Exploring Perceptrons in Branch Prediction

February 28, 2002
Nick Deibel
Kevin Sikorski
Branch Prediction

- CPU speeds are increasing
- Pipeline lengths are increasing
- What to do with a branch:
  - Stall? 😞
  - Predict?
    - If correct: 😊
    - If not: 😞 😞 😞
Better Branch Prediction

- Current branch predictors do really well: 90+ percent accuracy
- How do they do it: Industry Secrets
- Is a 0.5% percent accuracy improvement really that helpful?

100,000 branches

⇒ 500 less pipeline flushes
⇒ More throughput
But to quote Mark…

- “It’s all a HACK!”
- But it’s an interesting hack for *Machine Learning* due to the requirement of High Accuracy with Low Cost
- Most machine learners can’t do this, except…

**PERCEPTRONS**
Perceptrons

- Simple model of a human neuron.
- Contains a weight vector $w$.
- Takes a vector $x$ of inputs.
- Outputs $\text{sgn}(x \cdot w)$
Perceptrons

- So how do we get the weights?
  - Use Machine Learning!
  - Perceptron Training Rule:
    \[ w_i \leftarrow w_i + \Delta w_i \]
    \[ \Delta w_i = \eta(t - o)x_i \]
  - Guaranteed to converge to an optimal weight vector within finite time if:
    - We use a small \( \eta \).
    - The dataset is linearly separable.
Linear Separability

- Perceptrons can only learn functions of the form: $w_1x_1 + w_2x_2 + \ldots + w_{n-1}x_{n-1} + w_n = 0$.
- Means that we must be able to divide classes of data using $n$-dimensional hyperplanes.
- Can still learn a lot of things:
  - AND, OR, NAND, NOR, NOT.
Linear Separability
Why use Perceptrons for Branch Prediction?

- Allows Dynamic Branch Prediction.
- Intrinsically robust to aliasing.
- Smaller hardware requirement than other AI techniques.
- Supply confidence values.
- Fast to train and predict.
  - Lots of multiplying by ±1 and adding
  - Lots of parallelism.
Previous Work: Jimenez & Lin, 2000

- First to adapt perceptrons to branch prediction.
- Simplified training rule: \( w_i \leftarrow w_i + t \cdot x_i \)
- Weight caps: \( \Theta = [1.93h+14] \)
- Can’t achieve 100% accuracy on linearly-inseparable branches.
  - Empirically, still do well on inseparable ones.
- Argue that prediction takes about 2 cycles on a 700MHz clock.
Jimenez and Lin also explored a gshare/perceptron hybrid predictor.
- Generally outperformed gshare or perceptron alone.

Found that some branches are best done with classical predictors.

Michaud and Seznec (2001) found that using a few bits from the branch address improves linear-separability.
How to do Branch Prediction with Perceptrons

- Hash the branch address to get an index into a table of perceptrons.
- Fetch the appropriate perceptron.
- Compute the branch prediction.
- Act on the prediction.
- Train the given perceptron on the outcome.
- Write the trained perceptron back to table.
Previous Implementation Approaches

- Tracing of SPEC Benchmarks:
  - Run a benchmark
  - Record each branch and outcome to a file
  - Feed this file into a predictor simulator
  - Compare performances for different predictors

- Pros:
  - Faster than a CPU simulator

- Cons:
  - Ignores speculative predictions and garbage history
Speculative Predictions and Garbage History

X and Y are branches. X is predicted taken.
Global History = 1001

<table>
<thead>
<tr>
<th>X’s Outcome</th>
<th>Global History</th>
<th>Correct History</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>0 1 0 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>NO</td>
<td>0 1 0 0</td>
<td>0 0 1 0</td>
</tr>
</tbody>
</table>
How to Deal with Speculation

- How do they do it in real processors?
  - They don’t. It’s too costly to fix.

- Doesn’t that affect how the predictor learns?
  - Yes, but these “errors” are consistent with its behavior.
Our Implementation

- Add a perceptron branch predictor to sim-alpha using the same design as the Jimenez paper
- Basic Configuration:
  - # of perceptrons
  - Size of the global history
  - Size of the local history
  - Threshold value on the weights
  - One input to every perceptron is always set to 1
A Bit about History Bits

- **Global History:**
  - A record of the last $n$ branches (1 = taken)
  - Shared by all perceptrons
  - Updated speculatively

- **Local History:**
  - IDEAL: A record of the last $m$ branches for a particular address
  - REALITY: A record of the last $m$ branches for a particular perceptron
  - Update speculatively
A Bit More about History Bits

- Gshare: After history length exceeds 10 bits, performance degrades
- Perceptrons: Performance increases with longer histories
- The Jimenez paper’s magic formula:

\[ \theta = \left\lfloor 1.93 \, h + 14 \right\rfloor \]
Hardware Tradeoffs

- Gshare and other predictors use a small amount of hardware: (usually 1024 2-bit SUD counters)
- Each perceptron must store its weights and its local history
- Compensation:
  - Keep the local history relatively small compared to the global history
  - Use less perceptrons
Benchmark Testing

- Currently using these SPEC2000 benchmarks:
  - CINT: vpr, gcc, parser, twolf
  - CFP: lucas

- Due to time concerns, using only the test inputs instead of the ref inputs
Our Experimental Method

- Compare the perceptron predictor’s performance to other predictors:
  - Always Taken Predictor
  - Gshare:
    - 1024 counters
    - Global history length: 8, 10, 16
  - 21264 Predictor:
    - Sim-alpha’s guess of how the 21264 really works
Perceptron Configurations

- # of Perceptrons: 512 vs 128
- Local History Size: 0, 5, and 10 bits
- Global History Size: 20 and 25 bits
- Threshold: with and without the magic formula
GCC Benchmark
VPR Benchmark
TWOLF Benchmark
## An Up Close Look at the Data

<table>
<thead>
<tr>
<th>GCC Benchmark</th>
<th>Direction Hits</th>
<th>Direction Misses</th>
<th>Total</th>
<th>Percentage Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptron: 512,0,20,52</td>
<td>238195317</td>
<td>79164520</td>
<td>317359837</td>
<td>75.06%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,52</td>
<td>265314537</td>
<td>52053250</td>
<td>317367787</td>
<td>83.60%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,62</td>
<td>265157819</td>
<td>52207845</td>
<td>317365664</td>
<td>83.55%</td>
</tr>
<tr>
<td>Perceptron: 512,0,25,62</td>
<td>238519623</td>
<td>78841970</td>
<td>317361593</td>
<td>75.16%</td>
</tr>
<tr>
<td>Perceptron: 128,0,20,52</td>
<td>232351139</td>
<td>85012059</td>
<td>317363198</td>
<td>73.21%</td>
</tr>
<tr>
<td>Perceptron: 128,5,20,52</td>
<td>253213011</td>
<td>64152929</td>
<td>317365940</td>
<td>79.79%</td>
</tr>
<tr>
<td>Perceptron: 128,5,20,62</td>
<td>253003396</td>
<td>64363919</td>
<td>317363715</td>
<td>79.72%</td>
</tr>
<tr>
<td>Gshare: 1,1024,10,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Gshare: 1,1024,8,1</td>
<td>259465549</td>
<td>57910011</td>
<td>317375560</td>
<td>81.75%</td>
</tr>
<tr>
<td>Gshare: 1,1024,16,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Always Taken</td>
<td>193367708</td>
<td>123988295</td>
<td>317356003</td>
<td>60.93%</td>
</tr>
<tr>
<td>21264</td>
<td>295950630</td>
<td>21422272</td>
<td>317372902</td>
<td>93.25%</td>
</tr>
</tbody>
</table>
## GCC Benchmark

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Direction Hits</th>
<th>Direction Misses</th>
<th>Total</th>
<th>Percentage Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptron: 512,0,20,52</td>
<td>238195317</td>
<td>79164520</td>
<td>317359837</td>
<td>75.06%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,52</td>
<td>265314537</td>
<td>52053250</td>
<td>317367787</td>
<td>83.60%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,62</td>
<td>265157819</td>
<td>52207845</td>
<td>317365664</td>
<td>83.55%</td>
</tr>
<tr>
<td>Perceptron: 512,0,25,62</td>
<td>238519623</td>
<td>78841970</td>
<td>317361593</td>
<td>75.16%</td>
</tr>
<tr>
<td>Perceptron: 512,10,20,81</td>
<td>272613439</td>
<td>44760254</td>
<td>317373693</td>
<td>85.90%</td>
</tr>
<tr>
<td>Perceptron: 128,0,20,52</td>
<td>232351139</td>
<td>85012059</td>
<td>317363198</td>
<td>73.21%</td>
</tr>
<tr>
<td>Perceptron: 128,5,20,52</td>
<td>253213011</td>
<td>64152929</td>
<td>317365940</td>
<td>79.79%</td>
</tr>
<tr>
<td>Perceptron: 12,5,20,62</td>
<td>253003396</td>
<td>64363919</td>
<td>317367315</td>
<td>79.72%</td>
</tr>
<tr>
<td>Perceptron: 128,0,25,62</td>
<td>232594537</td>
<td>84767405</td>
<td>317361942</td>
<td>73.29%</td>
</tr>
<tr>
<td>Perceptron: 128,10,20,81</td>
<td>258691841</td>
<td>58677654</td>
<td>317369495</td>
<td>81.51%</td>
</tr>
<tr>
<td>Gshare: 1,1024,10,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Gshare: 1,1024,8,1</td>
<td>259465549</td>
<td>57910011</td>
<td>317375560</td>
<td>81.75%</td>
</tr>
<tr>
<td>Gshare: 1,1024,16,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Always Taken</td>
<td>193367708</td>
<td>123982895</td>
<td>317356003</td>
<td>60.93%</td>
</tr>
<tr>
<td>21264</td>
<td>295950630</td>
<td>21422272</td>
<td>317372902</td>
<td>93.25%</td>
</tr>
</tbody>
</table>
## The Magic Formula Flops

### GCC Benchmark

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Direction Hits</th>
<th>Direction Misses</th>
<th>Total</th>
<th>Percentage Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptron: 512,0,20,52</td>
<td>238195317</td>
<td>79164520</td>
<td>317359837</td>
<td>75.06%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,52</td>
<td>265314537</td>
<td>52053250</td>
<td>317367787</td>
<td>83.60%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,62</td>
<td>265157819</td>
<td>52207845</td>
<td>317365664</td>
<td>83.55%</td>
</tr>
<tr>
<td>Perceptron: 512,0,25,62</td>
<td>238519623</td>
<td>78841970</td>
<td>317361598</td>
<td>73.21%</td>
</tr>
<tr>
<td>Perceptron: 128,0,20,81</td>
<td>272613439</td>
<td>44760254</td>
<td>317373693</td>
<td>85.90%</td>
</tr>
<tr>
<td>Perceptron: 128,0,20,52</td>
<td>232351139</td>
<td>85012059</td>
<td>317363198</td>
<td>73.21%</td>
</tr>
<tr>
<td>Perceptron: 128,5,20,52</td>
<td>253213011</td>
<td>64152929</td>
<td>317365940</td>
<td>79.79%</td>
</tr>
<tr>
<td>Perceptron: 12,5,20,62</td>
<td>253003396</td>
<td>64363919</td>
<td>317367315</td>
<td>79.72%</td>
</tr>
<tr>
<td>Perceptron: 128,0,25,62</td>
<td>232594537</td>
<td>84767405</td>
<td>317361942</td>
<td>73.29%</td>
</tr>
<tr>
<td>Perceptron: 128,10,20,81</td>
<td>258691841</td>
<td>58677654</td>
<td>317369495</td>
<td>81.51%</td>
</tr>
<tr>
<td>Gshare: 1,1024,10,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Gshare: 1,1024,8,1</td>
<td>259465549</td>
<td>57910011</td>
<td>317375560</td>
<td>81.75%</td>
</tr>
<tr>
<td>Gshare: 1,1024,16,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Always Taken</td>
<td>193367708</td>
<td>123988295</td>
<td>317356003</td>
<td>60.93%</td>
</tr>
<tr>
<td>21264</td>
<td>295950630</td>
<td>21422272</td>
<td>317372902</td>
<td>93.25%</td>
</tr>
</tbody>
</table>
### GCC Benchmark

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Direction Hits</th>
<th>Direction Misses</th>
<th>Total</th>
<th>Percentage Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptron: 512,0,20,52</td>
<td>238195317</td>
<td>79164520</td>
<td>317359837</td>
<td>75.06%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,52</td>
<td>265314537</td>
<td>52053250</td>
<td>317367787</td>
<td>83.60%</td>
</tr>
<tr>
<td>Perceptron: 512,5,20,62</td>
<td>265157819</td>
<td>52207845</td>
<td>317365664</td>
<td>83.55%</td>
</tr>
<tr>
<td>Perceptron: 512,0,25,62</td>
<td>238519623</td>
<td>78841970</td>
<td>317361593</td>
<td>75.16%</td>
</tr>
<tr>
<td>Perceptron: 128,0,20,52</td>
<td>232351139</td>
<td>85012059</td>
<td>317363198</td>
<td>73.21%</td>
</tr>
<tr>
<td>Perceptron: 128,5,20,52</td>
<td>253213011</td>
<td>64152929</td>
<td>317365940</td>
<td>79.79%</td>
</tr>
<tr>
<td>Perceptron: 12,5,20,62</td>
<td>253003396</td>
<td>64363919</td>
<td>317367315</td>
<td>79.72%</td>
</tr>
<tr>
<td>Perceptron: 128,0,25,62</td>
<td>232594537</td>
<td>84767405</td>
<td>317361942</td>
<td>73.29%</td>
</tr>
<tr>
<td>Perceptron: 128,10,20,81</td>
<td>258691841</td>
<td>58677654</td>
<td>317369495</td>
<td>81.51%</td>
</tr>
<tr>
<td>Gshare: 1,1024,10,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Gshare: 1,1024,8,1</td>
<td>259465549</td>
<td>57910011</td>
<td>317375560</td>
<td>81.75%</td>
</tr>
<tr>
<td>Gshare: 1,1024,16,1</td>
<td>256427183</td>
<td>60948583</td>
<td>317375766</td>
<td>80.80%</td>
</tr>
<tr>
<td>Always Taken</td>
<td>193367708</td>
<td>123988295</td>
<td>317356003</td>
<td>60.93%</td>
</tr>
<tr>
<td>21264</td>
<td>295950630</td>
<td>21422272</td>
<td>317372902</td>
<td>93.25%</td>
</tr>
</tbody>
</table>
Set-Associative Perceptron Tables

- Usually, we use a hash to index into a table of perceptrons.
  - This is exactly like indexing into a direct-mapped cache.
- Try applying 4-way set associativity to perceptron tables.
Set-Associative Perceptron Tables

- It is not immediately clear if associativity will be effective:
  - Set-Associativity is a tool for avoiding aliasing.
  - Perceptrons are already robust to aliasing.
  - Is there a better way to spend hardware budget?
  - What to do on a replace? Load a blank perceptron? Maintain a “common” perceptron? Do nothing?
  - Victim caching? L2 Cache?
Learning Rule Enhancements

- Imagine a human training a perceptron by hand - what would the human do?
  - Perceptron Mispredicted: Decrement the Weight.
  - Perceptron Mispredicted: Decrement the Weight.
  - Perceptron Mispredicted: Decrement the Weight.
  - Perceptron Mispredicted: Decrement the Weight.

- Human would cheat - If perceptron is way off, decrement by a larger number.

- Hopefully, this would speed convergence.
Learning Rule Enhancements: First Approach

First Approach:

- Employ a Saturating Counter.
- Set to zero on a correct prediction, increment on a misprediction.
- When we are saturated and mispredict, adjust weight by 2 instead of by 1.
Learning Rule Enhancements: Second Approach

- See how far off the perceptron is in predicting the outcome.
- If we are VERY far off, adjust weight by 2 instead of 1.

\[ \left\| (w \cdot x) - w_n \right\| \gg n \]
Learning Rule Enhancements: Risks

- Increased chance of oscillation.
- Increased hardware complexity.
- Good if we are in a tight loop. Bad if we aren’t.
Immediate Plans

- Implement and collect data on:
  - Set-associative predictor
    - With or without the common perceptron
    - With or without the victim cache
  - All designs with the advanced learning rule

- Determine what specific data to collect:
  - Dependence on # of perceptrons
  - Proper threshold values when using local history
  - Good sizes for the victim cache
And some future plans…

- Add all of our perceptron predictors as a configurable option in sim-alpha
- Potentially investigate:
  - Hybridizing our predictors with gshare
  - Using perceptrons as part of a tournament predictor like the Alpha 21264 predictor
  - The performance of a perceptron predictor in a multithreaded environment
And the Oracle says…

THE END