Final Report on Computer Architecture Project
EE 5364 Advanced Computer Architecture

Department of Electrical and Computer Engineering
University of Minnesota, Twin Cities

Report Prepared by:
Amanda Skinner (3749048)
and
Nirav Desai (4280229)

Course Instructor: Prof. Pen-Chung Yew
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>3</td>
</tr>
<tr>
<td>Literature Review</td>
<td>7</td>
</tr>
<tr>
<td>Look Ahead Program Counter based Speculative Address Estimation and Data Prefetching</td>
<td>8</td>
</tr>
<tr>
<td>Data Prefetching Example ( As given in the reference paper )</td>
<td>9</td>
</tr>
<tr>
<td>Basic Block Diagram for Data Pre-loading</td>
<td>10</td>
</tr>
<tr>
<td>Reference Prediction Table</td>
<td>12</td>
</tr>
<tr>
<td>Branch Predictor:</td>
<td>14</td>
</tr>
<tr>
<td>Loop Counter Branch Predictor:</td>
<td>14</td>
</tr>
<tr>
<td>The O-GEHL Branch Predictor:</td>
<td>15</td>
</tr>
<tr>
<td>The Branch Decision Table:</td>
<td>16</td>
</tr>
<tr>
<td>Duplicate ALU</td>
<td>17</td>
</tr>
<tr>
<td>Testing Methodology:</td>
<td>18</td>
</tr>
<tr>
<td>References</td>
<td>19</td>
</tr>
</tbody>
</table>
Background

The goal of this project is to design a branch predictor for a microprocessor. In order to understand why a branch predictor would be necessary, we carried out a few simulations in the Simple SCALAR microprocessor simulator to show the impact of the branch predictor on the microprocessor performance. These simulations were part of the first homework assignment in the course.

SPEC 2000 Benchmark Characteristics

Figure 1 above shows the instruction mix for the SPEC CPU 2000 AMMP Floating Point Benchmark. The total number of jump instructions in the SPEC CPU 2000 AMMP benchmark are $20.66777 \times 10^7$. This shows that if the processor performance at the jump instructions is improved, it will have sizeable impact on the overall processor performance.

Jump instructions adversely affect the microprocessors performance mainly from the aspect of the
performance of the cache and the execution pipeline. Modern microprocessors have execution pipelines that are up to 20 stages[1] long and these need to filled in advance so as to get the maximum throughput from the micro-processor. Branch instructions need to be predicted correctly in order to fill the pipeline and avoid pipeline flushes that cost additional clock cycles [2].

Figure 2: Branch Predictor Performance with Branch Predictor Type for the SPEC CPU 2000 MESA benchmark.
Figure 3: Cache Misses as a function of the Branch Predictor Performance for the SPEC CPU 2000 MESA benchmark.

Figure 4: Cache Performance with Branch Target Buffer Size for the Bimod Branch Predictor on the SPEC CPU 2000 GCC Benchmark.

Figure 2 above indicates the branch predictor performance for existing branch predictors on the SPEC CPU 2000 MESA benchmark and figure 3 indicates the number of Cache misses with the branch predictor type for the same benchmark. As can be seen, the number of cache misses
decrease with increasing branch predictor size. Figure 4 above indicates that the number of cache misses decrease with increasing Branch Target Buffer Size for the bi-modal branch predictor on the SPEC CPU 2000 GCC benchmark.

A cache miss has a very high penalty on the microprocessor performance as the pipeline has to stall till the required data is available and the access time for the second level caches and main memory is usually very high. Increasing the cache size [3] is one of the ways to decrease the number of cache misses. However, the cache size cannot be increased arbitrarily as the read access time of the cache will go up for higher cache sizes. Cache partitioning into blocks [4 Rab96 Page 558] is another approach used to increase the cache size without increasing the read access time. In any case, having the information for the correct branch to take will help to prefetch the right block of data to the cache. Block prefetching to the cache takes up a lot of bandwidth and if block prefetching goes wrong, the cache miss penalty is much higher than without pre-fetching. So in general, having the branch prediction information can be very beneficial in getting the full throughput from the microprocessor – both from the point of view of full pipeline utilization and reduced number of cache misses.

![Figure 5: Pipeline Efficiency with increasing Branch Predictor Accuracy](image)

**GCC Benchmark on Simple SCALAR sim-outorder**

**Pipeline Efficiency with increasing Branch Predictor Accuracy**
Literature Review

We started off this project with a literature review of the existing branch predictors and their performance evaluation. The preceding section shows various existing branch predictors studied and their performance impact on the SimpleSCALAR microprocessor configured as an Out of Order Issue Microprocessor with a decode width of 4. The branch predictors studied are 1) Always Taken, 2) Always Not Taken and 3) Bi Modal Branch Predictor with varying Branch Target Buffer Size. These are all included in the SimpleSCALAR microprocessor and studied in the text book for the course: Computer Architecture: A Quantitative Approach by Patterson and Hennessy.

A history based branch predictor with multi-level look ahead[5] was the first branch predictor considered for this project especially because the multi-level look ahead would allow cache prefetching to be done well in advance of the executing Program Counter[6]. However, this approach is also used for the Branch Target Buffer based bi-modal branch predictors studied in the class lectures and included in the SimpleSCALAR microprocessor simulator. This was our first project proposal.

The second approach we developed from the start of this project was to use a look-ahead branch predictor which does all the computation to make the branching decision before the executing Program Counter reaches the instruction. One of the approaches for a Look-Ahead Branch Predictor that does data prefetching for the cache would be to use a Data-Decoupled Architecture[7]. Data decoupled architectures use 2 parallel instruction streams – one for data access and another for instruction execution. By decoupling data access from instruction execution, these architectures make Vector Processing possible and so are used in in the CDC CYBER 200 series of vector micro-processors[8].

We finally decided to implement a Look-Ahead Program Counter based branch predictor as outlined in [9]. The Look-Ahead Program Counter does a stride calculation based on the instruction history and uses this information to pre-fetch selected data to the cache. The branch predictor in [9] uses a Branch Target Buffer with 2 bit state encoding for branch prediction. We are planning on doing the branch prediction based on calculation of the branching condition using the pre-fetched data. Since most branch decisions are based on keeping track of a local loop counter, using the value of the loop counter to calculate the branch condition could give theoretically very accurate results. We plan to use a duplicate register stack accessed only by the Look Ahead Program Counter on the instructions of load, store and loop counter update to make the actual comparison to decide whether to branch or not. In addition, branch prediction for random jumps with long histories will require an auxiliary TAGE branch predictor as given in [10]. The next 2 sections in this mid-term report describe the over-all approach we plan to use for this project.
Look Ahead Program Counter based Speculative Address Estimation and Data Prefetching

This idea has been developed by Prof. Tien Fu Chen at the University of Washington and described in his paper: An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty[9]. The basic approach is to use a Look Ahead Program Counter that keeps track of the load and store instructions in a Reference Prediction Table (RPT). The LA-PC does speculative address estimation by keeping track of the stride at each load-store instruction in the RPT.

There are 4 types of Stride Patterns that are covered by the Look Ahead Program Counter. These are all based on the idea of there being multiple nested for-loops. The LA-PC based speculative address estimation works best for nested for loops based design.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>Simple Variable Reference</td>
<td>Index, count</td>
</tr>
</tbody>
</table>
| Zero Stride  | Reference to a subscripted array element with the subscript being an invariant at that loop level but modifiable at an outer level. | for(i=0;i<10;i++){  
for(j=0;j<10;j++){  
a[j]=a[j]+b[i];  
}  
//b[i] here is Zero Stride |
| Constant Stride | The subscript of the array element increases linearly at that loop level.  | for(i=0;i<10;i++){  
for(j=0;j<10;j++){  
a[i][j] = a[i][j]+b[i];  
}  
//a[i][j] here is constant stride |
| Irregular    | None of the above                                                          | Will need TAGE here for branch prediction.  
No preloading done by LA-PC    |

Table 1: Stride Patterns covered by the Look Ahead Program Counter

As described in the paper, this scheme works best for high performance micro-processors with relatively small first level caches with a small block size running programs where the data access patterns are regular but not necessarily of stride 1. The data access patterns of load-store instructions will be kept in a Reference Prediction Table (RPT) which will be accessed ahead of time by the Look-Ahead Program Counter (LA-PC). The LA-PC will run ahead of the PC and will use the branch predictor to determine which instructions to execute. The branch predictor will be a combination of a loop counter and a TAGE predictor for predicting regular loops and irregular jumps.
Data Prefetching Example (As given in the reference paper):

Consider the sequence of nested for loops:

```c
int A[100][100], B[100][100], C[100][100];
for (i=1;i<100;i++){
    for (j=1;j<100;j++){
        for (k=1;k<100;k++){
```

This will map to the following assembly language code:

<table>
<thead>
<tr>
<th>ADDR</th>
<th>INSTRUCTION</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>lw r4, 0(r2)</td>
<td>Load B[i,k]</td>
</tr>
<tr>
<td>504</td>
<td>lw r5, 0(r3)</td>
<td>Load C[k,j]</td>
</tr>
<tr>
<td>508</td>
<td>mul r6, r5, r4</td>
<td>B[i,k]*C[k,j]</td>
</tr>
<tr>
<td>512</td>
<td>lw r7, 0(r1)</td>
<td>Load A[i,j]</td>
</tr>
<tr>
<td>516</td>
<td>addu r7, r7, r6</td>
<td>+=</td>
</tr>
<tr>
<td>520</td>
<td>sw r7, 0(r1)</td>
<td>Store A[i,j]</td>
</tr>
<tr>
<td>524</td>
<td>addu r2, r2, 4</td>
<td>Ref B[i,k]</td>
</tr>
<tr>
<td>528</td>
<td>addu r3, r3, 400</td>
<td>Ref C[k,j]</td>
</tr>
<tr>
<td>532</td>
<td>addu r11, r11, 1</td>
<td>Increase k</td>
</tr>
<tr>
<td>536</td>
<td>bne r11, r13, 500</td>
<td>loop</td>
</tr>
</tbody>
</table>

Table 2: Assembly language code for the above nested for loops for Matrix Multiplication

The technique described in the paper keeps the LA-PC ahead of the PC by δ where δ is the latency to access the next level in the memory hierarchy.

For the above sequence of instructions, at steady state, the RPT will contain entries for the 3 load lw and the sw instructions. Since in each iteration of the inner loop A[i,j] accesses the same memory location, it will have a stride of 0 and no pre-load will be requested for it. Depending on the block size, references to B[i,k] (constant stride) will be pre-loaded at every iteration (block size = 4) or at every other iteration (block size = 8). References to C[k,j] (constant stride with stride larger than block size) will generate a pre-load instruction every iteration.
Basic Block Diagram for Data Pre-loading

Figure 6: Block Diagram of the scheme discussed in the paper (Adapted from the paper) The branch Prediction Table will be replaced by a Loop Counter + TAGE Predictor in our implementation

- The Branch Prediction Table in the figure will be replaced by a Loop Counter with TAGE Predictor in our implementation
- The LA-PC is the secondary PC used to predict the execution stream. The LA-PC uses the Branch Predictor to decide which instructions to pre-fetch.
- The Reference Prediction Table is used for loading blocks in the data cache. The adder does stride computation.
- The Outstanding Request List (ORL) holds the addresses of in-progress or outstanding requests.

Each RPT entry corresponds to a load/store instruction and contains the address of the operand to be loaded. When the LA-PC hits a load-store instruction that has already been stored in the RPT, a check is made to see whether

1. The state of the entry is for no prediction.
2. The data is already in the cache.
3. The preloading of the block is in progress (by checking the ORL).

If none of these three conditions is true, a request to load the operand is performed and its address is stored in the ORL. If the LA-PC hits a branch instruction, it uses the Loop-Counter Branch Predictor to determine whether to branch or not. In this case, a predicted value is given to the LA-PC and it will be corrected later if the prediction was wrong. When the PC hits a load-store it modifies or enters the corresponding entry in the RPT. The same occurs for a branch instruction in the BPT.

On a store, a write-allocate, copy back mechanism is employed. On a replacement, the replaced dirty line is put in the write buffer and handled in the usual way. However, flow dependencies require that
each preload or data miss be checked with the contents of the write buffer.
Reference Prediction Table

Figure 7: Reference Prediction Table and State Diagram adapted from the reference paper

The Reference Prediction Table is used to keep track of previous reference addresses and associated strides for load and store instructions. The paper discusses an RPT implementation using a cache and the RPT has the following fields:

- **tag** – corresponds to the address of the load-store instruction
- **prev_addr** – The last operand address that was referenced when PC reached that instruction
- **stride** – the difference between the last 2 addresses that were generated when a state transition occurred from init to transient
- **state** – 2 bit encoding of the past history. It indicates how further preloading should be generated.

States of the State Machine:

- **Initial**: Set at first entry in the RPT or after the entry experienced an incorrect prediction from steady state.
- **Transient**: Corresponds to the case when the system is not sure whether the previous prediction was good or not. The new stride will be obtained by subtracting the previous address from the currently referenced address.
- **Steady**: Indicates that the prediction should be stable for a while.
- **No Prediction**: Disables pre-loading for this entry for the time being.

When the LA-PC encounters a load instruction, there are 2 mutually exclusive possibilities:

1. **No Action**: There is no corresponding entry in the RPT or there is an entry in the state no prediction.
2. **Potential Preload**: There is a corresponding entry. A block address (prev_addr + stride) is generated. If the block is uncached and the address is not found in the Outstanding Request List (ORL), a preload is initiated. This implies sending a request to the next level of memory hierarchy or buffering it if memory channel is busy. The address of the request is entered in the ORL.
When the PC encounters a load/store instruction with effective operand address \( addr \), the RPT is updated as follows: (The paper denotes correct by \( addr = prev\_addr + stride \) and incorrect by \( Addr \neq prev\_addr + stride \).)

1. There is no corresponding entry. The instruction is entered in the RPT. The \( prev\_addr \) field is set to the present address, the stride to 0 and the state to initial.
2. There is a corresponding entry about that instruction in the RPT:
   a. Transition –
      When incorrect and state=initial: Set \( prev\_addr \) to \( addr \), stride to \( (addr-prev\_addr) \) and state to transient.
   b. Moving to / Being in Steady State – When correct and (state = initial, transient or steady): Set \( prev\_addr \) to \( addr \), leave stride unchanged and set state to initial.
   c. Steady state is over: back to initialization – When incorrect and state = steady: Set \( prev\_addr \) to \( addr \), leave stride unchanged and set state to initial.
   d. Detection of irregular pattern – When incorrect and state = transient: Set \( prev\_addr \) to \( addr \), stride to \( (addr-prev\_addr) \) and state to no prediction.
   e. No prediction state is over; back to transient: When correct and state = no prediction: Set \( prev\_addr \) to \( addr \), leave stride unchanged, and set state to transient.
   f. Irregular pattern: When incorrect and state = no prediction: Set \( prev\_addr \) to \( addr \), stride to \( (addr-prev\_addr) \) and leave state unchanged.

The paper discusses the idea of having a look ahead distance less than \( d \) so that the cache pre-fetching does not overwrite useful unused data. The cache sizes are small for these simulations.
Branch Predictor:
There are 4 components of this Branch Predictor: Loop Counter, Optimized Geometric History Length Predictor, Branch Decision Table and Duplicate ALU. Each will be described in details in the following paragraphs.

The approach that we used to identify branch prediction strategy is as follows. There are 2 kinds of branches: conditional and unconditional. Unconditional branches like jump, jump and link, etc. are always taken and the branch target is always within the instruction as an offset to the Program Counter.

Loop Counter Branch Predictor:
For a conditional branch, there usually is a comparison between 2 registers and the third register contains the branch offset as an offset relative to the Program Counter. Here we initially assume that all conditional branches belong to loop counters and use a Loop Counter Branch Predictor on them. The Loop Counter branch predictor tracks all the registers tagged as being used by the branch instruction in a Duplicate ALU that does not have access to cache and main memory.

The Intel Core 2 Duo microprocessor has a loop counter branch predictor that tracks the loop for number of iterations on the first pass and uses this count for prediction on the second pass. This branch predictor will not give very high efficiencies on loops that have few iterations of long lengths. (Reference from Course Slides on Branch Predictor Design from University of Wisconsin.)

Figure 8: Algorithm for Loop Counter Branch Predictor for MIPS we designed
The O-GEHL Branch Predictor:
If the Loop Counter branch predictor is wrong twice at the same conditional branch address, that would imply that the register value used for the comparison is being updated dynamically by a value not in the register stack. In this case the Optimized Geometric History Length (O-GEHL) Branch Predictor [10] is used. The O-GEHL branch predictor is designed on the assumption that all conditional branches that don’t belong to loops will fit in one of the history lengths for a geometric progression of history lengths given by: \( L(i) = \alpha^{i+1} L(1) + constant \). A maximum history length of around 266 will be enough to cover all possible branch patterns.

In the O-GEHL branch predictor, each history length is indexed by a 4 bit counter that counts from -8 to +7. So a history length of 2 will have 2 4-bit counters, history length of 4 will have 4 4-bit counters and so on. The counter indices are incremented modulo the length of the history table on each branching decision. For each prediction, all current index counter values are added and compared to a threshold to decide which way to branch. By default, all counters are initialized to 0 giving the default prediction as taken. If there is a prediction error, the counter used for making that prediction is updated as follows: For a predicted taken branch which is in error, decrement the corresponding counter and for a predicted not taken branch which is in error, increment the counter. Each counter is a saturating counter and so its value will not go higher than +7 or less than -8. The counter sum for all the history lengths is computed using a tree adder to minimize the delay in computing the branch prediction and to give the LA-PC as much lookahead over the PC as possible.

- **History Lengths go in Geometric Progression given by** \( L(i) = \alpha^{i+1} L(1) + constant \)
- **Best Series found from experiments:** 2, 4, 9, 12, 18, 31, 54, 114, 145, 266
- **Dynamic History length fitting with variable \( \alpha \) also possible.

```
• j,k,l.. Are incremented on every unconditional branch.
• j increments are modulo 2, k increments are modulo 4, l increments are modulo 266.
• Each C(i)(j) is a 4 bit saturating counter that counts -8 to 7.
• Counter Update given by:
  if(branch==taken) c(i)(j)++
  if(branch!=taken) c(i)(j)--
• Dynamic Threshold (\( \theta \)) Fitting possible
• Threshold(\( \theta \)) by default is 0.
  Sum > \( \theta \) then p = taken
  Sum < \( \theta \) then p = not taken
```

Figure 9: Structure of O-GEHL branch predictor scaled down for our project.
The Branch Decision Table:
The Branch Decision Table is used to index the prediction and counter value for each branch address and also used to compare this against the actual decision of the Program Counter. The Look Ahead Program Counter for the Reference Prediction Table which does cache prefetching and the Look Ahead Program Counter for the instruction pipeline that prefetches to the pipeline both use this Branch Decision Table to make their branch predictions.

<table>
<thead>
<tr>
<th>Branch Address</th>
<th>Predicted Direction</th>
<th>Predicted Branch Target</th>
<th>Actual Direction</th>
<th>Actual Branch Target</th>
<th>Counters Used C(i)(j)</th>
<th>Tag</th>
<th>Counters Used C(i)(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entered by LA-PC</td>
<td>Entered by Loop Counter or O-GEHL</td>
<td>Entered by Duplicate ALU</td>
<td>Entered by PC</td>
<td>Entered by PC</td>
<td>Entered by O-GEHL</td>
<td>Entered by O-GEHL</td>
<td></td>
</tr>
</tbody>
</table>

if prediction ! = actual decision

Prediction computed by Loop Counter?
Yes – Incorrect Duplicate Register Values
Re-Initialize Duplicate Register Stack
Set LA-PC to PC
After 2 successive errors make an entry in O-GEHL
Also tag the branch address in Branch Decision Table to be used with O-GEHL

Prediction computed by O-GEHL?
Yes – Run the update equation on counters listed in table
Set LA-PC to PC

Figure 10: Branch Decision Table and decision structure to determine when to use O-GEHL.
**Duplicate ALU**

All of the decision making and arithmetic is done in a duplicate ALU which is a scaled down version of a MIPS ALU optimized for addition and subtraction. The figure below describes the Duplicate ALU that the Look Ahead Program Counter uses:

- If register flags set, do the computation for:
  - Op-Code: 0 bits(5:0) 32: add r1, r2, r3
  - Op-Code: 0 bits(5:0) 34: sub r1, r2, r3
  - Op-Code: 0 bits(5:0) 33: addu r1, r2, r3
  - Op-Code: 0 bits(5:0) 35: subu r1, r2, r3
  - Op-Code: 8: addi r1, constant
  - Op-Code: 9: addiu r1, constant

The Busy Bit of the Duplicate ALU is set after it fetches an instruction and it is reset after the required operation on the instruction is finished. The Op-Code and Function fields of the MIPS instructions are used to determine which operations to perform on the tagged registers and to determine the nature of the branch.

Each conditional branch instruction is referred to the Branch Decision Table and an appropriate Branch Predictor (Loop Counter or Optimized Geometric History Length) is used.
**Testing Methodology:**

The original plan was to test the Reference Prediction Table with the Loop Counter Branch Predictor in the first phase. Since this branch predictor is designed for highly regular loops, matrix multiplication is the example that we planned to use.

Instead, we did a VERILOG implementation of the Optimized Geometric History Length Branch Predictor with the Tree Adder to add 10 4-bit 2's complement numbers in the first phase. This is a standalone design not interfaced to an ALU. Testing this would require an ALU interface and an instruction stream in a binary format. The O-GEHL Branch Predictor will have to be tested along side the Loop Counter on the ALU. So a good strategy could be test for matrix multiplication first and then test for more complex programs.

The Verilog Project Files for the O-GEHL Branch Predictor are included in the attached .rar project file. The Verilog Design Files are all marked .v extension and can be opened in WORD to read.
References

(All references searched on GOOGLE )

1. The Microarchitecture of the INTEL Pentium 4 Microprocessor
INTEL Technology Journal, Q1, 2001
(Available on the course website for EE 5364)
2. The Microarchitecture of the INTEL Pentium 4 Microprocessor
INTEL Technology Journal, Q1, 2001
(Available on the course website for EE 5364)
3. Norman P. Jouppi and Steven J.E. Wilton, "Trade-offs in Two-level On-chip caching",
4. Digital Integrated Circuits by Jan Rabaey, Borivoje Nikolic and Anantha Chandrakasan
5. Increasing the Lookahead of Multilevel Branch Prediction
Alexander V. Veidenbaum
Dept. of Information and Computer Science
University of California Irvine
Copyright IEEE 1991 (Cannot find name of the publication. Available on GOOGLE.)
6. Instruction Cache Prefetching Using Multilevel Branch Prediction
Alexander V. Veidenbaum
Dept. Of Electrical Engineering and Computer Science
University of Illinois at Chicago
alexv@eecs.uic.edu
(Available on GOOGLE)
7. Decoupled Access/Execute Computer Architectures
James Smith, University of Wisconsin
8. Technology and Design Trade-Offs in the creation of Modern Super Computer
Lincoln N.R.
IEEE Transactions on Computers C-31, 5 May 1982, 349-362
9. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty
Jean-Loup Baer, Tien Fu Chen
Department of Computer Science and Engineering
University of Washington Seattle, WA 98195
In Supercomputing, pages 176-186
IEEE, November 1991
10. The O-GEHL Branch Predictor
Andre Seznec
The 1st JILP Championship Branch Prediction Competition CBP1 (2004)
Available from www.jilp.org