unweighted average. The global settings are in effect the same as taking a weighted average, by comparison. Section 4.3 only includes the predictors that were in the top five most accurate for a given size.

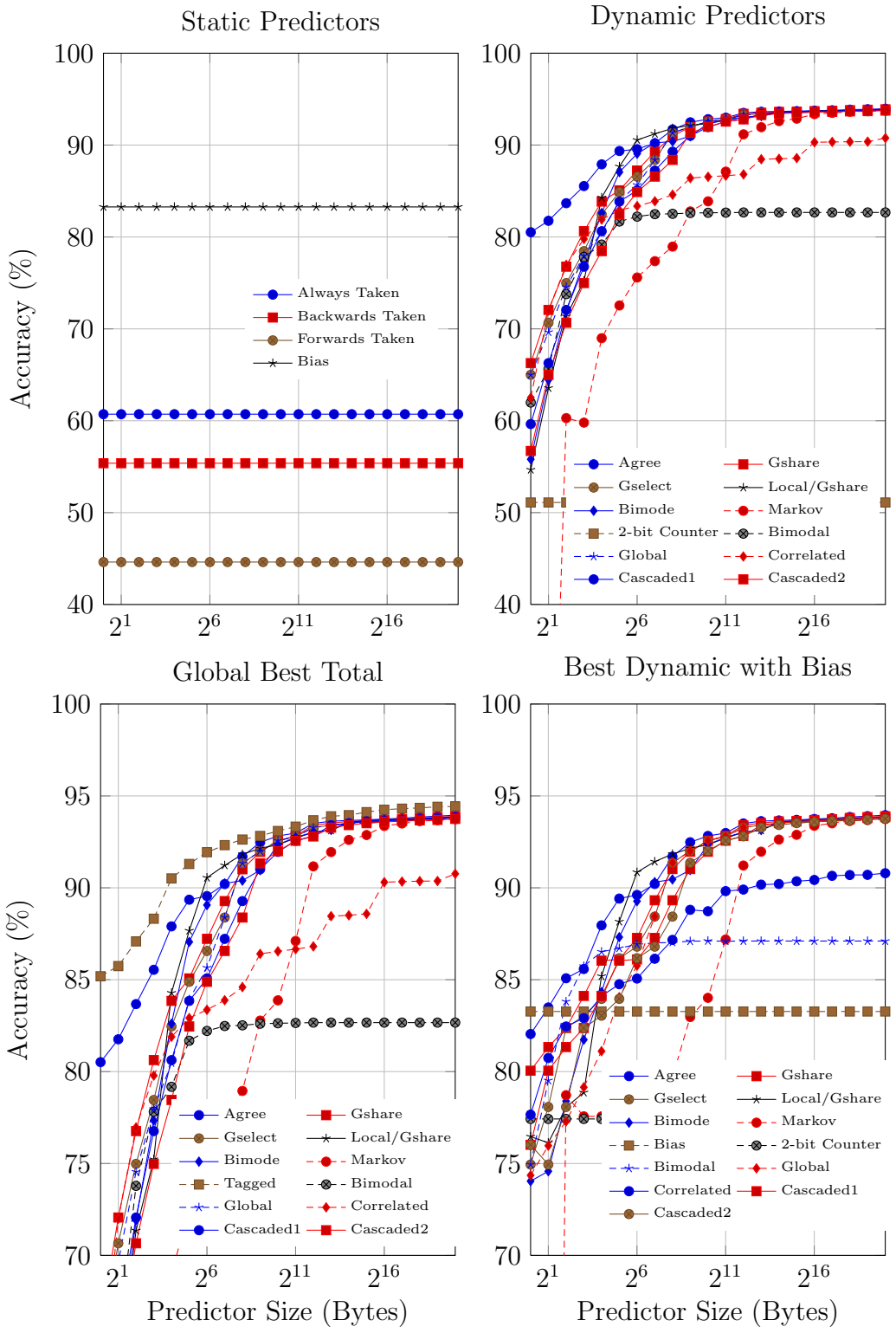
4.1 Best Predictors per Benchmark

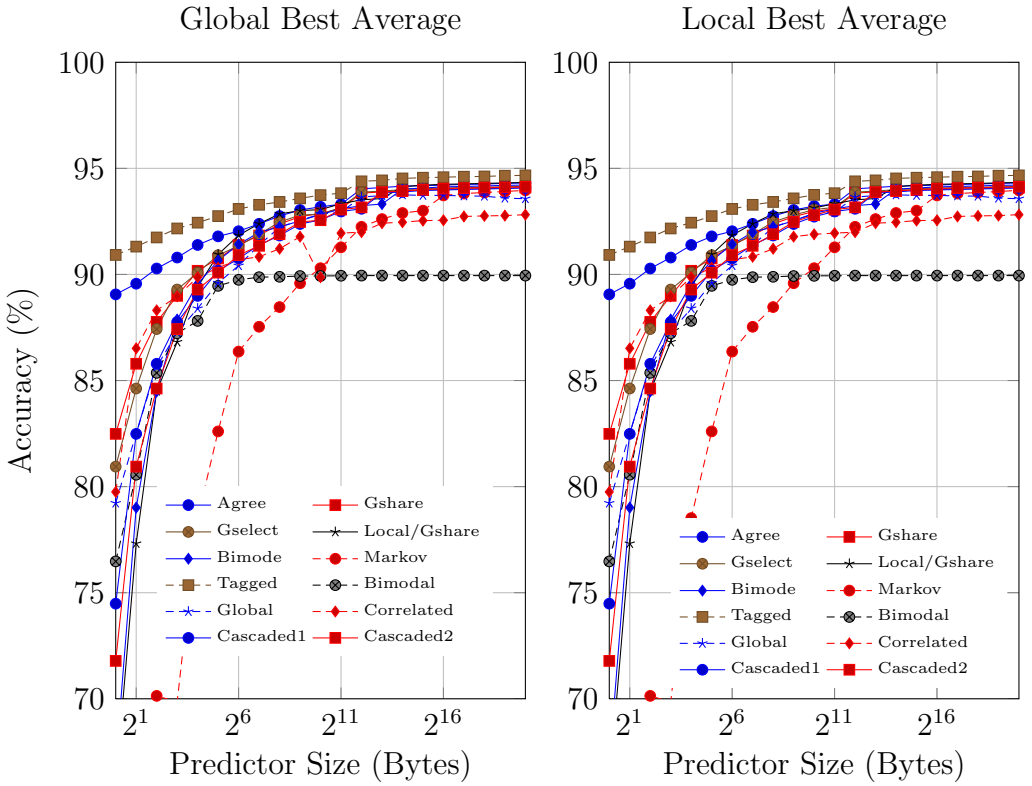



Jpeg (14.8M)
Patricia (2.9M)
Qsort (1.9M)
Rijndael (2.4M)
Accuracy (%)
Predictor Size (Bytes)
100
90
80
70
60
50
40
2^1
2^6
2^{11}
2^{16}
Agree
Gshare
Gselect
Local/Gshare
Bimode
Markov
Tagged
Bimodal
Global
Correlated
Cascaded1
Cascaded2


Stringsearch (378k)
Susan (33M)
Accuracy (%)
100
90
80
70
60
50
40
2^1
2^6
2^{11}
2^{16}
Predictor Size (Bytes)
Agree
Gshare
Gselect
Local/Gshare
Bimode
Markov
Tagged
Bimodal
Global
Correlated
Cascaded1
Cascaded2

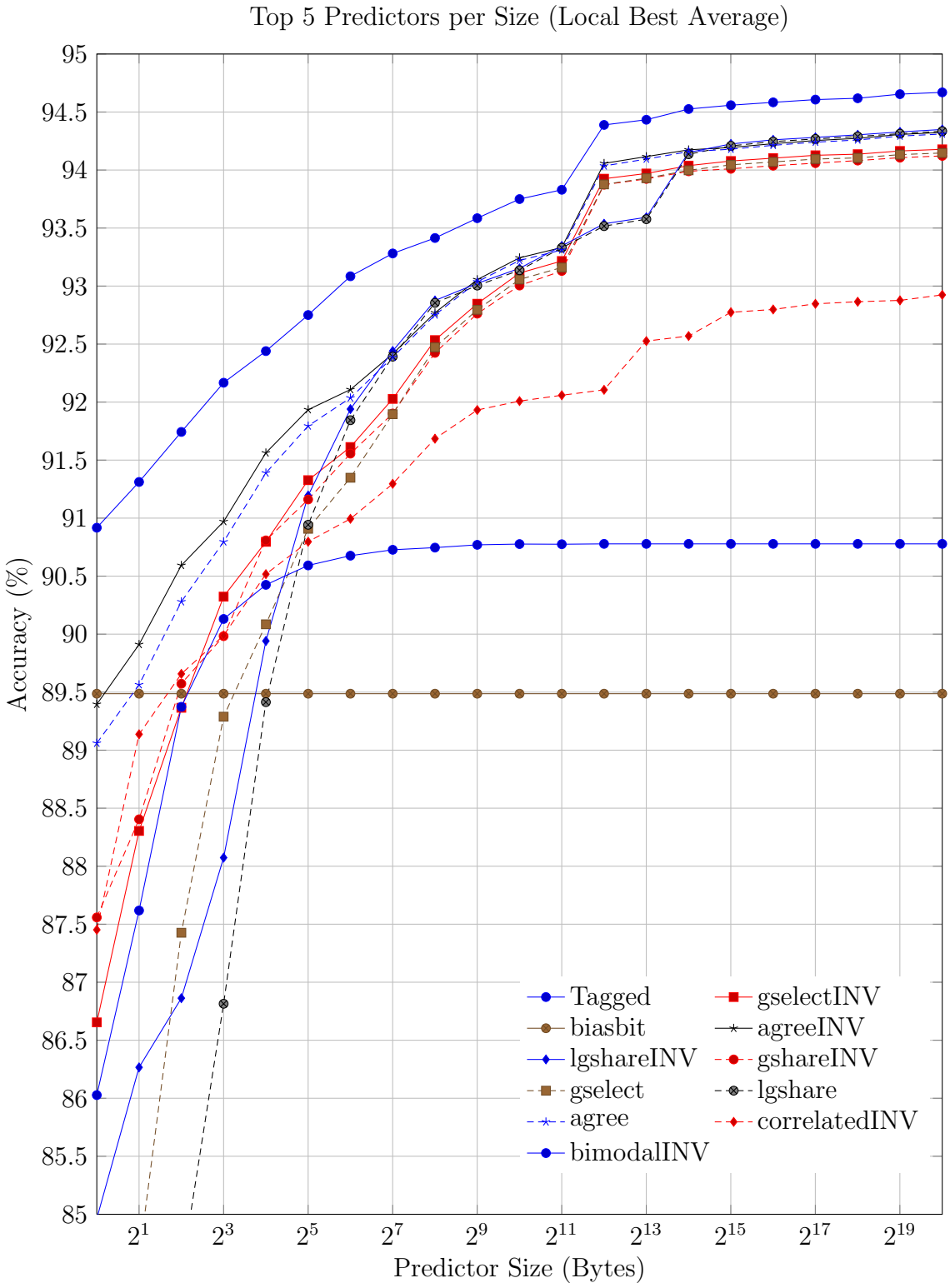
4.2 Overall Predictors

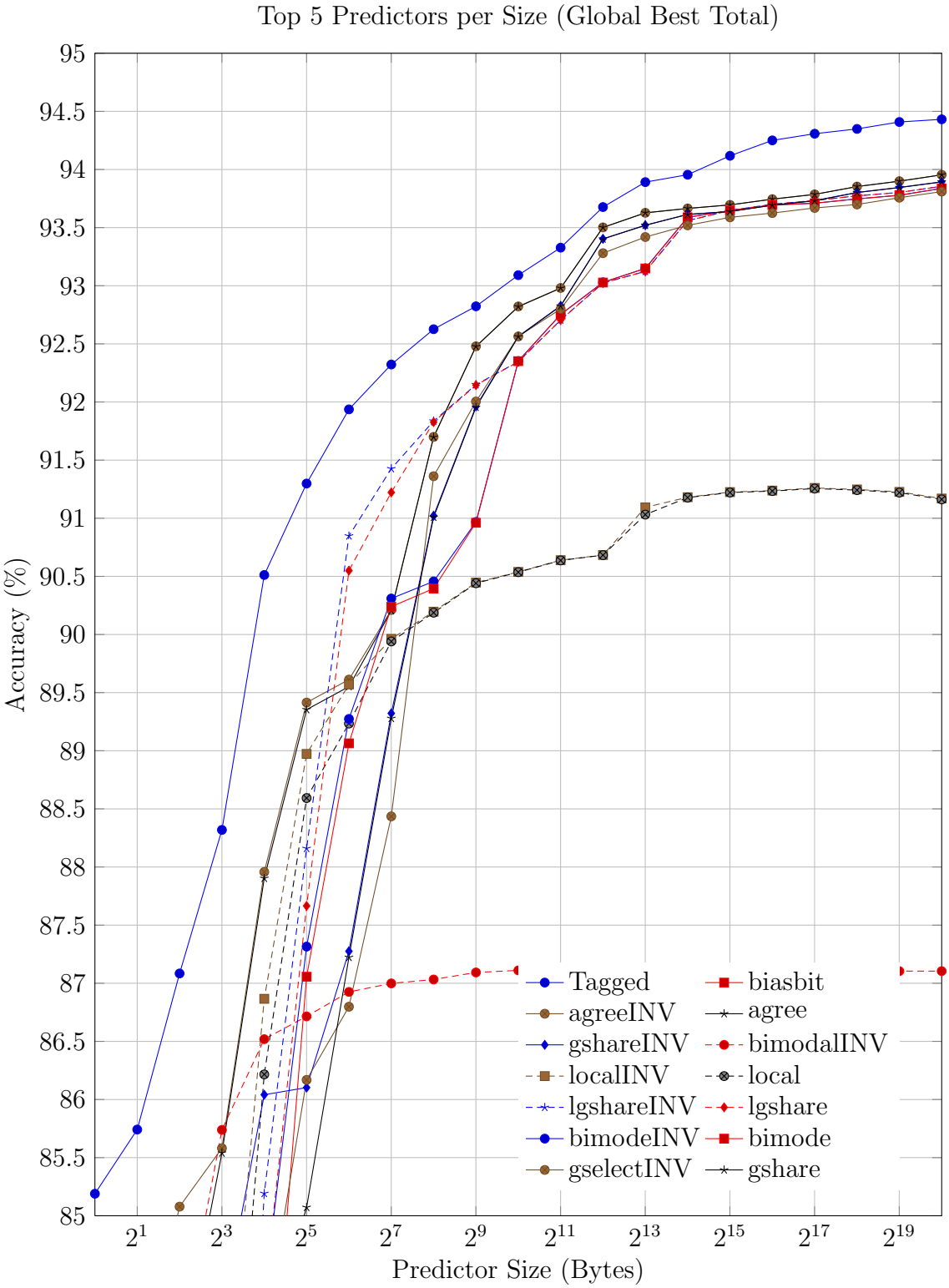




Results tables can be found in Appendix C.

4.3 Top 5 Predictors for Sizes



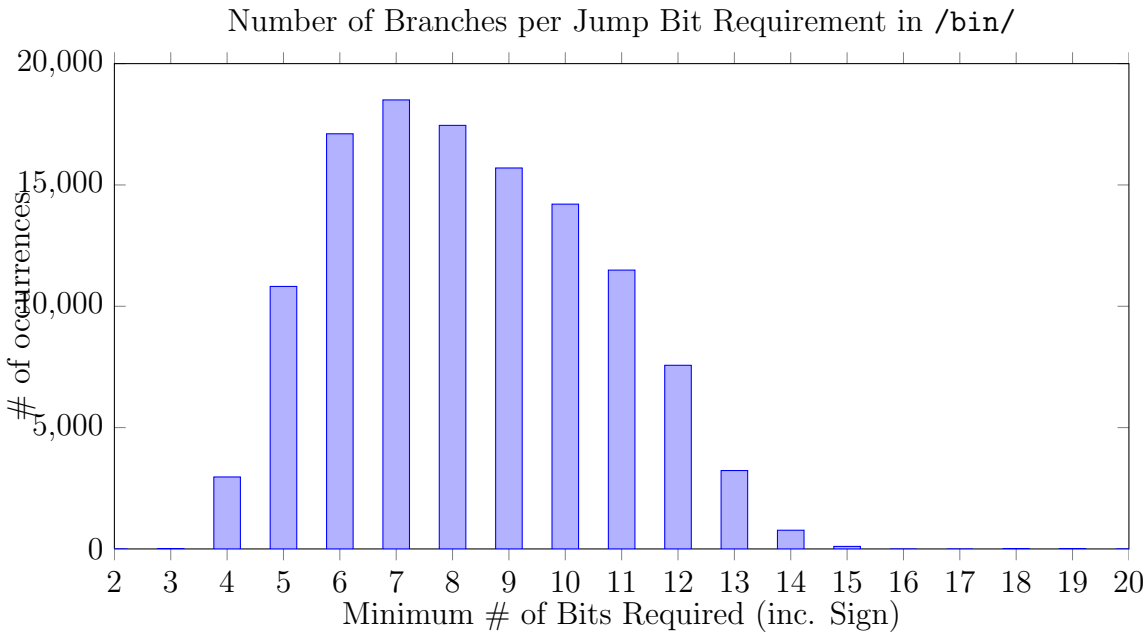

Top 5 Predictors per Size (Global Best Total)
Accuracy (%)
Predictor Size (Bytes)
95
94.5
94
93.5
93
92.5
92
91.5
91
90.5
90
89.5
89
88.5
88
87.5
87
86.5
86
85.5
85
2^1
2^3
2^5
2^7
2^9
2^{11}
2^{13}
2^{15}
2^{17}
2^{19}
Tagged
biasbit
agreeINV
agree
gshareINV
bimodalINV
localINV
local
lgshareINV
lgshare
bimodeINV
bimode
gselectINV
gshare

4.4 Branch Density

Table 4.1: Branches Per Instruction

Benchmark	Branches	Instructions	IpB	BpI
adpcm	293871549	1272182080	4.32	0.2320
dijkstra	47731814	174683579	3.66	0.2732
FFT	3596462	45261274	12.58	0.0795
ispell	85117867	[2]	6.4	0.1562
jpeg	14806832	136502823	9.21	0.1085
patricia	2909457	20144176	6.92	0.1444
qsort	1879057	14903057	7.93	0.1261
rijndael	2448387	407340139	166.37	0.0060
stringsearch	376842	1572823	4.17	0.2396
susan	32680929	489816121	14.99	0.0667

4.5 Jump Distance



[2]The `ispell` benchmark did not finish correctly, causing the `hotspot` tool to not output any results. The figures given here for BpI and IpB are the weighted average of the other benchmarks.

Chapter 5

Evaluation

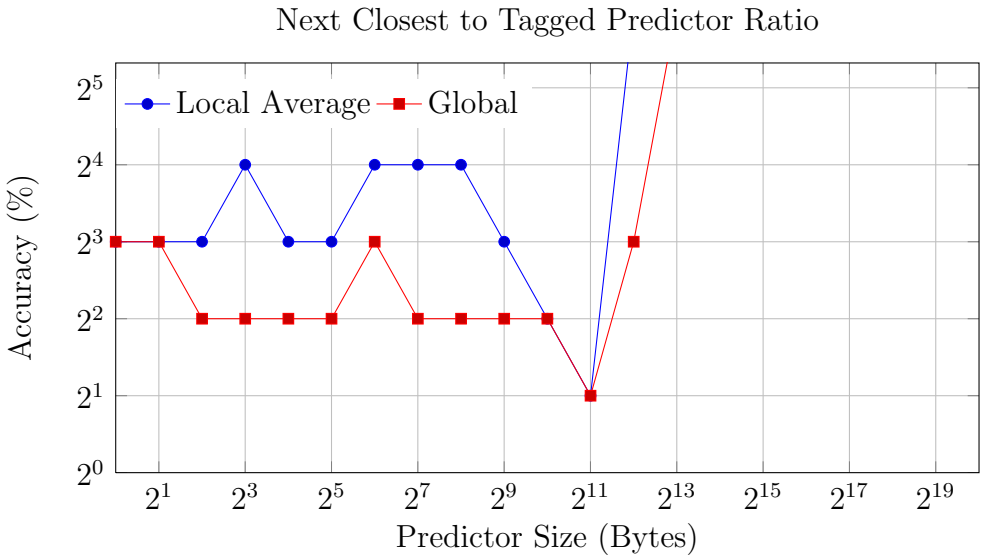
The results obtained in these experiments are not directly comparable to those published by authors. For example, Yeh and Patt reported 97.1% accuracy for their predictor, and McFarling 98.1% for his, and yet here they achieved a best performance of 91.2% and 93.8% at 1MB, respectively. One reason for this is the bias towards difficult-to-predict benchmarks in the MiBench subset, which accounted for almost eighty percent of the benchmark suite.

One very interesting result was the seeming inability of any predictor to accurately predict `qsort`, `fft` or `adpcm`. The reason for this is the same for both benchmarks: `adpcm` has a very small number of branch sites which deal with greatly varying data; likewise `qsort` and `fft`. Other benchmarks exhibiting similar behaviour were `ispell`, `jpeg`, `patricia` and `susan`, though for a different reason: all of these benchmarks had hundreds of active sites against which data is continually tested, making accurate prediction difficult.

Also of interest where the large jumps in accuracy that `susan` and `fft` both experienced around 4KB predictor size. It is unlikely that the predictors accuracy jumping noticeably at 4KB, and the fact that there were 4498 branch sites in the benchmark suite, are unrelated. The reason for this jump is likely to be the reduction in collisions caused by aliasing in the PHTs, and is probably the cause for the jump around 4KB in the total results, though this jump is somewhat smoother in the global total compared to the local average. This is because `fft` only accounted for 0.6% of the branches encountered, and is given unfair weighting in the local average. The jump from `susan` still has a noticeable effect however, since it accounts for approximately 6% of encountered branches.

For predictors that were already very accurate, the biased version produced negligible improvements. However, lower accuracy predictors, notably the static predictors, showed much greater improvements. Having said this, using a bias bit on a static predictor is unnecessary, as it produces the same accuracy as using the

bias bit itself to make the prediction. Nevertheless, at the cost of an instruction bit, biased predictors, as proposed here are not worth the architecture change.



The same can not be said, however, about the Tagged Branch Predictor. It consistently outperformed the other predictors, to such an extent that it outperformed much larger predictors in all except one size bracket (2KB). Figure 5.1 shows the ratio between TBP size and the size required by the next best performer to outperform it. Under 2KB, the TBP uses half, or sometimes a quarter, of the amount of chip space to outperform the next predictor. This has serious ramifications for low-power chips. A 4KB TBP outperforms the very best 16KB predictor tested; and none of the predictors tested at any size, even 1MB, could outperform an 8KB TBP.

The TBP is based on the predictors it was tested against, taking the best of them and combining them into a single predictor. As expected, Agree and local/gshare fared consistently well, and this accuracy was reflected in the TBP. This means that, as even more accurate base predictors are developed, a TBP units performance could be improved by adding it to the subset. For any given predictor size, a TBP will never be out-predicted by a single predictor unit.

5.1 Branch Distance Considerations

The main drawback of the TBP is the architecture change required to implement it - two bits in the branch instruction need to be assigned to be control bits. In a 64-bit system, this is perhaps not a major issue. However, consider a 32-bit

branch instruction, which typically consists of 12 instruction bits, and 20 target bits. If we were to include the TBP control bits in the target section, this reduces the available branch distance to 18 bits, or +/- 128K. Given modern programming paradigms, however, it is theorised that it is unlikely that any branch will exceed this jump distance, and if it were to, this could be got around in an architecture supporting the TBP by using a shorter branch to a jump instruction. This would result in a minor performance impact. This is also not a new proposal: another variant proposed by Calder and Grunwald is the use of indirect branches to facilitate long jump distances [5].

Section 4.5 shows the jump distances of `/bin/*` on the test system `LUNACY`. No branches used more than 18 bits, and in fact, only 6/134217 (0.004%) used more than 15 bits. The impact then, of changing to a TBP on the test system has nothing but positive effects.

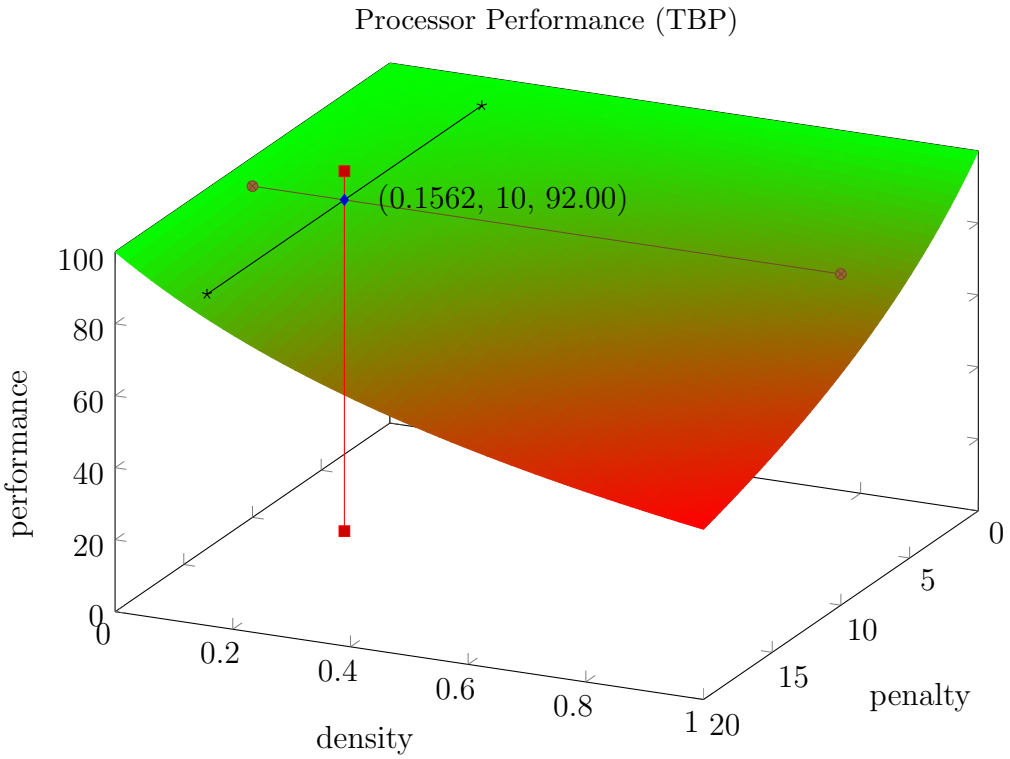
And the TBP provides more advantages than just increased accuracy: For example, any given program can be reconfigured by an external profiler, if required, without a recompile; the TBP will actually execute faster than a combined predictor, due to the single MUX required to implement its selection mechanism; the lower amount of predictor hardware required will engender itself towards low-power chips; the performance increase at larger sizes will have a notable effect on high-end processors; and programs do not have to be profiled on compilation - in this scenario the TBP will merely be a wrapper for the underlying base predictor, which should be the best performer of it's components.

5.2 Processor Performance from Predictor Accuracy and Branch Density

Predictor accuracy, however, doesn't have a direct relationship with CPU performance [37]. Other factors, most notably branch penalty and branch density, also have a significant impact, as seen in Figure 5.2. As pipeline depth increases, due to the associated mis-prediction penalty, processor performance decreases accordingly. The real performance of a processor, as a percentage of its speed, is given by the following formula:

$$perf = \frac{100\%}{1 - density + density * (accuracy + (1 - accuracy) * (1 + penalty))}$$

From this formula, we can calculate that a TBP with an accuracy of 94.5%, on a machine with a 10-cycle penalty (such as the DEC Alpha had [4]), and



running the MiBench sub-suite (BpI = 0.1562), will effectively run at 92.00% of maximum stated speed. This predictor has a mis-prediction rate of 5.5%, almost double the 3% Yeh and Patt claimed was unacceptable. Though, their predictor was also tested, and found to have a mis-prediction rate of 8.84% (see Appendix C, "mcf_local"). As has been discussed earlier, this was due to the highly predictable nature of their benchmarks [34]. In any case, their predictor for the same conditions has an effective CPU speed of 87.87% of maximum. In fact, the best predictor tested (a 1MB biased-Agree predictor, which also uses two instruction bits), had an effective CPU performance of 91.37%. This makes a CPU running a TBP 0.6% faster than one running an Agree predictor. In real terms, were the processors 10GHz, that's an extra 63M instructions processed at no extra cost.

5.3 Restrictions of Study

When looking at the results of this study, it is necessary to keep in mind the restrictions under which the study was done. Many of the criticisms leveled against earlier studies also apply to this one.

First, MiBench hasn't been updated since 2001, and only a subset of

the benchmark was used. Some elements of the benchmark suite, `rijndael`, `stringsearch`, and `dijkstra` were highly predictable, and account for approximately twenty percent of the branches encountered. The issue this presents is, what level of predictability is acceptable in a benchmark? Whilst it may seem preferable to have a benchmark suite that is as difficult to predict as possible, is this actually indicative of real performance? The answer to this question is beyond the scope of this study, but it bears consideration, as the results of this study are not directly comparable to the results of others for this very reason.

In a similar vein, another restriction is the small set of branches present in the benchmark suite. The MiBench subset used had only 4498 branch sites, which disadvantaged predictors larger than 4KB that did not use any form of history. It also didn't contain any indirect branches, which disadvantaged path-based predictors, such as Nair's Correlated predictor.

Another restriction was that libraries were not instrumented. As noted in Section 2.2.1, several authors claim the results from a single program are not really comparable to real-world application in a multi-tasking system. This is the main motivation behind benchmarks/traces which include system calls and multiple programs. However, whilst this means that the results of this study are not directly comparable to those of other studies, that does not reduce their usefulness for comparative analysis.

The most recent predictor (aside from the TBP) was developed in 1998. There have been other advances in predictor algorithms made since, notably in perceptron-based predictors, and other exotic offerings from the Championship Branch Prediction competition. Nevertheless, regardless of later predictor's performance, they could be easily assimilated into the TBP, and so their relative accuracies are not actually that important.

The final restriction is on the veracity of the results presented. Every possible effort has been made, given the time constraints, to present each tested predictor in its optimal settings. Whilst this was achieved for all the stand-alone predictors, the biased and Tagged predictors were only tested across a subset of their possible configurations. This subset consisted of using the very best settings per component predictor, but it is possible that sub-optimal components may have produced better combined performance. Further investigation may be warranted.

Chapter 6

Conclusion

In this research, optimal settings have been found for a range of predictors. These settings have been used in conjunction with a benchmark suite, consisting of a subset of the MiBench benchmark. The predictors have then been compared for a range of sizes and optimisation levels. A Tagged Branch Predictor has been developed, which uses a small architecture change along with underlying predictor technology to improve branch prediction accuracy. Using a single tag bit to indicate predictor accuracy has also been investigated.

It has been shown that, for the predictors implemented, a biased predictor which uses only a single instruction bit has limited benefit. It has also been shown, however, that the use of more bits for a *tagged* branch predictor results in substantially better predictor performance. If ultimate performance is the aim, the results obtained indicate that a large TBP could be a viable option. Conversely, if the aim is efficiency, a smaller TBP outperforms single predictors over four times larger. Branches larger than that afforded by a TBP, assuming an 18-bit branch offset, are very rare, and could be handled using a short branch to a jump, at very little cost. And there is no cost to using programs either profiled, not-profiled, or re-profiling them either.

This project was also subject to some limitations - particularly in the range of predictors tested. The most recent branch predictor utilised in this study was proposed in 1998. Several more advanced predictors have been proposed since. It would be useful to see how they compare, and affect the TBP implementation proposed.

Bibliography

[1] Quad core test processor chip doing 10 ghz, August 2013. `https://www.youtube.com/watch?v=MUq50N9w5D8`.

[2] Eniac - wikipedia, November 2014. `http://en.wikipedia.org/wiki/ENIAC`.

[3] Mibench version 1.0, March 2014. `http://wwweb.eecs.umich.edu/mibench/`.

[4] Thomas Ball and James R Larus. *Branch prediction for free*, volume 28. ACM, 1993.

[5] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 2–11. IEEE, 1994.

[6] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 287–296. IEEE, 1995.

[7] Po-Yung Chang, Marius Evers, and Yale N Patt. Improving branch prediction accuracy by reducing pattern history table interference. *International journal of parallel programming*, 25(5):339–362, 1997.

[8] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 22–31. ACM, 1994.

[9] Pohua Chang and Upal Banerjee. Profile-guided multi-heuristic branch prediction. In *1995 International Conference on Parallel Processing*, pages 215–218, 1995.

[10] J Bradley Chen, Michael D Smith, Cliff Young, and Nicolas Gloy. An analysis of dynamic branch prediction schemes on system workloads. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 12–12. IEEE, 1996.

[11] Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 167–178. IEEE Computer Society, 1998.

[12] Karel Driesen and Urs Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 249–258. IEEE Computer Society Press, 1998.

[13] AN Eden, J Ringenberg, S Sparrow, and T Mudge. Hybrid myths in branch prediction. In *Proc. 7th International Conference on Information Systems Analysis and Synthesis*, 2001.

[14] Joel S Emer and Douglas W Clark. A characterization of processor performance in the vax-11/780. In *ACM SIGARCH Computer Architecture News*, volume 12, pages 301–310. ACM, 1984.

[15] Marius Evers. *Improving branch prediction by understanding branch behavior*. PhD thesis, The University of Michigan, 2000.

[16] Marius Evers, Po-Yung Chang, and Yale N Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 3–11. ACM, 1996.

[17] Dirk Grunwald, Donald Lindsay, and Benjamin Zorn. Static methods in hybrid branch prediction. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 222–229. IEEE, 1998.

[18] WW Hwu, Thomas M Conte, and Pohua P Chang. Comparing software and hardware schemes for reducing the cost of branches. In *ACM SIGARCH Computer Architecture News*, volume 17, pages 224–233. ACM, 1989.

[19] Erik Jacobsen, Eric Rotenberg, and James E Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 142–152. IEEE Computer Society, 1996.

[20] Chih-Chieh Lee, I-CK Chen, and Trevor N Mudge. The bi-mode branch predictor. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 4–13. IEEE, 1997.

[21] Scott McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[22] Scott McFarling and J Hennesey. *Reducing the cost of branches*, volume 14. IEEE Computer Society Press, 1986.

[23] Trevor N Mudge, I-Cheng Chen, and John Coffey. *Limits to branch prediction*. University of Michigan, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, 1996.

[24] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 15–23. IEEE Computer Society Press, 1995.

[25] Ravi Nair. Optimal 2-bit branch predictors. *Computers, IEEE Transactions on*, 44(5):698–702, 1995.

[26] Shien-Tai Pan, Kimming So, and Joseph T Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 27, pages 76–84. ACM, 1992.

[27] Jason RC Patterson. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Notices*, volume 30, pages 67–78. ACM, 1995.

[28] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. The role of adaptivity in two-level adaptive branch prediction. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 264–269. IEEE Computer Society Press, 1995.

[29] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[30] Michael D Smith. *Tracing with pixie*. Computer Systems Laboratory, Stanford University, 1991.

[31] Eric Sprangle, Robert S Chappell, Mitch Alsup, and Yale N Patt. The agree predictor: A mechanism for reducing negative branch history interference.

In *ACM SIGARCH Computer Architecture News*, volume 25, pages 284–291. ACM, 1997.

[32] David W Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.

[33] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM, 1991.

[34] Tse-Yu Yeh and Yale N Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 124–134. ACM, 1992.

[35] Tse-Yu Yeh and Yale N Patt. A comparison of dynamic branch predictors that use two levels of branch history. *ACM SIGARCH Computer Architecture News*, 21(2):257–266, 1993.

[36] Cliff Young, Nicolas Gloy, and Michael D Smith. *A comparative analysis of schemes for correlated branch prediction*, volume 23. ACM, 1995.

[37] Cliff Young and Michael D Smith. Improving the accuracy of static branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 29, pages 232–241. ACM, 1994.

Appendices

Appendix A

Sample Predictor Code Listing (`gshare`)

```
/*  mcfarling_gshare.c
 *  ==================
 *  By Neil Bradley
 *
 *  All rights reserved.
 *
 *  This predictor XORs the PC and GHR to index into the PHT.
 *
 *  See McFarling, S.; "Combining Branch Predictors", 1993.
 */

#include "predictors.h"
#include <stdio.h>
#include <math.h>

/* MACROS */
#define MCF_GSHARE_PINDEX(a, b)    (((a) * mcf_gshare_variants) + (b))

/* VARIABLES */
int mcf_gshare_size;           // How large the PHTs are
int mcf_gshare_variants;       // How many different variants there are
unsigned int mcf_gshare_GHR;   // The global history register
char *mcf_gshare_PHT;          // [size][variants]
long *mcf_gshare_totals;       // The tallies for each predictor

/* Setup the variables for basic predictor. */
void mcf_gshare_SETUP(int max_size) {
    int i;

    // Calculate sizes
    mcf_gshare_size = max_size;
    i = max_size; mcf_gshare_variants = 1;
    while ((i >>= 1) > 0) {
        mcf_gshare_variants++;
    }

    // Allocate tables
    mcf_gshare_totals =
        (long *)calloc(mcf_gshare_variants, sizeof(long));
```

```
    mcf_gshare_PHT =
        (char *)calloc(max_size * mcf_gshare_variants, sizeof(char));

    // Initialise totals and PHT
    for (i = 0; i < mcf_gshare_variants; i++) {
        mcf_gshare_totals[i] = 0;
    }
    for (i = 0; i < mcf_gshare_variants * max_size; i++) {
        mcf_gshare_PHT[i] = 3; // Start at fully used.
    }
    mcf_gshare_GHR = 0xFFFFFFFF;
} // mcf_gshare_SETUP ================================================

/*  The variables passed in provide all the information need to
    calculate the prediction retrospectively. */
int mcf_gshare(unsigned int address, unsigned int target, int taken) {
    int predicted = 0, i, best = 0, prediction = 0, n, m, index;

    // Find best predictor
    for (i = 0; i < mcf_gshare_variants; i++) {
        if (mcf_gshare_totals[i] > mcf_gshare_totals[best]) {
            best = i;
        }
    } // Finding best predictor

    // Cycle through predictors
    for (i = 0; i < mcf_gshare_variants; i++) {
        // Calculate parts
        n = address & (mcf_gshare_size - 1);
        m = mcf_gshare_GHR & (~(0xFFFFFFFF << i));
        index = MCF_GSHARE_PINDEX(n ^ m, i);

        // Calculate prediction
        prediction = mcf_gshare_PHT[index] >> 1;
        if (i == best) {
            predicted = prediction;
        }

        // Update PHT
```

```
        if (taken == YES && mcf_gshare_PHT[index] < 3) {
            mcf_gshare_PHT[index]++;
        } else if (taken == NO && mcf_gshare_PHT[index] > 0) {
            mcf_gshare_PHT[index]--;
        }

        // Update totals
        if (prediction == taken) {
            mcf_gshare_totals[i]++;
        }
    } // Cycling through predictors

    // Update GHR
    mcf_gshare_GHR = (mcf_gshare_GHR << 1) | taken;

    return predicted;
} // mcf_gshare ====================================================


// Publishes the predictor's current statistics to stdout.
void mcf_gshare_DUMP(void) {
    int i;
    _OP_DISPLAY("gshare (McFarling) predictor active:");
    for (i = mcf_gshare_variants; i > 0; i--) {
        _OP_DISPLAY("\t%d:%d", i - 1, mcf_gshare_totals[i - 1]);
    }
    _OP_DISPLAY("\n");
} // mcf_gshare_DUMP ================================================


// Cleans up the predictor's memory space.
void mcf_gshare_CLOSE(void) {
    free(mcf_gshare_totals);
    free(mcf_gshare_PHT);
} // mcf_gshare_FINALISE ============================================
```

Appendix B

Sample Predictor Totals (`gshare`)

```
../adpcm_1.trace: Total=146406294 Local={GHR=14b}134471978 LocalInv={GHR=14b_INV}134471978 Global={GHR=14b}134471978 GlobalInv={GHR=14b}134471978
../adpcm_2.trace: Total=133087388 Local={GHR=14b}120686831 LocalInv={GHR=14b_INV}120686831 Global={GHR=14b}120686831 GlobalInv={GHR=14b}120686831
../adpcm_3.trace: Total=7531492 Local={GHR=14b}6475721 LocalInv={GHR=14b_INV}6475721 Global={GHR=14b}6475721 GlobalInv={GHR=14b}6475721
../adpcm_4.trace: Total=6846375 Local={GHR=14b}5769607 LocalInv={GHR=14b_INV}5769607 Global={GHR=14b}5769607 GlobalInv={GHR=14b}5769607
../dijkstra_1.trace: Total=39838870 Local={GHR=4b}39616742 LocalInv={GHR=4b_INV}39616743 Global={GHR=14b}39615129 GlobalInv={GHR=14b}39615130
../dijkstra_2.trace: Total=7892944 Local={GHR=4b}7848955 LocalInv={GHR=4b_INV}7848956 Global={GHR=14b}7848461 GlobalInv={GHR=14b}7848462
../fft_1.trace: Total=1556528 Local={GHR=14b}1420509 LocalInv={GHR=14b_INV}1421853 Global={GHR=14b}1420509 GlobalInv={GHR=14b}1421853
../fft_2.trace: Total=1589296 Local={GHR=14b}1453273 LocalInv={GHR=14b_INV}1454619 Global={GHR=14b}1453273 GlobalInv={GHR=14b}1454619
../fft_3.trace: Total=143398 Local={GHR=12b}134565 LocalInv={GHR=12b_INV}134658 Global={GHR=14b}134416 GlobalInv={GHR=14b}134585
../fft_4.trace: Total=307240 Local={GHR=12b}289820 LocalInv={GHR=12b_INV}290022 Global={GHR=14b}289815 GlobalInv={GHR=14b}289891
../ispell_1.trace: Total=892998 Local={GHR=12b}846648 LocalInv={GHR=12b_INV}846863 Global={GHR=14b}844978 GlobalInv={GHR=14b}845236
../ispell_2.trace: Total=84224869 Local={GHR=14b}81094140 LocalInv={GHR=14b_INV}81094921 Global={GHR=14b}81094140 GlobalInv={GHR=14b}81094921
../jpeg_1.trace: Total=10468313 Local={GHR=14b}9912855 LocalInv={GHR=14b_INV}9913556 Global={GHR=14b}9912855 GlobalInv={GHR=14b}9913556
../jpeg_2.trace: Total=1069396 Local={GHR=14b}1023442 LocalInv={GHR=14b_INV}1024002 Global={GHR=14b}1023442 GlobalInv={GHR=14b}1024002
../jpeg_3.trace: Total=2975508 Local={GHR=14b}2762802 LocalInv={GHR=14b_INV}2763445 Global={GHR=14b}2762802 GlobalInv={GHR=14b}2763445
../jpeg_4.trace: Total=293615 Local={GHR=14b}275372 LocalInv={GHR=14b_INV}275924 Global={GHR=14b}275372 GlobalInv={GHR=14b}275924
../patricia_1.trace: Total=402418 Local={GHR=14b}375233 LocalInv={GHR=14b_INV}375253 Global={GHR=14b}375233 GlobalInv={GHR=14b}375253
../patricia_2.trace: Total=2507039 Local={GHR=14b}2353816 LocalInv={GHR=14b_INV}2353835 Global={GHR=14b}2353816 GlobalInv={GHR=14b}2353835
../qsort_1.trace: Total=150437 Local={GHR=5b}105094 LocalInv={GHR=5b_INV}105096 Global={GHR=14b}105072 GlobalInv={GHR=14b}105074
../qsort_2.trace: Total=1728620 Local={GHR=14b}1370008 LocalInv={GHR=2b_INV}1377615 Global={GHR=14b}1370008 GlobalInv={GHR=14b}1370010
../rijndael_1.trace: Total=117270 Local={GHR=4b}117192 LocalInv={GHR=4b_INV}117215 Global={GHR=14b}117157 GlobalInv={GHR=14b}117180
../rijndael_2.trace: Total=97767 Local={GHR=4b}97705 LocalInv={GHR=4b_INV}97715 Global={GHR=14b}97670 GlobalInv={GHR=14b}97680
../rijndael_3.trace: Total=1218168 Local={GHR=4b}1218090 LocalInv={GHR=4b_INV}1218113 Global={GHR=14b}1218055 GlobalInv={GHR=14b}1218078
../rijndael_4.trace: Total=1015182 Local={GHR=4b}1015120 LocalInv={GHR=4b_INV}1015130 Global={GHR=14b}1015085 GlobalInv={GHR=14b}1015095
../stringsearch_1.trace: Total=15743 Local={GHR=1b}15525 LocalInv={GHR=1b_INV}15526 Global={GHR=14b}15364 GlobalInv={GHR=14b}15384
../stringsearch_2.trace: Total=361099 Local={GHR=3b}357484 LocalInv={GHR=3b_INV}357485 Global={GHR=14b}356855 GlobalInv={GHR=14b}356856
../susan-c_1.trace: Total=39122 Local={GHR=9b}37394 LocalInv={GHR=9b_INV}37415 Global={GHR=14b}37175 GlobalInv={GHR=14b}37195
../susan-c_2.trace: Total=1062352 Local={GHR=10b}991386 LocalInv={GHR=10b_INV}991399 Global={GHR=14b}989056 GlobalInv={GHR=14b}989071
../susan-e_1.trace: Total=74427 Local={GHR=10b}72309 LocalInv={GHR=10b_INV}72345 Global={GHR=14b}72074 GlobalInv={GHR=14b}72113
../susan-e_2.trace: Total=2869864 Local={GHR=14b}2673272 LocalInv={GHR=14b_INV}2673408 Global={GHR=14b}2673272 GlobalInv={GHR=14b}2673408
../susan-s_1.trace: Total=1756847 Local={GHR=14b}1749474 LocalInv={GHR=14b_INV}1749498 Global={GHR=14b}1749474 GlobalInv={GHR=14b}1749498
../susan-s_2.trace: Total=26878317 Local={GHR=14b}26767324 LocalInv={GHR=14b_INV}26767360 Global={GHR=14b}26767324 GlobalInv={GHR=14b}26767360
GLOBAL Total=485419196 Global={GHR=14b}453392019 GlobalInv={GHR=14b}453398851
```

Appendix C

Totals Tables

Total	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
mcf_localINV	75.3771	73.7892	73.7836	82.8783	86.8661	88.9726	89.5704	89.9625	90.1964	90.4465	90.5384	90.6394	90.6838	91.0934	91.1807	91.2250	91.2379	91.2604	91.2477	91.2267	91.1696
mcf_gselectINV	74.9432	78.0754	82.3712	83.0421	83.9568	86.1686	86.7982	88.4353	91.3616	92.0042	92.5640	92.8002	93.2800	93.4190	93.5195	93.5883	93.6251	93.6692	93.6989	93.7573	93.8097
biasbit	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705
agreeINV	82.0439	83.4887	85.0784	85.5810	87.9573	89.4145	89.6121	90.2109	91.6998	92.4791	92.8221	92.9810	93.5019	93.6274	93.6649	93.6954	93.7448	93.7859	93.8531	93.8995	93.9540
saturating	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047	51.1047
mcf_lgshareINV	76.4629	76.1264	78.0627	78.8600	85.1923	88.1596	90.8477	91.4263	91.8365	92.1489	92.3470	92.7053	93.0237	93.1237	93.5597	93.6477	93.7008	93.7333	93.7735	93.8027	93.8560
forwards	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228	44.6228
mcf_bimodal	61.9786	65.6260	73.7827	77.8175	79.1724	81.6791	82.2127	82.4809	82.5174	82.6058	82.6304	82.6469	82.6665	82.6665	82.6665	82.6665	82.6665	82.6665	82.6665	82.6665	82.6665
mcf_bimodalINV	74.9432	79.5160	83.8119	85.7386	86.5191	86.7151	86.9256	86.9986	87.0323	87.0929	87.1108	87.0956	87.1044	87.1044	87.1044	87.1044	87.1044	87.1044	87.1044	87.1044	87.1044
mcf_global	64.9945	69.6400	74.5478	77.9639	80.4679	83.9295	85.6532	88.3845	91.3351	91.9919	92.5561	92.7932	93.2730	93.4122	93.4897	93.5274	93.5864	93.6185	93.6928	93.7514	93.7993
mcf_globalINV	74.3559	75.9690	77.2908	79.1479	81.1101	84.0524	85.7630	88.4353	91.3616	92.0042	92.5640	92.8002	93.2800	93.4190	93.4960	93.5336	93.5924	93.6244	93.6989	93.7573	93.8051
saturatingINV	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237	77.4237
cascadedINV	76.0160	80.0502	81.3351	82.3712	84.1174	86.0411	86.1026	87.2753	89.3219	91.0219	91.9623	92.5651	92.8277	93.4036	93.5209	93.6141	93.6388	93.6932	93.7321	93.8025	93.8455
correlated	62.5056	71.9947	76.9595	79.7833	81.8945	82.9148	83.3611	83.8855	84.5982	86.4069	86.5458	86.6617	86.8122	88.4581	88.5044	88.5820	90.3057	90.3382	90.3621	90.3711	90.7637
all	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144	60.7144
mcf_gshareINV	80.0502	81.3351	82.3712	84.1174	86.0411	86.1026	87.2753	89.3219	91.0219	91.9623	92.5651	92.8277	93.4036	93.5209	93.6141	93.6388	93.6932	93.7321	93.8025	93.8455	93.8938
mcf_gshare	66.2711	72.0610	76.7617	80.6244	83.8474	85.0720	87.2227	89.2781	91.0084	91.9547	92.5618	92.8253	93.4022	93.5195	93.6133	93.6379	93.6922	93.7310	93.8014	93.8444	93.8927
bimodeINV	74.0397	74.5791	78.3908	81.7353	84.3285	87.3155	89.2738	90.3096	90.4567	90.9678	92.3522	92.7535	93.0296	93.1497	93.5889	93.6472	93.6946	93.7099	93.7470	93.7783	93.8376
mcf_gselect	64.9945	70.6590	74.9782	78.4552	82.4605	84.8856	86.5626	88.3845	91.3351	91.9919	92.5561	92.7932	93.2730	93.4122	93.5160	93.5851	93.6224	93.6678	93.6953	93.7514	93.8068
allINV	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705
cascaded	59.6376	66.2711	72.0610	76.7617	80.6244	83.8474	85.0720	87.2227	89.2781	91.0084	91.9547	92.5618	92.8253	93.4022	93.5195	93.6133	93.6379	93.6922	93.7310	93.8014	93.8444
backwards	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772	55.3772
mcf_lgshare	54.6688	63.5622	71.3504	75.2288	84.2848	87.6652	90.5496	91.2221	91.8273	92.1448	92.3452	92.7044	93.0230	93.1233	93.5595	93.6473	93.7006	93.7331	93.7731	93.8023	93.8552
cascaded2	56.7124	64.9945	70.6590	74.9782	78.4552	82.4605	84.8856	86.5626	88.3845	91.3351	91.9919	92.5561	92.7932	93.2730	93.4122	93.5160	93.5851	93.6224	93.6678	93.6953	93.7514
mcf_local	60.8970	66.4562	70.0944	81.9418	86.2159	88.5929	89.2339	89.9400	90.1876	90.4408	90.5361	90.6370	90.6825	91.0314	91.1776	91.2213	91.2337	91.2554	91.2420	91.2202	91.1621
comb	85.1895	85.7417	87.0841	88.3190	90.5118	91.2985	91.9365	92.3225	92.6261	92.8230	93.0910	93.3277	93.6768	93.8913	93.9549	94.1177	94.2503	94.3075	94.3489	94.4086	94.4315
markov	00.0000	00.0000	60.2871	59.7945	68.9840	72.5432	75.5870	77.3652	78.9483	82.7754	83.8767	87.1089	91.1668	91.9461	92.6062	92.8734	93.3663	93.5002	93.6269	93.6796	93.7303
backwardsINV	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705
forwardsINV	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705
markovINV	00.0000	00.0000	78.7177	77.5658	77.5620	77.4169	78.1832	78.7335	79.6828	82.9703	84.0141	87.1884	91.2133	91.9671	92.6237	92.8843	93.3769	93.5103	93.6367	93.6887	93.7389
biasbitINV	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705	83.2705
correlatedINV	77.6601	80.7426	82.4548	82.9105	84.1133	84.7563	85.0636	86.1415	87.1670	88.8024	88.7250	89.8201	89.9049	90.1734	90.2074	90.3563	90.4282	90.6526	90.6976	90.7067	90.7941
bimode	55.8111	64.4524	71.9158	77.3369	82.6073	87.0564	89.0636	90.2391	90.3951	90.9611	92.3512	92.7523	93.0275	93.1488	93.5884	93.6468	93.6941	93.7097	93.7468	93.7781	93.8372
agree	80.5150	81.7587	83.6741	85.5393	87.9026	89.3537	89.5517	90.2044	91.6975	92.4767	92.8196	92.9792	93.5013	93.6268	93.6649	93.6953	93.7447	93.7858	93.8531	93.8995	93.9540
cascaded2INV	76.0160	74.9432	78.0754	82.3712	83.0421	83.9568	86.1686	86.7982	88.4353	91.3616	92.0042	92.5640	92.8002	93.2800	93.4190	93.5195	93.5883	93.6251	93.6692	93.6989	93.7573

Average	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
mcf_localINV	85.9206	85.0225	85.6168	88.5094	90.2406	91.0450	91.5981	91.9445	92.1489	92.3397	92.4227	92.5778	92.6881	93.0563	93.2145	93.1937	93.1361	93.0722	92.9926	92.9038	92.8012
mcf_gselectINV	86.6542	88.3040	89.3653	90.3236	90.7957	91.3272	91.6126	92.0276	92.5345	92.8476	93.1119	93.2160	93.9252	93.9702	94.0380	94.0779	94.1024	94.1271	94.1367	94.1644	94.1785
biasbit	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867
agreeINV	89.3955	89.9111	90.5929	90.9706	91.5623	91.9345	92.1079	92.4163	92.7697	93.0555	93.2435	93.3324	94.0573	94.1158	94.1740	94.1945	94.2272	94.2545	94.2731	94.3066	94.3243
saturating	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440	66.8440
mcf_lgshareINV	84.9636	86.2663	86.8626	88.0736	89.9411	91.1946	91.9398	92.4403	92.8754	93.0231	93.1526	93.3460	93.5369	93.5928	94.1514	94.2248	94.2603	94.2814	94.3035	94.3283	94.3497
forwards	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643	28.0643
mcf_bimodal	76.4755	80.5623	85.3620	87.2265	87.8097	89.4616	89.7467	89.8706	89.8892	89.9248	89.9339	89.9382	89.9452	89.9452	89.9452	89.9452	89.9452	89.9452	89.9452	89.9452	89.9452
mcf_bimodalINV	86.0273	87.6182	89.3736	90.1308	90.4250	90.5923	90.6762	90.7271	90.7458	90.7698	90.7764	90.7750	90.7781	90.7781	90.7781	90.7781	90.7781	90.7781	90.7781	90.7781	90.7781
mcf_global	79.2236	82.5123	85.6461	87.2722	88.3970	89.5989	90.4147	91.3811	92.2254	92.5560	92.8408	92.9382	93.6476	93.7164	93.7516	93.7249	93.7255	93.7002	93.6888	93.5965	93.5692
mcf_globalINV	85.2786	86.0329	86.8439	88.0462	88.9021	89.7993	90.5559	91.4691	92.2903	92.6110	92.8968	92.9943	93.7099	93.7791	93.8089	93.7802	93.7811	93.7373	93.7207	93.6293	93.6073
saturatingINV	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671	84.7671
cascadedINV	86.6652	87.5584	88.4042	89.5727	89.9827	90.8082	91.1607	91.5559	91.9015	92.4242	92.7622	93.0045	93.1302	93.8736	93.9234	93.9892	94.0106	94.0374	94.0592	94.0815	94.1066
correlated	79.7513	86.5164	88.3081	88.9700	89.8663	90.3169	90.6938	90.8313	91.2108	91.7800	91.8916	91.9471	91.9941	92.4152	92.4606	92.5374	92.5527	92.7318	92.7588	92.7735	92.8159
all	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663	63.6663
mcf_gshareINV	87.5584	88.4042	89.5727	89.9827	90.8082	91.1607	91.5559	91.9015	92.4242	92.7622	93.0045	93.1302	93.8736	93.9234	93.9892	94.0106	94.0374	94.0592	94.0815	94.1066	94.1218
mcf_gshare	82.4856	85.7894	87.7597	88.9896	90.1719	90.7510	91.3855	91.8344	92.3776	92.7238	92.9671	93.0941	93.8364	93.8869	93.9579	93.9787	94.0059	94.0277	94.0500	94.0751	94.0904
bimodeINV	83.6578	85.5811	87.6038	89.2047	90.1921	90.9978	91.7108	92.0996	92.3311	92.5761	92.9097	93.1042	93.2655	93.3350	94.0439	94.0896	94.1487	94.1632	94.1895	94.2146	94.2281
mcf_gselect	80.9430	84.6235	87.4264	89.2887	90.0849	90.9081	91.3480	91.8957	92.4720	92.7952	93.0567	93.1602	93.8763	93.9294	93.9977	94.0447	94.0699	94.0945	94.1047	94.1328	94.1468
allINV	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867
cascaded	74.4838	82.4856	85.7894	87.7597	88.9896	90.1719	90.7510	91.3855	91.8344	92.3776	92.7238	92.9671	93.0941	93.8364	93.8869	93.9579	93.9787	94.0059	94.0277	94.0500	94.0751
backwards	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357	71.9357
mcf_lgshare	65.8841	77.3192	84.4998	86.8143	89.4140	90.9426	91.8439	92.3918	92.8550	93.0043	93.1363	93.3302	93.5166	93.5763	94.1363	94.2083	94.2452	94.2680	94.2894	94.3131	94.3347
cascaded2	71.7842	80.9430	84.6235	87.4264	89.2887	90.0849	90.9081	91.3480	91.8957	92.4720	92.7952	93.0567	93.1602	93.8763	93.9294	93.9977	94.0447	94.0699	94.0945	94.1047	94.1328
mcf_local	75.1933	82.2155	84.4913	88.0172	89.9202	90.8649	91.4362	91.9107	92.1283	92.3225	92.3134	92.4740	92.6549	92.8340	93.1531	93.1211	93.0558	92.9799	92.8901	92.7915	92.6768
comb	90.9173	91.3116	91.7429	92.1669	92.4395	92.7505	93.0845	93.2811	93.4142	93.5849	93.7498	93.8298	94.3887	94.4334	94.5259	94.5588	94.5835	94.6070	94.6185	94.6540	94.6694
markov	0.0000	0.0000	70.1381	69.8547	78.5168	82.6024	86.3641	87.5333	88.4622	89.5850	90.2944	91.2792	92.2460	92.6084	92.8955	93.0058	93.7242	93.8116	93.8614	93.9082	93.9642
backwardsINV	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867
forwardsINV	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867
markovINV	0.0000	0.0000	85.3572	85.9616	86.4885	87.3833	87.9013	88.4251	89.0776	89.8125	90.4408	91.3906	92.3276	92.6743	92.9619	93.0676	93.7862	93.8737	93.9225	93.9681	94.0150
biasbitINV	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867	89.4867
correlatedINV	87.4518	89.1370	89.6576	89.9887	90.5171	90.7965	90.9940	91.2957	91.6845	91.9317	92.0087	92.0589	92.1057	92.5260	92.5694	92.7741	92.7984	92.8469	92.8650	92.8766	92.9240
bimode	66.8923	79.0117	84.4848	87.8724	89.5504	90.6512	91.4151	91.9856	92.2384	92.5256	92.8879	93.0854	93.2440	93.3153	94.0238	94.0697	94.1291	94.1448	94.1729	94.1979	94.2114
agree	89.0600	89.5641	90.2795	90.7960	91.3893	91.7944	92.0362	92.3862	92.7513	93.0370	93.2209	93.3110	94.0354	94.0945	94.1587	94.1794	94.2132	94.2404	94.2591	94.2926	94.3103
cascaded2INV	86.2683	86.6542	88.3040	89.3653	90.3236	90.7957	91.3272	91.6126	92.0276	92.5345	92.8476	93.1119	93.2160	93.9252	93.9702	94.0380	94.0779	94.1024	94.1271	94.1367	94.1644