Runtime Adaptation: A Case for Reactive Code Alignment

Michelle McDaniel
University of Virginia

Kim Hazelwood
University of Virginia

ABSTRACT

Static alignment techniques are well studied and have been incorporated into compilers in order to optimize code locality for the instruction fetch unit in modern processors. However, current static alignment techniques have several limitations that cannot be overcome. In the exascale era, it becomes even more important to break from static techniques and develop adaptive algorithms in order to maximize the utilization of every processor cycle. In this paper, we explore those limitations and show that *reactive realignment*, a method where we dynamically monitor running applications, react to symptoms of poor alignment, and adapt alignment to the current execution environment and program input, is more scalable than static alignment. We present fetches-per-instruction as a runtime indicator of poor alignment. Additionally, we discuss three main opportunities that static alignment techniques cannot leverage, but which are increasingly important in large scale computing systems: microarchitectural differences of cores, dynamic program inputs that exercise different and sometimes alternating code paths, and dynamic branch behavior, including indirect branch behavior and phase changes. Finally, we will present several instances where our trigger for reactive realignment may be incorporated in practice, and discuss the limitations of dynamic alignment.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Measurement

1. INTRODUCTION

In the exascale era, where millions of cores enable exaflop performance, it becomes increasingly important to optimally utilize every processor cycle. As the number of microarchitectural, application, compiler and run-time system choices increase, static optimization approaches no longer suffice. In particular, static approaches to code alignment are severely limited. How sequences of instructions are aligned in the instruction cache have been shown to affect the overall runtime of applications by up to 2x [12]. Poor code alignment can increase the number of branch mispredictions, cache misses, memory stalls, and instruction fetches, all of which will degrade performance. In large scale computing systems, we cannot rely on the static alignment determined by the compiler, in particular when the microarchitecture of the cores is not uniform and the inputs to the applications vary. We must dynamically adapt the alignment of these instruction sequences in order to greatly improve the utilization of the ever-increasing number of cores used in large scale computing systems.

Many researchers have specifically worked to improve code alignment to reduce the number of cache misses [9, 14, 16], to reduce the memory footprint of applications [4], or to decrease the number of memory stalls caused by poor code locality [9]. However, by focusing on just one of these issues, it is easy to introduce other issues that did not originally exist in the program. For example, for maximum fetch throughput, code should be aligned at 16-byte boundaries as often as possible, as shown in Figure 1a. However, if we align only for maximum fetch throughput, we may introduce branch collisions that result in branch mispredictions. On Intel Core based processors, if two branches exist in the same fetch line, they will collide in the branch predictor. To avoid branch collisions, the code should be aligned as shown in Figure 1b. In order to correctly align programs, we must consider all alignment issues and how they may affect one another. In our original example, if we only align for maximum fetch throughput, we may inadvertently push two branches onto the same fetch line, leading to poor branch behavior.

Large scale computing systems often may be made up of many non-uniform cores where each core has its own microarchitectural features that must be either exploited in order to improve performance or avoided in order to not degrade performance. These features include instruction fetch length, which determines the boundaries at which we must align, hardware optimizations like the Loop Stream Detector, which significantly improves the performance of loops by bypassing the instruction fetch and branch prediction units, and the branch prediction unit, which is designed differently in many different microarchitectures. Additionally, reasonably complex applications that are executed in these large

EXADAPT '12 March 3, 2012, London, UK.

Addr	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
400570	mov			cmp			jne		add			cmp			je	
400580																

(a) Aligning for maximum fetch throughput

Addr	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
400570			mov			cmp			jne		add			cmp		
400580	je															

(b) Aligning to avoid branch collisions

Figure 1: The 16-byte layout of a code snippet aligned for 1a) maximum fetch throughput and 1b) the branch predictor. Note that each block in the diagram is one byte, so a three byte `cmp` takes up three blocks in the diagram.

scale computing systems can have a wide variety of inputs which exercise many different code paths in the application. In these environments, we cannot assume that static techniques will scale. Because static techniques choose one fixed alignment for a program, in order to align code for each the exponential combinations of microarchitectural features and program inputs, we would have to create a new binary for each possible pair, wasting both computing resources and space.

Because static approaches no longer suffice in the exascale era, we must look to dynamic approaches to solve the issues that arise. While creating binaries for each possible pair of microarchitectural feature and program input is unreasonable, applying a dynamic approach can overcome the limitations of a single static alignment, and account for many alignment issues that may arise during execution. In a dynamic approach, we would adapt the alignment of the program based on an alignment trigger that informs the system that there are alignment issues. An adaptive code alignment system could uniquely align the program during execution, accounting for both microarchitectural features and program inputs.

Additionally, an adaptive approach would be self-tuning. It would monitor a running program, detect poor behavior, and change the alignment of the program. If the application continues to exhibit poor behavior, dynamic techniques can attempt to realign the code again. In this way, dynamic techniques overcome the main issue that static alignment faces:the inability to adapt to changes in the execution of the program.

Our main contributions in this paper are:

- The Basic Block Code Alignment score: a novel metric for quantifying the alignment quality of entire programs,
- The identification of effective dynamic indicators of poor alignment, and
- The identification of alignment opportunities that can only be leveraged at runtime.

The rest of this paper is structured as follows: in Section 2, we discuss static alignment techniques and their limitations. Section 3 describes the Basic Block Code Alignment score (BBCA score), a novel mechanism for scoring the alignment of programs given various inputs. Then in Section 4, we present reactive code alignment and discuss potential triggers. In Section 5, we describe opportunities where reactive realignment can improve upon static alignment. Next, Section 6 discusses how reactive realignment can be used in practice and its practical limitations. Finally, we present related work in Section 7 and conclude in Section 8.

2. STATIC ALIGNMENT TECHNIQUES AND LIMITATIONS

Static alignment techniques are very well studied and have been incorporated into modern compilers. Many compiler alignment techniques sweep through the assembly level source code and align code at the function, loop, and instruction level. For example, the four GCC alignment directives are `-falign-functions`, `-falign-loops`, `-falign-branches`, and `-falign-labels`. All of these directives seek to align either functions, loops, branch targets, or labels at power-of-two byte boundaries. For each of these cases, they calculate how many `nops` need to be inserted in order to allow that section of code be well aligned. For most compilers, if too much padding would need to be inserted, the compiler will simply give up. For example, the Intel C compiler will insert at most 8 `nops` for any particular piece of code. Additionally, code will be aligned to a specified byte boundary, rather than inserting additional padding that may allow future code to be aligned as well. Because static alignment techniques must rely on heuristics about the code, important runtime paths may not be aligned appropriately.

Another static alignment decision is whether to swap the `then` and `else` case of an `if` loop, based on some path frequency heuristic. `then` and `else` scheduling is also difficult for static compilers because it requires profile information about the program in order to accurately schedule the two cases. Often, this profile information is not available, so the compiler must rely on path frequency heuristics which are rarely perfect. Finally, where the compiler places a branch target will modify the alignment of the code due to both the size of jump, and the amount of code that is displaced by the content of the branch target.

Prior work focuses on inserting `nops` in order to improve the compiler-decided alignment of code [3] [10] [11]. In MAO [10], there are three alignment passes: short loop alignment, loop stream detector, and branch alignment. These three alignment passes are done sequentially, and for each, the placement of the code is changed by inserting `nops` in the appropriate places in order to improve the performance of the code. Boehm *et al.* [3] study alignment issues on the Power 6, and deal with the effects of pipeline dispatch. Since a fetch buffer contains 8 instructions, the alignment can change what instructions are fetched together. In order to change the alignment, they insert gaps between chains of instructions to place frequent branch targets on the buffer

Addr	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
400560															mov	
400570		cmp			jne											

Figure 2: The 16-byte layout of a basic block that could span only one fetch line, but actually spans two. The gray cells represent instructions that are not executed.

boundary. Jiménez focuses on branch alignment [11]: after determining code regions where adding `nops` will not negatively affect performance, the system determines the effects of each possible number of one-byte `nops` from 0 to some maximum number. It then choses the number of `nops` to insert that maximizes the number of branches assigned to their correct pattern history table partition. While all of these works determine the number of `nops` in different ways and to affect different alignment issues, they are all similar in that their technique to perform realignment is `nop` insertion.

3. BASIC BLOCK CODE ALIGNEMENT SCORE

In order to understand how much room there is for dynamic realignment, we must first understand how well or poorly applications are aligned by the compiler. To do so, we developed a novel scoring mechanism which calculates an alignment score for an application, given various inputs. In the rest of this section, we will describe the Basic Block Code Alignment Score (BBCA score), which we use to score programs based on particular inputs to show that alignment is input-dependent.

Intuitively, the BBCA score calculates an individual alignment score for each basic block, based on the minimum cache lines for that basic block and the actual number of cache lines it fits on. It then takes the average of those scores where each basic block is weighted by their execution count. We must first score the individual basic blocks of the program. The score S of a basic block B is determined by calculating:

$$S_B = \frac{cachelines_{minimum}}{cachelines_{actual}},$$

where $cachelines_{minimum}$ is determined by calculating the minimum number of cache lines for a basic block of this size, assuming that it is aligned at the alignment boundary determined by the alignment factor:

$$lines_{minimum} = \left\lceil \frac{size}{alignment\_factor} \right\rceil,$$

and $lines_{actual}$ is determined by mathematically counting the actual number of lines the basic block spans:

$$lines_{actual} = \left\lceil \frac{address + size}{alignment\_factor} \right\rceil - \left\lfloor \frac{address}{alignment\_factor} \right\rfloor.$$

For example, in Figure 2, the basic block has a score of 0.5 because it could fit on one fetch line, but based on compilation alignment, actually spans two. In these equations, $alignment\_factor$ refers to the alignment boundary at which we wish to align the program. For example, if the instruction fetch unit fetches 16 bytes at a time, the $alignment\_factor$ would be 16.

The BBCA score of a program is a weighted average of the scores of the individual basic blocks. Specifically, the score, S, of program P is:

$$S_P = 100 \times \left(2 \times \frac{\sum_{b=0}^{\#bbl} S_b \times E_b}{totalexecutions} - 1 \right),$$

where S_b is the score of basic block b, E_b is the execution count for basic block b and $totalexecutions$ is the sum of the execution counts of all the basic blocks. Because factors like branch collisions are microarchitecture-dependent, we only consider maximum fetch throughput when calculating the BBCA score.

The average BBCA scores for the SPEC CPU2006 Integer benchmarks compiled with the optimization level `-O2` are shown in Figure 3. For each benchmark, we show the results for the `test`, `train`, and `ref` inputs in order to show the wide range of scores these benchmarks attain. We have included error bars that represent the highest and lowest scores that the benchmark achieved. While some benchmarks have only a $1-2\%$ swing in their scores, five of the benchmarks have a range of over 14%. With a wide range of inputs, we can conclude that these scores would not remain constant. We can conclude from Figure 3 that there is no one static alignment for which a program will have the same BBCA score for multiple inputs. Therefore, we need to know the dynamic execution path to find the best alignment for each input.

4. TRIGGERS FOR REACTIVE CODE ALIGNMENT

Because static compilers cannot accurately predict the microarchitecture on which the program will be executed, inputs to the program, or the program's dynamic branch behavior, we must perform many alignment optimizations dynamically. We refer to this type of alignment as *reactive* code realignment. A system performing reactive realignment would react to certain triggers that suggest poor alignment and then take steps to improve the alignment of the affected code. In this way, reactive code alignment is also *adaptive*: as the program executes, the system can adapt to changes in program execution and make different decisions based on the current execution path.

In order to reactively realign code, we must first recognize when code is poorly aligned. There are several ways code can be poorly aligned: 1) loops are not aligned at fetch line boundaries and therefore are not compressed into the fewest number of fetch lines as is strictly necessary; 2) for particular microarchitectures, including the Intel Core2 and the Intel i7, loops are not aligned appropriately for the Loop Stream Detector (LSD) that eliminates the need to fetch the instructions for every execution; and 3) branch instructions are aligned such that multiple branches are on the same fetch line, resulting in an increased number of branch mispredictions which require significantly more fetches.

There are several triggers that may be used to monitor alignment quality. For example, we can use the number of

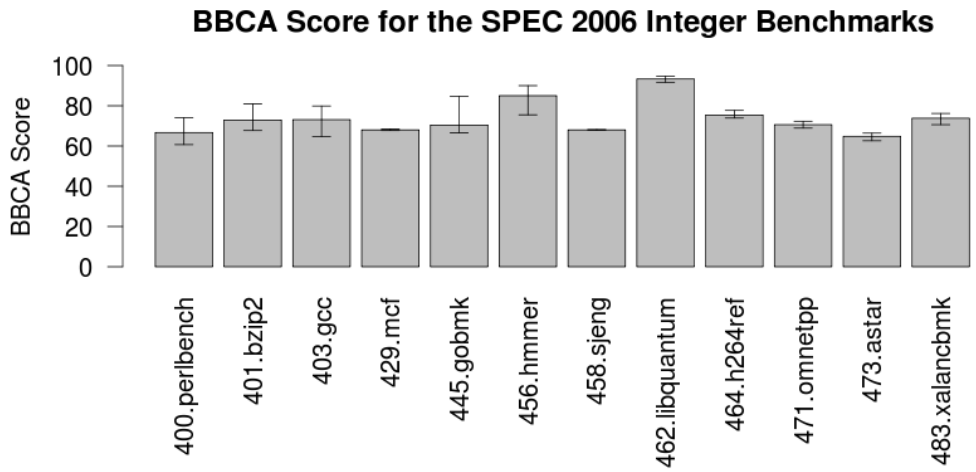


Figure 3: BBCA Scores for the SPEC 2006 Integer benchmarks running the `ref` inputs. The bars represent the average BBCA score while the error bars represent the minimum and the maximum score. For benchmarks with multiple `ref` inputs, this figure shows that there is a large range of BBCA scores for, showing us that the important code snippets change with the different inputs.

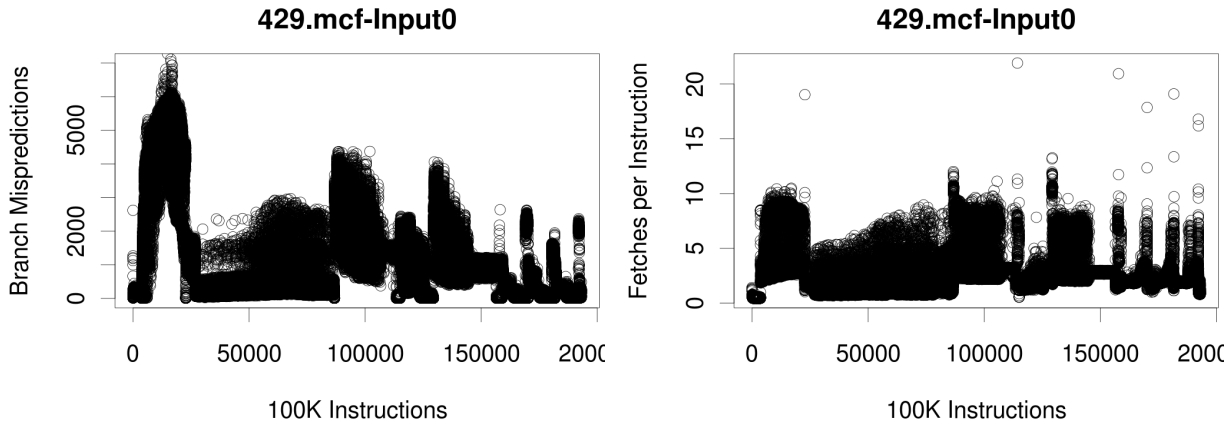
instruction fetches per instruction executed to monitor general code alignment, the number of branch mispredictions per branch to monitor branch alignment, and the number of uses of the Loop Stream Detector for those microarchitectures that have this technology. When we notice spikes, or dips in the case of the LSD, in the behavior of these triggers, we can adapt to those sudden changes by realigning the code in the previously executed code region. While not all of the code in the previous code region will be executed again, it is likely that the executed code is part of a loop that will be executed in the future. By realigning code as we notice problems, we can catch the most serious code alignment issues, i.e. highly executed code segments that were not aligned appropriately, while ignoring most of the insignificant code alignment issues, i.e. code that is only executed a few times at most.

Figure 4 shows the relationship between branch mispredictions (4a) and instruction fetches (4b) for the train input to `429.mcf` on an Intel Core2 processor. As we can see, the patterns for the behavior of these two performance counters are closely related: when the number of branch mispredictions increases, the number of instruction fetches per instruction also increases. This is intuitive because when a branch is mispredicted, all of the instructions that had been fetched on the mispredicted path are thrown away and the correct instructions must be then fetched. While we give `429.mcf` as an example, the other benchmarks exhibit this behavior as well.

If we were only interested in branch alignment, we could use only branch mispredictions as a trigger for reactive realignment. However, branch mispredictions do not always imply poorly aligned code, and may just represent a changing code pattern. Additionally, branch alignment is not the only code alignment issue that we must contend with. For example, all of the branches in a program may be appropriately aligned, but the important loops may be aligned across fetch boundaries, leading to an increased number of fetches for that section of code. For this situation, we must refer back to the number fetches per instruction, which can inform the system when the number of fetches is higher than what we expect it to be.

Using a performance counter that monitors the Loop Stream Detector as a trigger is also very limiting in the types of code alignment issues that we can catch. Additionally, knowing if the infrequent use of the Loop Stream Detector is actually an alignment issue requires the overhead cost of calculating if there are any loops that missed the LSD that should have been executed by it. A loop will be executed by the LSD on the Core2 if: 1) it fit into no more than four 16-byte fetches; 2) it has no more than 18 instructions; 3) it has no more than four taken branches and none of those branches may be RETs; and 4) it is executed for at least 64 iterations. Finally, the LSD is only available on the Intel Core2 and Intel i-series processors, and therefore this trigger is not generalizable. Even between the Core2 and the i-series, requirements for the LSD were changed to allow the LSD to detect more loops that the Core2's LSD often missed.

While neither branch mispredictions nor the Loop Stream Detector can predict all alignment issues, instruction fetches per instruction can point out all the alignment issues that those two triggers can, and additionally can indicate other alignment issues. These issues include when larger loops or loops with internal control flow are not aligned at fetch boundaries, when indirect branch targets cross fetch boundaries, and when both edges of branches with phase changes are not aligned optimally. By studying several benchmarks, we determined that the average number of fetches per instruction should be no greater than 1.4 on today's systems. In most cases, that is a high estimate, as other factors, including using the LSD, can decrease the average number of fetches per instruction. Figure 6 shows the instruction fetches per instruction of the example code snippet from Figure 5 with two different alignments on an Intel Core2 processor: in 6a, the branch of the inner loop and the branch of the outer loop collide, leading to millions of branch mispre-



(a) Branch Mispredictions for 429.mcf

(b) Fetches per Instruction for 429.mcf

Figure 4: Comparison of branch misprediction rate for 100,000 instructions and fetches per instructions in 100,000 instruction chunks.

```
int a = 0;
int i;
int ii;
for(ii = 0; ii < 500000000; ii++){
    for(i = 0; i < 5; i++)
    {
        a++;
    }
}
```

Figure 5: Code for the doubly nested loop microbenchmark.

dictions; in 6b, the two branches do not collide. By assuming that the expected number of instruction fetches per instruction should not exceed 1.4, we can easily notice that the code in 6a is poorly aligned. While we chose a threshold of 1.4, a reactive realignment system can self-tune: by monitoring a program and detecting if the average number of fetches per instruction is significantly lower than the existing threshold, it can lower the threshold.

5. EXPERIMENTAL RESULTS

In this section, we describe our experimental results and demonstrate the opportunities available for reactive code realignment. We first describe the opportunities that static alignment misses when considering the microarchitectural differences of different processors. Then, we highlight how different program inputs affect alignment decisions. Finally, we discuss the opportunities for reacting to dynamic branch behavior.

5.1 Experimental Setup

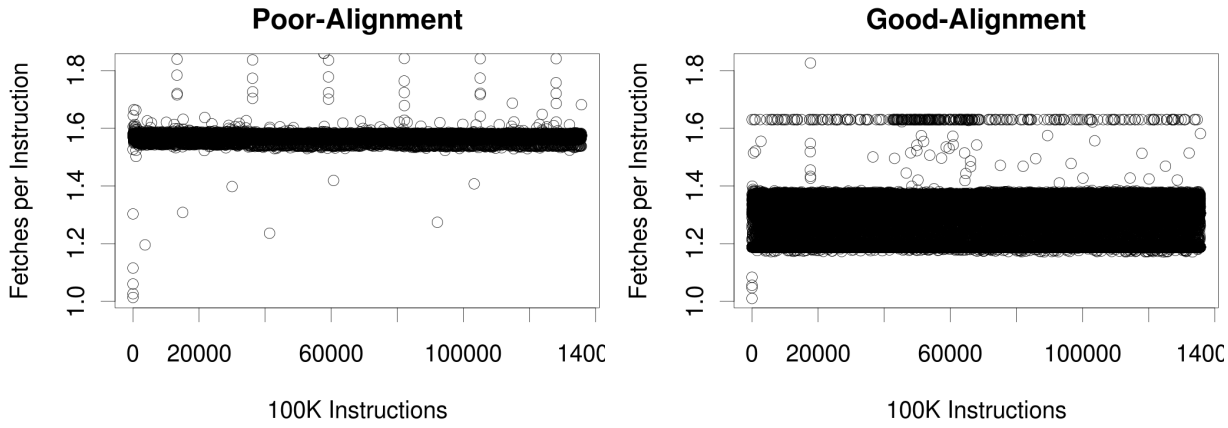
We conducted our experiments on a 1.86GHz Intel Dual Core2 processor with 4GB of RAM. This processor is based on the Penryn microarchitecture. The machine ran Ubuntu 11.04 with Linux kernel version 2.6.38. The Intel Core2 processor has many unique features that are often affected by the alignment of programs including the Loop Stream Detector and the branch predictor. For our experiments, we used the SPEC CPU2006 Integer benchmarks running both the ref and train input sets. We ran both ref and train input sets to see how different program inputs are affected by the alignment.

All benchmarks were compiled with GCC version 4.5.2 at optimization level `-O2`. We collected all basic block and branch profile information using Pin version 2.10 and a modified version of the `insmix` Pintool. We used the gnu utility `objdump` to create the object file for each of the benchmarks with address information for all instructions. Finally, to collect performance counter data, we used the Linux utility `perf`, and the `libpfm4` library to obtain performance counter data at finer granularity.

In order to compare performance on multiple microarchitectures, we used on three additional machines. We conducted our experiments on a 3.4GHz Intel Core i7 2600, codenamed Sandy Bridge with 16GB of RAM. While the Intel i7 processor also has the Loop Stream Detector, the requirements for the LSD were improved to allow more loops to hit the LSD. Also, the branch predictor was updated to avoid the branch misprediction issues that the Intel Core2 machine faced. Additionally, we conducted experiments on a 1.60GHz Intel Xeon E5310, codenamed Clovertown, with 8GB of RAM. This processor is based on the Intel Core microarchitecture. Finally, we conducted experiments on a 3.2GHz Intel Xeon, codenamed Dempsey, with 8GB of RAM. This processor is based on the Intel Netburst microarchitecture. All of these machines ran Ubuntu 11.04 with Linux Kernel version 2.6.38. All benchmarks were compiled on the Intel Core2 machine and then run on the other microarchitectures.

5.2 Reacting to Microarchitectural Differences

As summarized in Section 1, while modern processors execute `x86` code, there are several microarchitectural difference between these `x86` machines that can either be exploited to improve performance of executing code, such as the Loop Stream Detector, or must be accommodated in order to



(a) Poorly Aligned Branch Test

(b) Correctly Aligned Branch Test

Figure 6: Comparison of fetches per instruction for the code in Figure 5 where the code has been aligned to induce branch collisions (6a), and where the code has been realigned to avoid branch collisions (6b).

avoid poor performance, like the branch prediction unit on the Intel Core2. Many of these microarchitectural differences are difficult to account for statically if the compiler does not know *a priori* the destination microarchitecture. However, because dynamic approaches actually monitor the application behavior on the destination microarchitecture, they can a) recognize what microarchitecture they are executing on and adjust alignment of the entire program before execution begins, and b) adapt to the issues that the currently executing application is experiencing by modifying the alignment during execution.

To demonstrate how alignment issues are microarchitecture dependent, consider the code in Figure 5. As we described in Section 1, on the Intel Core2, a branch collision occurs when aligned poorly such that the two branches span the same fetch line. When a branch collision occurs, the performance of the microbenchmark is 20% worse than when they do not span the same fetch, which avoids the collision. Likewise, on the Intel Core-based microarchitecture, we see the same phenomenon. However, when we run these two versions on either the Intel i7 or the Intel Netburst microarchitecture, we do not experience this difference in performance. In fact, on the Netburst microarchitecture, the performance of the version that was poorly aligned actually performed 2% faster than the version correctly aligned for the Core2. This can be attributed to the fact that the version correctly aligned for the Core2 spans one additional fetch line, compared to the poorly aligned version. This shows us that what is "correct" for one processor may in fact cause performance issues on another processor.

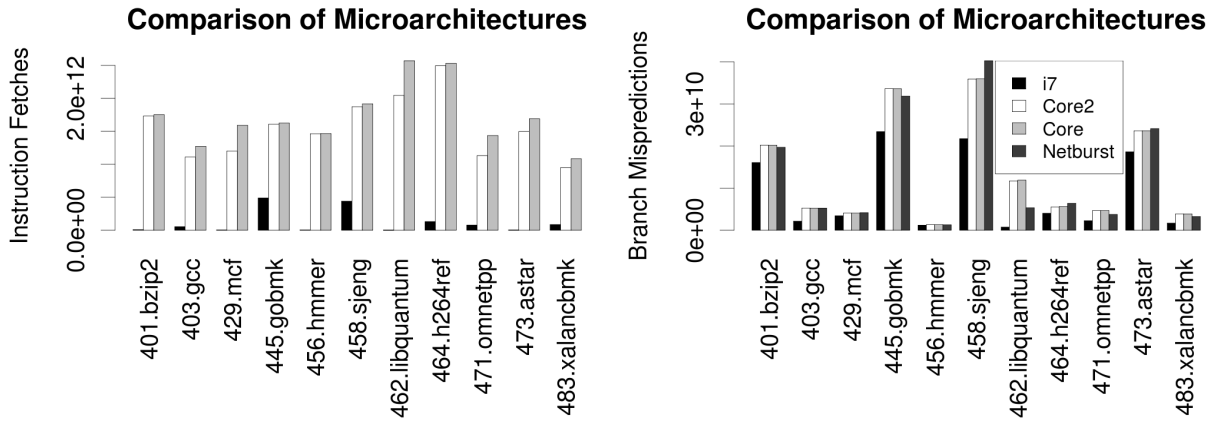
Figure 7 shows the comparison of several microarchitectures based on instruction fetches and branch mispredictions. We ran the SPEC CPU2006 Integer benchmarks executing the train inputs on the Intel Core i7, Intel Core2, Intel Core, and Intel Netburst microarchitectures. We can see from these results that for both instruction fetches and branch mispredictions, the behavior on all of these microarchitectures is very different. In particular, the Core2 and the Core microarchitecture have a similar branch misprediction issue that is not evident on the i7 or Netburst microarchitectures, which do not suffer from branch collisions. If we were to compile only considering the behavior of one of these microarchitectures, we may dramatically affect the performance of the application on other microarchitectures. In a large scale computing system made up of cores that are not uniform, adapting to the current microarchitecture can improve the overall throughput of the entire system.

5.3 Reacting to Program Inputs

At compile time, there is little that we can assume about the inputs to a program. Compiler heuristics do their best to align as much of the code as possible, but are restricted to minimize code bloat. Therefore, many code segments are not aligned appropriately due to incorrect assumptions by the compiler as to how often they will be executed versus the cost of aligning. Additionally, many compilers restrict how much padding can be used to align a particular code segment. Rather than spreading out that alignment cost over all of the code before a particular point (by using more padding than is strictly necessary at several different points), the compiler simply gives up on aligning that block of code. This can lead to poorly aligned, frequently executed code.

Additionally, since the compiler must rely on heuristics to perform alignment, it will not align for all possible inputs. Where one input may take one code path, a different input, not predicted by the compiler, may take a completely different code path. This can lead to the program being appropriately aligned in many cases, but poorly aligned in a very important case, depending on how the compiler chose to align the program. Dynamically, we can track how the program is executing, and how well or poorly the code is aligned for a given input. We can use that information to then adapt the alignment of the code so that it is more appropriate for the given input. In a large scale computing system where each core may be executing the same program with different inputs, adapting to those inputs becomes extremely important in order to maximize total throughput of the system, in particular when those inputs exercise different code paths.

Figure 8 shows the fetches-per-instruction for 100K chunks



(a) Comparison of Instruction Fetches

(b) Comparison of Branch Mispredictions

Figure 7: Comparison of instruction fetches and branch mispredictions for several different microarchitectures running the SPEC CPU2006 Integer benchmarks with the reference input. Note, Figure 7a does not include data for the Netburst microarchitecture because `perf` cannot interpret raw performance counter codes on Netburst.

of the execution of `401.bzip2` for each of its three train inputs. As we can see, none of these inputs have the same fetches-per-instruction behavior. Whereas input 1, shown in Figure 8a, has very regular behavior, generally staying under 2 fetches-per-instruction over the execution of the benchmark, the behavior of input 2, Figure 8b, is spiked: there are periods where the number of fetches-per-instruction is low, but there are spikes of poor fetch-per-instruction performance. The behavior of Input 3, Figure 8c, is between inputs 1 and 2: Its behavior gradually improves and declines over the course of execution with a few peaks of bad behavior. For all of the benchmarks with multiple train inputs, we can see these marked differences between the fetch-per-instruction behavior for all of the inputs. Additionally, we can also see these differences when comparing train inputs for the benchmarks to the ref inputs. These differences show us that, while a benchmark may be aligned well for a particular input or code path, it is not well aligned for all inputs. Since we cannot know all of the possible code paths for all inputs for a reasonably complex application, we must adapt to the runtime behavior in order to align the program well for all inputs, rather than attempting to predict runtime behavior at compile time.

5.4 Reacting to Runtime Branch Behavior

One specific instance of input dependent program behavior is branch behavior. Based solely on the input to the program, branches may be taken or not taken, loop iteration counts may change, and indirect branch targets may change. Statically, we cannot know what the specific branch behavior will be, so decisions are made based on heuristics. However, these heuristics can be misguided for many executions, leading to runtime slowdowns for important inputs.

Additionally, branches may exhibit phase changes. A phase change occurs when a branch consistently is either taken or not taken, and then switches to the opposite behavior for a significant amount of time. It may then exhibit another phase change and switch back, or it may finish execution in the second phase. Figure 9 demonstrates a phase change.

```
int a = 0;
int i;
for(i = 0; i < 50000; i++){
    if(a < 25000)
        a++;
    else
        a+=2;
}
```

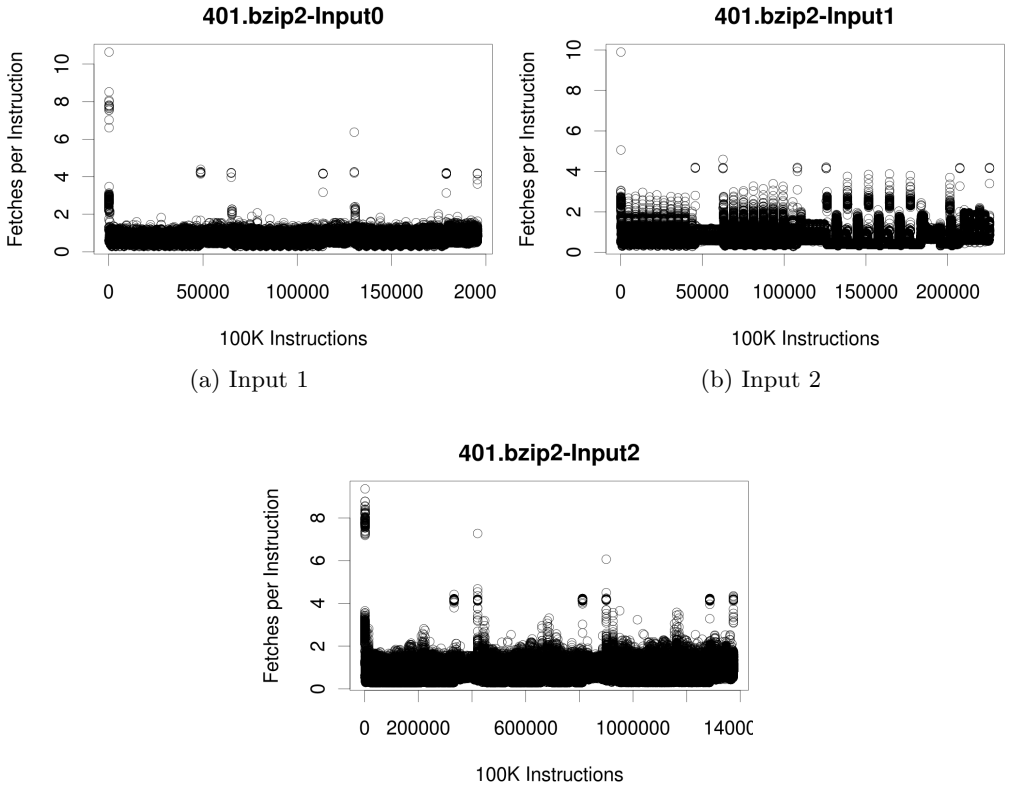
Figure 9: Code snippet demonstrating a phase change.

For half the execution, `a` is increased by one. For the second half of the execution, `a` increases by 2. When a phase change occurs, we want both edges of the `if` statement to be aligned appropriately. While this is a simple operation, current compiler alignment directives will only align the branch target, and not the fall through edge.

5.4.1 *Indirect Branches*

Indirect branch target alignment is difficult for static compilers because, at compile time, the most frequent target of an indirect branch is often difficult to predict. At runtime, the situation is simplified. In particular, reactive realignment can be informed by the triggers that we described in Section 4, as well as frequency information about the code. Additionally, compilers may attempt to align all of the possible branch targets for an indirect branch, introducing a significant amount of padding in the program. In a dynamic system, we could remove this padding, and only align those targets that are both poorly aligned and frequently executed.

The static number of indirect branches for the SPEC CPU 2006 Integer benchmarks are shown in Table 1. Additionally, we show the static average number of targets per branch and the dynamic average of targets per branch. As we can see, the dynamic count of targets is significantly lower than the static average. In particular, there are significantly fewer targets at runtime that require proper alignment for



(a) Input 1

(b) Input 2

(c) Input 3

Figure 8: Fetches-per-instruction for 100K chunks of the execution of `401.bzip2` for each of its 3 train inputs.

`445.gobmk` than at compile time. This is due to the fact that `445.gobmk` is made up of several huge `switch` statements to make decisions for future moves while playing the game go. This shows us that at runtime, we can align significantly fewer code segments than we may at compile time. Additionally, at runtime, we can decrease the code bloat by eliminating this unnecessary padding required for aligning all of the indirect branch targets. Finally, several of the benchmarks show that significantly fewer of the indirect branches executed at runtime. Therefore, in order to decrease the size of the benchmarks, alignment of indirect branches should be left to dynamic systems.

5.4.2 Branch Phase Changes

Phase changes occur in several of the SPEC 2006 Integer benchmarks. Table 2 displays the number of train inputs, and the average, minimum, and maximum number of branches that have phase changes over all of the inputs for the specific benchmark. Of the SPEC 2006 Integer benchmarks, the two benchmarks with the most obvious branches with phase changes are `429.mcf` and `458.sjeng`. Figure 10a is a graphical display of the phase changes for one of the most important branches in `429.mcf`. In this figure, 0 represents that the branch was not taken, while 1 represents taken. When we look at this branch in the source code, we discover that it is a branch nested in a while loop. Similarly, we discover that the most important branch with distinct phases in `458.sjeng`, shown in Figure 10b is a `case` within a `switch` statement.

When a branch experiences a distinct phase change, we want each of edges of the branch to be well aligned at the time that it is being executed. However, due to some limitations of the compiler, we may end up with one or both of the edges of the branch poorly aligned. For example, the example branch of `429.mcf` shown in Figure 10a jumps to address `0x401e38` when compiled with the optimization level `-O2`. The basic block that the branch jumps to ends with a `callq`, and ends on address `0x401e52`. Because of limitations of the compiler, the basic block is aligned such that it spans one more fetch line than is necessary. Dynamically, we could recognize that the basic block is frequently executed, and shift it 8 `nops` so that it spans the minimum number of fetch lines.

We also need to be aware of phase changes so that we can guarantee that the super block containing both the branch and the fall through edge is well aligned. Rather than requiring the fall through edge to be properly aligned, we want the basic block containing the branch before the fall through edge, plus the basic block containing the fall through edge to be properly aligned. If the compiler's heuristics do not predict that the fall through edge will be frequently executed, it may only align the previous block well, which could push the entire super block onto an extra fetch line. Dynamically, we can recognize when the fall through edge is frequently executed and adapt by appropriately aligning the entire super block. Because half of the SPEC CPU2006 Integer benchmarks have at least one branch that experiences

Benchmark	Static Branches	Dynamic Branches	Average Static Indirect Targets per Branch	Average Dynamic Indirect Targets per Branch
400.perlbench	78	60.6	19.7	7.8
401.bzip2	2	1	24.0	7.7
403.gcc	412	91	13.3	7.6
429.mcf	0	0	0	0
445.gobmk	13	8	17.9	2.8
456.hmmer	24	1	9.2	1
458.sjeng	15	8	8.7	7.8
462.libquantum	0	0	0	0
464.h264ref	11	3	6.8	4.7
471.omnetpp	12	11	7.9	3.1
473.astar	0	0	0	0
483.xalancbmk	82	40	10.7	1.2

Table 1: Static and dynamic counts of indirect branches and indirect branch targets for each of the SPEC CPU2006 Integer benchmarks running the train inputs.

Benchmark	Input Count	Average	Minimum	Maximum
400.perlbench	5	3.2	0	11
401.bzip2	3	0	0	0
403.gcc	1	2	2	2
429.mcf	1	2	2	2
445.gobmk	8	0.625	0	2
456.hmmer	1	0	0	0
458.sjeng	1	3	3	3
462.libquantum	1	0	0	0
464.h264ref	1	0	0	0
471.omnetpp	1	0	0	0
473.astar	1	1	1	1
483.xalancbmk	1	0	0	0

Table 2: Average, minimum and maximum number of phase changes for the SPEC 2006 Integer benchmarks based on many different program inputs.

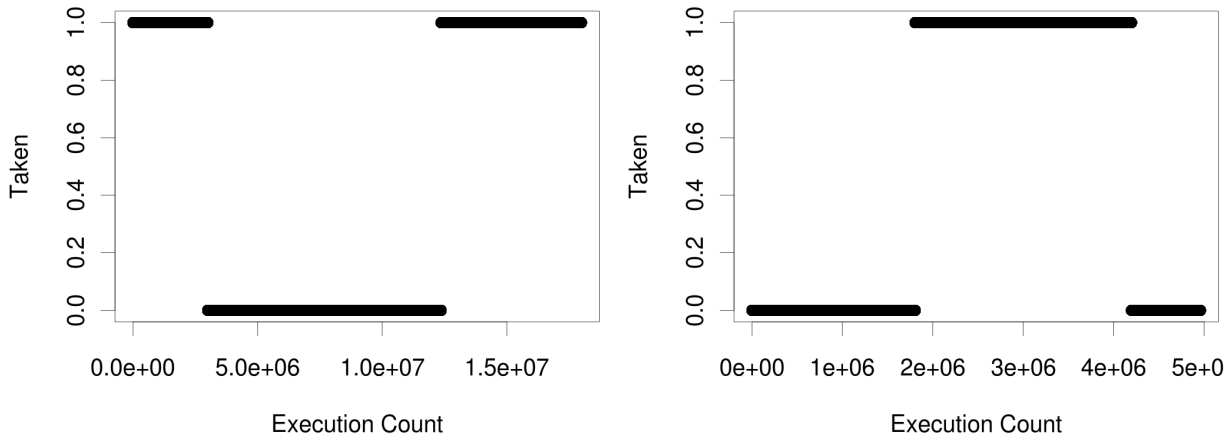
phase changes, we can conclude that branches with phase changes are an opportunity for reactive realignment.

6. REACTIVE REALIGNMENT IN PRACTICE

Reactive realignment requires an added abstraction layer to monitor hardware performance while executing applications. This layer would sit between the hardware and software, like a virtual machine, and modify the software as it recognizes poor performance. There are several ways this could be achieved with off the shelf tools currently available. Reactive realignment techniques require two stages: 1) a method of monitoring hardware performance based on the triggers we described in Section 4, and 2) a way of modifying the running application and instructing the hardware to execute the new, correctly aligned code. In Section 4, we suggested using low overhead performance counters to monitor the performance of the application. In our experiments, we used `libpfm4` to monitor the behavior of the hardware at a finer granularity than the entire program execution. We found that by checking the performance counters every 100,000 instructions, we introduced little overhead in the overall application performance. For example, the overhead of running at the sampling rate for `429.mcf` is about 1%. Additionally, this granularity allowed us to observe the performance of the benchmarks at fine enough granularity that we could notice when performance was degrading as the most important regions of code were executed significantly more often than 100,000 times.

In addition to monitoring the behavior of the applications, we need a way to modify the running application. There are several mechanisms that can achieve this: a system-level VM like VMware, a dynamic binary translator (DBT) which could periodically invalidate cached traces that exhibit poor alignment, or a Java Virtual Machine (JVM) where this approach could be added to its advanced optimization phase. Because of the overhead of constructing a control flow graph dynamically, using any of these solutions exclusively for alignment purposes would be unlikely to provide net gains for anything more than the most basic insertion of padding. However, our alignment technique should integrate well with existing infrastructures like a JVM with multiple optimization phases or as a preprocessing phase in a dynamic binary translator in order to reduce the overhead of the DBT system.

For example, by integrating this technique into the advanced optimization phase of a JVM, we could recognize both when the hardware performance counters suggest there is an alignment issue and when the JVM recognizes a hot loop on which it has decided to perform more optimizations. Since the JVM knows this is a loop, after it performs the optimizations which change the underlying code, it could compute the proper alignment for this code as well. Com-



(a) Branch with phase changes for 429.mcf

(b) Branch with phase changes for 458.sjeng

Figure 10: Phase changes for one of the most important branches in `429.mcf` running the train input and one of the most important branches in `458.sjeng`. There are two phase changes for both of these branches. Ref inputs show phase changes in both branches.

pared to the other optimizations, this computation would be cheap, and it would help to improve the performance of the running application. Furthermore, while we can compute the proper alignment multiple times if necessary, our approach will generally only be required once for a particular code region. Therefore, once an appropriate alignment is found, our dynamic approach will essentially go to sleep and incur little to no additional overhead.

7. RELATED WORK

There has been significant work on runtime effects due to cache performance; however, most of this research focuses on minimizing cache misses [1, 2, 8, 17, 18, 19]. By minimizing cache misses, energy spent in accessing memory is decreased, and the overall application runtime is improved. For example, Huang *et al.* show that by using dynamic code reordering techniques, we can decrease the number of instruction cache misses and improve performance [8]. Huang *et al.* also studied three different code layout algorithms which they incorporated in a JIT compiler [9]. Their system modified the call graph of a program, identifying the heaviest edges in the graphs and merging the nodes connected by those edges, specifying their code layout. Their work differs from ours because we propose using performance counter data to trigger code realignment.

Prior work has also focused on data alignment for better reuse of data items [13, 16]. For example, Panda *et al.* use a data alignment technique that is based on the padding of arrays in order to stabilize cache performance and maximize cache utilization, resulting in better performance [16]. Instruction alignment has also been used to improve branch prediction [6, 7, 11]. Using profile information, Gloy *et al.* find they can minimize any negative runtime effects of static correlated branch prediction, and that by doing so, their technique actually performs better than traditional static branch prediction. Similarly, Jiménez uses pattern history table partitioning, which uses a feedback directed code placement technique that place conditional branches where they will not interfere with other branches, in order to improve branch prediction accuracy [11]. While all of these techniques improve runtime, they only address one particular issue, and none of them consider how different issues negatively affect one another.

Some research has been done that looks at the effects of code alignment on performance considering microarchitectural features [3, 5, 15]. Conte *et al.* [5] show that for highly-parallel microarchitectures, `nop` insertion can play a role in improving performance, but that inserting `nops` at basic block boundaries can cause too much code inflation to be worthwhile. However, their work looks at incorporating their techniques into hardware by inserting the alignment stage into the pipeline between the instruction cache and the instruction decoder. Similarly, Merten *et al.* [15] look to both profile and realign code in hardware. We, however, consider a software approach where we monitor runtime behavior and only modify code alignment if the program is experiencing poor fetch-per-instruction behavior. This allows the program to be modified only when necessary, decreasing the overhead of the technique.

8. CONCLUSION AND FUTURE WORK

In the exascale era, static techniques are often ineffective for most decisions. In particular, for large scale computer systems made up of cores of many different microarchitectures running different program inputs on the same program, static alignment decisions cannot account for all of the possible combinations of microarchitecture and program input. In this paper, we have presented the opportunities for runtime realignment. We have shown that there are many instances in which static alignment is insufficient, and have presented three major dynamic opportunities that a reactive alignment system can leverage. We have shown that microarchitectural differences, changing program inputs, and runtime branch behavior can all impact alignment decisions, none of which may be known at compile time.

Additionally, we have identified the instruction fetch per-

formance counter as a trigger that monitors executing programs and identifies when alignment issues are present. By monitoring the program during execution, the instruction fetch performance counter can account for many different alignment issues that may be present in the binary including code segments with high execution count that are misaligned, colliding branches which result in increased fetch counts, and loops that may miss the Loop Stream Detector or other similar microarchitectural features. Additionally, by using the instruction fetch performance counter as a trigger, we can adapt to the execution of the program and make different decisions based on the code that is currently executing, overcoming the limitations of static alignment techniques.

Our plans for the future are to incorporate our dynamic feedback scheme into dynamic optimization schemes, including JIT compilers and dynamic binary translators. In order for reactive realignment to have a positive effect on application performance, the feedback mechanism and the realignment scheme must must have low overhead. Therefore, we also intend to study the overhead of these stages of our system in order to maximize the benefit of the system while minimizing the cost.

9. REFERENCES

[1] E. Athanasaki and N. Koziris. Fast indexing for blocked array layouts to reduce cache misses. *International Journal on High-Performance Computer Networks*, 3:417–433, March 2005.

[2] K. W. Batcher and R. A. Walker. Dynamic round-robin task scheduling to reduce cache misses for embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 260–263. ACM, 2008.

[3] O. Boehm, G. Haber, and H. Kosachevsky. Code alignment for architectures with pipeline group dispatching. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 23:1–23:7. ACM, 2010.

[4] J. B. Chen and B. D. D. Leupen. Improving instruction locality with just-in-time code layout. In *In Proceedings of the USENIX Windows NT Workshop*, pages 25–32. USENIX, 1997.

[5] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344. ACM, 1995.

[6] N. Gloy, M. Smith, and C. Young. Performance issues in correlated branch prediction schemes. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 3 –14, November 1995.

[7] C. Hu, J. McCabe, D. A. Jiménez, and U. Kremer. The camino compiler infrastructure. *SIGARCH Computer Architecture News*, 33:3–8, December 2005.

[8] X. Huang, S. M. Blackburn, D. Grove, and K. S. McKinley. Fast and efficient partial code reordering: taking advantage of dynamic recompilation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 184–192, 2006.

[9] X. Huang, B. T. Lewis, and K. S. McKinley. Dynamic code management: improving whole program code locality in managed runtimes. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 133–143, 2006.

[10] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1 –10, april 2011.

[11] D. A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107–116. ACM, 2005.

[12] D. Kreitzer. Personal communication, 2010.

[13] J. Li, C. Wu, and W.-C. Hsu. An evaluation of misaligned data access handling mechanisms in dynamic binary translation systems. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 180–189. IEEE Computer Society, 2009.

[14] A. Mendlson, S. S. Pinter, and R. Shtokhamer. Compile time instruction cache optimizations. *SIGARCH Computer Architecture News*, 22:44–51, March 1994.

[15] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W.-m. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 59–70. ACM, 2000.

[16] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142 –149, Feb. 1999.

[17] K. Rajan, R. Govindarajan, and B. Amrutur. Dynamic cache placement with two-level mapping to reduce conflict misses. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 422. IEEE Computer Society, 2007.

[18] H. Stolberg, M. Ikekawa, and I. Kuroda. Code positioning to reduce instruction cache misses in signal processing applications on multimedia risc processors. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 699–702. IEEE Computer Society, 1997.

[19] C. Zhang. Reducing cache misses through programmable decoders. *ACM Transactions on Architecture and Code Optimization*, 4:5:1–5:31, January 2008.