Reverse Engineering Branch Predictors

Don Browne

April, 2012

B.A. (Mod.) Computer Science

Final Year Project

Supervisor: Dr. David Gregg

School of Computer Science and Statistics

Trinity College Dublin, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

Don Browne, 30th April, 2012
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Don Browne, 30th April, 2012
Acknowledgments

I’d like to thank my supervisor, Dr. David Gregg, for his assistance, advice and encouragement throughout this project.

I’d like to think my friends, family and classmates for their support, and helping me stay (somewhat) sane throughout my final year.
Abstract

Modern CPUs are heavily reliant on branch prediction techniques to alleviate the performance penalties associated with waiting for a branch instruction to propagate through the pipeline. While extensive work has gone into the design of highly efficient and accurate branch prediction techniques, the full details are not disclosed by the designers of CPUs. Reverse engineering of the branch prediction structures allows for code optimizations to mitigate branch-related performance penalties. This project will build on the work of others in the field, and seeks to adapt prior experiments, and devise new ones, in order to examine the branch prediction structures of three modern Intel CPUs.
Contents

1 Introduction 1

2 Methods of Branch Prediction 4

3 Experimental Methodology 14

4 Reverse Engineering the Intel Atom 20

5 Reverse Engineering the Core 2 Duo 29

6 Reverse Engineering the Sandy Bridge 44

7 Conclusions and Future Work 48

A Appendix - Generalized Outcome Predictor Experiments 54
1 Introduction

Branches are a type of processor instruction that are responsible for flow control in a program by changing the current Program Counter/Instruction Pointer to a desired address. Branch instructions are very common in program code of all kinds; on average, they make up 10-20% of a program’s instructions.

Modern CPUs are heavily reliant on instruction level parallelism to boost performance.[4] The most common example of this is the technique of pipelining. This is where the execution stages of the CPU are divided into separate stages so that different instructions can occupy each execution stage at once. A pipeline based on Hennessy and Patterson’s DLX architecture is shown below -

<table>
<thead>
<tr>
<th>Fetch Instruction From Memory</th>
<th>Decode Instruction and read from registers</th>
<th>Execute Instruction</th>
<th>Read/Write to Memory</th>
<th>Store Results to registers</th>
</tr>
</thead>
</table>

Furthermore, many modern processors are said to be super scalar, where there are multiple execution units to allow multiple instructions to be fetched and executed at once, and out of order CPUs, which can rearrange the order of instructions fetched by the CPU to make the most use of the super-scalar capability.

A pipeline increases the throughput of the CPU provided that it can be kept full. In certain cases, such as a memory access, or waiting for a result from an instruction further along in the pipeline, an instruction can stay in a single pipeline stage longer than is expected, meaning that instructions in the stages behind it have to wait in place until the offending instruction is ready to move on to the next stage. This is called a pipeline stall, and can have a serious impact on performance depending on how long it takes to resolve.

Branch instructions are problematic for pipelines, as the target of the branch instruction is not known till the branch has propagated its way through the pipeline, and thus the pipeline must stall until the CPU knows where to fetch the next instructions. Due to the prevalence of branch instructions, this caused a serious obstacle for effective instruction level parallelism in regular code which lead some to think it was unfeasible.
In order to compensate for this problem, CPUs are equipped with mechanisms to allow them to guess whether a branch is taken or not, and where it branches to, so the CPU can execute this predicted path while waiting for a definitive evaluation of the branch instruction. This is called Branch Prediction. However, in the case where the CPU engages in speculative execution based on incorrect assumption about the behaviour of a branch, the CPU will end up executing the wrong instructions while the branch propagates through the pipeline. When this is discovered, the CPU will have to flush its pipeline before proceeding to execute the instructions from the correct location. The so-called misprediction penalty for a CPU is proportional to the length of the pipeline, and is often in the region of 15-20 clock cycles for most x86 CPUs designed in the past decade[7]. Consequently, the ability of a branch predictor to provide accurate predictions has a significant effect on the performance of a CPU.

Due to the impact of branch predictors on the performance of the CPU, there has been a considerable amount of research, in both academia and industry, in providing increasingly complicated and elaborate branch prediction units to handle different types of branches. However, the manufacturers of CPUs generally do not provide specific details of the branch prediction units in their processors, quite possibly to prevent competitors from copying these crucial aspects of successful CPU design. If details of the branch predictor structures were properly known, it would allow programmers to optimize their code to minimize the number of branch mispredictions that occur, or allow the designers of compilers to implement CPU-specific optimizations to generate object code that produced minimal branch mispredictions. Some research based on exposed branch predictor details have shown performance increases, and algorithms for optimal code generation.[17, 13]

Research has been carried out into the reverse engineering of branch prediction units, in order to determine their structure and design. Milenkovic et al. devised a series of benchmarks with precise branch behaviour, given parameters such as branch type, number of of branches, distance between branches and branch outcomes[3]. Using performance counting registers that are built into most modern CPUs, Milenkovic ran these tests on the Pentium 3 and Pentium 4, establishing properties of various aspects of the branch predictor organization. Further papers have built upon the work of Milekovic, including Ramadoss[6], and Uzelac[1, 2], who carried out a thorough reverse engineering of the Intel Pentium M processor.
In this project, we aim to reverse engineer more recent Intel CPUs. We will investigate branch prediction techniques, and the benchmarks that have been described by Milenkovic and Uzelac, changing them, or devising our own as needed. We hope to develop a series of benchmarks and experimental methodologies that can be applied generally. In carrying out the experiments across several CPU types, we will get the opportunity to see the experiments applied to different branch prediction configurations, and provide as general a reverse engineering as possible.
2 Methods of Branch Prediction

In order to carry out reverse engineering work on branch predictors, and interpret the results, we need to be familiar with the sort of prediction that CPUs are capable of, and how they may be implemented.

When discussing branch instructions, it is necessary to be familiar with the following classifications that can be applied to them -

- **Unconditional Branches** – these are branches that are always executed.

- **Conditional Branches** – these are branches that are only executed if some condition holds true. Determining whether or not the branch is taken generally occurs in one of the later stages of the pipeline.

- **Direct Branches** – these are branches whose target is known at compile/assemble time, and does not change. The target can be determined during the instruction decode stage of the pipeline.

- **Indirect Branches** – these are branches whose target is calculated at runtime, and consequently, successive executions of the branch may yield different targets. The target is pulled from memory, from a register, or calculated using the contents of memory and/or a register, meaning that the target cannot be determined till the later execution stages of the pipeline.

When discussing Indirect Branches, we can introduce classifications for their behaviour [5]-

- **Monotonic** – these are indirect branches that generally jump to the same target every time.

- **Data-driven** – these are indirect branches that regularly change target, but the change is correlated with the behavior of other branches.

- **Non-correlated** – the indirect branch changes target in a manner that is not correlated with any other branch behaviour.
When we discuss branch prediction, we are referring to the ability to offer two types of prediction -

- Target Prediction – where is this branch jumping to?
- Outcome Prediction – is this branch taken or not? (in the case of conditional branches)

Branch Prediction techniques can be classified into two main categories -

- Static prediction – the branch is predicted according to certain rules. These rules may be specified by the programmer using special compiler or assembler directives, or they may be hard-wired rules built into the processor[10], e.g. Assuming that a backwards-jumping branch is taken (as it is likely implementing a loop.) The details of static prediction are usually well documented by manufacturers. As the same rules are constantly followed, static prediction is only really useful when encountering a branch for the first time, where there is no dynamic prediction information.

- Dynamic prediction – the CPU uses various cache structures to keep track of the behaviour of the branches that it has encountered. When a branch is encountered, its target and outcome are predicted according to either the prior behaviour of the particular branch, the prior behaviour of other branches, or a combination thereof.

Due to the prevalence of cache-like structures in branch prediction, it is important to remember the different designs of cache that one might encounter, how they work, and their relative advantages and disadvantages -

- Fully Associative – when an item is inserted into a fully associative cache, it is inserted into the next free space. When the cache is looked up for a given item, the cache is searched for a matching entry. If an item is inserted into a full cache, an item is ejected based on a replacement policy such as Least Recently Used. The space in a fully associative cache is always used optimally, but the process of inserting, looking up, and evicting an item are slow, and require relatively complicated circuitry so they are generally not used.
• Direct Mapped – in a direct mapped cache, a portion of the item’s address is used an index to the cache. Consequently, the item is always placed in the same place, which makes insertion and look-ups very quick. However, multiple items may map to the same index, resulting in the eviction of the previous item, even if there is free space elsewhere in the cache. Consequently, there is a trade off between speed and simplicity against the efficiency at which free space is used.

• Set Associative – a combination of the two previous schemes. Instead of using a flat table, each entry maps to a set of spaces. A portion of the item’s address is used to determine which set the item should be inserted into. The set works like a fully associative cache, so the item fits into the next free space in the set. An additional portion of the item’s address, called the tag, is used to identify that item within the set. A set associative cache whose sets hold n entries are usually described as n-way caches.

Generally, in the case of a set associative cache, the lower address bits are used as the set index, and the higher bits are used as the tag.

Outcome prediction attempts to guess whether a branch is taken or not, allowing the CPU to speculatively execute code from the predicted path until a definitive outcome can be determined. The most basic method of outcome prediction involves taking a portion of the branch address, and using it to look up a table of 1-bit counters. When the counter is equal to 1, the branch is predicted as taken, and likewise, when equal to 0, the branch is predicted not taken. When a misprediction is detected, the bit is flipped. This scheme is simple to implement, but is too basic to be useful. Consider a repeatedly executed branch that consistently behaves with the pattern \{Taken, Taken, Taken, Not Taken\} Irrespective of the default initial behaviour, the initial branch execution will set the one bit counter to 1, and the 2nd and 3rd executions will be correctly predicted. The fourth execution will be mispredicted as being taken, and the bit will be flipped. On the next iteration of this pattern, the 1st execution will be mispredicted as not taken due to the effect of the 4th execution. Consequently, for every four executions of this branch, we will expect to see a branch misprediction rate of 50%. This example illustrates the two failures of this scheme:
• It is too sensitive to divergent behaviour – there is one misprediction for when the branch changes behaviour, and another when it returns to the regular behaviour.

• It can only keep track of how the branch behaved the last time, and assume it will behave the same way the next time. It cannot offer any prediction as to when a branch’s behaviour may change, even for a simple pattern as depicted above.

As such, the 1-bit counter by itself is only effective when dealing with a branch that is either never taken, or always taken.

To address some of the failings of the 1-bit counter, a 2-bit counter is generally used in most branch prediction schemes. The 2-bit scheme is generally called a Saturation Counter, or a Bi-modal Predictor. The 2 bits are used to implement a finite state machine with four states – Strongly Taken, Weakly Taken, Weakly Not Taken and Strongly Not Taken. The most significant bit provides a prediction. When a branch is taken, the counter is incremented by 1, and when not taken, the counter is decremented by 1.

The advantage of this scheme over the 1-bit counter is that, from one of the strong states, the branch has to diverge from expected behaviour twice before it changes the prediction offered by the saturation counter. In the example given for the 1-bit counter, the bi-modal predictor will mispredict the 4th execution, but will not mispredict the 1st execution of the next iteration. This will mean that pattern will only have a 25% misprediction rate compared with 50%.

The bi-modal predictor still suffers the other failing of the 1-bit counter – it cannot predict changes in behaviour of the branch. In fact, the bi-modal predictor is worse than the 1-bit counter in situations where the branch behaviour changes, and remains changed, as it takes longer to change the prediction offered. Consider the pattern \{Taken, Taken, Taken, Not Taken, Not Taken, Not Taken\}
If the bi-modal predictor starts at the strongly taken state, it will correctly predict the first three executions of the branch, but then mispredict the fourth and fifth while it transitions from strongly taken to weakly not taken. The sixth execution will transition to strongly not taken, meaning that on the next iteration through the pattern, the first and second executions will be mispredicted while it transitions back to a taken state. Consequently, for every six executions of this branch, we’d expect to see four mispredictions, instead of the two we’d expect if a one bit counter was used.

In order to get around this problem, the branch predictor needs to be able to keep track of a pattern of execution. Most pattern based outcome predictions are based around a mechanism referred to as a Two-Level Adaptive Predictor. In this scheme, the table of bi-modal predictors is augmented with a history register. The register stores the details of prior executed branches – either storing what their outcomes were, or storing a portion of their address. The table of bi-modal predictors is then looked up either the history register, or a combination of the history register and the branch address. There are two main types of two-level predictor schemes -

- **Global**: A single history register is updated with information relating to all executed conditional branches.
- **Local**: There are a number of history registers. When a branch is executed, a history register is selected using the branch’s address, and is updated accordingly.

Global and Local schemes may or may not use multiple tables of bimodal predictors, which are selected through a portion of the branch address, but this practice seems to have fallen out of use, possibly due to the additional latency introduced in selecting a table, and the fact that some commonly used tables may have capacity clashes while other tables lay vacant.

Local predictors aim (given capacity issues) to provide pattern-based prediction for individual branches, and are good at handling branches whose behavior regularly diverges in a predictable manner, but obviously increases CPU complexity due to the need for multiple registers, and the need to select between them. Furthermore, there is often a correlation between the behaviour of different branches, which could potentially allow the CPU to predict changes that cannot be detected by looking at the prior behavior of the branch alone[12].
Consequently, most modern CPUs are based around global prediction. Most use what is known as a Gshare scheme, depicted below, which uses a single history register, and a single table of bimodal predictors. When a conditional branch is encountered, the address of the branch and the history register are combined, usually using an XOR-based hash function, and used to look up the history table.

Some sources encountered during the research suggest that Intel has moved back to 1 bit counters on newer processors, based on the claim that most bimodal counters stay in a strongly taken or weakly taken state, and consequently the largely redundant extra bit can be used to provide double the number of 1 bit counters.[16] This observation is likely related to the use of two-level predictors, where the particular outcomes of a given branch tend to end up with their own counter, instead of having one predictor counter for one branch.
In addition to these general purpose pattern-based predictors, there may also be a specialized loop predictor. This was a mechanism that originally appeared on the Intel Pentium M processor, and attempts to identify branches with loop-like behaviour – where it is taken a number of times, not taken once, then taken once again for the same amount of times. The implementation described by Intel consists of a counter which holds the current iteration number, and another holding the iteration number on which the branch diverges. This may be used in place of local prediction. The following is a diagram from an Intel article on the Pentium M.

The number of different outcome predictor designs, and their sophistication raises practical issues. Latency is a serious problem with branch predictor design, as if it takes too long to make a prediction, the number of saved clock cycles is diminished to the point where it is not worth the effort. Consequently, many modern CPU designs will use a combination of relatively inaccurate but fast predictors, and accurate but slower ones. One such scheme is the Hybrid Outcome Predictor, where a Gshare predictor and a simple bimodal table are used in unison, and some sort of chooser logic (often a bimodal predictor itself) is used to determine which structure to act on. This means that branches with consistent behaviour can be quickly predicted using the bimodal table, and branches with more complicated behaviour can be predicted by the Gshare predictor.
Another popular scheme is the BLG (Bimodal, Local, Global - the order in which the predictors are used) scheme. In this, the branch is predicted by quick but inaccurate predictors before moving on to increasingly accurate ones[8]. Each successively more accurate predictor can override the prediction of the prior structure. Each stage only stores information relating to branches that it was able to predict more accurately than the prior stage, meaning that the more accurate predictors do not get clogged up with branches that can be correctly predicted by less accurate structures.

The main unit of target prediction is the Branch Target Buffer (BTB), which may also be referred to as a Branch Target Array, or a Branch Target Address Cache. This, as depicted in the diagram below, is a cache that maps a portion of the branch address to the target that it was last taken to.
Usually allocation only happens when the branch is actually taken, meaning that never taken branches do not get an entry in the cache. The cache may also contain information in regards to what sort of branch it is, in order to determine if specialized predictors such as the loop or indirect predictor are to be consulted. The Branch Target Buffer is usually consulted at the instruction fetch stage, so it is possible for it to erroneously believe that a non branch instruction is a branch instruction if it happens to map to an entry on the Branch Target Buffer.

A Branch Target Buffer may be work in conjunction with the the Outcome Predictor, so that the Outcome Predictor only predicts branches that have been encountered before and static prediction is used otherwise. Modern CPUs often dispense with this mechanism and simply use the outcome predictor to predict all branches, as determining whether or not a branch needs static prediction can take so long that it can end up being quicker to have the outcome predictor guess randomly, and take the hit if its wrong.

The Intel Nehalem processor reputedly had a two level BTB[7] – a smaller but faster one for an initial prediction, and a slower but larger (and therefore likely more accurate) one to provide a later prediction[9]. There is some indication that the Sandy Bridge may use a table with variable target lengths, so that branches to nearby targets do not occupy as much space as a far jump, meaning that more smaller jumps can be recorded into the BTB.[16]

The Branch Target Buffer operates under the assumption that a branch will jump to the same target every time it is executed.
This is a reasonable assumption in most cases, but falls down when dealing with indirect branches that change target regularly. Consequently, many modern CPU designs incorporate a special Indirect BTB to handle indirect branches whose target changes in a manner that is correlated with the behaviour of other branches. To do this, the address of the indirect branch is used in conjunction with the outcome predictor’s history table to look up a dedicated cache of target addresses\[15\]. Depending on the number of branch patterns preceding the indirect branch, it will have multiple entries, and thus pattern-based target changing can be correctly predicted by the CPU.
3 Experimental Methodology

In reverse engineering the branch prediction units in the selected processors, we set out to answer the following questions -

- What sort of prediction structures does the CPU employ? How do they work in conjunction with each other?
- In regards to the caches and tables used in the CPU – what is their capacity? How many sets and ways do they have? Which address bits are used to look up the table? When do entries in the table get evicted?
- In regards to the history registers – how long are they? How are they updated, and what are they updated with? What sort of hashing function is used if they are combined with branch addresses as part of a look up? What sort of branches affect them?

In order to test these properties, we write a series of benchmarks in C with inline assembly. The assembly is used to give very precise control over the sort of branches used, and to insert NOP instructions to control distances between branch instructions, and to pad instructions so that they are located on memory addresses of our choosing. Some benchmarks do not use assembly, and indeed, a lot of useful information can be determined using only C. The code was compiled using gcc, with the -O0 flag passed to turn off optimizations. Even with this, gcc’s compilation and optimization techniques still interfered with the generated code. Two major issues that cropped up again and again were as follows -

Firstly, the compiler would attempt to do its own address padding with NOPs, which would interfere with the padding that was implemented in the benchmark. Often times the padding that gcc would increase as the amount of padding in the benchmarks was increased. Consequently, if two branches were separated by a block of NOPs such that the distance between them was $2^n$ bytes, it wouldn’t necessarily be correct to just add $2^n$ more NOPs to increase the distance to $2^{n+1}$ bytes.

Secondly, in order to simplify the control flow of the program, gcc avoids generating code with multiple indirect branches, instead opting to generate one indirect branch, and have other places that used indirect branches do a direct jump to the original indirect jump. For example, the following code -
It would end up as something to the effect of -

```
goto *T1;
goto *T2;
```

```
mov (T1), %rax
L1: jmp *%rax
mov (T2), %rax
jmp L1
```

It is interesting to note that from a branch prediction point of view, this behaviour is very problematic, as it increases the number of possible targets for the indirect branch without necessarily guaranteeing unique branch information for the new paths. Experimentation has shown that the Sandy Bridge may have solved this issue in the practical context by considering the addresses of direct branches as part of the indirect prediction process, however, in order for our experiments to work, it has to be solved, usually by inserting a goto, examining the generated assembly code used to load the target into the register, and then replacing the goto with inline assembly that did an indirect jump to that register’s contents.

It became absolutely vital to make regular use of a disassembly tool in order to ensure that the code was generated as expected. The objdump disassembler was used. This is a command line tool that when given an executable, will show the machine code listings, the matching assembly code, and the addresses on which the instructions are aligned. A GUI front end named dissy is available, however, using the command line tool in conjunction with grep and less proved to be the most efficient way of finding out the desired details, especially in programs where large numbers of NOPs were inserted.

The programs developed used a large number of branches (sometimes hundreds or thousands) and an even larger number of NOPs (into the tens of millions in some tests). Furthermore, the number of these instructions needed to be changed in order to carry out the experiments. For these benchmarks, automated code generation was required. Two techniques were used:
Python scripts – used to generate entire C and Assembly benchmarks, compile them, and run them.

Assembler Directives – the work of Milenkovic and Uzelac seemed to imply that the programmer should use inline assembly to insert the NOPs. In practice, even with code generation scripts, this proved to be unworkable as A) the generated code couldn’t be read or modified without considerable hassle and B) every compiler tried (including gcc, clang, icc and MSVC) would throw an error after several thousand NOPs were inlined directly into code. The solution found was to use the .rept directive of GAS. This allowed a large number of NOPs to be specified in a concise, easy to edit form, and furthermore did not create errors. Assembler errors were encountered with benchmarks where .rept was used to generate in excess of $2^{22}$ NOPs. The solution found was to spread the NOPs across a number of .rept directives. Programs with more than $2^{24}$ NOPs were successfully generated in this manner.

In order to measure the outcome of these experiments, we use the hardware-based performance counter registers that are present inside most modern CPUs. These registers can count a number of hardwired events. These include clock cycles, data/instruction cache misses, instructions retired, but those of us relevance in this context are ones concerning the numbers of branches executed, and the how many were mispredicted. The branch related events included general-purpose counts of the number of mispredicted branches of any kind, down to specific events such as mispredicted indirect or conditional branches, bogus branches, or branches that were mispredicted at the decode stage. Issues that arose with these hardware events included -

They are subject to inaccuracy, and will not work properly unless the program runs for a sufficiently long time (an interrupt needs to be triggered before the counting takes places). Consequently, the benchmark is run inside a loop for a large number of times, usually a million. During experimentation, misprediction counts in the thousands, or low ten-thousands were found to correspond with negligible misprediction rates, likely being contributed by monitoring overhead, or the initial period required to train the predictor to correctly predict the branches in the program. Consequently, we design the benchmarks so that the number of expected predictions is large, as small numbers of mispredictions (less than about
100,000) may vary too much from run to run to provide meaningful results. Even still, successive tests of the same program may yield different results, so it is necessary to run the process a few times to determine what the most consistent result is.

- Different CPU microarchitectures will have a different set of events available, and seemingly identical events may work differently on different CPU microarchitectures. If a benchmark is dependent on a particular event, it may need to be modified to work on another CPU.

- The large number of NOP instructions in some of the benchmarks was shown during experimentation to cause severe interference with the accuracy of the event counting (in some cases, it would cause the event monitor program to report a number of mispredicted branches that was several orders of magnitude higher than the total number of branches in the program). In order to avoid this, the benchmarks have to be designed in such a way that large NOP blocks are not executed, usually by jumping over them. This had to be designed in such a way that any jumping over of the NOP blocks did not interfere with the branch behaviour that was being measured.

The program likwid was used to benchmark the program using the performance registers. The likwid-perfctr command is used to specify the CPU/CPU core to be monitored, the program to be run, the events to be monitored, and which performance counter register to use. It can also be used to display the set of events that the CPU can monitor. In order to eliminate all potential issues involving threading and multiple cores, the systems tested had Hyper Threading and all but one CPU core disabled. Other possible measurement tools are Intel’s vTune, and the open source tool PAPI. vTune was tried, but was found to be inaccurate, and failed to give consistent or expected results. PAPI involved inserting function calls into the code, and thus increased the complexity of the benchmark design.

The CPUs tested were:

- Intel Core 2 Duo (Conroe)
- Intel Atom (Bonnell)
- Intel Core i7 2nd-gen (Sandy Bridge)
Various tests are required to determine various properties of different branch prediction structures. Some commonly occurring methods are as follows -

• Address testing – writing a program with two branches with different behaviour. Using NOP instructions, the distance between them is controlled such that their addresses only differ by a single bit (if we want them to differ only by bit k, first we insert NOPs before the first branch so that the bit k of its address is set to zero, then we adjust the distance between the two branches to be $2^k$, which will set bit k in the 2nd branch to one). We test each bit starting from the lowest and proceeding upwards until they appear to be the same branch to the branch predictor, and consequently cause mispredictions to occur. Once we establish what this bit $n$ is, we note that bit $n-1$ is the most significant bit of the branch address used by the predictor structure for this prediction type. With the two branches still separated such that they only differ by bit n, we then adjust the distance so that it also differs by a second bit $m$, starting from the lowest bits and working up. When the misprediction rate drops, we note that bit $m$ is the lowest bit used by the branch prediction structure.

• Capacity and Set testing – creating a program with a large number of a certain type of branch in order to determine how many branches the appropriate predictor can hold. Once the number of branches exceeds the capacity, we will start to see mispredictions. Set testing is somewhat similar, except we use a smaller number of branches, widely spaced out, in order to try and insert multiple items to the same cache. Once we exceed a certain number of items, items in the set will be evicted and we can infer the the capacity of the set from this.

• Pattern testing – in order to determine the behaviour and properties of structures that use a history register, we need to control the contents of the history register. Usually some initial tests will establish the capacity of the register, and the types of branches that affect it, meaning that we can write code that will set it to what we want. In many cases, this is used to clear the register of any useful information so that we can prevent it interfering with experiments. In other cases, it is used to create different patterns that can be used to determine the properties of predictor structures that use it.
In the benchmarks, we will refer to “Spy Branches” which are the branches whose misprediction rate we attempt to control and monitor through the structure of the benchmark. There are also “Setup Branches” whose misprediction rate is not of interest, but serve to control the misprediction rate of the Spy benchmarks in one way or another. In some cases, the Setup branches have misprediction rates, sometimes foreseeable during the design of the experiment, and not in others. It is good practice to run the benchmark with the spy benchmarks commented out in order to determine what background mispredictions are occurring, and either take action to remove them, or take them into account if it is not possible to remove them.
4 Reverse Engineering the Intel Atom

We start the reverse engineering process of the Atom CPU by determining the type of outcome prediction that it makes use of. We follow the outcome predictor experiment flow detailed in the appendix. The first test is to determine the maximum length of pattern that the Atom can successfully predict. Results are shown below.

<table>
<thead>
<tr>
<th>Pattern Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>12.6%</td>
<td>11%</td>
<td>9.8%</td>
<td>9.1%</td>
<td>8%</td>
</tr>
</tbody>
</table>

According to our results, the Atom is capable of predicting up to a pattern of \{6x Taken, 1 x Not Taken\}. At this point we attempt to determine if this prediction has been offered by global prediction, or some form of local prediction by inserting an always taken conditional branch before the spy branch when the pattern length is seven. The produced misprediction rate is the same when the pattern length is 8. We conclude that the Atom is using global prediction, which can keep track of the details of the last 12 branches. We then attempt to determine if the Atom has any local prediction in addition to the global predictor, by inserting 12 always taken branches before the spy branch, and testing the patterns of length 2-7. The results of the experiment, as detailed in the appendix, are given below.

<table>
<thead>
<tr>
<th>Pattern Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>80%</td>
<td>35%</td>
<td>26%</td>
<td>21%</td>
<td>17%</td>
<td>14%</td>
</tr>
</tbody>
</table>

Our conclusion is that the Atom relies solely on Global Prediction to provide outcome prediction. We now modify the experiment to determine which sort of branches affect the history register. We draw the following conclusions -

- The history register is affected by taken, and not taken conditional branches.
- The history register is not affect by calls, returns, unconditional or indirect branches.
The reverse engineering of the Pentium M carried out by Uzelac showed that the Pentium M’s history register is updated a portion of the address of taken conditional and indirect branches. The behaviour described above indicates that the Atom is using a different scheme, quite possibly one that is updated with the outcomes of conditional branches only, and not addresses (and that it more than likely does not have an indirect predictor, which will be investigated later).

In order to verify this hypothesis, we adapt one of Uzelac’s global predictor experiments. The spy branch follows a pattern \{16x Taken, 16x Not Taken\}. The expected misprediction rate for such a pattern if it solely relies on bimodal prediction is one-eighth (2 mispredictions when it transitions from strongly taken to weakly not taken and 2 mispredictions when it transitions from strongly not taken to weakly taken, out of a total of 32 executions per pattern iteration) We provide two separate paths leading up to the spy branch, one is always traversed through when the branch is taken, the other when the branch is not taken. The outcomes of the branches are constant for both paths – always taken. The idea is that if the addresses of the conditional branches affect the history register, the CPU will be able to correctly predict the outcome of the spy branch based on the correlation between the addresses of the always taken branches, and the outcome of the spy branch. If the CPU relies solely on branch outcome, we will see a misprediction rate of one-eighth for the spy branch (the setup branch used to determine which path is taken will suffer this misprediction rate, so we will expect to see approximately 125,000 mispredictions if the addresses affect the history register, and approximately 250,000 if only outcomes are considered).

The program is ran, measuring mispredictions. A result of approximately 260,000 mispredictions is yielded each time, suggesting that the paths have had no effect on the history register. Changing one of the paths so that any one of the always taken branches is always not taken will halve the misprediction rate. This suggests that the Atom has a 12-bit register which stores the outcomes of the last 12 encountered conditional branches.

We then attempt to determine the properties of the global predictor table. As the table is based solely on outcome information, we have to devise a new benchmark.
int i = 1000000, a=0;
do
{
(12x) if (a==0) a=1; //Repeat 12 times, used to clear history register
if (a==0) a=1; //Always taken spy branch
(11x) if (a==0) a=1;
//Carry out padding inside taken branch so CPU doesn’t execute NOPs
if (a==0)
{
asm(".rept 0x8000"); //Control distance between spy branches
asm("nop");
asm(".endr");
}
if(a==1) a=1; //Always not taken spy branch
i--;
}

Two spy branches, one always taken, the other never taken, are preceded with 12 always taken branches so that their lookup behaviour is solely affect by their address. As their behaviour never alters, they can be predicted with 100% accuracy using a bi-modal predictor. We then begin to adjust the distance between the two branches so that their addresses only differ by a single bit. It was discovered the setup branches were contributing to the misprediction rate, meaning that a misprediction rate of about 50% meant that the two spy branches were not clashing. The relevant portion of results are shown below:

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>50%</td>
<td>51%</td>
<td>75%</td>
<td>80%</td>
</tr>
</tbody>
</table>

When the addresses only differ by the bit 12, the misprediction rate goes high, indicating that the spy branches are mapping to the same address. Thus, we conclude that bit 11 is the most significant bit of the portion of the address lookup. We then set the addresses that so that they only differ by the 13th bit,
and then set the lower address bits to differ. Results are shown below -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>...</td>
<td>N/A</td>
</tr>
</tbody>
</table>

All the lower bits cause the mispredictions to disappear, implying that bits 11:0 of the branch address are used as an index to a table. Attempts to determine the exact hashing function proved inconclusive. As the tests based on Uzelac are designed for a CPU that uses address bits in the history register, it is possible in that instance to make two different paths appear to be the same by adjusting the distance between them. In a scheme that is purely outcome based, two different paths will always be different. Thus, if we wanted to carry out a capacity test for such a table, we’d need a benchmark that would create every permutation of 12 taken and not taken branches, in order to create each possible state for the history register. Which would prove difficult.

However, there are some suggestions found during research that the Atom may have a Gshare predictor with 4096 entries[17]. According to our results, this is a plausible claim. Given that 12 bits are used to index the table, it is possible that the 12 lowest address bits are used as a lookup to a direct mapped cache of bimodal predictors.

To test the presence of an indirect predictor, we test the misprediction rates for the following pieces of code. This code shows an implementation for two targets, but we test it for multiple amounts of targets.

```c
int mod, i=1000000;
void * targets[] = {&&T1,&&T2};
do{
    mod = i % 2;
goto * targets[mod]; //Spy indirect branch
T1: asm("clc");
T2: asm("clc");
i--
} while (i>0);
```

The results are shown below.
<table>
<thead>
<tr>
<th>No. of Targets</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

The Atom can predict an indirect branch that jumps to the same place, but not one whose behaviour changes, even if we provide a conditional branch whose behaviour is correlated to that of the indirect branch. We conclude that the Atom lacks a dedicated indirect predictor, and that it instead inserts the last target into the Branch Target Buffer, and assumes the target remains constant.

We then attempt to determining the organization of the Branch Target Buffer. Milenkovic devised an experiment for the Pentium III and Pentium 4 where a large number of always-taken spy branches, separated by a certain distance are used to stress the BTB. When the number of spy branches exceeds the capacity of the BTB, there is a high misprediction rate. When the number of spy branches equals the capacity of the BTB, we see that there are \( m \) different distances of separation between the spy branches which cause low mispredictions. This is because larger distances are causing the branches to be mapped to the same space in the set associative cache, meaning that the cache’s free space is not being fully used. Milenkovic notes that the degree of set associativity of the cache is equal to \( m + 1 \).

The above experiment depends on branches evicting an older branch straight away once a capacity clash occurs. Uzelac showed in his experiments that the Pentium M’s BTB’s replacement policy was designed in such a way that eviction may not happen on the first clash. In order to get around this, he revised Milenkovic’s so instead of executing all the spy branches once each in each loop iteration, he would execute a single one per test loop iteration, but executed them with the pattern \{1st, 1st, 2nd, 2nd, 3rd, 3rd,...\} In this way, each branch would be executed twice before proceeding on to the next branch.

When attempting to adapt this experiment to the Intel Atom, a problem was encountered. Both Uzelac and Milenkovic carried out their experiments on CPUs where it was possible to measure the number of mispredictions at the instruction decode stage. As pointed out during the introduction to branch prediction, target prediction is usually carried out during the instruction fetch stage of the pipeline, and a definitive outcome can be determined during the instruction decode stage. Experimentation has shown that the standard branch misprediction monitoring events concern themselves with mispredictions that happen later on in the pipeline, relating to outcome prediction, and the resolution of indirect
targets. This means that a direct branch whose outcome is predicted correctly, but whose target is incorrectly predicted will not generate a misprediction event.

However, we have shown previously that Indirect Branches are most likely handled by the Branch Target Buffer, and that the branch misprediction event monitor on the Atom can monitor the misprediction of indirect branch targets. Therefore, we devise our own BTB test using Indirect Branches to test the capacity of the BTB. In this test, we set up the following structure (shown for two branches, with a distance of 2 bytes between them)

```c
int i, mod;
void * targets[] = {&B0,&B1};
for (i = 0; i < 1000000; i++)
{
    mod = (i%4) >> 1; //Pattern of 0,0,1,1
    goto *targets[mod]; //Setup branch used to select spy branch
    B0: asm("lea t0, %eax"); //Note: If using 64-bit CPU, use register rax instead
        asm("jmp *%eax"); //Spy branch 1
        asm(".rept 0x2"); //Assembler macro used insert NOPs
        asm("nop");
        asm(".endr");
    B1: asm("lea t1, %eax");
        asm("jmp *%eax"); //Spy branch 2
        asm(".rept 0x2");
        asm("nop");
        asm(".endr");
    asm("t0: clc"); //Target for first spy branch
    asm("t1: clc"); //Target for second spy branch
}
```

To avoid encountering the issues Uzelac encountered, we executed each branch with the same double execution pattern as described above. A setup indirect branch is used to select which branch will be executed on a given iteration of the test loop. This branch will generate mispredictions itself, and thus we need to take them into account during our measurements. As the branch jumps to each target twice before changing target, we’d expect it to be mispredicted on every second iteration of the test loop, and consequently we’d expect a misprediction.
count of approximately 500,000 in the below implementation. A number of indirect branches that jump to unique targets are used as spy branches. NOPs to separate the branches by a given distance. The branch instruction and the load effective address instruction occupy two bytes of memory each, so we set the number of NOPs against this (however, as pointed out in Chapter 3, the unpredictability of gcc requires that some of the larger distance need to be adjusted by different amounts).

In Uzelac’s experiments where indirect branches jumped to different targets, he used an elaborate and lengthy piece of inline assembly to manually allocate a target stack, to which he pushed the addresses on to and pulled them out again, also using inline assembly. Here, we use the non-standard ability of gcc to have pointers to labels, and use a goto statement to jump to a target picked from the table using the pattern value mod. Uzelac used Visual Studio’s compiler, which does not support this construct (or the same sort of assembler directives that our experiments rely on) and consequently his solution was needed for his environment, but is unnecessary for ours.

When the number of spy branches exceeds the capacity, we would expect that once the BTB is initially filled, each subsequent branch will be mispredicted the first time it gets executed. Consequently, there will be one misprediction per every second iteration of the test loop, raising the expected misprediction count from 500,000 to 1,000,000. This behaviour will also occur when the distance between branches is large enough that they do not fill all available spaces in the BTB. Test results are shown for 64, 128 and 256 branches respectively:

<table>
<thead>
<tr>
<th>Inter-branch Distance</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-branch Distance</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-branch Distance</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

When 256 branches are used, the misprediction rate is high for all distances. When at 128, the misprediction rate is approximately 700,000 for three distances. This deviation from expected behavior can be explained by the fact that all the spy branches fit into the BTB, but the setup branch, and the loop
branch are clashing with some of the spy branches. This behaviour could be avoided by removing two spy branches, but then the loop and setup branches are not executed according to our desired pattern, which would affect the accuracy of our results. Furthermore, this behaviour reaffirms the hypothesis that the maximum capacity of the BTB is 128 entries. As there are three “fitting distances” for 128 entries, we conclude that the cache is a 4-way cache with 32 sets.

There are two further experiments based on the capacity test that we can use to glean more information, and verify the initial results. First, we take the above experiment with only two spy branches. We then use these to determine the address bits used in the lookup by adjusting the distance between the two branches in the same manner that we did with the outcome predictor experiments. The results are shown below, first for the most significant bit, and then for the least significant bit when the distance between the branches is high enough to cause them to clash to the same entry -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

We conclude that bits 12:4 are used to look up the BTB. The most likely configuration is that bits 8:4 are used as an index (as there are $2^5$ sets), and 12:9 are used as a tag.

We verify the results of the capacity test using what Uzelac and Milenkovic describe as the “set test”. This is based on the same test structure as the capacity test, except we use a smaller number of spy branches, and larger distance between them. The aim is to discover the number of branches needed to fill a set, and the distance between the branches so that they map into the same set. The results are shown below for inter-branch distances 128, 256, 512 bytes -

<table>
<thead>
<tr>
<th>Inter-branch distance</th>
<th>2</th>
<th>...</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.6</td>
<td>0.67</td>
<td>0.75</td>
<td>0.8</td>
<td>0.86</td>
<td>0.91</td>
<td>0.96</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-branch distance</th>
<th>2</th>
<th>...</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.67</td>
<td>0.8</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>Inter-branch distance</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mispredictions (millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the addresses differ only by bit 9, we are only able to hold four branches. This is explained by the earlier conclusion that we have a 4-way cache indexed by bits 8:4, and so when the distance is 512 bytes, the addresses only differ by bit 9, and thus they map to the same set. We conclude that our earlier results are valid.
5 Reverse Engineering the Core 2 Duo

We first seek to establish the behaviour of the outcome predictor. We get the following results for the pattern length test as described in the appendix.

<table>
<thead>
<tr>
<th>Pattern Length</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>63</th>
<th>64</th>
<th>65</th>
<th>66</th>
<th>...</th>
<th>127</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.5%</td>
<td>1.46%</td>
<td>...</td>
<td>0.81%</td>
<td>0.78%</td>
<td></td>
</tr>
</tbody>
</table>

We now wish to determine whether or not the pattern has been predicted by global prediction. To do so, we take the benchmark when the pattern length is 64, and insert an always taken branch before the spy branch. This does not change the misprediction rate, causing us to conclude that some sort of loop or local predictor is used.

We now wish to determine whether the pattern has been predicted by a loop predictor, or a local predictor. We carry out the test to distinguish between them as outlined in the appendix. We receive a misprediction rate of approximately 80%, indicating that the Core 2 uses a loop and global predictor, but no local predictor.

We establish the presence of the indirect predictor in the same way we attempted to establish it on the Atom. Results are shown below -

<table>
<thead>
<tr>
<th>Number of Targets</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

We conclude that the Core 2 has some means of indirect branch prediction. From Intel’s own description of their Indirect Branch Prediction scheme, we expect that the global predictor’s history register is used as part of the indirect predictor lookup. By inserting a number of conditional branches between the conditional branch that offers the correlation information, and the indirect branch, we determine the number of conditional branches that the history register can keep track of at once. Across this experiment, and others, we establish this figure to be 6-7, but we usually use 8 when we want to be certain that we are clearing out the register of any information that may interfere with our experimentation.

We proceed to reverse engineer the indirect prediction structure, using Uzelac’s benchmarks for the Pentium M as a starting point. It was found during the experimentation that the benchmarks described were sometimes rather ambiguous, incorrectly described (particularly the source code listings), and/or that the descriptions in different papers contradicted each other. [2, 1]
We first set out to determine the behaviour of the history register. Most of the tests are based on the structure below. The indirect spy branch has two targets, which it alternates between on every iteration of the test loop. There are two blocks of always taken conditional branches, and it is set up so that every time the indirect branch targets the first target, the program will run through the first block beforehand, and likewise for the second ones. A sufficiently large number of conditional branches are used so that the indirect predictor’s ability to predict the spy branch is based entirely on the addresses of the conditional branches. Running the below code with no distance between the blocks results in negligible indirect misprediction events.

```c
int a = 1;
int long unsigned mod, i = 1000000;
void *targets[] = {&&Target1,&&Target2};
do {
    mod = i % 2;
    if (mod==0) {
        /* 8 occurrences of if(a==0) a=1; */
        if (a==0) {  //Use this branch to jump over NOPs
            asm(".rept 0x7FF8C");
            asm("nop");
            asm(".endr");
        }
    } else {
        /* 7 occurrences of if(a==0) a=1; */
        //Use this branch to adjust the distance of the last branch
        if (a==0) {
            asm(".rept 0xC");
            asm("nop");
            asm(".endr");
        }
        //Adjust this branch’s address when misprediction
    }
}
```

30
//rate is high to determine lower bits
if (a==0) a=1;
}

/*/Insert always taken branches to create mispredictions here*/
goto *targets[mod]; //Spy indirect branch
Target1: asm("clc");
Target2: asm("clc");
i--;
}while (i>0);

We now determine the address bits of the conditional branches used to update the register by separating out the blocks, and then moving forward the last branch in the second block to determine the least significant bit. Results are shown below -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>N/A</td>
<td>N/A</td>
<td>0.9</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Our conclusion is that bits 18:4 of the conditional branch affect the history register. We then insert some always taken conditional branches between the path blocks, and the spy branch. These will affect both paths, and will shift some of the correlation information out of the register (assuming that the register is shifted before update). For every always taken conditional branch inserted after the path blocks and before the spy branch, two upper address bits of the last conditional branch in each block cease to affect the history register, causing mispredictions where there were none before.

<table>
<thead>
<tr>
<th>Different Address Bit (with 1 If)</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

We conclude that the register is shifted two places to the left on update, and that there is a total capacity of 15 bits in the register based on the size of the portion of the address used.

A slightly modified benchmark is used to determine the effect of the indirect branch and target address bits on the history register. We set the distance between them so that the conditional branches in the two blocks appear to be
the same to the be predictor. We then shift the target of the second block so that the misprediction rate decreases. Once the significant bits of the target are established, we move the targets away from their branches so that we can adjust the location of the branches in memory without affecting the target. Results are shown below, first for the target -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Then for the address bits -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Our conclusion is that bits 6:0 of the target, and 18:10 of the branch affect the register. This adds up to more than 15 bits, and so it seems likely that one of the bits, possibly from the target, is creating a false positive.

We now wish to establish the mechanism by which branches update the register. We expect that the address bits are combined with the register with an XOR operation, and from prior results, we expect that the register is shifted two places before this takes place. In order to verify this behaviour, we reuse the conditional branch address bit benchmark. We set the distance between the two blocks so that the branches appear to be the same, and that the lower bits of the second last and last branches of each block are all zeroes. We then set the address of the second last branch of the second block so that the fifth bit is one (thus making it distinct from the first block’s equivalent). We then adjust the address of last branch of the second block so that the 7th bit is equal to one. If our assumption is correct, the XOR will cancel these two bits out, and the misprediction rate will be high. Results show some of the pairings carried out during testing which achieved this effect.

<table>
<thead>
<tr>
<th>Last Branch Bit</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>4 &amp; 5 (Together)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd Last Branch Bit</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6 &amp; 7 (Together)</td>
</tr>
</tbody>
</table>

We conclude that the register is updated by shifting by two places, and XORing the address of the conditional branch with the register. The test can be similarly
modified to show that bits 18:10 of an indirect branch are concatenated with bits 5:0 of the target, and that is XORed with the register.

The below benchmark is used to establish the address bits of the indirect branch that are used to look up the indirect predictor table. Two indirect branches with distinct targets are laid out in memory. The branches leading up to them are separated out so that they appear to be the same to the predictor so that the history register has the same contents for both branches, however, we insert branches whose behaviour correlates with the behaviour of the branch, so that the indirect branches can be accurately be predicted as long as the two branches map to different entries in the predictor table. By adjusting the distance between the two spy branches, but not those of the conditional branches, we determine which bits are used in the look up of the predictor structure.

```c
int a = 1;
int long unsigned mod, iter = 1000000;
void *targets1[] = &Target1, &Target2;
void *targets2[] = &Target3, &Target4;
do {
    mod = iter % 2;
    /*6x always taken branches*/
    if (mod==0) a=1; //This branch gives correlation information to the spy branch
    if (a==0) a=1;
    goto *targets1[mod]; //Spy branch
    Target1: asm("clc");
    Target2: asm("clc");
    asm("jmp next");
    asm(".rept 0x7FF84");
    asm("nop");
    asm(".endr");
    asm("next:");
    /*6x always taken branches*/
    if (mod==0) a=1;
    if (a==0) a=1;
    //The compiler does not turn this into an indirect branch, so we do a goto manually
    //Exact code required to do this varies from compiler, and compiler version.
    //To figure it out, put in a goto, disassemble the code, and look at how it loads the
// goto *targets2[mod];
asm("mov -0x18(%rbp),%rax");
asm("mov -0x40(%rbp,%rax,8),%rax");
asm("jmpq *%rax"); //Spy branch 2
Target3: asm("clc");
Target4: asm("clc");
iter--;
}while (i>0);

Results show that bits 18:4 are used to look up the table.

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

If we set the paths so that they clash to the same indirect predictor entry, but change them so that they only have a single target, then the misprediction rate goes away. This is an indication that the indirect predictor is only used when a branch has been seen to alternate between targets, as is predicted by the BTB otherwise.

We then modify the benchmark in the same way that we modified the conditional branch benchmark in order to determine the hash function used to create a lookup for the table (Uzelac uses a different piece of code to carry out this experiment, but it is not described sufficiently to allow the experiment to be reproduced, nor is it clear why it needs a different benchmark layout). We conclude that the bits of the history register are XORed with the address bits as followed -

<table>
<thead>
<tr>
<th>History Register Bits</th>
<th>0-5</th>
<th>6-13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Address Bits</td>
<td>12-18</td>
<td>4-11</td>
<td>12</td>
</tr>
</tbody>
</table>

Finally, we will examine the structure of the indirect predictor itself. We use a benchmark that is somewhat similar to the indirect-based BTB benchmark we devised for the Atom. There is a single spy benchmark that has multiple targets. There are a number of conditional setup branches used to provide correlation to the indirect branch predictor. We use the addresses of those branches to control the history register. We take an arrangement with two such branches
and adjust the distance between them. An indirect setup branch is used to select which path we use. This is set up to be always mispredicted, so we'd expect the misprediction rate to double when the two paths collide to the same indirect target buffer entry.

```c
int a = 1;
int long unsigned mod, i = 1000000;
void *targets1[] = {&Target1, &Target2};
void *targets2[] = {&Target3, &Target4};
do {
    mod = i % 2;
    /*8x Always taken branches to clear out the history register*/
    goto *targets1[mod]; //Setup branch, always mispredicted
    Target1: asm("clc");
    asm("jne Spy"); //Setup branches to generate unique history information
    asm(".rept 0x7FFFFA");
    asm("nop"); asm(".endr");
    Target2: asm("clc");
    asm("jne Spy"); //Setup branch
    asm("Spy: mov -0x10(%rbp),%rax"); //The old inline goto trick...
    asm("mov -0x30(%rbp,%rax,8),%rax");
    asm("jmp *%rax"); //Spy indirect branch
    Target3: asm("clc");
    Target4: asm("clc");
    i--;
} while (i>0);
```

Our conclusion from the results is that bits 13:6 of the history register are used as index bits.

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>

Uzelac notes that bits 5:0 of the address bits of the history register in the Pentium M are used as a tag that is stored alongside the entry in a direct
mapped cache, and his results show a ~40% misprediction rate when the spy branches differ only by these bits. Either this scheme is not used in the Core 2, or they are used with a different replacement policy. If this was the case, then the indirect predictor table likely has a direct mapped cache indexed with 13:6, and thus having 256 entries. If we wished to validate this, we would make configurations of this benchmark with a large number of path branches to test the capacity of the table.

We expect the global predictor to use the same register as the indirect predictor. We modify the above experiments relating to the history register to use conditional branches and test. The benchmarks are the same, except for some differences -

- We use a pattern of \{16x Taken, 16x Not Taken\} for mod, as the pattern used for the indirect predictor can be predicted by the loop predictor. This produces a misprediction rate of one-eighth when relying on bi-modal behaviour.

- The setup branch is preceded with always taken branches to force it to be mispredicted. The spy branch is changed to if(mod==0) like the setup branch. If the spy branch is mispredicted, we expect to see the misprediction rate double.

- It was found during experimentation that the spy branch needs to be preceded with a number of NOPs to allow the prior branches to retire (96 were used).

- When testing for the address bits of conditional branches used in the lookup, we attempt to collide two spy branches with the same behaviour. When this happens, the misprediction rate decreases instead of increasing, as one misprediction for each branch is enough to make the bimodal counter change direction. Uzelac uses a different benchmark which requires the benchmarker to distinguish between two small misprediction rates, which raises issues of measurement accuracy.

Results demonstrate that the history registers for the two mechanisms are the same.

Uzelac provides a benchmark for investigating the organization of the global predictor table, which works along similar lines to the benchmark for the indirect
predictor table benchmarks. However, the benchmark has to take into account the behaviour of the indirect predictor, the loop predictor, and the bimodal table present in the Pentium M that is looked up using address bits of the branch only. This necessitated producing a rather elaborate setup path, and measure to fool the loop predictor. Furthermore, Uzelac ran into similar replacement policy issues as with the BTB, requiring that the these paths be run through twice on every execution. Attempts to recreate this rather elaborate benchmark proved inconclusive, and it was decided to leave it to future work in order to pursue other areas of experimentation.

The Core 2 allows us to monitor mispredictions at the instruction decode stage. We will first carry out the capacity experiments of Milenkovic to see if the Core 2 architecture exhibits the same issues that were encountered by Uzelac when benchmarking the Pentium M. The results leave it unclear as to whether the CPU has a 4-way cache with 1024 entries, or a direct mapped cache with 2048 entries, or possibly even a 4-way cache with 2048 entries. Consequently we will use the experiment that we initially developed for the Atom (we could attempt to recreate Uzelac’s experiment, but the new experiment should yield the same results)

When carrying out this experiment, we need to contend with the fact that the Core 2 has an indirect predictor which may initially be able to predict the setup indirect branch, but eventually will cause mispredictions when one or both of the following happen -

- The distance between the spy branches gets to a stage where the paths leading up to the indirect branch appear to be the same.
- The number of spy branches gets to a stage where there are too many targets and paths for the indirect target buffer to keep track of.

In order to prevent these scenarios skewing our results, we proceed the setup branch with eight always taken conditional branches so that the setup branch is always mispredicted. This way, the misprediction rate is constant, and affects our results in a predictable way.

First we carry out the capacity test. Results are shown for branch counts 1024, 2048 and 4098 respectively -
Now the set test is carried out. Results are shown for inter-branch distances 2048, 4096, 8192.

<table>
<thead>
<tr>
<th>Inter-branch Distance</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-branch Distance</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-branch Distance</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (Millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

We test the most and least significant bit -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.97</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

We draw the following conclusions -

- The BTB is indexed by bits 21:4
- The cache is a 4-way cache with 2048 entries.
- Bits 12:4 are used as the set index, and bits 21:13 are used as the tag.

Furthermore, we investigate what happens when a non branch instruction has the same tag and address bits as a branch. We set up a simple benchmark where
a branch and a NOP are laid in memory so that they only differ in address by
bit 22, thus colliding them into the same BTB entry.

Measuring mispredictions at the front end, the benchmark generates a mispre-
diction rate of one per iteration of the test loop, implying that a non branch
instruction can cause the eviction of a valid branch entry in the BTB. This
phenomenon is referred to as a “bogus branch”.

We proceed to investigate the behaviour of the loop predictor. The general
design of the loop predictor has been detailed by Intel themselves, and Uzelac
devised a series of benchmarks of benchmarks for determining particular proper-
ties of the loop predictor and how it interacts with other prediction mechanisms.
We will test these on the Core 2 to see how they apply to it.

As we expect there to be a table of loop predictor elements, we carry out the
typical array of tests for capacity, organization and the address bits used to look
up the cache. We test this by creating a series of benchmarks where we vary the
number and distance between a series of loops with distinct pattern lengths.¹

```c
int long unsigned i = 1000000;
do {
    _asm
    {
        mov eax, 63
        10: sub eax, 1
        cmp eax, 0 jne 10
    }
    asm(".rept 0x3F3"); //Inter-loop distance of 1024 bytes
    asm("nop");
    asm(".endr");
    _asm{
        mov eax, 62
        11: sub eax, 1
        cmp eax, 0 jne 11
    }
}

¹This benchmark design was developed during a phase of the project where Intel’s icc and
vTune were being used. It would be necessary to ‘translate’ them to AT&T syntax assembly
to run them on gcc. Running on icc involves using the -use-msasm flag.
We test the address bits by separating two loops, test the capacity with a large number of loops, and the set associative by separating a small number of branches by larger distances to try and collide them to the same set. When collisions occur in each of these cases, the branch predictor will mistake one loop for another, and will mispredict the point at which it is not taken. Consequently, when the number of branches exceeds capacity, or when the distance is such that not all sets are used and that capacity is exceeded, we see a number of mispredictions equal to the number of loops multiplied by the number of iterations of the test loop. First we test the address bits used -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Then we carry out a capacity test. Results are shown below for distances of 8, 16, 32 and 64 bytes -

<table>
<thead>
<tr>
<th>Loop Branches</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
</tr>
</tbody>
</table>

For the smallest distances - 8 and 16, we see a maximum capacity of 128 branches, and suggesting a 2 way cache. However, the results suggest are not entirely conclusive as the size of the loop construct means that we cannot place
the branches together closely enough to test for smaller distances, and thus the number of ways is inconclusive. We then carry out the set test. Uzelac uses a different benchmark code for the set test, but it was found during testing that the capacity test code could be reused without issue. By virtue of testing the address bits, we have effectively carried out a set test for 2 branches, so we now try for larger numbers. The below shows the results for the set test using three branches, and varying the distance between them -

<table>
<thead>
<tr>
<th>Different Address Bit</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mispredictions (millions)</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Given a capacity of 128 entries, these results are consistent with a 2-way cache design that is indexed by bits 9:4 (for a total of 64 sets).

From these, we draw the following conclusions -

- Bits 15:4 are used to look up the loop predictor table.
- It is a 2-way cache with a total of 128 entries.
- Bits 9:4 are used as an index, and 15:10 as a tag.

There are further pieces of information which are of interest. First, we investigate the logic used to allocate a branch into the loop predictor. Ideally, the branch predictor mechanism will only consider a branch to be a loop branch when it has fully established that it behaves like a loop branch. However, the question is raised as to how long the branch should be monitored for such behaviour, and it’s more likely that the loop predictor allocates a branch when it is not taken after being taken a number of times. To verify this behaviour, we use three branches, two with loop behaviour, and a third that has the pattern {3x Taken, 2x Not Taken}. They are laid out such that they compete for the one loop buffer entry. It has been demonstrated beforehand in regards to testing for the local predictor that the latter pattern cannot be predicted by the loop predictor. However, if the loop predictor tries to allocate it into the buffer, it will cause the eviction of the real loops. Consequently, we’d expect a misprediction rate equal to the misprediction rate of the unpredictable branch, plus those of the evicted loops.
int mod1, mod2, mod3, a=1, i = 1000000;

do 
{
    mod1 = i % 3;
    mod2 = i % 4;
    mod3 = (i%5) & 0xFE; //Non-loop pattern
    /*block of always taken branches to prevent global predictor interference, and nops*/
    if (mod1==0) a=1; //Loop branch
    /*nops to adjust distance between branches to 1024 bytes, block of always taken branches*/
    if (mod2==0) a=1; //Loop branch
    /*nops to adjust distance between branches to 1024 bytes, block of always taken branches*/
    if (mod3==0) a=1; //Spy branch with expected misprediction rate of 60%
    i--;
} while (i > 0);

Test results give a misprediction rate approximately equal to the misprediction rate of the two loop branches being incorrectly predicated, plus the misprediction rate of the spy branch. It is concluded that branches are allocated into the buffer the first time they go from taken to not taken. As the loop predictor cannot handle these branches, it is likely that the design relies on the LRU policy to eventually dispose of erroneous entries.

Then we determine the relationship between the BTB and the loop predictor. Uzelac demonstrated that the BTB in the Pentium M stored branch type information, and that the loop predictor was only consulted if the BTB believed the branch to be a loop branch. Five branches – four always taken, and one with loop behaviour are laid out in memory to compete for entries in the one BTB set (which holds four entries). If the loop predictor works independently of the BTB, the branch with loop behaviour will be correctly predicted. If not, the BTB will believe the branch to be the same branch as the always not taken branch, and the loop branch will be predicted using a bimodal predictor, leading to a misprediction rate proportional to the pattern length of the loop branch.
int mod,a=1, i = 1000000;
do
{

    mod = i % TEST_VAL; //Adjust value to create different patterns
    if(a==0)
    {

        //Distance of 8192 bytes to map each branch to the same BTB set
        asm(".rept 0x1FF6");
        asm("nop");
        asm(".endr");
    } if(a==0) {
        //NOPs to create distance of 8192 bytes
        } if(a==0) {
            //More NOPs as above
            } if(a==0) {
                //More NOPs as above
                } if(mod==0) a=1; //Spy loop branch
                i--;

    }
    while (i > 0);

Results are show below, tested for multiple pattern lengths.

<table>
<thead>
<tr>
<th>Loop Pattern Length</th>
<th>4</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>0.25</td>
<td>0.125</td>
<td>0.1</td>
</tr>
</tbody>
</table>

From the results, we conclude that the loop predictor is dependent on the BTB being able to identify the branch as a loop branch. This arrangement is likely used as it is far less likely that a collision between a loop and non-loop branch will occur in the larger BTB compared with the smaller loop prediction table.
6 Reverse Engineering the Sandy Bridge

First we test for an indirect predictor, and establish that the Sandy Bridge has one -

<table>
<thead>
<tr>
<th>Number of Targets</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

We carry out the same indirect branch tests as we did for the Core 2. The results are very similar to those provided by the Core 2, so we do not list them fully here. Instead we will discuss the differences -

- In the Core 2, the bit 18 was the most significant bit used in lookups, and to update the history register with conditional and indirect branches. In the Sandy Bridge, it appears to be bit 17. Consequently, the history register appears only to be 14 bits long. Results showed that a shift count of two was used as with the Core 2.

- Unconditional direct branches affect the history register. As described in Chapter 3, this may have been a way of dealing with situations where compilers attempt to reuse the one indirect branch to simplify the control flow of the program, however, it means we need to be careful of using unconditional jumps over branches, and handling the else part of an if statement (a good way is to inline an unconditional jump at the end of both the if and else part of the branches which will jump to the same target after the block.)

- Attempts to determine the hashing function proved inconclusive. It may use something other than XOR, but it was not possible to discern what it may have used instead. A more likely explanation was that some unforeseen interference from elsewhere in the predictor structure was preventing mispredictions when we tried to collide address bits.

We now wish to determine properties of the outcome predictor used in the Sandy Bridge. We run the test described in the appendix to establish the maximum pattern length. Unusually, it was necessary to increase the test loop iterations from 10 million to 100 million to get conclusive results. Results are shown below

| Pattern Length | 2 | 3 | 4 | ... | 36 | 37 | 38 | 39 | ... | 64 | 65 |
|----------------|---|---|---|-----|----|----|----|----|-----|----|----|----|
| Misprediction Rate | N/A | N/A | N/A | N/A | N/A | N/A | 5.1% | 4.7% | ... | 3.1% | 3% |
The Sandy Bridge is capable of making a prediction of up to a pattern length of 37. We then insert an always taken conditional branch before the spy branch, which raises the misprediction rate to be the same as when the pattern length of 38. We now test small pattern lengths with the spy branch preceded with 72 always taken conditional branches in order to establish the presence of a local predictor which handles smaller patterns. Results are shown below. We do not continue testing as far as 37 as it becomes apparent that the Sandy Bridge relies on global prediction, and has no loop or local predictor.

<table>
<thead>
<tr>
<th>Pattern Length</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misprediction Rate</td>
<td>90%</td>
<td>66%</td>
<td>51%</td>
<td>39%</td>
</tr>
</tbody>
</table>

The results show something odd – when the pattern cannot be predicted, the misprediction rate is twice what we would expect it to be if bimodal predictors would use. This could be explained by the fact that Sandy Bridge is using 1 bit counters instead. Research into this led to the discovery of the article discussed in Chapter 2 about the use of 1 bit counters in newer Intel processors.\[16\] To verify this hypothesis, we set up the following benchmark.

```c
int mod, a=0, i=1000000;
do
{
    mod = (i % 32) >> 4;
    /* insert a block of if (a==0) a=1; here */
    if (mod==0) a=1; //Always mispredicted correlation branch
    /* insert a variable number of always taken branches here */
    if (mod==0) a=1; //Spy branch
    i--;
}
```

The setup branch provides correlation to the spy branch, and is preceded with a large number of conditional branches to force it to be mispredicted. If the outcome predictor uses one bit prediction, we’d expect a misprediction rate of one-sixteenth (compared with one-eighth when a bimodal predictor is used). As we increase the number of conditional branches between the spy and setup...
branch, we will reach a number that clears out the history register, preventing
the spy branch to be mispredicted.

The misprediction rate is approximately one-sixteenth with no separation branch,
and doubles when the spy and setup branch are separated by 7 always taken
conditional branches. From this, we draw the following high level conclusions
about the Sandy Bridge’s outcome predictor -

- The outcome predictor uses 1 bit counters instead of bimodal predictors.
- 7 indirect, taken conditional, or unconditional branches will clear out the
  history register.
- While the Sandy Bridge does not have a Loop Predictor, it appears that
  its outcome predictor can predict a branch with a relatively long loop-like
  pattern. A likely explanation for this is that the history register
  is not updated with outcome or address information every time one is
  encountered, possibly in conjunction with some loop detection logic. This
  idea is backed up by some information found in one of the sources.[16]

The similarities between the above behaviour, and the behaviour of the indirect
predictor suggest that as with the Core 2, the two share the same outcome
predictor. As with the Core 2, we do not develop benchmarks to determine the
organization of the global predictor table due to the complexity in cancelling out
interference between the other prediction units in the processor. In particular,
the ability of the Sandy Bridge to have different pattern prediction properties
depending on branch behaviour would cause all sorts of issues during testing.
If we wanted to flesh out benchmarks to show that the global predictor and
the indirect predictor used the same register, we would probably pick a tighter
pattern, such as \{8x Taken, 8x Not Taken\} to give a higher misprediction rate
with the single bit counter, as the misprediction rate of one-sixteenth when
using the same pattern as used for the Core 2 is rather close to the point of
being statistically insignificant.

As with the Atom, the Sandy Bridge does not have a means for counting mis-
predictions at the instruction decode stage. We apply the same benchmark that
we used for the Atom and Core 2 in order to establish the organization of the
BTB cache. The results are effectively the same as the ones yielded for the Core
2, thus implying that Sandy Bridge BTB is -
- A 4-way cache with 2048 entries.
- Uses branch address bits 21:4
- Indexed by bits 12:4, tagged with bits 21:13

As mentioned in Chapter 2, some sources suggested that the Sandy Bridge has a target buffer that supported variable target sizes, allowing it to vary the amount of branches it stored depending on how far the branches jumped. Yet, this property does not appear to affect the experiment as described above, except by possibly making the BTB appear to have more or less capacity that it actually does depending on the definition of what constitutes a short or long branch in the context of the suggested BTB design. In order to verify this behaviour, we would need to determine how the CPU distinguishes between different target sizes, how the different sizes are stored (is the table divided into sections for different types? Or do multiple targets share the one cache entry? How are they distinguished?) and attempt to devise a benchmark that used both small and large target branches. In the end, due to the relatively vague information on this concept, and the possibility that it could be wrong or incorrectly described, we do not investigate further, and leave it for future work.
7 Conclusions and Future Work

The project has been successful in revealing an extensive set of details about the branch prediction units in the modern Intel x86 CPUs which we have tested. For the Intel Atom, data suggests the following structure -

For the Core 2 Duo
History Register - 15 bits, whenever CPU encounters a taken conditional or indirect branch, shift two places to the left, and XOR with bits 18:4 for conditional, 18:10 for indirect + 5:0 of indirect target

XOR Lookup Hash - Bottom half of register XOR'd with upper half of address, and vice-versa

Indirect Branch Target Buffer

256 Entries

Non Monotonic Indirect Branch? (From BTB)

XOR Lookup Hash - Same as used in Outcome Predictor

History Register - Same used in Outcome Predictor
For the Sandy Bridge -

Branch Target Buffer

- 2048 Entries
- 512 Sets
- 4 Ways
- Tagged with bits 21:13
- Indexed with 12:4

Instruction of Last Byte of Branch Instruction

Indirect Target Table

- 256 Entries
- Non Monotonic Indirect Branch? (From BTB)

Lookup Hash

History Register - 14 bits, whenever CPU encounters a taken conditional or indirect branch, shift two places to the left, and update with bits 17:4 for conditional, 17:10 for indirect + 5:0 of indirect target
Whilst the work builds on the research of others, prior research was primarily focused on a single CPU microarchitecture (particularly the Pentium M in the case of Uzelac, and the Pentium 4 in the case of some other papers encountered.) In carrying out the experiments on a number of CPUs, we have provided the following contributions -

Experiments and results that provide specific information on the details of branch predictors in the CPUs analyzed. Whilst some of the details uncovered here have been speculated upon in the media or academia, this project provides conclusive results based on experimentation. Implementing benchmarks that are not tied down to the quirks of a single CPU, and providing an experimental methodology that makes as few assumptions as was feasible. Providing, as much as is possible, enough information to carry out these experiments. Much of the prior work in this field concerned itself with algorithms, and not the practical details of implementation required to implement the benchmarks properly. Demonstrating an experimental methodology applied to three quite different CPUs, we identify the different situations that may arise, and demonstrate how they should be interpreted and further analyzed.

The primary difficulties encountered during the project were issues relating to implementing benchmarks correctly (see Chapter 3 for a discussion of the issues encountered.) Any benchmark that required the use of inline assembly required extensive hand tweaking in order to perfect. A single branch address being different to the expected place by a single byte was enough to cause a correctly
designed benchmark to appear inconclusive, or more problematically, an incor-
rectly designed benchmark to be appear to be correct. In addition, memory bit
addresses required extensive tweaking in order to set arbitrary bits to desired
values. Consequently, every benchmark generated had to be run through the
disassembler, and any changes would also require disassembly to ensure that
new discrepancies did not crop up.

As the project progressed, the experience gained allowed these problems to
be solved expediently, but it still ate into large quantities of time that could
have been used to develop more benchmarks, or experiment on other processor
architectures. A possibly way to mitigate some of these issues would have been
to write pure assembly programs, in order to avoid any compiler interference.
This, however, would have required a substantial amount of up front effort, and
would be only be recommended for those building upon this work, as opposed
to people starting afresh.

The initial plan of work was an iterative process whereby a single area of branch
prediction would be worked on before moving on to the next. It was found,
however, that implementing more complicated ones (particularly the indirect
predictor benchmarks) shed a lot of light on both the practical issues around
benchmark implementation, and the relationship between the different parts of
the system. This knowledge then allowed issues with the more straightforward
benchmarks to be resolved quickly. Consequently, it would have made more
sense to research the entire system up front, rather than taking a divide and
conquer approach.

In regards to future work, even when considering some of the above experiments,
further experimentation could be carried out, including -

- Thorough capacity testing of the indirect and global outcome predictors.
- Examining some of the novel features of the Sandy Bridge, including the
  logic by which a loop pattern can be recognized by the global predictor,
  and the claims of the variable target length BTB.
- Some untested areas, such as the prediction of calls and returns.

The CPUs experimented upon were all Intel-made ones, and whilst there was
some considerable design difference between the ones which we examined, similar
overall principles applied, allowing us to make certain assumptions about how
things worked based on prior work on this topic. If we were to experiment with non Intel designs, we would discover that some of these benchmarks would not be suitable. For example, the BTB in certain AMD CPUs is tied into the instruction cache, and there is selector logic which limits the number of branches that can be correctly predicted to 3[17]. If we had benchmarked an AMD CPU, this may have caused our experiments to give inconclusive results, and would have required extensive redesigning and experimentation.

It would be particularly interesting to apply these experiments to a non-x86 architecture, as most research into branch prediction reverse engineering or code optimizations has focused on x86 CPUs. This of course would require redesigning the assembly-based benchmarks to the intricacies of different instruction set architectures, in addition to dealing with different designer’s ideas of how branch prediction should be implemented. Another problem is finding the appropriate tools to carry out the experimentation, particularly disassemblers and performance counter monitors. There is no shortage of different open source tools for x86, but for less popular architectures, finding suitable equivalents for other architectures may prove difficult.

Carrying out this project gave a great insight into the design of branch prediction units, their relative strengths and weaknesses, and the trade-offs that may be made by manufacturers when combining different predictor types together. The design of benchmarks provided experience with writing of x86 assembly, program generation techniques using Python and assembler directives, and the use of disassembly tools. The reverse engineering process itself required a lot of research into branch prediction techniques, and being able to design tests and interpret results based on this knowledge.
A Appendix - Generalized Outcome Predictor Experiments

Milenkovic presented a general workflow for determining the sort of outcome predictor used in a CPU. However, it only could determine if local or global prediction was used, and didn’t take into account a situation where a loop predictor was present, or if no sort of pattern based recognition was used at all. Uzelac presented benchmarks for the loop predictor, but due to published information on the Pentium M processor that he reverse engineered, he was able to assume the presence of a loop predictor and proceeded to experiment with the properties of the predictor without showing conclusively that one existed. Here, a more general workflow is presented that should allow some useful information to be concluded when no details about the CPU’s predictor structure is known. The below descriptions assume the use of bimodal prediction. In the case of a CPU like the Sandy Bridge which does not use such bimodal predictors, the expected misprediction rate will differ, but the same principles outlined should still stand.

First we wish to determine if the branch predictor uses some form of pattern-based prediction, and if so, what is the length of the largest pattern it can keep track of. To do so, we use a piece of code as follows -

```c
int a, i, j=TEST_VALUE;
for (i=0; i<1000000; i++)
{
    if ((i%j)==0) a=0; /*Spy branch*/
}
```

The Spy Branch has the pattern {((TEST_VALUE – 1) x Taken, 1 x Not Taken)

We create versions of this program with TEST_VALUE ranging from 2 to 128 (in theory, we may need more than 128 if the CPU was capable of predicting a very large pattern, but 128 was found to be more than enough.) If we run these applications, and monitor their misprediction rate, we will expect to see one of two behaviours -
1. The misprediction rate is high for TEST\_VALUE equal to 2 (about 50%-100%), and the misprediction rate slowly decreases as we increase TEST\_VALUE. This is an indication that the CPU does not have pattern-based prediction, and is merely using the address of the branch to look up a table of bi-modal predictors. At this point, we would carry out tests to determine the address bits used, and the organization of the table, and no more of the further described experiments are of relevance.

2. The misprediction rate is negligible for all values of TEST\_VALUE less than some value $n$. Once we increase the TEST\_VALUE beyond $n$, the misprediction rate slowly drops. This is an indication that the CPU is capable of handling a pattern of up to $n-1$ (we will refer to this value as $L$)

For both cases 1 and 2, we’d expect any versions of the benchmark that generate mispredictions, we’d expected the misprediction rate to be the number of times the branch is executed divided by TEST\_VALUE – for example, if TEST\_VALUE is 100, and the CPU cannot predict that the 100th execution is not taken, then there is a misprediction for every 100 executions of the branch. If bimodal prediction was not used, the misprediction rate would be double.

Once we have established the maximum pattern length $L$ that can be predicted correctly, we need to determine how exactly the CPU is able to predict it. There are three possible explanations -

The CPU has a loop predictor that can detect a pattern of $L$. The CPU has local prediction that is capable of keeping track of the last $(L-1)$ outcomes of a given branch. The CPU has a global predictor that can store the details of the last $2(L-1)$ (this is because the loop in the benchmark would also be contributing the global predictor)

Milenkovic goes further by saying that the local predictor will have a register that holds $L$ bits, and likewise that the global predictor will have $2(L-1)$ bits in its register. However, as is demonstrated on the experiments for the Core 2 and Sandy Bridge, this claim only holds true if the register only holds outcomes represented with a single bit (as with the Atom.) In the case of more sophisticated processors, the register may be shifted several places before update, and it may only take the addresses of taken branches into account. More details experiments are consequently needed to establish the length of the history register.
To determine if global prediction is used to predict this pattern, we take the above benchmark with TEST_VALUE equal to \( L \). We then insert a number of always taken conditional branches (such as an if statement whose condition always evaluates to false) before the Spy Branch, and measure the misprediction rate. If the misprediction rate stays low, this is an indication that the CPU has local or loop prediction. If it goes high, it is an indication that the CPU is using global prediction (if we insert \( n \) always taken branches, we’d expect the misprediction rate to be the same as it would be if we set TEST_VALUE to equal \( L + n \))

In the case where global prediction is found to be used, we need to determine if it is making use of any local prediction for smaller patterns. To do so, we modify the original benchmark to feature \( 2L \) always taken conditional branches. We then test the program for values of TEST_VALUE from 2 to \( L \). We expect one of two behaviours -

The misprediction rate is low from 2 to some value \( M \) where \( M < L \). This is an indication that there is a local predictor capable of predicting a pattern of length \( M \). The misprediction rate is high for all values from 2 to \( L \). The misprediction rate is approximately equal to the number of spy branch executions divided by TEST_VALUE. This is an indication that only global prediction is used, as the always taken branches have displaced any information from the history register required to correctly predict the Spy branch.

If we have a loop predictor, we would want to establish whether any local prediction sits underneath it. To do so, we’d pick the smallest non-loop-like pattern – \{2x Taken, 2x Not Taken\}, assuming that any local predictor located alongside a loop predictor be at least capable of predicting that. We proceed the spy branch branch using this pattern with a large number of always-taken conditional branches to prevent the global predictor predicting the pattern. A high misprediction rate (around 75%) indicates that the CPU relies on a global predictor for this situation, and that there is no local predictor. In this case, removing the always-taken branches should eliminate the misprediction rate. If the pattern is predicted correctly irrespective of the number of branches before it, then the CPU has local prediction, and we would increase the pattern length to determine what length of pattern the local predictor can predict. If it is capable of predicting a pattern of \{ \frac{L}{2} \times \text{Taken}, \frac{L}{2} \times \text{Not Taken} \} then we conclude that the local predictor was providing the prediction we saw for the original pattern experiment, and that there is no loop predictor.
Tests relating to local prediction, and determining if there is a global predictor if we have established the presence of a local predictor, were not developed in this project, as local prediction was not found in any of the CPUs that were experimented upon, and the experiments would be little different to those described by Milenkovic. However, Milenkovic’s experiments are only concerned with pattern length, and not with issues of the address bits used, or the size of the caches. If we were to experiment on a local predictor, it would proceed along similar lines to the experiments on the global predictor, except we would need to determine the mechanism by which a local history register is selected, and how many the CPU has.

We would first have two Spy branches with different behaviour that could be predicted by the local predictor, and separate them out so that they end up using the same register, causing mispredictions. Using this, we could determine the address bits used. We could then use a capacity benchmark with conditional branches with varying patterns to determine how many history registers are in use.
References


