Compiler Techniques to Improve Dynamic Branch Prediction for Indirect Jump and Call Instructions

JASON MCCANDLESS and DAVID GREGG, LERO@TCD, Trinity College Dublin

Indirect jump instructions are used to implement multiway branch statements and virtual function calls in object-oriented languages. Branch behavior can have significant impact on program performance, but fortunately hardware predictors can alleviate much of the risk. Modern processors include indirect branch predictors which use part of the target address to update a global history. We present a code generation technique to maximize the branch history information available to the predictor. We implement our optimization as an assembly language transformation, and evaluate it for SPEC benchmarks and interpreters using simulated and real hardware, showing indirect branch misprediction decreases.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms: Performance, Algorithms, Experimentation

Additional Key Words and Phrases: Branch prediction, interpreters, object oriented

ACM Reference Format:

McCandless, J. and Gregg, D. 2012. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM Trans. Architect. Code Optim.* 8, 4, Article 24 (January 2012), 20 pages. DOI = 10.1145/2086696.2086703 <http://doi.acm.org/10.1145/2086696.2086703>

1. INTRODUCTION

Indirect jumping is used to implement common programming constructs such as switch statements, virtual function calls and calls through function pointers. The performance of programs that make a lot of use of these features can depend heavily on the underlying hardware's ability to correctly predict the target address of the indirect branch. Predicting the outcome of an indirect branch is often more difficult than conditional branch prediction, because indirect branches can have many different targets.

The simplest type of indirect branch predictor simply uses the branch target buffer (BTB) which is used to predict the (unchanging) target of conditional and unconditional direct branches. When used with indirect branches, the BTB simply predicts that the indirect branch will jump to the same target as last time it was executed. This works well for branches that usually jump to the same target, which are often known as *monomorphic indirect branches*. However in the case of indirect branches which jump to many different targets (*polymorphic indirect branches*) the prediction accuracy is usually poor.

To address this problem, researchers have adapted the idea of *two-level branch predictors* to indirect branches [Driesen and Hölzle 1998; Chang et al. 1997]. Rather

This work was supported, in part, by the Irish Research Council for Science Engineering & Technology and by Science Foundation Ireland grant 03/CW3/I303.1 and 10/CE/I1855 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

Authors' address: J. McCandless and D. Gregg, Computer Science Department, Trinity College Dublin, Dublin 2, Ireland; email: mccandjm@cs.tcd.ie.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/01-ART24 \$10.00

DOI 10.1145/2086696.2086703 <http://doi.acm.org/10.1145/2086696.2086703>

than having only a single branch prediction entry per branch, a two-level predictor uses both the address of the branch itself and a history of recent branch outcomes to index a table and find a prediction. In the case of indirect branches, the history of recent branches usually consists of some bits selected from the addresses of recent indirect branch targets, and combined together into a single history register. By capturing information about the outcome of recent branches, a two-level branch predictor is able to exploit correlations between different indirect branches, or between multiple executions of the same branch.

A limitation of two-level indirect branch predictors is that it is only possible to select a subset of the bits from each indirect branch target. For example, the indirect branch predictor in the Intel Pentium M processor selects the lower six bits of the indirect branch target, and hashes them into the branch history register [Uzelac and Milenkovic 2009]. A problem is that if more than one target has the same pattern in those bits, then the predictor will not be able to distinguish between them. In this case, the predictor will have less context information to make the prediction, and the likely outcome is poorer prediction accuracy.

In this paper we present two compiler-based software techniques to improve the predictability of indirect branches that are used to implement switch statements. Both techniques have the goal of ensuring that different targets of the switch branch have bit patterns that can be used to distinguish the branches in the indirect branch predictor. The paper makes the following contributions.

- We show that there is an opportunity to reduce collisions in the indirect branch predictor using compiler techniques.
- We present a solution based on NOP insertion and describe the problem formally, and present heuristic solutions.
- We present a second solution based on re-ordering the cases inside the switch statement and present heuristic solutions.
- We present a hybrid of both these techniques.
- We implement both strategies in an assembly optimizer tool, and evaluate them using both simulation and real hardware.
- We provide preliminary experiments that show these techniques can also be applied to improving branch prediction accuracy for virtual function calls.

2. BACKGROUND

Programming language switch (or case) statements can be implemented using different techniques, depending on the number of cases and the distribution of the switch constant values. For small numbers of cases, a sequence (or tree) of conditional branches suffice. Even for larger numbers of cases, a binary tree of branches can work well. However, in general the most efficient switch code can be generated for large number of cases when the *density* of the switch constant values is high. The density is high if the constant values fall within a fairly narrow range, and there are few gaps between the constants. In this situation, the compiler can create a jump table, where the table contains the addresses of the code that implements each case, and the table is indexed by the switch value. Jump tables are highly efficient, and require only a table lookup and an indirect jump to implement a switch statement. They are so efficient that GCC generates a switch statement for case statements with as low as 10% density when optimizing for speed [Sayle 2008].

In order to correctly predict the indirect branches used to implement switch statements and other constructs such as virtual functions and function pointers, researchers have developed increasingly sophisticated indirect branch predictors. Driesen and Hölzle [1998] adapted two-level conditional branch correlation techniques

Fig. 1. Bits of the indirect branch target address used to update the path information register in Intel processors.

to indirect branches. Kalamatianos and Kaeli [1999] showed that correlation with previous branches using data compression techniques could further improve accuracy. Kim et al. [2009] propose a low-cost mechanism to adapt an existing two-level conditional branch predictor to indirect branches.

CPU pipelines have grown over time, and modern processors have highly sophisticated branch predictors that attempt to avoid branch misprediction and a flush of the pipeline. In recent years, two-level indirect branch predictors have started to appear in real processors. The earliest example we are aware of is the Intel Pentium M [Gochman et al. 2003] which was introduced in 2003. More recent examples include the AMD Barcelona processor [Advanced Micro Devices 2008], and various Intel Core 2 processors.

Effective branch predictors are so important for performance that they are a competitive edge for CPU manufacturers, so their specific behavior is seldom publicly documented. There have been attempts to reverse-engineer the exact details of branch predictors [Milenkovic et al. 2002]. Uzelac and Milenkovic [2009] provide experiment flows and microbenchmarks for reverse-engineering cache-like branch predictor structures. In particular, they provide details of the branch predictor structures of the Intel Pentium M, including the indirect branch target buffer (iBTB). They find that the Pentium M takes the six lower bits of the address of the branch target (Figure 1), and hashes them into a branch history register (called the *path information register* (PIR) by Intel). This PIR is used in the prediction of future branches by the Pentium M.

The PIR in the Pentium M is a fifteen bit register, which is updated by both taken conditional branches and indirect branches as follows:

$$\begin{aligned} \text{PIR}[14 : 0] = & (\text{PIR}[12 : 0] \ll 2) \mathbf{xor} \\ & (\text{cbt} \cdot \text{IP}[18 : 4] \mathbf{or} \\ & \text{ibt} \cdot (\text{IP}[18 : 10] \text{concat } \text{TA}[5 : 0])), \end{aligned}$$

where *cbt* is a taken conditional branch, *ibt* is a taken indirect branch, *IP* is the memory address of the branch instruction, and *TA* is the target address of an indirect branch. In other words, when a new conditional or indirect branch is executed, the existing value of the PIR is shifted right by two places, and a new value is *xor*'d into the PIR. In the case of a conditional branch, this new value is simply 15 bits of the address of the branch in memory. In the case of an indirect branch, 9 bits are taken from the address of the branch, and six least significant bits from the address of the branch target. It has been reported that the Core microarchitecture includes the same indirect branch predictor [Kim et al. 2009].

Given that indirect branch predictors use only a selection of bits from the branch target when updating their history register, the effectiveness of two-level indirect branch prediction schemes depends on the bit pattern being different for the common targets. The actual addresses at which the branch targets appear in memory depends on choices made by the compiler and linker. In general, compilers and linkers make no attempt to optimize for the indirect branch predictor. In some systems the locations of the branch targets in memory are simply arbitrary, and whether the chosen bits are distinct for different branch targets is a matter of luck.

```

        .p2align 4,,10
        .p2align 3

#NOP bytecode implementation
.L4:
        addq    $8, %rsi        #, ip
        jmp     .L41
        .p2align 4,,10
        .p2align 3
#LOAD IMMEDIATE bytecode implementation
.L5:
        movl    4(%rsi), %eax
        addq    $8, %rsi        #, ip
        movl    %eax, (%r8)
        addq    $4, %r8         #, sp
        jmp     .L41

```

Fig. 2. Typical jump-table assembly compiled by GCC for our experimental interpreter. The jump targets will be 16-byte aligned by the linker, which means that the lower four bits of the address will be zero.

Sometimes compilers actually make the situation worse, by aligning branch targets for supposed cache gains—if the target is at the lowest address, then more code may end up fitting on a cache line. However, this leads to less information being available to the branch predictor hardware, and consequentially, worse predictions. Figure 2 shows an example extract of assembly code compiled by GCC for a switch statement. The branch targets of cases in the switch statement are labels .L4 and .L5. The compiler has inserted directives (the .p2aligns, whose first argument is the number of lower bits) to align these branch targets on a 16-byte boundary, which will ensure that the lower four bits always have the value zero. If the indirect branch predictor uses these bits, then they will be useless for distinguishing between different branch targets. In rare cases, aliasing can be helpful to the indirect branch predictor, but this is unlikely.

3. NOP INSERTION

3.1. Idea

Our first technique to modify indirect branch target addresses is to insert NOP instructions directly before a target instruction in order to move its memory address to a higher memory location that will not conflict with other targets.

3.2. Problem Formulation

It is clear that we can only have a bijective mapping if the number of targets is less than or equal to the number of “buckets”.

Definition 3.1 (Unique Target Address Keys). Given a positive integer k , a positive integer t , and a sequence of distinct nonnegative integers $X_1, \dots, X_n$ is there a sequence of nonnegative integers $D_1, \dots, D_n$ such that there does not exist $1 \leq i < j \leq n$ such that $X_i + D_i + \dots + D_1 \equiv X_j + D_j + D_i + \dots + D_1 \pmod{k}$ where the sum of $D_1, \dots, D_n$ is less than t .

CONJECTURE 3.2. *Deciding if there is a solution to the NOP insertion problem less than t is NP-complete (where t is the total number of inserted NOPs).*

Unfortunately a proof for this conjecture is not known to exist. A brute-force algorithm to evaluate all possible NOP insertion permutations will have $O(\binom{t+n-1}{n-1})$ complexity, where t is the total “budget” and n is the number of target positions. Enumerating all the possible solutions in this manner is the problem of integer compositions. An algorithm for generating integer compositions is described by Knuth [2005].

3.3. Heuristics

On the assumption that the problem is NP-hard, we now present three heuristics that perform well, and we show simulation results.

3.3.1. Greedy. The simplest heuristic, shown in Algorithm 1, is a greedy algorithm. It simply iterates through the addresses from lowest to highest (therefore making updated address calculation easy), and inserts NOPS until the current address no longer conflicts with previously processed addresses. The average case for this algorithm can be analyzed in the same manner as a series of unsuccessful searches in linear probing, and an aggregation of the following formula from Knuth [1998] will provide an average for the number of NOPS inserted:

$$C_N \approx \frac{1}{2} \left(1 + \left(\frac{1}{1 - \frac{N}{M}} \right)^2 \right),$$

where N is the number of targets already considered and M is the total number of targets. The worst case of number of NOPS inserted is $M(M + 1)/2$.

ALGORITHM 1: Greedily insert NOPS.

Input: Target memory addresses (T).

Output: New target memory addresses (T').

uniqueKeys = $\emptyset$;

offset $\leftarrow$ 0;

for each target address T_i in T **do**

$T'_i \leftarrow T_i + \text{offset}$

while $T'_i \pmod{2^6}$ in uniqueKeys **do**

 insertNop();

 offset $\leftarrow$ offset + 1;

end

$T'_i \leftarrow T_i + \text{offset}$;

 uniqueKeys $\leftarrow$ uniqueKeys $\cup \{T'_i \pmod{2^6}\}$;

end

3.3.2. Minimize Conflicts. The greedy algorithm can be improved upon—Algorithm 2 also iterates through target addresses from lowest to highest and it inserts enough NOPS to minimize the number of all conflicting targets at each stage while being cautious about how many NOPS it inserts by minimizing (conflicts $\times$ number of NOPS).

3.3.3. Maximize “Clean Distance”. A third heuristic is shown in Algorithm 3. This algorithm will also iterate through targets from lowest to highest, and it will choose the number of NOPS to insert that maximizes the distance from the current target to the next target that has a conflict, while also avoiding aggressively inserting NOPS by maximizing $\frac{\text{distance}}{\text{NOPS}}$.

3.4. Simulation Results

To compare three heuristics (and variants thereof), we simulated input target addresses using random numbers. First, 60 random variables α with a uniform distribution within the interval (5, 29)¹ are generated. These are used as the sizes to generate a sequence of memory addresses. This is repeated 1000 times for each heuristic that is being

¹These sizes are representative of test programs we looked at.

ALGORITHM 2: Insert NOPS, minimizing the conflicts at each point.

Input: Target memory addresses (T).

Output: New target memory addresses (T').

uniqueKeys $\leftarrow \emptyset$;

offset $\leftarrow 0$;

for each target address T_i in T **do**

 smallest $\leftarrow 0$;

 bestNN $\leftarrow 0$;

for $nn = 0 \rightarrow (2^6 - 1)$ **do**

 nextAddrs $\leftarrow \text{getNextAddresses}(\text{offset}, nn, T)$;

 conflicts = countConflicts(nextAddrs, uniqueKeys);

if $T_i + \text{offset} + nn \pmod{2^6}$ in uniqueKeys **and** $(\text{conflicts} \times nn) < \text{smallest}$ **then**

 bestNN $\leftarrow nn$;

 smallest $\leftarrow \text{conflicts} \times nn$;

end

end

 offset += bestNN;

$T'_i \leftarrow T_i + \text{offset}$;

 uniqueKeys $\leftarrow \text{uniqueKeys} \cup \{T'_i \pmod{2^6}\}$;

end

ALGORITHM 3: Insert NOPS, maximizing the distance until a conflict occurs with the offsetting at each point.

Input: Target memory addresses (T).

Output: New target memory addresses (T').

uniqueKeys $\leftarrow \emptyset$;

offset $\leftarrow 0$;

for each target address T_i in T **do**

 greatest $\leftarrow 0$;

 bestNN $\leftarrow 0$;

for $nn = 0 \rightarrow (2^6 - 1)$ **do**

 nextAddrs $\leftarrow \text{getNextAddresses}(\text{offset}, nn, T)$;

 dist = getCleanDist(nextAddrs, uniqueKeys);

if $T_i + \text{offset} + nn \pmod{2^6}$ not in uniqueKeys **and** $\frac{\text{dist}}{nn+1} > \text{greatest}$ **then**

 bestNN $\leftarrow nn$;

 greatest $\leftarrow \frac{\text{dist}}{nn+1}$;

end

end

 offset += bestNN;

$T'_i \leftarrow T_i + \text{offset}$;

 uniqueKeys $\leftarrow \text{uniqueKeys} \cup \{T'_i \pmod{2^6}\}$;

end

evaluated. Table I shows simulated results for four of the best heuristics. The second and fourth rows show heuristics that also run the greedy algorithm for each set of addresses and choose the answer with the smallest number of NOPS. The best heuristic, with an average of 116 NOPS inserted, is the combination of maximize “clean distance” heuristic and greedy.

Table I. Simulated Average NOP Insertions for Various Heuristics for 60 Target Addresses and Modulo 64

Heuristic	Average number of NOPs
Greedy	137
Minimize conflicts & greedy (pick best)	128
Maximize “clean distance”	127
Maximize “clean distance” & greedy (pick best)	116

4. RE-ORDERING

4.1. Idea

It may be wasteful to insert NOP instructions when memory size is severely constrained, so another technique to change the memory addresses of the indirect branch targets (when the targets are placed sequentially in memory) is to re-order the code blocks.

4.2. Problem Formulation

Definition 4.1 (Unique Target Address Keys Ordering where $n = m$). Given a sequence of positive integers $S_1, \dots, S_n$, is there a permutation of S , such that for $\forall x \in X$, where $x_i = S_0 + S_1 + \dots + S_{i-1}$, every x_i is unique mod k .

LEMMA 4.2. *For $n < m$, there is not always a permutation that results in no collisions in the hash values.*

PROOF. If the address of two cases p_i and p_{i+1} differ by a value v , such that $v \bmod k \equiv 0$, then both will hash to the same number. This happens when $\text{length}(p_i) \bmod k \equiv 0$. If we have more than one such case, then we will always have at least one collision in the hashed values. $\square$

LEMMA 4.3. *The decision problem for unique targets when $n = m$ is NP-hard.*

PROOF. To ensure a particular x_i is unique mod k , we must choose a subset S_j of $S_1, \dots, S_n$ such that its sum mod k equals a particular integer. This sub-problem is the NP-complete problem of subset-sum on a finite field [Nathanson 1996]. $\square$

4.3. Heuristics

Since we also assume the problem of re-ordering in the general case is NP-hard, we present two heuristics to order the blocks in the program. These heuristics do not attempt to remove all conflicts, but to reduce them.

The simplest heuristic to reduce the number of conflicts is a “pair-swap”. We perform a single pass over all the triplets of target addresses (pairs of ‘blocks’²) and swap them if it will mean less conflicts with target addresses already processed. Algorithm 4 shows this heuristic.

A second, more effective heuristic to reduce conflicts is to greedily schedule them: Select the next block that causes no conflicts and place it after the previously scheduled one. If there is no block that does not cause a conflict, then just pick the first block in the list. Algorithm 5 shows this heuristic.

For these heuristics, the worst case would be a new ordering where all targets conflict with each other. This could happen if the input consists of blocks that are all the size of the “buckets”.

5. HYBRID

Re-ordering attempts to reduce conflicts by only moving blocks. A more sophisticated approach is to combine that with NOP insertion to get the best of both. Blocks are

²These are not basic blocks in the CFG sense, since there may be arbitrary control flow between two branch targets.

ALGORITHM 4: Re-order “blocks” using a “pair-swap”.

Input: Target memory addresses (T).**Output:** New target memory addresses (T').blocks $\leftarrow \emptyset$;**for** each pair of target addresses T_i, T_{i+1} in T **do** blocks $\leftarrow$ blocks $\cup \{T_i, T_{i+1}\}$;**end**offset $\leftarrow 0$;

output(offset);

for each pair of blocks b_i, b_{i+1} in blocks **do** **if** $\text{size}(b_i) \neq \text{size}(b_{i+1})$ **then** **if** $\text{offset} + \text{size}(b_i) \pmod{2^6}$ in uniqueKeys **and** $\text{offset} + \text{size}(b_{i+1}) \pmod{2^6}$ not in uniqueKeys **then**

swapBlocks();

end **end** offset $\leftarrow$ offset + size(b_i);

output(offset);

end

ALGORITHM 5: Re-order “blocks” using a greedy fit.

Input: Target memory addresses (T).**Output:** New target memory addresses (T').blocks $\leftarrow \emptyset$;**for** each pair of target addresses T_i, T_{i+1} in T **do** blocks $\leftarrow$ blocks $\cup \{T_i, T_{i+1}\}$;**end**offset $\leftarrow 0$;

output(offset);

while blocks $\neq \emptyset$ **do** choice $\leftarrow$ blocks.first; **for** each block b in blocks **do** **if** $\text{offset} + \text{size}(b) \pmod{2^6}$ not in uniqueKeys **then** choice $\leftarrow b$; **end** **end** offset $\leftarrow$ offset + size(choice);

output(offset);

end

re-ordered to reduce collisions, and NOPs inserted if no block is the required size. Algorithm 6 shows this hybrid algorithm.

6. PERFORMANCE EVALUATION**6.1. Implementation**

We implemented our two techniques using the assembly language optimization framework developed by Hundt et al. [2011]. For NOP insertion, we implemented the greedy algorithm (Algorithm 1) for simplicity. For re-ordering, we implemented Algorithm 5. In some cases, the compiler (GCC) inserts `.p2align` assembler directives before indirect branch targets (as shown in Figure 2, aligning them to 16-byte boundaries—this is especially detrimental to the performance of the branch predictor, as only 2 unique

ALGORITHM 6: Hybrid—re-order “blocks” using a greedy fit, insert NOPS if necessary.

Input: Target memory addresses (T).
Output: New target memory addresses (T').
 blocks $\leftarrow \emptyset$;
for each pair of target addresses T_i, T_{i+1} in T **do**
 blocks $\leftarrow$ blocks $\cup \{T_i, T_{i+1}\}$;
end
 offset $\leftarrow 0$;
 output(offset);
while blocks $\neq \emptyset$ **do**
 choice $\leftarrow$ nil;
 nopsNeeded $\leftarrow 0$;
 for each block b in blocks **do**
 if offset + b_{size} (mod 2^6) not in uniqueKeys **then**
 choice $\leftarrow b$;
 end
 end
 if choice = nil **then**
 nopsNeeded $\leftarrow \infty$;
 for each block b in blocks **do**
 if countNopsNeeded(b) < nopsNeeded **then**
 choice $\leftarrow b$;
 nopsNeeded $\leftarrow$ countNopsNeeded(choice);
 end
 end
 end
 offset $\leftarrow$ offset + choice.size;
 for $i=1$ to nopsNeeded **do**
 insertNop();
 offset $\leftarrow$ offset + 1;
 end
 output(offset);
end

bits of the target address would be used in the predictor history. In our implementation we remove these .p2align before the indirect branch targets.

For re-ordering, the code between target labels may include arbitrary control-flow, so we make a simplification, and treat the code between any two target labels as a ‘block’ to be moved (unless there is a fall-through path from one block to another, in which case we coalesce the two).

A small adaption was required when inserting NOPS for the problem cases where $N > M$, that is, where the number of branch targets is greater than the number of ‘buckets’ (64). If $N > M$, for the first 64 targets we insert NOPS to make each unique, for the next 64 we insert nops until each target only conflicts with one other target, etc.

6.2. PIN Simulation

We created a simulation of the indirect branch target buffer and the path information register using PIN [Luk et al. 2005]. Figure 3 shows our model. We then evaluated our techniques on the SPEC2006 C benchmarks [Standard Performance Evaluation Corporation 2011]. Figure 4 shows the misprediction rate reductions of the programs with NOPS inserted to increase target uniqueness, the programs using re-ordering, and the programs using the hybrid technique compared to the baseline (100%). The majority of the benchmarks show reductions in the misprediction rate, some of them large (>90%

Fig. 3. Our PIN model for the indirect branch target buffer. Dashed lines represent the path information register update.

Fig. 4. Simulated iBTB misprediction rate for three techniques compared to baseline. (Smaller is better.)

reduction for the *mcf* benchmark). The worst performing case was re-ordering targets for the *sphinx3* benchmark, where the misprediction rate was increased by 40%.

We did not observe a significant change in branch mispredictions from running our techniques on the SPEC2006 benchmarks on real hardware. We believe this is because on our test machine, conditional branches share the path information register with the iBTB. This is then compounded by the low ratio of indirect branches to conditional branches in those programs. For example, the *gcc* benchmark, which has a significant number of indirect branches, has 28 times more conditional branches than indirect branches. Given that taken conditional branches update the branch history, and assuming a significant number of the conditional branches are taken, and the history depth is ≈ 8 , updates to the history by indirect branches will be ‘flushed’ by the time another indirect branch is reached. In the next subsection, we will look at programs that make intensive use of indirect branches.

Table II. Simple Interpreter Results

Event	Baseline	NOPs inserted	Re-ordered	Hybrid
Branches executed	4,400,625,607	4,500,244,910	4,499,465,407	4,501,921,449
Indirect branches	1,396,710,949	1,394,288,997	1,399,635,984	1,400,056,605
Indirect branch mispredictions	399,755,351	22,221	82,269	88,948
L1 Instruction cache reads	6,196,712,454	6,192,419,852	6,605,654,635	6,606,098,075
L1 Instruction cache misses	43,864	32,038	12,755	15,209
Execution time (seconds)	6.42	2.89	2.76	2.76

```

enum {OP_LDI, /* ... */};
typedef struct inst {
    int32_t op, arg;
} inst;
void engine() {
    inst program[]={ { OP_LDI,    0 } /* ... */ };
    int *sp=s;
    inst *ip = program;
    while(1) {
        switch(ip->op) {
            case OP_LDI:
                *sp+=ip->arg; ++ip; break;
                /* ... */
        } } }

```

Fig. 5. Our simple interpreter.

6.3. Indirect-Branch-Heavy Programs

An important use of multiway branches is in interpreter implementation. We therefore evaluate our techniques on three interpreters: a simple interpreter written in C; Lua, a switch-based dynamic language interpreter; and JamVM, a Java language interpreter (switch implementation).

Our first interpreter is a simple interpreter we designed to experiment with optimizations (Figure 5). It has a full set of opcodes, but only a few are exercised in our test bytecode program, which is just a nested loop with arithmetic in the inner loop. The results of applying our code transformation technique on our simple interpreter are enumerated in Table II. These results are for the Intel Core 2. We see a significant decrease in the number of branch mispredictions, and a pronounced reduction in execution time for all three techniques. The re-ordering and hybrid show a 6% increase in level 1 instruction cache reads, but have less misses than either the baseline or the version with NOPs inserted. The hybrid algorithm shows more indirect branch mispredictions than the other techniques; this is likely due to the other techniques producing addresses that are more fortuitous for that benchmark, but the overall effect is not as significant as the change from the baseline. The instruction cache miss rates are tiny ($<0.001\%$), so we do not infer anything from the small changes in misses.

The second interpreter we look at is Lua, a dynamically typed language interpreter. We evaluate both our techniques for Lua using a selection of benchmarks from the Computer Language Benchmarks Game [CLBG 2011] and The Great Win32 Computer Language Shootout on the Intel Core 2. Figure 6 shows the misprediction rate of the interpreter before and after performing the NOP insertion technique. Figure 7 shows speedups resulting from the reduction in indirect branch misses. Figure 8 shows the misprediction rate of the interpreter before and after performing re-ordering and Figure 9 shows the misprediction rate of the interpreter before and after the hybrid technique. The reduction in mispredictions compared to the baseline for the three techniques is shown in Figure 10. All three techniques lead to a substantial decrease

Fig. 6. Lua misprediction rate for baseline vs. NOP insertion for a selection of benchmarks.

Fig. 7. Lua speedups (due to fewer mispredictions) for the NOP insertion technique. Showing an average speedup of 7%. The *mandelbrot* benchmark achieves an almost 80% speedup.

in mispredictions for the majority of Lua benchmarks. Figure 11 shows the level 1 instruction cache results for Lua. The size of the instruction cache (32 KB) dwarfs the code size of the interpreter.

The third interpreter we investigate is the JamVM Java interpreter [Lougher 2011]. We compile the ‘switch’ version of the interpreter by disabling the threaded code

Fig. 8. Lua misprediction rate for baseline vs. re-ordered indirect branch targets.

Fig. 9. Lua misprediction rate for baseline vs. hybrid indirect branch targets.

mode. We evaluated our techniques on this interpreter using benchmarks from the Java Grande Forum [Mathew et al. 1999]. Figure 12 shows the misprediction rates of the baseline interpreter, the interpreter using NOP insertion, the interpreter using re-ordering and the interpreter using the hybrid technique on the Intel Core 2. The effect of reducing branch target address collisions is not as pronounced for this interpreter,

Fig. 10. Lua indirect branch misprediction rate for the baseline vs. NOP insertion vs. re-ordering vs. hybrid for a selection of benchmarks. (Smaller is better).

Fig. 11. Lua level 1 instruction cache miss rate compared to baseline for the three techniques for a selection of benchmarks. (Smaller is better.)

perhaps because there are many more opcodes (>100), and consequentially many more indirect branch targets. Figure 13 shows speedups attained from the reduction in indirect branch misses. Figure 14 shows the level 1 instruction cache results for Java. These have a similar scale as the Lua instruction cache results.

Fig. 12. Java misprediction rate for baseline vs. NOP insertion vs. re-ordering for a selection of benchmarks.

Fig. 13. Java speedups for the three techniques compared to the baseline (Larger is better).

Fig. 14. Java level 1 instruction cache miss rate for the baseline vs. NOP insertion vs. re-ordering vs. hybrid for a selection of benchmarks. (Smaller is better).

6.4. C++ Programs

In addition to multiway branches, indirect branches are also used to implement indirect function calls. Function calls are used in programs that use function pointers where the target cannot be resolved statically at compile time. Indirect function calls are also used to implement run-time dispatching which is required for polymorphism in object-oriented languages.

For C++, the GCC compiler implements virtual method calls using dynamic dispatch with *virtual function tables*, which involve an indirect call instruction. Indeed, this may be one of the greatest motives for the addition of indirect branch prediction hardware to recent Intel and AMD CPUs.

To apply our techniques to this use-case of indirect calls, we adapted our assembly language optimization passes to work with function branch targets. We parse the virtual tables in our optimization pass and determine the start labels of all virtual functions. We also include *thunks* as these are possible targets of indirect calls.

GCC puts functions into separate sections in the assembly file, and these are usually 16-byte aligned by the linker. In our first implementation, we took this constraint into account, and inserted enough NOP instructions to ensure each branch target had unique lower bits. However, we did not observe a significant decrease in indirect branch misses, so we changed the implementation to move all the virtual functions into one section before making targets unique. To evaluate our optimization pass, we created a C++ version of our simple interpreter, using an abstract base Instruction class which has a virtual `execop` method that child classes override to implement the various opcodes. Thus the simple interpreter loop: `while (1){ (*ip)->execop(); }`

The results for our C++ simple interpreter are enumerated in Table III. The baseline has a branch misprediction rate of 13%, and the version with NOPS inserted has a misprediction rate of 0.004%. Our re-ordering technique gives a misprediction rate of 0.006%. These are both much better than the baseline, which has a misprediction rate of 13%. Both techniques show a reduction in the number of L1 i-cache reads—with less mispredictions, less instructions are executed, and consequently the i-cache is accessed less.

Table III. Simple Interpreter (C++) Results

Event	Baseline	NOPS inserted	Re-ordered
Branches executed	4,293,992,704	4,301,493,720	4,300,948,941
Indirect branches	1,400,250,180	1,399,109,088	1,399,971,862
Indirect branch mispredictions	186,606,245	58,434	84,555
L1 Instruction cache reads	20,911,541,414	14,275,657,052	14,524,382,937
L1 Instruction cache misses	30,678	103,103	115,485
Execution time (seconds)	8.73	6.05	6.05

6.5. Hardware Configuration

We evaluate our optimization on the following system: Intel Core 2 Q6600 (Core Kentsfield) 2.4 GHz CPU with 6 GB memory, running Ubuntu GNU/Linux 11.04 with kernel version 2.6.38-8. GCC version 4.5.2. Performance counters were collected using the *perf* tool, an interface to the Linux kernel perf events and PAPI [Browne et al. 2000]. For indirect branch misses, we recorded the BR_IND_MISSP counter.

7. RELATED WORK

7.1. Reducing Branch Interference

Chen and King [1999] adjust addresses to reduce collisions in 2-bit counters for conditional branches. They also add NOP instructions to perform the address adjustment. They present a constrained and a relaxed method, where the constrained method will only insert NOPS following an unconditional branch and relaxed will insert NOPS where they may get executed by the processor. They also describe *branch classification*, which maps branches with the same history pattern to the same counter. Our work similarly inserts NOP instructions to move branch addresses, but we instead consider indirect branches, and we also present a technique using code re-ordering.

Jiménez [2005] build upon that work, describing *pattern history table partitioning*, a feedback directed technique that moves conditional branch addresses to avoid destructive interference. We have not considered any feedback techniques for the current work.

7.2. Code Placement for Branches

Krall [1994] uses profiling to collect information about branches and then uses code replication to reduce the branch misprediction rate. We are concerned with indirect branches, and do not investigate profiling methods.

Uh and Whalley [1999] describe a transformation that coalesces conditional branches into indirect jumps. They show that this transformation to indirect jumps can improve prediction. We are not concerned with how indirect jumps are generated, we always try to maximize their predictability.

Yang et al. [2002] re-order branches using profile data to reduce the number of conditional branches executed by a program. Our work is concerned with indirect branches whose execution can generally never be avoided by code rearrangement.

There has been much research on code rearrangement to improve cache behavior. Related to branches, Calder and Grunwald [1994] present a profile-based technique that ‘aligns’ branches: It will minimize the number of taken branches, therefore ensuring frequently executed code is closer together, thereby improving cache performance.

7.3. Target Predictability for High-Level Languages

In previous work [McCandless and Gregg 2011], we demonstrated that re-ordering the cases of a switch statement has a significant impact on indirect branch prediction for interpreters. We hypothesized this to be due in part to aliasing in the indirect branch predictor.

Li et al. [2005] improve the target predictability of Java, particularly for virtual methods by proposing a rehashable branch target buffer that dynamically adapts branch target storage for polymorphic branches. They show improvements over traditional techniques, including target caches. Our work is focused on improving the performance of such code on already existing hardware.

7.4. Compiler Assisted Branch Prediction

Deitrich et al. [1998] concern themselves with static branch prediction from the compiler's perspective. They present heuristics, evaluate them for a selection of benchmarks and provide insights about their efficacy.

Mahlke and Natarajan [1996] present a compiler technique for branch prediction by using profile feedback to insert a prediction function that captures information about the current context that is then used to make predictions. They show performance comparable to that of two-level hardware branch predictors.

7.5. "Weird" Behavior

The effect of linking order as a source of measurement bias has been described by Mytkowicz et al. [2009]. They attribute measurement bias due to link order as an alignment issue, and using the m5 simulator they see instruction cache miss variances when the link order is changed. They suggest branch prediction may also be a cause in general. Our work investigates optimizing code by maximizing information available in branch predictor history. We do not propose that measurement bias may be eradicated, but that bias produced by branch predictor aliasing may be decreased. In a later paper [Knights et al. 2009], the authors propose "blind optimization". We believe this will be a useful technique, but there remain traditional hardware-aware optimizations to be discovered and exploited.

Hundt et al. [2011] describe changes in execution time from *nopinizer*, a random NOP insertion pass they implemented using their assembly language optimization framework. They also observed performance variations from a *nop killer* pass. We do not insert NOPS at random, but instead attempt to utilize them to provide as much information to the predictor as possible. However, for any program, there will always be an unknown optimal address layout that maximizes entropy for a particular history. Indeed, constructive interference [Young et al. 1995] causes branches that would have been mispredicted to be correctly predicted.

8. CONCLUSIONS

In this article we presented an optimization opportunity where the indirect branch predictor does not perform as well as possible, due to indirect branch target aliasing that can be removed or reduced. We described our optimization to improve the branch prediction of indirect jumps and calls. We presented two techniques to increase the uniqueness of the bits in target addresses that are used by the predictor, the first based on NOP insertion, and the second based on re-ordering; and we describe them formally. We provided heuristics to approximate these techniques and implemented two as assembly language optimizations. Experimental results, both simulation and hardware, were provided for C programs using SPEC2006 and our own intensive indirect branch programs, showing that providing more information to the predictor by making targets unique can reduce the number of indirect branch mispredictions. We also provided preliminary results for virtual function calls.

In our future work, we intend to further experiment with branch target addresses and implement a profile-based optimization that will automatically determine good target addresses for a given profile to maximize entropy in the global history.

REFERENCES

- Advanced Micro Devices 2008. *Software Optimization Guide for AMD Family 10h Processors*. Advanced Micro Devices.
- BROWNE, S., DONGARRA, J., GARNER, N., LONDON, K., AND MUCCI, P. 2000. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '00)*. IEEE Computer Society, Los Alamitos, CA.
- CALDER, B. AND GRUNWALD, D. 1994. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. ACM, New York, 242–251.
- CHANG, P.-Y., HAO, E., AND PATT, Y. N. 1997. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*. ACM, New York, 274–283.
- CHEN, C.-M. AND KING, C.-T. 1999. Walk-time address adjustment for improving the accuracy of dynamic branch prediction. *Comput. IEEE Trans.* 48, 5, 457–469.
- CLBG. 2011. The computer language benchmarks game. <http://shootout.alioth.debian.org/>.
- DEITRICH, B., CHENG, B.-C., AND HWU, W.-M. 1998. Improving static branch prediction in a compiler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*. IEEE Computer Society, Los Alamitos, CA, 214.
- DRIESEN, K. AND HÖLZLE, U. 1998. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*. IEEE Computer Society, Los Alamitos, CA, 167–178.
- GOCHMAN, S., RONEN, R., I. ANATI, A. B., KURTS, T., NAVEH, A., SAEED, A., SPERBER, Z., AND VALENTINE, R. 2003. The Intel Pentium M processor: Microarchitecture and performance. *Intel Tech. J.* 7, 2.
- HUNDT, R., RAMAN, E., THURESSON, M., AND VACHHARAJANI, N. 2011. MAO—an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10.
- JIMÉNEZ, D. A. 2005. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, 107–116.
- KALAMATIANOS, J. AND KAEI, D. 1999. Indirect branch prediction using data compression techniques. *J. Instruct. Level Paral.* 1, 6.
- KIM, H., JOAO, J. A., MUTLU, O., LEE, C. J., PATT, Y. N., AND COHN, R. 2009. Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. *IEEE Trans. Comput.* 58, 1153–1170.
- KNIGHTS, D., MYTKOWICZ, T., SWEENEY, P. F., MOZER, M. C., AND DIWAN, A. 2009. Blind optimization for exploiting hardware features. In *Proceedings of the 18th International Conference on Compiler Construction (Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS) (CC '09)*. Springer-Verlag, Berlin, 251–265.
- KNUTH, D. E. 1998. *The Art of Computer Programming, volume 3: (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA.
- KNUTH, D. E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional.
- KRALL, A. 1994. Improving semi-static branch prediction by code replication. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, 97–106.
- LI, T., BHARGAVA, R., AND JOHN, L. K. 2005. Adapting branch-target buffer to improve the target predictability of java code. *ACM Trans. Archit. Code Optim.* 2, 109–130.
- LOUGHER, R. 2011. JamVM virtual machine. <http://jamvm.sourceforge.net>.
- LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, 190–200.
- MAHLKE, S. AND NATARAJAN, B. 1996. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Los Alamitos, CA, 153–164.
- MATHEW, J. A., CODDINGTON, P. D., AND HAWICK, K. A. 1999. Analysis and development of Java grande benchmarks. In *Proceedings of the ACM Conference on Java Grande (JAVA '99)*. ACM, New York, 72–80.

- MCCANDLESS, J. AND GREGG, D. 2011. Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ '11)*. ACM, New York.
- MILENKOVIC, M., MILENKOVIC, A., AND KULICK, J. 2002. Demystifying Intel branch predictors. In *Proceedings of the Workshop on Duplicating, Deconstructing and Debunking*, John Wiley & Sons, 52–61.
- MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. 2009. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. ACM, New York, 265–276.
- NATHANSON, M. B. 1996. Additive number theory: Inverse problems and the geometry of sumsets. In *Graduate Texts in Mathematics*. Springer.
- SAYLE, R. 2008. A superoptimizer analysis of multiway branch code generation. In *GCC Developers' Summit*, 103.
- STANDARD PERFORMANCE EVALUATION CORPORATION. 2011. SPEC CPU2006. <http://www.spec.org/cpu2006/>.
- UH, G.-R. AND WHALLEY, D. B. 1999. Effectively exploiting indirect jumps. *Softw. Pract. Exper.* 29, 1061–1101.
- UZELAC, V. AND MILENKOVIC, A. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*. 207–217.
- YANG, M., UH, G.-R., AND WHALLEY, D. B. 2002. Efficient and effective branch reordering using profile data. *ACM Trans. Program. Lang. Syst.* 24, 667–697.
- YOUNG, C., GLOY, N., AND SMITH, M. D. 1995. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, 276–286.

Received July 2011; revised October 2011 and December 2011; accepted December 2011