ECE 486 Final Project:
Branch Target Predictor

Andy Goetz & Kevin Riedl

April 20, 2013

1 Abstract

LaTeX *A branch target predictor was developed for an unknown ISA. Computer simulations using novel heuristics were used to evaluate thousands of possible cache hierarchies, in order to produce the optimal cache result. This cache had a target miss rate of 13.071 branches per thousand instructions, while the taken/not-taken predictor had a miss rate of 6.243 branches per thousand instructions.*

Keywords: Branch Prediction, Simulation, Heuristics, ISA, 0xBEEFA55

2 Acknowledgements

The authors would like to thank Tyler Tricker, Bradon Kanyid, and Eric Krause for the many hours of fruitful discussions. Additionally, they would like to thank Beth Krause for the delicious cookies provided.

3 Background Information

The purpose of this project was to develop a branch predictor for an unknown ISA. We were required to copy the branch predictor used in the Alpha 21264 processor. We were then required to design a branch target predictor, with the requirement that it use only 8 kilobytes of state. This branch predictor was then tested against an array of 20 test traces, in order to determine its performance.

4 Branch Predictor

The branch predictor used in this project was modeled after the one described in R. E. Kessler's paper on the Alpha 21264 processor. Figure 1 shows the diagram of our implementation of the alpha predictor. The biggest difference between the alpha predictor specified in the paper and our implementation is that ours does not speculatively load the path history with the predictions from the output of our alpha predictor.

The alpha predictor is what is known as a tournament predictor. This means that it implements more than one branch predictor and has a predictor-predictor to choose the most accurate branch predictor. This tournament predictor (otherwise known as the "Choice Predictor") is a 4096 entry table filled with 2-bit saturating counters that select between the global or the local predictor based on the history of the last 12 conditional branches (known as the "Path History")

The global predictor is simply a 4096 entry table of 2-bit saturating counters indexed by the path history (similar to the choice predictor) that determine whether or not a branch is taken.

The local predictor is 1024 entry table of local histories indexed by the least significant 10-bits of the program counter. This history is then used as an index into a 1024 entry table of 3-bit saturating counters which determines if a branch is taken or not.

The overall design takes advantage of local history when a series of branches tend to stay within a limited address space and takes advantage of the global history when a series of branches tend to use a larger address space.

Since the alpha paper does not specify a few details about how the predictor works, liberties were taken in determining the best prediction values.

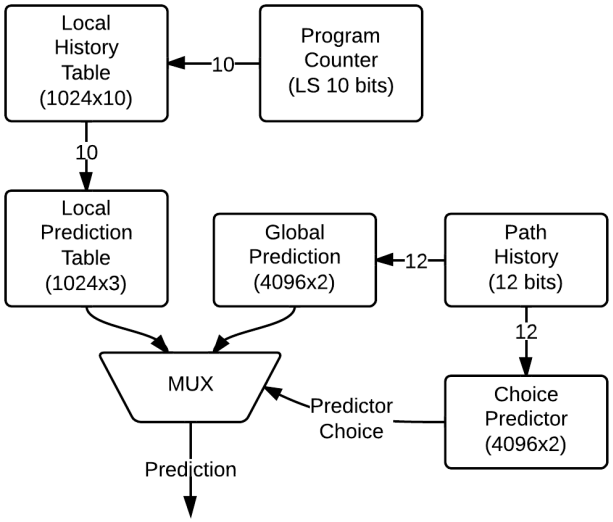

Figure 1: Alpha Predictor

4.a Initialized Values

The Alpha paper did not specify which values to initialize the tables with. Fortunately, once many branches have occurred, the initial values do not matter when looking at a system which processes billions of branches between power downs. For the relatively small trace data set used in these simulations however, the initialization values were important for determining more accurate branch prediction. For example, Changing the Global Predictor from weakly taken to weakly not taken changes the average $\frac{\#\,of\,mispredicts}{1000\,\text{instructions}}$ from 14.10950 to 14.07025 respectively, which is a significant performance improvement.

4.b Unconditional Branch Inclusion in the Path History

The path history is important in determining which predictor to use in the case of the choice predictor and it is important in determining which saturating counter to use in the case of the global predictor. The alpha predictor described in the paper did not specify whether or not unconditional branches are included in the path history (i.e. jumps), but could change the results of these predictors significantly. Testing both possibilities determined that the exclusion of the unconditional branches yielded more accurate branch prediction, thus this implementation is using a path history that ignores unconditonal branches.

5 Branch Target Predictor

In order to be truly successful, a branch predictor must be carefully tuned to its target workload. The fact that the set of all traces our branch predictor would be tested against was known ahead of time presented us with an unbridled opportunity for "benchmarketing", however, we still needed to gather information about the traces, in order to design an effective branch target predictor.

5.a Displacement Cache

Figure 2 shows the ECDF (Empirical Cumulative Distribution Function) of the first four benchmarks. As can be seen from this graph, most branches have destinations that are relatively close to the source program counter. From this, it follows that a branch target buffer that only stores the displacement to branch destination could achieve close to the same performance as a cache with direct-mapped branch destinations, while consuming less memory.

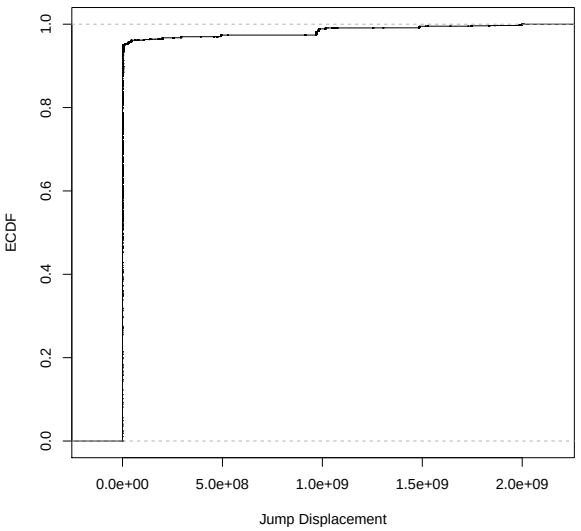

Figure 2: CDF of Branch Displacement Size

Table 1 shows percentage of the branch targets that could fit in a buffer entry of 4, 8, 12, 16, and 24 bits. As can be seen by the table, while a larger displacement field can store a greater percent of the branch targets, even a 24 bit entry can only hold 95% of branch targets. However, an 8 bit entry can store over half of the possible branch targets. This suggests a hybrid approach: use two caches, one that only

Entry Size (bits)	Storeable Branches
4	3.5%
8	51.9%
12	66.3%
16	81.8%
24	95.1%

Table 1: Displacement Cache Capacity

holds branches that can fit in a small displacement, and another that stores branch targets that are too large for the displacement field. This allows us to get performance that is close to that of a larger cache, while using less space. A 64 entry by 10 way cache uses 77056 bits, while a combination of a 8 way main cache and 2 way displacement cache only uses 30720 bits, and achieves similar performance.

5.b Return Address Stack

Another way that we sought to increase branch target performance was by using a Return Address Stack (RAS). Figure 3 shows the distribution of branch types in four of the benchmark programs. As can be seen from the graph, subroutine returns make up a not insubstantial part of each trace file, especially the server benchmarks.

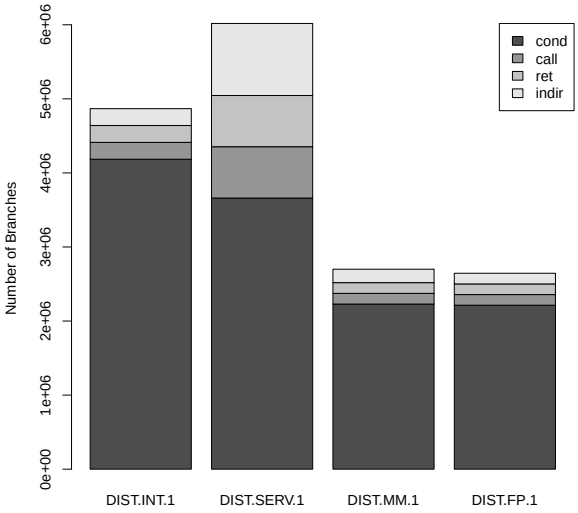

Figure 3: Branch Type Distribution

The size of the return address stack is important. Figure 4 shows that the test traces have widely varying maximum depths. There is a diminishing return on creating a larger RAS, but the relationship between stack size and predictor performance is not intuitive.

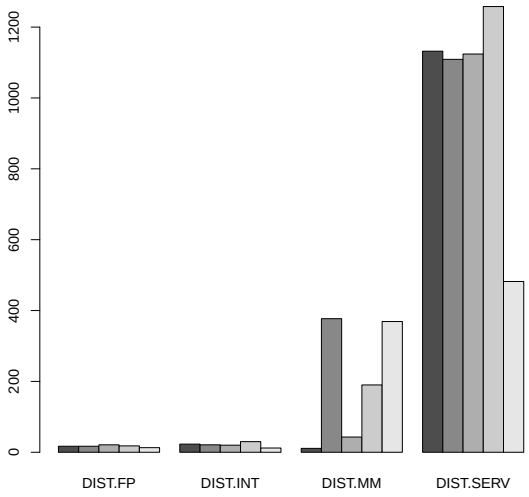

Figure 4: Maximum Callstack Depth per Trace

5.c Target Predictor Implementation

The branch target predictor used in the simulator can be seen in figure 5. It is based on a hierarchy of two separate caches, combined with a return address stack. A small displacement cache (64 entries by 2 ways) holds entries whose target address is less than 128 bytes from the program counter. If a target address has a displacement that is too large to fit in this cache, or an address is evicted from the displacement cache, it is placed in another, larger cache that contains direct-mapped addresses. This larger cache is organized as 64 entries by 8 ways.

The branch target predictor also contains a return address stack. This stack stores the return address of the last 21 calls. This allows return addresses to be predicted, regardless of whether or not they are in the cache. In addition to storing return addresses in the RAS, return addresses are also placed in the displacement and direct caches.

6 Size Budget

As mentioned above, we were limited to 8 kilobytes of storage space. A description of how this space was

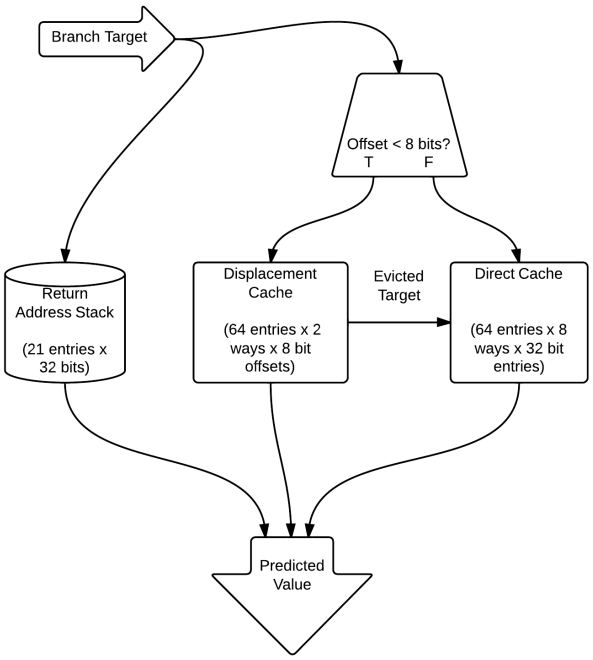

Figure 5: Branch Target Predictor

Alpha Predictor	
Local History Tables	10240
Local Predictor Counters	3072
Global History Counters	8192
Choice Predictors	8192
Global History	12
Subtotal:	29708
Branch Target Buffer	
Displacement Cache	4352
Direct Cache	30720
RAS	682
Subtotal:	35754
Total:	64462

Table 2: Space Budget in Bits

used can be found in table 2.

7 Testing Methodology

In order to determine the optimal branch predictor, a generic predictor framework was designed that used environment variables to determine the cache hierarchy used by the branch target predictor. The tunable parameters can be seen in table 3.

In addition, a replacement predictor framework was written using plaintext tracefiles, allowing for much faster traces, as well as more detailed predictor statistics. These statistics included a breakdown of the percentage of misses that were caused by function calls, conditional branches, and indirect branches, among other values. Also, an oracle predictor was developed. This oracle was capable of perfect prediction of a given branch's taken/not-taken status, which allowed us to evaluate branch target buffer performance without being impacted by design decisions in the alpha predictor.

The decision was made to test the solution space exhaustively. The tuneable parameters on our cache meant that there were over 400,000 unique cache hierarchies possible. This was far too many to test in the time alotted. However, we were able to use heuristics to reduce the size of the solution space. Figure 6 shows the miss rate for 2500 random cache

Variable Name	Description
`BTB_MAIN_SIZE`	Index bits of direct cache
`BTB_MAIN_WAYS`	Number of ways of direct cache
`BTB_DISP_ENTRIES`	Size of entry for displacement cache
`BTB_DISP_SIZE`	Index bits of displacement cache
`BTB_DISP_WAYS`	Number of ways of displacement cache
`BTB_WAY_ALGO`	Way Eviction Policy (LRU or Round Robin)
`BTB_FUNC_CAP`	Number of entries in RAS

Table 3: BTB Tunable Parameters

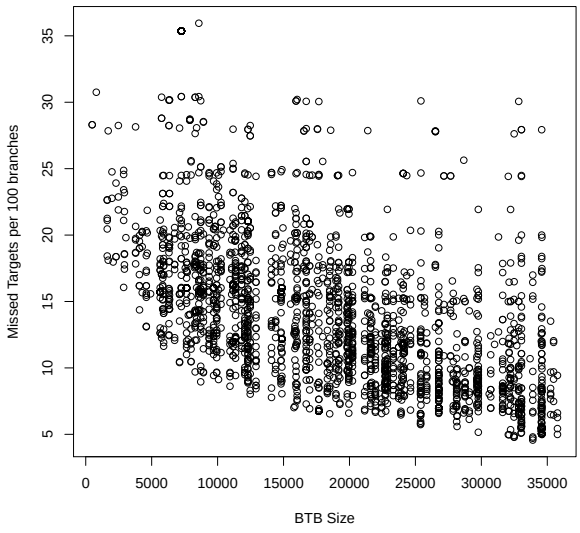

Figure 6: Performance of Random Sized Target Buffers

Trace	*bpredict*	*tpredict*
DIST-FP-1	3.363	3.376
DIST-FP-2	1.321	1.339
DIST-FP-3	0.523	0.549
DIST-FP-4	0.267	0.290
DIST-FP-5	2.474	2.483
DIST-INT-1	8.598	8.622
DIST-INT-2	11.868	16.260
DIST-INT-3	12.618	12.952
DIST-INT-4	2.896	3.460
DIST-INT-5	0.477	0.666
DIST-MM-1	8.906	8.923
DIST-MM-2	10.830	11.176
DIST-MM-3	1.565	9.102
DIST-MM-4	2.150	2.929
DIST-MM-5	6.447	13.853
DIST-SERV-1	9.781	34.708
DIST-SERV-2	10.144	34.715
DIST-SERV-3	8.145	18.377
DIST-SERV-4	10.944	37.397
DIST-SERV-5	11.542	40.241
Total:	6.243	13.071

Table 4: Predict Rates for Individual Traces

hiearchies. This graph clearly shows that, in general "Biggest Cache is Best Cache." That, is the best performing caches tend to be the largest. Based on this data, we decided to limit our search to cache organizations that consumed at least 30,000 bits of storage space. Using the MCECS computer labs as an impromptu server farm, we were able to determine to optimal cache hierarchy in less than 3 days, using approximately 1200 computer-hours.

8 Results

Optimizing the cache hierarchy using a server farm paid off admirably. We were able to achive a rate of 6.243 missed predicts per thousand instructions, and 13.071 missed targets per thousand instructions. Individual results are detailed in table 4.

The optimal cache hierarchy we found was surprising. Initially, we thought that the optimal target predictor design would have a much larger RAS. Figure 4 shows that the server traces routinely have stack depths that range into the hundreds. There is no way that a 21 entry return address stack could hold the entire callstack. However, this hierarchy produced the best performing target predictor. Also, we originally thought that a larger displacement cache would result in better hit rates. However, the best predictor topology only had 128 entries.

9 Conclusion

The branch target predictor we developed provided very high accuracy, considering the (comparative) inaccuracy of the alpha predictor. Our initial assumption that limiting the total size of the whole design to less than 8 kilobytes would significantly limit the accuracy of our target predictor proved false. As it turned out, the limited size still provided frequently accurate predictions. Exhaustive testing (as seen in the previous sections) provided results that yielded a branch target predictor design that had a higher prediction accuracy than we had initially anticipated. However, this solution is not scaleable. Exhaustive testing beyond 8 kilobytes will not scale well considering the large amount of time it took to calculate the branch target predictor configurations that yielded the highest accuracy. A better way to determine the most accurate configurations would be with meta-heuristics which would more intelligently (and thus more quickly) determine the most accurate branch target predictor configurations.

Predictor.cc

```
#include <stdint.h>
#include "predictor.h"
#include <cassert>
#include <stdlib.h>
#include <map>
#include <stdint.h>
#include <stdlib.h>
#include <vector>
#include <deque>

// Table size values for the Alpha Predictor
//local history table
#define LHT_SIZE 1024
//local prediction table
#define LPT_SIZE 1024
//global prediction table
#define GPT_SIZE 4096
//choice predictor table
#define CPT_SIZE 4096

// Global variables used in multiple files:

static int util_debugmode = 0;  // Whether or not BTB_DEBUG is set
static FILE *oraclefd = NULL;   // The file descriptor of the oracle
                                // file, if it exists
static std::map<uint,uint> addr_hist; // hash table of branch
                                      // targets. Used in the oracle
                                      // predictor

// macro to print debug statement. If debug mode is turned on...
#define debug(...) do {if(util_debugmode)fprintf(stderr, __VA_ARGS__);} while(0)

// implements ceil(log2(x)). Used to calculated table sizes ONLY
int log2(int value)
{
        value--;
        if (value == 0)
                return 1;
        for(int i = 32; i >= 1; i--) {
                if(0x80000000 & value)
                        return i;
                value = value << 1;
        }
        return 0;
}

// Simple factorial function, used to number of LRU bits needed in
// cache
int factorial(int n){
        if(n <= 1)
                return 1;
        else
```

```
                return n*factorial(n-1);
}

// Helper function to get the value of an environment variable, in
// order to set a parameter of the framework
void getparam(const char* name, int *value)
{
        char* tmpval = NULL;
        tmpval = getenv(name);
        if (tmpval)
                *value = atoi(tmpval);
}

// Enum describing two possible cache-replacement strategies
// implemented.
enum WayAlg_t {WAY_RROBIN = 0, WAY_LRU = 1};


// picks which way to replace, based on either lru or round robin. The
// LRU implementation uses counters, but sizes itself with the FSM
// implementation in order to reduce code size
class ReplacementPolicy {
private:
        int numways;            // the number of ways in the cache
        WayAlg_t algorithm;     // which replacement algorithm to use
        int *counters;          // pointer to counter object
public:
        // constructor. must provide the number of ways in the cache
        ReplacementPolicy(int numways = 1, WayAlg_t algo = WAY_LRU)
                : numways(numways), algorithm(algo),
                  counters(new int[numways]) {
                assert(numways > 0);
                if (algorithm > WAY_LRU)
                        algorithm = WAY_LRU;
                else if (algorithm < WAY_RROBIN)
                        algorithm = WAY_RROBIN;
                for(int i = 0; i < numways; i++)
                        counters[i] = i;
        }
        // copy constructor
        ReplacementPolicy(const ReplacementPolicy& other)
                : numways(other.numways), algorithm(other.algorithm),
                  counters(new int[other.numways]){
                assert(numways > 0);
                for(int i = 0; i < numways; i++ )
                        counters[i] = other.counters[i];

        }

        // copy assignment
        ReplacementPolicy & operator= (const ReplacementPolicy & other) {
                if(this != &other)
                {
                        numways = other.numways;
```

```
                assert(numways > 0);
                algorithm = other.algorithm;
                delete[] counters;
                counters = new int[numways];
                for(int i = 0; i < numways; i++)
                        counters[i] = other.counters[i];
        }
        return *this;
}

// destructor
~ReplacementPolicy() {
        delete [] counters;
}

// replacement function. Called when a way needs to be
// evicted. It returns the number of the way that has been
// evicted, and updates its current state.
int replace() {
        assert(numways > 0);
        // If we are using round robin, eviction is
        // easy. return the current value of the counter, and
        // increment to the next victim.
        if(algorithm == WAY_RROBIN) {
                int choice = counters[0];
                counters[0]++;
                counters[0] %= numways;
                return choice;

        } // If we are using LRU, it is a bit more
          // complicated. We need to find the way that has the
          // highest counter value. (equal to numways-1). We
          // return this way, and mark that way as LRU.
        else {
                for(int i = 0; i < numways; i++) {
                        if(counters[i] == numways-1) {
                                update(i);
                                return i;
                        }
                }
                // we should never get here.
                assert(0);
        }
}

// this function is used to update the LRU way.
void update(int way_used) {
        assert(numways > 0);
        // uses counter method: all ways with a counter value
        // smaller than the counter value of the LRU way are
        // incremented, and the LRU was has its counter set to
        // zero
        if(algorithm == WAY_LRU) {
                int oldval = counters[way_used];
```

```
                        for(int i = 0; i < numways; i++)
                                if (counters[i] < oldval)
                                        counters[i]++;
                        counters[way_used] = 0;
                }
        }
        // return the size of this waypicker
        int size() {
                if (algorithm == WAY_RROBIN)
                        return log2(numways);
                else
                        return log2(factorial(numways));
        }
};

// Helper class to implement a RAS
class ReturnAddressStack {
private:
        std::deque<uint> callqueue; // used to store RAS
        int capacity;               // size of RAS in entries

public:
        uint maxsize;           // counter indicating deepest stack seen
        uint call_overflow;     // incremented every time we try to
                                // add a call to the stack, and it is
                                // full.
        uint ret_underflow;     // incremented every time we try to
                                // return a return address from the
                                // RAS, and it is empty

        // constructor
        ReturnAddressStack()
                : capacity(21), maxsize(0), call_overflow(0), ret_underflow(0) {
                // get the size of the cache from an environment
                // variable
                getparam("BTB_FUNC_CAP", (int*)&capacity);
        }

        // if a call happens, the address of the next instruction is
        // passed to this function.
        void call(uint addr) {
                // if there is no RAS
                if (capacity == 0) {
                        call_overflow++;
                        return;
                }
                // always push the new value on the top of the stack
                callqueue.push_front(addr);
                uint size = callqueue.size();
                // if we are at capacity, we need to remove the oldest entry
                if(capacity > 0 && (int)size > capacity) {
                        callqueue.pop_back();
                        call_overflow++;
                }
```

```
                if(size > maxsize)
                        maxsize = size;
        }

        // called when we need to fetch the return address. If the
        // stack is empty, returns false
        bool ret(uint * addr) {
                if(callqueue.size() > 0) {
                        *addr = callqueue.front();
                        callqueue.pop_front();
                        return true;
                }
                ret_underflow++;
                return false;

        }

        // returns the size of the stack in bits
        int size() {
                // each entry is 32 bit, and we need 2 pointers, one
                // to the top of the stack, and a count of the number
                // of elements in the stack.
                if (capacity > 0)
                        return 32*capacity + 2*log2(capacity);
                else if (capacity < 0)
                        return -1;
                else
                        return 0;
        }
};

// This class implements a branch target buffer that can store
// elements in an arbitrary number of ways,
class BTB_CACHE {
private:
        int indexbits;          // the number of bits of the cache
                                // that are index bits
        size_t numways; // the number of ways in the cache
        size_t btbsize; // The size of the btb (numways * 2^indexbits)
        size_t tagsize; // number of bits that represent the tag

        int m_displacementbits;// the number of displacement bits
                                // stored in the cache. If (dest - PC
                                // > 2^displacementbits), the target
                                // cannot fit in this cache
        uint* targets;  // pointer to target address buffer
        uint* tags;             // pointer to tag bit buffer
        std::vector<ReplacementPolicy> policy_state; // vector of state bits for
                                        // way eviction policy


public:
        uint nummissed;         // The number of branch targets that
                                // were too large to fit in this
```

```
                        // cache: kept for statistical
                        // purposes

// returns the number of displacement bits this cache can store
int displacementbits() {
        if (m_displacementbits < 0 || m_displacementbits >= 32)
                return 32;
        else
                return m_displacementbits;
}

// simple constructor used to represent an cache with zero entries
BTB_CACHE()
        : indexbits(-1) {
}
// constructor
BTB_CACHE(int indexbits, int numways = 1,
          int displacementbits = -1, WayAlg_t way_algo = WAY_LRU)
        : indexbits(indexbits), numways(numways),
          btbsize(1 << indexbits),
          tagsize(32-indexbits),
          m_displacementbits(displacementbits),
          policy_state(btbsize, ReplacementPolicy(numways, way_algo)),
          nummissed(0){
        targets = (uint*)malloc(btbsize*numways*sizeof(uint));
        tags = (uint*)malloc(btbsize*numways*sizeof(uint));

        for(uint i = 0; i < btbsize*numways; i++) {
                tags[i] = 0xffffffff;
        }
}
// deconstructor
~BTB_CACHE() {
        if (indexbits < 0)
                return;
        free(targets);
        free(tags);
}
bool predict(uint instr, uint &target) {

        if(indexbits < 0)
                return false;

        uint index = instr & ((1 << indexbits) - 1);
        uint shiftedindex = index * numways;
        uint tag = instr >> indexbits;

        for(uint i = 0; i < numways; i++) {
                if(tags[shiftedindex+i] == tag) {
                        target =  targets[shiftedindex+i];
                        return true;
                }
        }
        return false;
```

```
}
bool update(uint addr, uint target,
            uint &evicted_addr, uint &evicted_target) {

        if(indexbits < 0)
                return false;

        uint index = addr & ((1 << indexbits) - 1);
        uint shiftedindex = index * numways;
        uint tag = addr >> indexbits;

        // need to check that the displacement can fit in the
        // displacement field
        if (displacementbits() != 32) {
                uint delta = target - addr;
                int64_t maxdisp =
                        ((int64_t)1 << (m_displacementbits - 1)) - 1;
                int64_t mindisp = -maxdisp - 1;
                if (delta > maxdisp || delta < mindisp) {
                        nummissed++;
                        return false;
                }
        }

        for (uint i = 0; i < numways; i++) {
                if (tags[shiftedindex+i] == tag){
                        policy_state[index].update(i);
                        return true;
                }
        }

        // insert into table
        // save evicted value
        uint way = policy_state[index].replace();

        // if the way was evicted
        if(targets[shiftedindex + way] != 0xffffffff) {
                evicted_addr = index;
                evicted_addr &= (1 << indexbits) - 1;
                evicted_addr |= tags[shiftedindex + way] << indexbits;
                evicted_target = targets[shiftedindex + way];
        }

        // insert new brach target into this cache instance
        tags[shiftedindex + way] = tag;
        targets[shiftedindex+ way] = target;

        return true;
}

// update function if eviction is irrelevant
bool update(uint addr, uint target) {
        uint evicted_a;
        uint evicted_t;
```

```
                return update(addr, target, evicted_a, evicted_t);
        }

        // returns size of the cache
        uint size(void) {

                if(indexbits < 0)
                        return 0;
                uint size = 0;
                size += btbsize * numways * tagsize;
                size += btbsize * numways* displacementbits();
                if(numways > 1) {
                        size += btbsize*policy_state[0].size();
                }
                if(numways > 8)
                        size *=2;
                return size;
        }
};


BTB_CACHE *maincache;
BTB_CACHE *dispcache;
static ReturnAddressStack rastack;
uint btb_predict(const branch_record_c *br)
{
        uint target;
        // check the displacement cache first
        if(!dispcache->predict(br->instruction_addr, target))
                // If it wasn't in the displacement cache, check the main cache
                if(!maincache->predict(br->instruction_addr, target))
                        // if we still failed, we do not have it in
                        // the cache. Return the next address instead.
                        target = br->instruction_next_addr;

        if(br->is_return) {
                if(rastack.ret(&target))
                        return target;
        }

        return target;
}

void btb_update(const branch_record_c *br, uint actual_addr)
{

        uint evicted_addr = 0;
        uint evicted_target = 0;
        if(br->is_call)
                rastack.call(br->instruction_next_addr);

        // if we cannot place the address in the displacement cache,
        // place it in the main cache (hierarchical caches!)
        if(!dispcache->update(br->instruction_addr, actual_addr,
```

```
                                evicted_addr, evicted_target)) {
                maincache->update(br->instruction_addr, actual_addr);
        }
        // if an address was evicted from the displacement cache,
        //put it in the main cache
        if(evicted_addr != 0){
                maincache->update(evicted_addr,evicted_target);
        }

}

// setup and destroy functions for the btb predictor
void btb_setup(void)
{
        //number of bits devoted to index
        int main_size = 6;
        //number of ways in the main cache
        int main_ways = 8;
        //how many bits of displacement are stored in the displacement cache
        int disp_entries = 8;
        //number of bits devoted to the index of the displacement cache
        int disp_size = 6;
        //number of ways in the displacement cache
        int disp_ways = 2;

        WayAlg_t way_algo = WAY_LRU;
        getparam("BTB_MAIN_SIZE", &main_size);
        getparam("BTB_MAIN_WAYS", &main_ways);
        getparam("BTB_DISP_ENTRIES", &disp_entries);
        getparam("BTB_DISP_SIZE", &disp_size);
        getparam("BTB_DISP_WAYS", &disp_ways);
        getparam("BTB_WAY_ALGO", (int*)&way_algo);

        //initialize the main cache
        maincache = new BTB_CACHE(main_size, main_ways, -1, way_algo);

        if (disp_size >= 0)
                dispcache = new BTB_CACHE(disp_size, disp_ways,
                                          disp_entries, way_algo);
        else
                dispcache = new BTB_CACHE();

        debug("Way Algo: ");

        //print information about eviction policy of both caches
        if(way_algo == WAY_RROBIN)
                debug("Round Robin\n");
        else
                debug("LRU\n");

        //prints main cache geometry information
        debug("Main: %d entries by %d ways, %d bit displacements\n",
                1 << main_size, main_ways, maincache->displacementbits());
```

```
        //print displacement cache geometry information
        if(disp_size >= 0)
        debug("Displacement: %d entries by %d ways, %d bit displacements\n",
                1 << disp_size, disp_ways, dispcache->displacementbits());
        else
                debug("No displacement cache.\n");

        //prints size of the caches
        debug("BTB size: %d\n",dispcache->size() +
              maincache->size() + rastack.size());

}

void btb_destroy(void)
{
        debug("Unable to cache %d branch targets.\n", dispcache->nummissed);
        debug("max func stack size: %d\n", rastack.maxsize);
        debug("call overflow: %d\n", rastack.call_overflow);
        debug("ret underflow: %d\n", rastack.ret_underflow);
        delete maincache;
        delete dispcache;
}

// These functions are called once at the begining, and once at the
// end of the trace
static void on_exit(void);
static void init(void);

//Local history table, only the least significant 10 bits will be used
unsigned short local_hist_table[LHT_SIZE];

//Local prediction bits for the saturated counter for the local branch predictor
//Only uses least significant 3 bits
unsigned char local_predict[LPT_SIZE];

//Global prediction bits for the sturated counter for the global
//branch predictor Only uses least significant 2 bits
unsigned char global_predict[GPT_SIZE];

//Choice prediction bits for the saturated counter that chooses the
//predictor that will be used to do the predicting. Only uses least
//significant 2 bits
unsigned char choice_predict[CPT_SIZE];

//Path history stores the history of the last 12 branches.
//Only the least significant 12 bits are used, with the lsb
//being the most recent branch
unsigned short path_history;

// helper function to increment a saturated counter
void inc_cnt(unsigned char &counter, uint size) {
        if (counter < (unsigned char)((1 << size) - 1))
                counter++;
}
```

```
// helper function to decrement a saturated counter
void dec_cnt(unsigned char &counter, uint size) {
        if (counter > 0)
                counter--;
}
//predicts taken/not taken branch based on local history
bool alpha_local_predict(const branch_record_c *br)
{
        //grabs bits 0-9 of the PC to index the local history table
        unsigned int PC = ((br->instruction_addr) & 0x3FF);

        //grabs the history value stored in the table
        unsigned int history = local_hist_table[PC];

        //predict true if the counter is more than 4
        //predict false if the counter is less than 4
        //i.e. (weakly|strongly) taken
        if(local_predict[history] >= 4)
                return true;

        else
                return false;

}

//predicts taken/not taken branch based on global history
bool alpha_global_predict(const branch_record_c *br)
{
        if(global_predict[path_history] >= 2)
                return true;

        else
                return false;
}

//This is the predictor predictor AKA the Choice predictor
//uses the path history to determine which predictor to use
bool alpha_predict(const branch_record_c *br)
{
        bool taken;

        //Choice Predictor:
        //Use the path history to index into the table of saturating counters
        //Use the Global Predictor if the counter is more than 2
        if(choice_predict[path_history] >= 2)
        {
                taken = alpha_global_predict(br);
        }
        //Use the Local Predictor if the counter is less than 2
        else
        {
                taken = alpha_local_predict(br);
        }
```

```
        return taken;
}

//updates the path history to reflect whether the last branch
//was actually taken
void alpha_update(const branch_record_c *br, bool taken)
{

        unsigned int PC = (br->instruction_addr) & 0x3FF;
        bool g_correct = (alpha_global_predict(br) == taken);
        bool l_correct = (alpha_local_predict(br) == taken);

        // update choice predictor
        if (!g_correct && !l_correct) {
                // do nothing
        } else if (!g_correct && l_correct) {
                dec_cnt(choice_predict[path_history], 2);
        } else if (g_correct && !l_correct) {
                inc_cnt(choice_predict[path_history], 2);
        } else if (g_correct && l_correct) {
                // do nothing
        }

        //Global Predictor:
        //updates the global predictor saturated counter
        if(taken)
                inc_cnt(global_predict[path_history], 2);

        else
                dec_cnt(global_predict[path_history], 2);

        //Local Predictor:
        //updates the local predictor saturated counter
        if(taken)
                inc_cnt(local_predict[local_hist_table[PC]], 3);

        else
                dec_cnt(local_predict[local_hist_table[PC]], 3);

        //Path History:
        //shift left by one and mask off the last 12 bits
        //so that any bits above the 12th will be zero
        path_history = (path_history << 1) & 0xFFF;

        if(taken)
                path_history++;

        //update the local history table with the newest history
        unsigned short &local_hist = local_hist_table[PC];
        local_hist = local_hist << 1;
        local_hist += taken ? 1: 0;
        local_hist &= 0x3ff;
}
```

```
void alpha_setup(void)
{
        int i = 0;

        //initialize the tables for the local predictor
        for(i = 0; i < LHT_SIZE; i++)
        {
                local_hist_table[i] = 0;
        }

        for(i = 0; i < LPT_SIZE; i++)
        {
                local_predict[i] = 0b011;
        }

        //initialize the global prediction table
        for(i = 0; i < GPT_SIZE; i++)
        {
                //sets the default global prediction to weakly not taken
                global_predict[i] = 0b01;
        }

        //initialize the choice prediction table
        for(i = 0; i < CPT_SIZE; i++)
        {
                //sets the default choice prediction to weakly taken
                choice_predict[i] = 0b10;
        }

        //initializes the path history to all not taken by default
        path_history = 0;

}

// This function is called once when the program exits. It is used to
// tear down any data structures used by the alpha predictor.
void alpha_destroy(void)
{
}

bool PREDICTOR::get_prediction(
        const branch_record_c* br,
        const op_state_c* os,
        uint *predicted_target_address)
{
        // need to only run this code once.
        static int initial_run = 1;
        if(initial_run){
                initial_run = 0;
                init();
        }
```

```
        bool taken = true;

        if(oraclefd == NULL) {
                if (br->is_conditional)
                        taken = alpha_predict(br);
        } else {
                uint instr_addr = 0xbeefa55;
                uint next_addr = 0xbeefa55;
                uint actual_addr = 0xbeefa55;
                uint status = 0x3f;
                fscanf(oraclefd, "%08x%08x%08x%02x\n",
                        &instr_addr,
                        &next_addr,
                        &actual_addr,
                        &status);
                assert(instr_addr == br->instruction_addr);
                taken = status & 0x1;
                // if it is not in the table yet
                if(addr_hist.count(br->instruction_addr)) {
                        *predicted_target_address = addr_hist[instr_addr];
                        addr_hist[instr_addr] = actual_addr;
                        //                      return taken;
                } else {
                        *predicted_target_address = br->instruction_next_addr;
                        addr_hist[instr_addr] = actual_addr;
                        //                      return false;
                }
        }

        *predicted_target_address = btb_predict(br);

        return taken;
}

// Update the predictor after a prediction has been made.  This should accept
// the branch record (br) and architectural state (os), as well as a third
// argument (taken) indicating whether or not the branch was taken.
void PREDICTOR::update_predictor(
        const branch_record_c* br,
        const op_state_c* os, bool taken, uint actual_target_address)
{

        // Update the BTB predictor
        btb_update(br, actual_target_address);

        // Update the Alpha predictor
        if (br->is_conditional)
                alpha_update(br, taken);
}

// This function is called at the end of the framework. It is used to
// print out cumulative statistics information in more detail than is
// captured by the benchmark.
static void on_exit(void)
```

```
{
        // if we are using an oracle, close the pipe.
        if (oraclefd)
                pclose(oraclefd);

        // run the  destroy functions:
        btb_destroy();
        alpha_destroy();
}

// This function is only called once. It is used to set up some global
// variables for the branch target predictor.
static void init(void)
{
        // determine if we should print debug messages...
        getparam("BTB_DEBUG", &util_debugmode);

        // If we decided to use an ORACLE, hook the oracle now.
        char* oraclefile = getenv("ORACLE");
        char  oraclecmd[1024];
        sprintf(oraclecmd, "xzcat %s",oraclefile);
        if (oraclefile){
                oraclefd = popen(oraclecmd, "r");
                debug("Hooking ORACLE... %s\n", oraclefile);
        }

        // hook the exit function, so that data structures can be
        // cleaned up.
        atexit(on_exit);

        // call the setup functions:
        btb_setup();
        alpha_setup();
}
```