Ispike: A Post-link Optimizer for the Intel®Itanium® Architecture

Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, Geoff Lowney

Massachusetts Microprocessor Design Center
Intel Corporation

Submitted to CGO'04

Abstract

Ispike is a post-link optimizer developed for the Intel®Itanium Processor Family (IPF) processors. The IPF architecture poses both opportunities and challenges to post-link optimizations. IPF offers a rich set of performance counters to collect detailed profile information at a low cost, which is essential to post-link optimization being practical. At the same time, the predication and bundling features on IPF make post-link code transformation more challenging than on other architectures. In Ispike, we have implemented optimizations like code layout, instruction prefetching, data layout, and data prefetching that exploit the IPF advantages, and strategies that cope with the IPF-specific challenges. Using SPEC CINT2000 as benchmarks, we show that Ispike improves performance by as much as 40% on the Itanium®2 processor, with average improvement of 7.7% and 9.1% over executables generated by the Intel®Electron compiler and by the Gcc compiler, respectively. We also demonstrate that statistical profiles collected via IPF performance counters and complete profiles collected via instrumentation produce equal performance benefit, but the profiling overhead is significantly lower for performance counters.

1 Introduction

Post-link optimization [4, 8, 17, 20, 22] is a technique to improve the performance of a program *after* it is compiled and linked. By directly operating on the executable, it has several advantages. First, it can see the entire program and perform optimizations across procedures that may be in different source modules. Second, it is relatively easy to use profile feedback since the same executable is being profiled and optimized. In contrast, mapping profile information back to the source code is more challenging due to the compiler transformations that have been done in between the source code and the executable. Third, it is applicable even when the program source is unavailable, which may be the case in some commercial or legacy codes. These advantages make post-link time optimization appealing, particularly in a production environment.

This paper is about post-link optimization on the Intel®Itanium Processor Family (IPF) processors, under the Linux®operating system (abbreviated as *IPF/Linux*). The IPF architecture provides a number of features that facilitate post-link optimization. In particular, its *fine-grain performance monitoring* can identify performance bottlenecks at the instruction level, and software can apply optimizations precisely to remove these bottlenecks. Moreover, the rich set of events that can be monitored on IPF enable detailed performance analysis. However, IPF also poses challenges to post-link optimizations. Specifically, *predication* makes post-link code transformation in general and branch inversion in particular a challenging task on IPF.

We have developed a post-link optimization tool called *Ispike* for IPF/Linux. Besides standard optimizations, it implements a number of key optimizations targeting *memory latency*, including code layout, instruction prefetching, data layout, and data prefetching. They are driven by the branch profiles, I-cache miss profiles, and D-cache miss profiles collected via the IPF performance counters. We apply these optimizations to IPF/Linux executables generated by the Intel®Electron [6] and the Gcc compilers. For SPEC CINT2000, these optimizations improve performance from 2% to 40% on the Itanium®2 processor, with average improvement of 7.7% and 9.1% over Electron and Gcc, respectively. We also demonstrate that statistical profiles collected with IPF performance counters provide the same performance benefit as complete profiles collected with in-

Name	What it captures	What are recorded
Branch Trace Buffer (BTB)	Last 4 to 8 branches	Branch's PC, branch target's PC, mispredict status
Instruction Event Address Register (I-EAR)	Last I-cache miss	Instruction PC, miss latency in cycles
	Last I-TLB miss	Instruction PC, who serviced the miss: L2 I-TLB, VHPT, or software
Data Event Address Register (D-EAR)	Last D-cache miss	Instruction PC, data address, miss latency in cycles
	Last D-TLB miss	Instruction PC, data address, who serived the miss: L2 D-TLB, VHPT, or software

Table 1: IPF hardware structures for instruction-level profiling. For I-EAR and D-EAR, we can monitor either cache misses or TLB misses, but not both at the same time.

strumentation.

The rest of this paper is organized as follows. First, we describe our profiling infrastructure in Section 2. Next, we discuss Ispike optimizations in Section 3. We then describe our solutions to a number of IPF implementation issues in Section 4. We report our experimental results in Section 5. Finally, we relate Ispike to other work in Section 6, and conclude in Section 7.

2 Profiling Infrastructure

In this section, we first introduce the IPF performance monitoring unit. We then describe a Linux tool called *pfmon* which we use to collect data from the performance counters. Finally, we discuss the profiling support inside Ispike itself.

2.1 IPF Performance Monitoring

A design philosophy of the IPF architecture is that software plays a major role in optimizing program performance. Many software optimizations require information about the program's run-time behavior. To provide this information, IPF includes performance monitoring hardware [9] that supports two complementary usage models: *workload characterization* and *instruction-level profiling*. Workload characterization is measuring the performance characteristics of the workload under study. Two types of information are of particular interest: how often an event occurs, and how the cycles are spent (so called *cycle accounting*). For event counting, the Itanium®2 processor provides four 48-bit performance counters and over 100 events that can be monitored with these counters. For cycle accounting, IPF provides a way to break down the total cycles into various categories of stalls and flushes. We will explain these categories when we do the cycle breakdown for our results in Section 5.2.2. As for instruction-level profiling, the hardware attributes events like branches, cache and TLB misses to individual instructions so that software can know exactly where to optimize in the program. IPF implements three hardware structures for this purpose: *Branch Trace Buffer (BTB)*, *Instruction Event Address Register (I-EAR)*, and *Data Event Address Register (D-EAR)*; they are described in Table 1. By performing statistical sampling on these structures, precise instruction-level profiling can be done at a low cost.

2.2 Perfmon and Pfmon

The IPF/Linux kernel provides an interface for controlling the performance monitoring hardware: the *perfmon* APIs [14]. A tool called *pfmon* [14] utilizes the perfmon APIs to do event counting, cycle accounting, and instruction-level profiling. We have enhanced pfmon in several ways to make it more suitable for profile-guided optimization. First, we incorporate a sample-aggregation mechanism into pfmon. This avoids dumping out raw samples in the middle of the profiling session and hence reduces the profiling overhead. Second, we extend the *per-task* mode in pfmon to monitor all processes forked by the task, instead of just the initial process. Third, we add a module that detects strides in load-miss addresses using the D-EAR. More details of this are in Section 3.4.

2.3 Ispike Profiling Support

Ispike accepts six types of profiles: I-cache misses, I-TLB misses, D-cache misses, D-TLB misses, load-miss strides, and branches. Processing the first five profile types is relatively simple as we mainly need to attach the miss count to the corresponding instruction. Processing branch profiles is more complicated. Since we sample only *taken* branches (including direct and indirect, branches and calls) in the branch trace buffer (BTB), Ispike derives the counts of fall-through edges based on Kirchoff's laws (i.e. flow in equals flow out at each basic block).

Besides processing profiles, Ispike also provides numerous tools for analyzing profiles and their impact on optimizations. Three of them are illustrated in

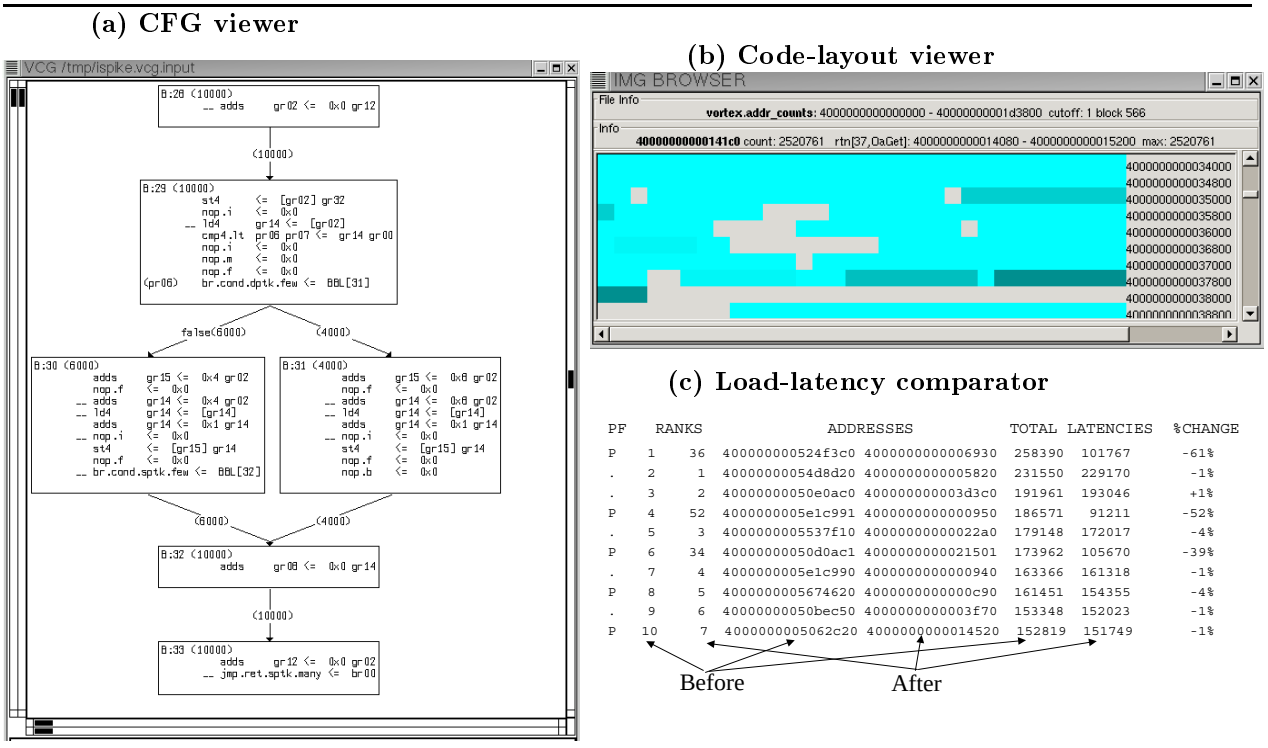

Figure 1: Three profile-analysis tools in Ispike.

Figure 1. The *CFG viewer* is a graphical display of the program's control-flow graph (CFG) based on the VCG tool [21]. The snapshot shows the basic blocks and edges annotated with execution counts. The tool is useful for debugging optimizations and inspecting code for new optimization opportunities. The *code-layout viewer* visualizes the execution frequency of instructions in an image at cache line granularity: frequently executed lines have a darker color. It allows for a quick visual validation of the effectiveness of the code-layout optimization. The *load-latency comparator* compares the latency of loads before and after an image has been processed by Ispike, assuming that load latency profiles for both images are collected. Since the position of a load in an image will change because of optimizations, Ispike must track loads while processing an image. Combined with prefetching reports produced by Ispike, the comparator becomes a powerful tool to evaluate the effectiveness of data prefetching. The snapshot above shows the top ten loads in the original image. A load that Ispike attempted to prefetch is marked with the letter 'P'. By looking at the %change in the load latency, we can tell whether a prefetching scheme is beneficial for a particular load.

3 Ispike Optimizations

The main goal of Ispike optimizations is to cope with *memory latency*, a major performance bottleneck on modern machines. There are two general approaches. The first is to *reduce* latency by improving locality. The second approach is to *tolerate* latency by prefetching. To improve locality, Ispike rearranges the layout of both code and data based on profiles. It also prefetches both code and data. The details of these optimizations are given in the rest of this section.

3.1 Code Layout

Our profile-driven code-layout optimization is inspired by Pettis and Hansen's algorithm [18]. It has three aims: (i) increasing I-cache performance by improving locality, increasing cache line utilization, and eliminating cache conflicts, (ii) reducing number of control flow changes, and (iii) reducing the number of active code pages and thereby increasing the I-TLB hit rate. This optimization is particularly important for programs with large instruction footprints.

Our algorithm consists of three steps, which we explain using the examples in Figure 2. The first step, *basic-block chaining*, tries to put basic blocks in sequence if there is a frequently executed control

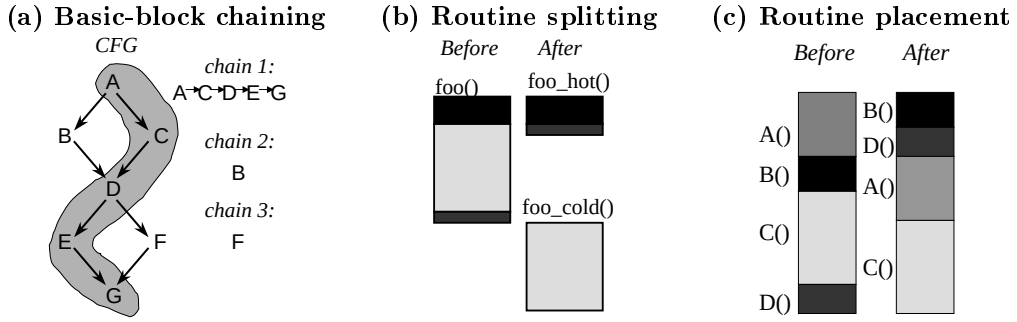

Figure 2: Three steps of our code-layout algorithm.

flow edge between them. For each routine, we sort the edges in the routine's flow graph in descending order of their execution counts. Some adjustments are made to the execution counts before the sorting to bias the algorithm when basic blocks have only a single successor or predecessor. We then walk down the sorted edge list in a greedy fashion chaining basic blocks together if they are the head and tail of their respective chains and those chains are different. For the flow graph shown in Figure 2(a), assuming that the shaded path is hot, then three chains will be formed. The second step, *routine splitting*, sorts the resulting chains by the maximum edge count within each chain. We compare this count against a threshold to determine whether a chain is hot or cold. Hot chains and cold chains are then placed in two separate regions, as shown in Figure 2(b). The third step, *routine placement*, arranges the regions resulting from the previous step by their density, which is defined as the average instruction execution frequency of each routine. Putting hot routines close together reduces their chance of conflicting with each other in the I-cache and reduces the number of code pages. This step is illustrated in Figure 2(c).

The Ispike code generator will preserve the basic block sequences computed above. Unconditional branches will be inserted as necessary and conditional branches will be *inverted* if the *taken* branch target happens to be the next basic block in the sequence. While branch inversion is straightforward on architectures that can branch on predicate true or false, it is non-trivial on IPF which can only branch on predicate true. We will discuss our methods of inverting branches on IPF in Section 4.2, when we talk about implementation issues. We have also experimented with excluding branch edges from the chaining process if the corresponding branches cannot be inverted. This did not yield any performance gain and is therefore disabled by default.

3.2 Instruction Prefetching

Even with code layout, there are still some I-cache misses that are not covered. They typically happen at call and branch targets that are far away. To cope with these misses, researchers have proposed using *instruction prefetching* [2, 12]. IPF provides two software-controlled mechanisms for prefetching instructions [9], namely *streaming prefetching* and *hint prefetching*.

Streaming prefetching initiates hardware prefetching of sequential cache lines at the targets of dynamically predicted taken branches. To invoke streaming prefetching upon a particular branch, we use the instruction `br.many target`. Sequential lines are prefetched starting at `target` plus 64 or 128 bytes (depending on the alignment of `target`). Ispike decides whether to use streaming prefetching for a branch by estimating the size of the *span* starting at the branch's target. A span runs from the target to the first statically predicted taken branch (including unconditional branch). Streaming prefetching is used if the span's size is at least 128 bytes. This makes sure that we will prefetch only instructions that are going to be used at the branch's target.

Hint prefetching allows software to prefetch a particular instruction line. A hint prefetch is either a `brp.few target` or `brp.many target`. A `brp.few` prefetches one cache line at `target`, whereas a `brp.many` prefetches two cache lines. Hint prefetching is intended to be used together with streaming prefetching such that the first one or two lines of a branch's target are prefetched via a hint prefetch, while the rest is via a streaming prefetch. In order to fully hide the latency of an I-cache miss that hits in the L2, the hint prefetch should precede the branch by at least 9 fetch cycles. To find out which instruction lines need to be prefetched, we collect the I-miss profile using the I-EAR. For each hot instruction line T, we put `brp.few T` in the predecessor basic blocks of T in the flow graph of the whole program that

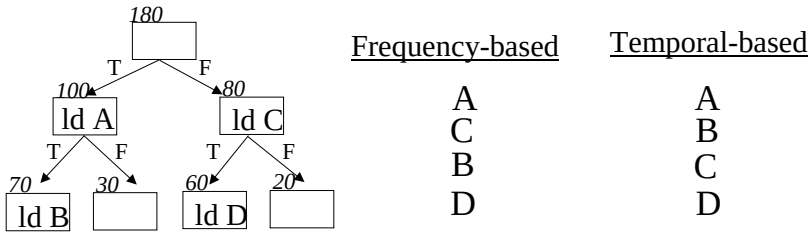

Figure 3: Frequency-based vs. temporal-based data layout. The execution count of each load is shown above the top-left corner of the block that contains the load.

satisfy the following two criteria: (i) the predecessor block P must be at least 18 instructions (9 cycles $\times$ 2, assuming an IPC of 2) ahead of T on an execution path between P and T, which may be in the same or different routines, and (ii) P must have at least one `nop` (no operation) slot where we can place the `brp`. This avoids any code size increase due to the additional `brp`s. We keep inserting `brp`s for T at the predecessors that satisfy both criteria until 95% of T's miss latency would be covered. This coverage estimation is based on path probabilities, which are derived from the edge counts.

One practical issue in using an I-miss profile to drive prefetching is that the code-layout optimization will change the I-cache behavior. Therefore, if we want to apply instruction prefetching *after* code layout, we need to recollect the I-miss profile of the program with code layout and then insert prefetches. This requires profiling and applying Ispike twice.

3.3 Data Layout

Similar to code layout, Ispike also rearranges data for better locality. However, reordering data is much more challenging in terms of preserving program correctness. Thus, we limit our scope to reordering *statically allocated* data, which we call *global*. Our global data reordering algorithm is based on the one recently proposed by Haber *et al.* [7]. The major difference is that their execution profiles are collected via instrumentation, while ours are collected via performance counters. Following is a brief description of this algorithm.

Global data is defined in the *data* sections of the image, including global variables, constants, switch-statement target addresses, or function addresses. On IPF/Linux, global data references are typically made through the special register *gp*. For each global data symbol v, we aggregate the execution frequencies of the instructions that access it. This aggregated value, denoted by $H(x)$, represents the hotness of x. Since data symbols are of different sizes, we normalize their hotness against their sizes: $NH(x) = H(x)/sizeof(x)$. In other words, $NH(x)$ measures the hotness per byte of x. Based on NH, we have two algorithms to reorder global data. The first one is to simply sort symbols in descending order of NH. We call this a *frequency-based* algorithm. The second one is a *temporal-based* algorithm, which packs together symbols that are accessed close to each other in time during execution. Figure 3 illustrates the different data layouts resulting from these two algorithms. The figure shows a flow graph corresponding to two levels of if-then-else, which contain loads of four global variables A to D. Assume the same size for the four variables. The frequency-based algorithm results in the data layout: A, C, B, D. In contrast, the temporal-based algorithm results in the data layout: A, B, C, D. The temporal layout is a better one in this example because A and B are on a disjoint path from C and D.

The temporal-based algorithm requires building a *data connectivity graph* (DCG). A node in the DCG corresponds to a global data symbol. Two symbols, say x and y, are connected by an (undirected) edge in the DCG if a reference to y immediately follows a reference to x on some path during program execution, and vice versa. The weight of the edge (x, y) is the total frequency of the x-followed-by-y instances and the y-followed-by-x instances. Once the DCG is built, we lay the global data out by traversing the DCG, starting with symbol that has the largest NH. For each symbol x visited, we place x at the current address in the global data area and increase this address by $sizeof(x)$. Then we select the next symbol to visit to be one of those that are connected to x in the DCG, which have not been visited so far. To ensure that symbols placed together are of comparable hotness, we add a criteria that the new symbol selected must have a $NH >= NH(x) * Threshold$, where $Threshold$ is a parameter between 0 and 1, and is chosen as 0.3 in our experiments. If there are multiple symbols connected to x that satisfy this criteria, we select the one that has the largest edge weight to

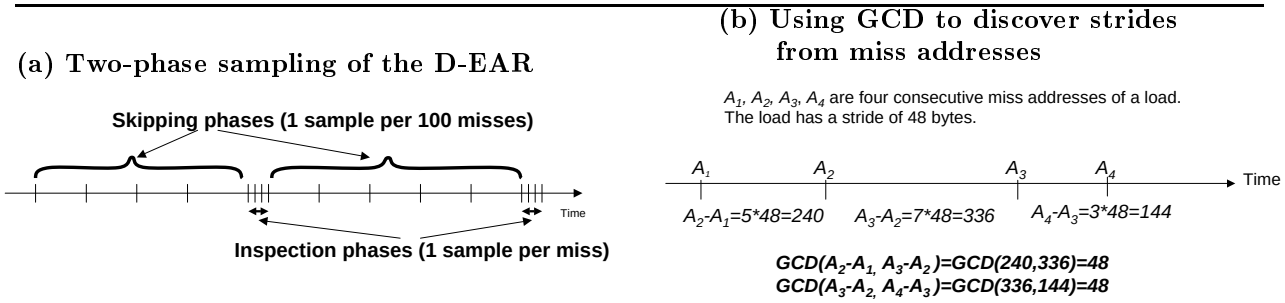

Figure 4: Stride profiling with the D-EAR.

visit next. However, if there is no such symbol, we will pick the symbol that has the largest NH among those that have not been visited, regardless of their connectivity to x. We repeat this process until all symbols in the DCG are visited.

In applying both frequency-based and temporal-based data layout to various applications, we find that the temporal-based algorithm consistently outperforms the frequency-based one. Therefore, we will present only the temporal-based results later in Section 5.

3.4 Data Prefetching

Ispike implements *stride-based prefetching*, targeting address strides that are not statically detectable by the compiler and thus need to be determined through *stride profiling*. Wu [23] performs stride profiling while the program is running and dynamically decides whether prefetching should be performed for a particular load based on the profiling results. To minimize the impact of stride profiling overhead on the overall performance, his scheme tends to be conservative in choosing which loads to profile and hence may miss some prefetching opportunities. In contrast, Luk *et al.* [13] perform stride profiling in a separate profiling pass. This allows Luk *et al.* to profile more loads than Wu. However, their profiling overhead is large (a 15x slowdown on average for SPEC CINT2000) because their profiling is based on *instrumentation*.

We have developed a stride profiling scheme [11] that can profile most of the loads that we would consider for prefetching at a relatively low cost (5% profiling overhead on average for SPEC CINT2000). We use the D-EAR performance counter to capture load events that *miss* in the D-cache. To achieve low profiling overhead, we sample the D-EAR in two phases with different sampling rates, as illustrated in Figure 4(a). The *skipping* phase uses a lower sampling rate: one sample per 100 misses. When enough samples have been collected, we switch to the *inspection* phase which uses a much higher sampling rate: one sample per miss. Stride detection is done in the inspection phase as follows. Each D-EAR sample provides three pieces of information about the miss: $< pc, daddr, lat >$, where pc is the PC of the load captured in the miss event, $daddr$ is the data address loaded, and lat is the miss latency in CPU cycles. For each load, we look at four consecutive misses and compute the deltas in the data addresses between each consecutive pair ($\Delta_i = daddr_{i+1} - daddr_i$, where $i = 1..3$). Standard stride detection checks whether $\Delta_1 = \Delta_2 = \Delta_3$. However, since our data addresses are *miss* addresses instead of *reference* addresses, we are uncertain about the number of strides actually included in each Δ_i. Fortunately, we are certain that if a stride S does exist, all these deltas should be some multiple of S. Therefore, we can discover S by computing the greatest common denominator (GCD) of these deltas. If $GCD(\Delta_1, \Delta_2) = GCD(\Delta_2, \Delta_3)$, then this GCD is the stride or a small multiple of it. Figure 4(b) illustrates this process.

Once the stride S of a load is found, Ispike inserts an instruction `lfetch R` immediately after the load, where `lfetch` is the data-prefetch instruction on IPF. `R` is a register assigned a value equal to the load's current data address plus the product $S * d$. We discuss how we allocate this register specifically on IPF in Section 4.3. The parameter d is the prefetching distance, which is either a user-specified constant or determined by some compiler heuristics [15]. If there are multiple strides detected for a load, we prefetch the most frequent two of them.

3.5 Other Optimizations

In addition to the four memory-oriented optimizations mentioned above, Ispike also includes numerous optimizations that improve performance by reducing the number of instructions executed. These include inlining, dead-code elimination, branch forwarding, store-load forwarding, and GOT-access optimization. Among these optimizations, *GOT-access optimization* has the biggest performance impact on

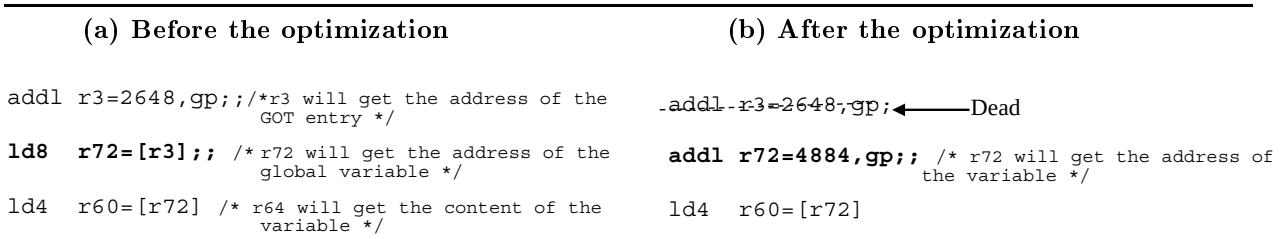

Figure 5: GOT-access optimization.

our benchmarks, and thus we explain it in more detail.

On IPF/Linux (and the run-time models for many other architectures), accesses to global variables often occur indirectly via the *Global Offset Table(GOT)* and a special reserved register *gp*, the global pointer which usually points somewhere into the middle of the GOT. Figure 5(a) gives a typical code idiom for reading a global variable. There are two reasons for the extra level of indirection. First, storing addresses of global variables in the GOT allows them to be resolved or even changed by the dynamic loader (a process called *symbol preemption*). Second, offsets encoded in the `addl` instructions are limited in size. The GOT is compact as it only contains 64-bit addresses and hence the offsets from the *gp* will be small enough for `addl`.

The optimization performed by Ispike will replace the code idiom with the one from Figure 5(b). Here we assume that the address of the variable accessed is equal to $gp + 4884$. To preserve correctness Ispike must ensure that the corresponding GOT entry cannot be changed (or preempted) by the dynamic loader and that the variable is close enough to the *gp* that the offset in the `addl` instruction will not overflow. The first condition is trivially satisfied for static images; for shared images Ispike consults relocations and symbols. The second condition is also easily verified by Ispike as the image is fully linked and all addresses are known.

It is worth noting that Ispike is in a unique position to perform this kind of optimization. The compiler alone cannot perform this transformation as it cannot make guarantees about preemption and offset ranges. A smart linker together with some cooperation from the compiler could probably substitute the code idiom but would not be able to eliminate the first (dead) instruction.

Applying the data-layout optimization from Section 3.3 can facilitate this GOT-access optimization since hot global variables can be grouped together in a smaller range of addresses close to the *gp*. A similar optimization opportunity was also observed by Haber *et al.* [7] in their data-layout work.

4 IPF Implementation Issues

In this section, we address four issues of implementing Ispike optimizations on IPF. First, we discuss a feature called *call shadow* which affects adding/deleting instructions in IPF binaries in general. Second, we discuss how to invert branches when performing code layout. Third, we discuss how to obtain free registers to compute data-prefetch addresses. Finally, we discuss the code scheduling required for IPF.

4.1 Call Shadow

On IPF, instructions are grouped into *bundles*, each of which typically contains three instructions. The targets of all control transfers, including calls and returns, are aligned at bundle boundaries. An instruction i is under a *call shadow* if it follows a *predicated* call instruction in the same bundle b. If the call is not taken, i will be executed; however if the call is taken, control will be returned to the bundle immediately following b after the call, and thus i will be skipped. For instance, the instruction `br elsewhere` in Figure 6(a) is under a call shadow. The presence of call shadows poses a challenge to post-link time optimization: if we add or delete instructions without special care, the instructions under call shadows could be pushed to the following bundles and so will be executed even after returning from the predicated calls, violating the original semantics. To solve this problem in existing binaries, we adopt the transformation proposed by Ramasamy and Hundt [19], which uses *trampolines* to eliminate call shadows before applying any optimizations. Figure 6(b) illustrates this transformation. To avoid this problem in future binaries, we have also worked with the Intel compiler team to make the tool chain free of call shadows.

4.2 Branch Inversion

As part of code layout, *branch inversion* is needed to convert the targets of frequently taken branches to fall-throughs. On IPF, the direction of a branch is determined by the value of its predicate register.

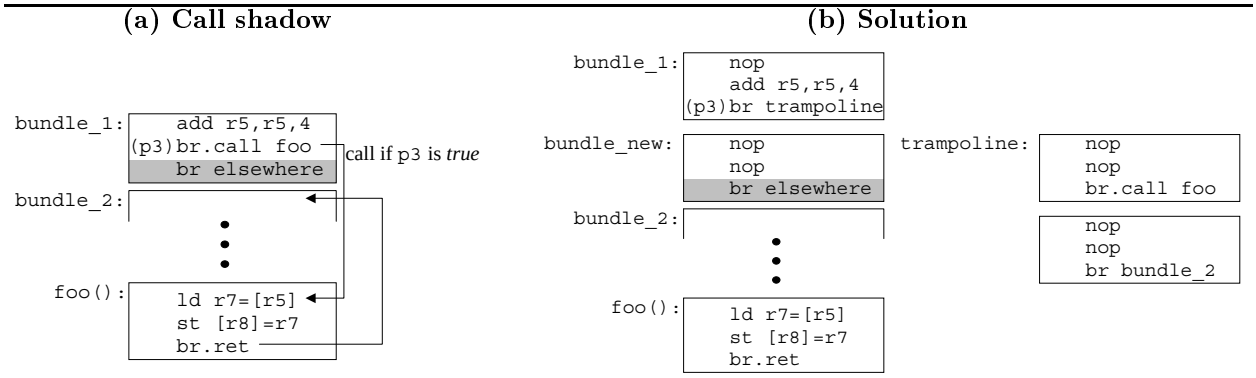

Figure 6: Handling call shadows.

To invert a branch, we need the *complement* of its predicate. Ispike finds the instruction that defines the branch's predicate, which is typically a compare instruction (`cmp`). If the complement is already computed by the compare and saved in another predicate register p_c, then we can use p_c to invert the branch provided that p_c is not redefined between the compare and the branch. Unfortunately, there are many compare instructions that either do not compute the complement or do not store it in another predicate register. One such case is shown in Figure 7(a), where the `cmp` instruction has predicate register `p0` as the complement, which cannot be used for branch inversion because `p0` is hardwired to 1. In that case, Ispike uses *liveness analysis* [16] to find a free predicate register for holding the complement for branch inversion, as shown in Figure 7(b). With this technique, the number of dynamic inverted branches out of the total number of dynamic branches that we want to invert is dramatically improved from 9.3% to 75.0% on average for the SPEC CINT2000 binaries compiled by the Electron compiler. For the remaining cases where we cannot find any free predicate register or the compare instruction does not compute a complement[1], although we cannot invert the branch, we can still move the branch's target closer to the branch by adding a new branch and switching the locations of the fall-through block and the branch target. This approach is shown in Figure 7(c). Even when Ispike cannot invert a branch, we can still increase I-cache locality at the expense of an unconditional branch.

[1] An additional problem is that comparisons with a `NaT` input are not invertible (A `NaT` is a value that indicates a speculative operation has occurred with a deferred exception). We avoid this problem by an agreement with the compiler to not generate conditional branches that are controlled by a comparison with a `NaT` input. The following scheduling rules accomplish this: (i) branches are kept in their original order, and (ii) speculation check instructions are placed in their home block. These rules ensure that the check for speculative exceptions occurs *before* the branch and the comparison will be re-evaluated with non-`NaT` inputs.

4.3 Finding Registers for Prefetch Addresses

One major issue in implementing post-link stride prefetching on IPF is the need for an extra register to hold the prefetch address. This is not an issue for architectures that have the base-plus-offset addressing mode in their prefetch instructions (e.g., `prefetch 32[R]`), because the prefetch address can be generated by adding a new offset to the base register of the load being prefetched. However, the prefetch instruction on IPF (`lfetch`) does not have an offset field and hence the whole prefetch address has to be explicitly computed and stored in a register. Our problem is how to get this register at the post-link level, where register allocation has already been done.

We have three solutions to this problem. First, we perform *liveness analysis* [16] to find free registers at the points where we want to prefetch. If no free register is found, we attempt our second method–allocating an additional register on the *register stack* [10] provided by the IPF hardware. Typically, a procedure allocates its own frame on the register stack by executing an `alloc` instruction as one of the first instructions in the procedure. Ispike allocates additional registers for prefetching by increasing the frame size of `alloc`, up to the maximum size of the register stack frame (96 registers). If there is no `alloc` or there are multiple `alloc`'s in the procedure or the original frame size has already reached the maximum, we cannot use this method and must turn to our final method. On IPF, both load and prefetch instructions have a *base-update-immediate* form, where an given immediate value is added to the base register after the memory access. Our last resort is to increment the load's base register by the prefetch offset, and then perform a prefetch with the same base register but a decrement of the same prefetch

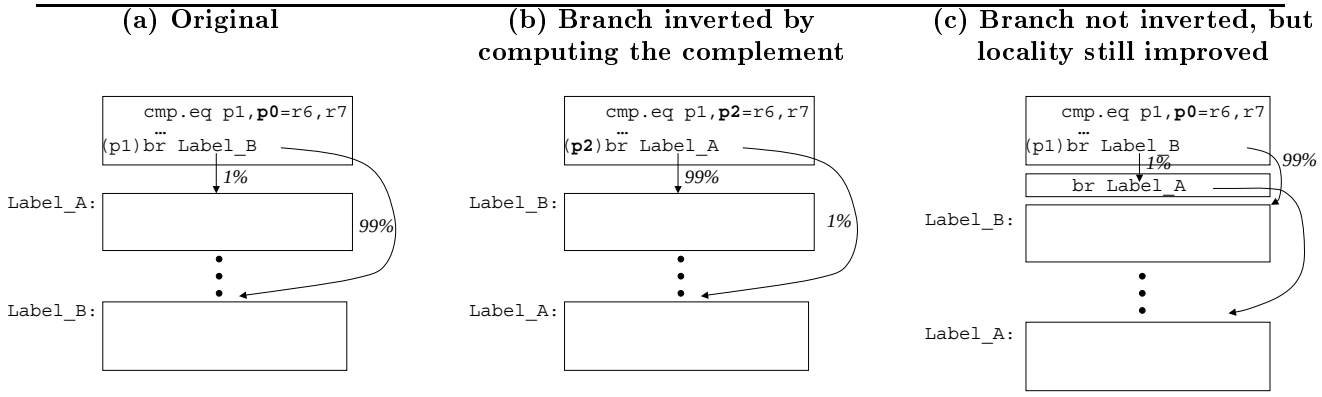

Figure 7: Branch inversion techniques.

offset. Hence, the base register's value is restored after the prefetch. This approach works as long as the load's base register is different from the destination register and the prefetch offset can fit in the 9-bit immediate field. If all three solutions fail, we do not prefetch that particular load.

4.4 Code Scheduling

High-quality code scheduling is important for performance on IPF. The scheduling rules are complicated and often quite subtle [5]. Also, the IPF instruction set encodes three instructions per bundle. To avoid the complexities of maintaining the original schedule and bundling throughout all the phases of Ispike, we reschedule all code. This makes the other optimizations simpler, and permits our intermediate representation to ignore bundling issues.

A compiler typically has more knowledge about the memory aliasing issues than Ispike, and it can use this knowledge when scheduling. The Ispike scheduler leverages the compiler's knowledge by favoring the original code order. The result is a scheduler that both maintains the performance of the original image and integrates the changes made by our optimizer with little or no overhead.

5 Experimental Results

We now report the performance impact of Ispike optimizations on the Itanium®2 processor. We first describe our experimental setup. Then we discuss the results of Ispike optimizations that are driven by performance-counter profiles. Finally, we compare statistical profiles collected via performance counters against complete profiles collected via instrumentation in terms of their profiling times and resulting performance impact.

5.1 Framework

The test bed of our experiments was a HP Everest server with four 1GHz Itanium®2 processors and 16GB memory. Each processor has three levels of on-chip caches: 16KB L1I/16KB L1D, 256KB L2, and 3MB L3. Only one processor was used throughout our experiments. Our system runs Red Hat Enterprise Linux AS with the 2.4.18 kernel. We used SPEC CINT2000 as our benchmarks. The *training* data sets were used to generate profiles, while the *reference* data sets were used for performance measurement. Each benchmark was run to completion five times, and the median execution time was reported. We applied Ispike optimizations to *non-shared* binaries generated by two different compilers: the Intel®Electron compiler (Ecc) version 8.0 Beta, and the GNU C compiler (Gcc) version 3.2. We used the O3 optimization level in both compilers, which produces aggressively optimized baselines for our profile-guided optimizations.

5.2 Results driven by Performance-counter Profiles

We used our modified pfmon to collect the performance-counter profiles needed to drive Ispike optimizations. We ran each baseline *once* to simultaneously collect three types of profiles: branch traces (the BTB counter), load misses, and strides (both use the D-EAR). Since the goal of the experiment in this section is to maximize performance, we use relatively high sampling rates: one BTB sample per 10,000 branches, one D-EAR sample per 100 load misses, and for stride profiling, one sample per 100 misses in the skipping phase and one sample per miss in the inspection phase. The total profiling overhead with these sampling rates is 59% on average. Later in Section 5.3, we will show that reducing these sam-

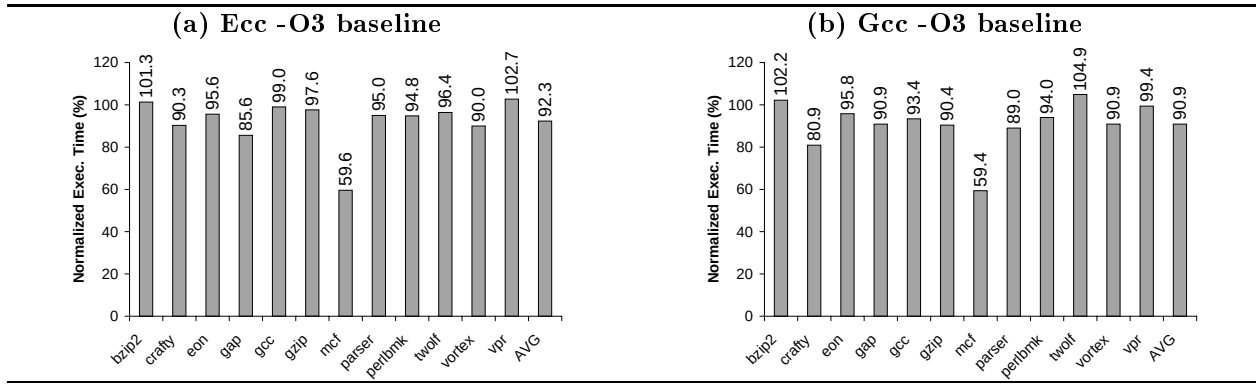

Figure 8: Performance improvement of Ispike optimizations on SPEC CINT2000 binaries compiled by (a) *Ecc* and (b) *Gcc*. AVG is the arithmetic mean.

pling rates by 10 times will lower the average profiling overhead to only 3%, at the expense of 1.7% less performance gain.

Figure 8 shows the results of applying Ispike optimizations to the Ecc-generated and Gcc-generated baselines. Code layout, streaming instruction prefetching, data layout, data prefetching, and the other optimizations discussed in Section 3.5 are all applied at the same time. As we mention in Section 3.2, hint instruction prefetching requires an extra profiling/optimizing pass. Nevertheless, we found that hint prefetching does not bring additional benefit in these programs. Therefore, we do not include it in Figure 8 to avoid the extra pass.

5.2.1 Overall Performance

Execution time is reduced by as much as 40%, with an average improvement of 7.7% in the Ecc baseline and of 9.1% in the Gcc baseline. Larger speedups are observed in the Gcc baseline since it is less optimized than the Ecc baseline. Nine out of 12 benchmarks (`crafty`, `eon`, `gap`, `gcc`, `gzip`, `mcf`, `parser`, `perlbmk`, and `vortex`) are sped up in both baselines. The slowdowns observed in the three benchmarks (combining both baselines) have different reasons: for `bzip2` and `twolf`, Ispike's instruction scheduler generates less optimized code than the compiler's scheduler; for `vpr`, the new data layout is somehow worse than the original. Overall, Ispike optimizations achieve significant speedups in both baselines.

5.2.2 How were the Cycles Spent?

To get insight into *how* our optimizations affect execution time, we used the cycle-accounting mechanism available on IPF to break down execution time into various stall and flush reasons. The breakdown for the Ecc case (i.e. Figure 8(a)) is shown in Figure 9 (the breakdown for the Gcc case is omitted due to space limitation). Each benchmark has two bars: one for the baseline version and one for the optimized version. Each bar represents the execution time normalized to the baseline case and is broken down into six categories. *Busy* is the cycles where the processor's backend is *not* stalled. That is, the processor retires at least one instruction in each of these cycles. The remaining five categories are all stall or flush cycles. *Front-end* is the stall cycles due to the lack of instructions delivered from the processor's front-end, usually because of I-cache misses and I-TLB misses. *L1D-access* is the stall cycles in accessing the D-cache due to various reasons such as a store in conflict with a returning fill. However, this does *not* include the stalled cycles experienced by the consumers of loads. Instead, these cycles are separately counted under *Load-to-use*. *Br-mispredict* are the cycles where the pipeline is flushed due to branch mispredictions and interrupts. All remaining stall cycles are lumped together as *Other*.

Figure 9 demonstrates that our various optimizations do help different components of the execution time. Our I-cache optimizations (code layout and prefetching) reduce *Front-end stalls* in all benchmarks except `vpr`. Our D-cache optimizations (data layout and prefetching) reduce *Load-to-use stalls* in eight benchmarks, most dramatically in `mcf` due to stride prefetching. Our other optimizations, in particular the GOT-access optimization, also reduce the *Busy* component by executing fewer instructions.

5.2.3 Contributions of Individual Optimizations

Having seen the *combined* improvement of Ispike optimizations, we now investigate the contributions of individual optimizations. Figure 10 shows the results for the Ecc baseline (Gcc baseline results are again

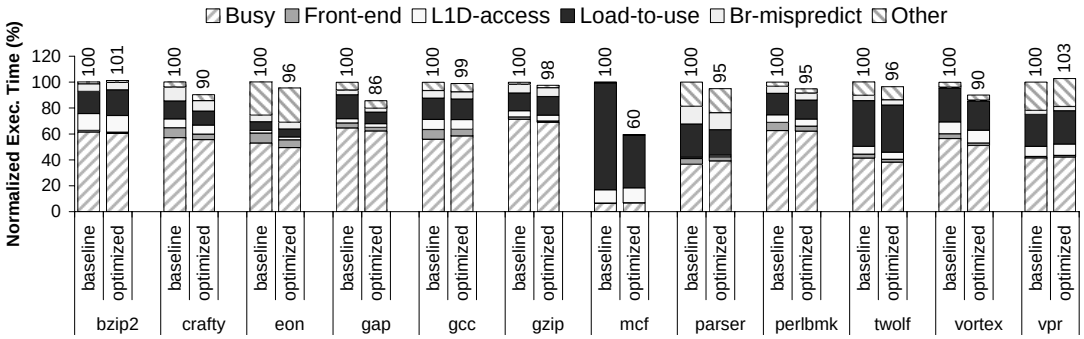

Figure 9: Cycle breakdown into various stall and flush reasons for the Ecc baseline and its optimized version.

omitted due to space limitation). In Figure 10(a), we first apply code layout and then incrementally add streaming prefetching and hint prefetching. Code layout alone improves `eon`, `gap`, and `vortex` by over 4%. Streaming prefetching helps a little, most noticeably in `perlbmk`. However, hint prefetching does not yield additional benefits. Instruction prefetching does not produce much gain in these benchmarks because only a small amount of time is spent on front-end stalls, especially after code layout, as evidenced in Figure 9. Nevertheless, we do observe larger speedups with instruction prefetching in a commercial database application which has a much bigger code footprint than our benchmarks. In Figure 10(b), the performance of data layout and data prefetching is shown separately. Except for `vpr`, the performance with data layout is either improved or unchanged. It is because this optimization has no instruction overhead. In contrast, data prefetching requires additional instructions for computing the prefetch addresses and for the prefetches themselves. As a result, while we enjoy substantial speedups in `gap`, `mcf`, and `parser` with data prefetching, we also suffer slowdowns in a few benchmarks. Finally, we show the improvements due to other optimizations alone in Figure 10(c). Among all other optimizations, the GOT-access optimization is the one that provides the biggest improvement. The gains in `crafty` and `vortex` largely come from this optimization. For these two benchmarks, we also note that the combined performance improvement is bigger than the sum of the improvements from individual optimizations. This is because the data-layout optimization creates more opportunities for the GOT-access optimization.

5.3 Profiling Overhead vs. Performance Impact

One practical consideration of profile-guided optimizations is the cost and accuracy tradeoff in profile collection. In general, more accurate profiles come at the cost of longer profiling time. In this section, we vary the profiling overhead to two extremes and measure their resulting performance. At one extreme, we use low sampling rates in pfmon. At the other extreme, instead of using performance-counter profiles, we use a *dynamic instrumentation* tool on IPF called *Pin* [3] to collect profiles that are functionally equivalent to pfmon profiles. Complete instrumentation-based profiles are the most accurate though the cost of collecting them is typically much higher.

Figure 11 shows the overhead and performance improvement of several profiling schemes. Figure 11(a) includes the run-time overhead of three statistical profiling schemes based on performance counters, with different sampling rates. The notation used is: BTB=$1/b$ means one BTB sample per b branches; D-EAR=$1/d$ means one D-EAR sample per d load misses; Stride=<$1/s$,$1/i$> means one D-EAR sample per s load misses in the skipping phase and one sample per i misses in the inspection phase. The first bar is the default sampling rates that we have been using so far. When we reduce the BTB sampling rate by 10 times in the second bar, the average profiling overhead is down to 24%. And when we also reduce both the D-EAR and stride sampling rates by 10 times in the third bar, the profiling overhead becomes only 3% on average. Figure 11(b) is the overhead of complete instrumentation-based profiling. We separately collect three types of profiles: edge profiles, load-latency profiles, and stride profiles. So, the total overhead of collecting all three profile types is at least as much as the maximum of collecting any of them. Finally, Figure 11(c) shows the performance improvements achieved with these various profiling

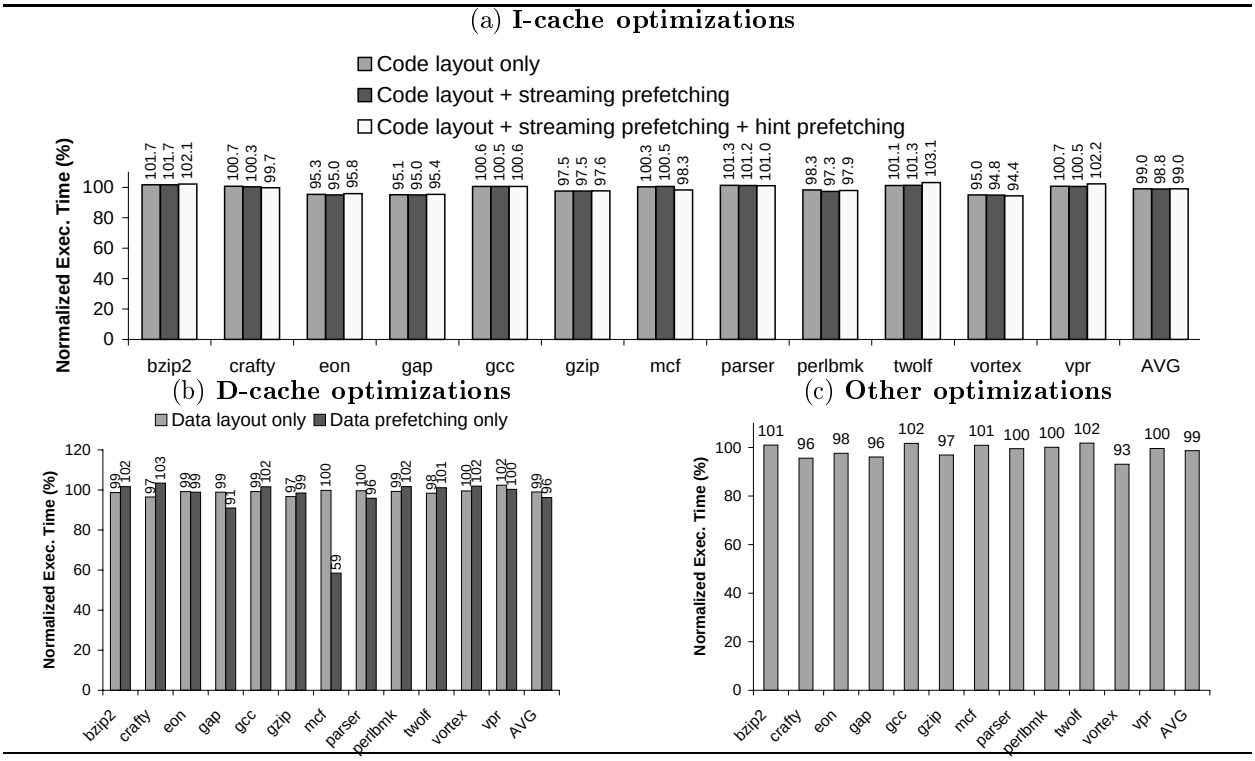

Figure 10: Performance impact of individual Ispike optimizations on the *Ecc* baseline.

schemes.

The profiling overhead with performance counters is two orders of magnitude less than that with Pin's dynamic instrumentation. Nevertheless, it should be noted that the current focus of Pin is on providing *general* instrumentation; little effort has been spent on minimizing profiling overhead. Thus, instrumentation profiling overhead will be reduced in the future. The really good news we learn from Figure 11 is that we do not need to sacrifice much performance for fast profiling. As Figure 11(c) shows, using instrumentation profiles to optimize does not provide additional performance over our default scheme that uses performance-counter profiles. In fact, performance-counter profiles result in a noticeably higher performance than instrumentation profiles in `mcf`. The reason is that stride profiling via D-EAR captures strides between *misses*, while stride profiling via instrumentation captures strides between *references*. It turns out that miss strides are better candidates for prefetching than reference strides. When we lower the BTB sampling rate from our default to the third bar in Figure 11(c), we suffer a 0.8% performance drop while the profiling overhead is significantly reduced from 59% to 24%. And when we also lower the D-EAR sampling rate by 10 times to the fourth bar where the profiling overhead is only 3%, although we observe substantial performance drops in `gap` and `parser` (because a number of strides become undetected), the average performance improvement is still within 1.5% of that with instrumentation profiles. Overall, performance-counter profiles produce as good speedups as instrumentation profiles, but incur substantially less overhead.

6 Related Work

A number of static post-link optimizers have been developed for different architectures, including FDPR for the IBM®Power® [8], Etch [20] for the Intel®Pentium®, and Alto [17] and Spike [4] both for the Compaq Alpha. To the best of our knowledge, Ispike is the first post-link optimizer developed for the Itanium architecture, and thus we need to cope with a new set of challenges in implementing post-link optimizations that are unique to this architecture.

Most studies on profile-guided optimizations are based on instrumentation profiles. Optimizers that use hardware-counter profiles include Morph [24] and Spike [4]. To collect profiles, Morph implements its own Morph Monitor, while Spike uses DCPI [1]. Both the Morph Monitor and DCPI were specially designed for *continuous* profiling, and thus their profiling overhead is remarkably low (less than 0.3% for Morph and 1-3% for DCPI). In contrast, pfmon is a general profiling tool and is not designed for continuous profiling. Yet, we show that our profiling overhead

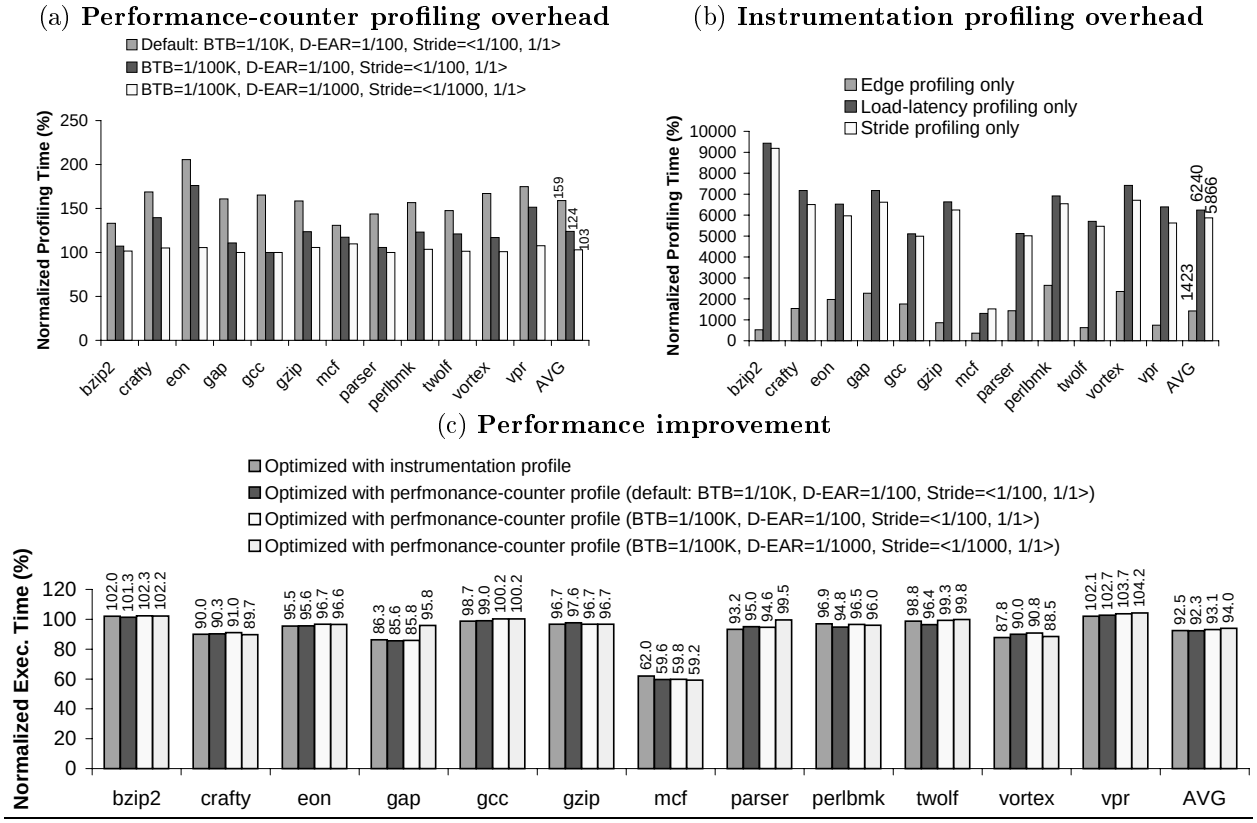

Figure 11: Overhead and performance improvement of various profiling schemes.

can be as low as 3%. In addition, we quantitatively compare instrumentation profiles and performance-counter profiles, and demonstrate that one can have both low profiling overhead and good speedups with performance counters.

The various Ispike optimizations (code layout and prefetching, data layout and prefetching, GOT-access optimization) have been separately investigated in the past. References to these studies can be found in Section 3. Ispike is a single tool that integrates all these optimizations, thereby maximizing overall performance.

7 Conclusions

In developing Ispike, the first post-link optimizer for the Intel®Itanium®, we have faced new opportunities and challenges. We have addressed Itanium-specific implementation issues including call shadows, branch inversion, getting free registers, and code scheduling. We have exploited the fine-grain performance monitoring on Itanium®to drive important optimizations including code layout, instruction prefetching, data layout, and data prefetching. We show that these optimizations contribute significant performance improvement to SPEC CINT2000: an average of 7.7% over the Intel®Electron compiler and 9.1% over the GNU Gcc compiler on the Itanium®2 processor. We also demonstrate that these speedups obtained with performance-counter profiles are essentially the same as those obtained with instrumentation profiles, and that the profiling overhead can be as low as 3% while the speedups are still substantial. Finally, we believe that the techniques we developed are not limited to static optimization, but are also applicable to dynamic optimization.

References

[1] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihi. Continuous profiling: Where have all the cycles gone. In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.

[2] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 281–290, Jan 2001.

[3] R. Cohn. *Pin User Manual.* Intel Corporation, Sept 2003.

[4] R. Cohn, D. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.

[5] J.-F. Collard and D. Lavery. Optimizations to prevent cache penalties for the intel itanium 2 processor. In *Proceedings of the 2003 International Conference on Code Generation and Optimization*, March 2003.

[6] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the intel ia-64 compiler. *Intel Technology Journal*, 4 th quarter:1–15, 1999.

[7] G. Haber, M. Klausner, V. Eisenberg, B. Mendelson, and M. Gurevich. Optimization opportunities created by global data reordering. In *Proceedings of the 2003 International Conference on Code Generation and Optimization*, March 2003.

[8] E. A. Henis, G. Haber, M. Klausner, and A. Warshavsky. Feedback based post-link optimization for large subsystems. In *Proceedings of the 2nd Workshop on Feedback Directed Optimization*, pages 13–20, November 1999.

[9] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2002.

[10] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Vol. 1: Application Architecture*, Oct. 2002.

[11] C-K Luk and P. G. Lowney. *Patent application for Methods and Apparatus for Stride Profiling: A Software Application.* Intel Corporation, 2003.

[12] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–193, December 1998.

[13] C-K Luk, R. Muth, H. Patil, P. G. Lowney, R. Cohn, and R. Weiss. Profile-guided post-link stride prefetching. In *Proceedings of 2002 International Conference on Supercomputing*, pages 167–178, June 2002.

[14] D. Mosberger and S. Eranian. *IA-64 Linux Kernel Design and Implementation*, chapter 9.3: Kernel Support for Performance Monitoring. Hewlett-Packard Company, 2002.

[15] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[16] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[17] R. Muth, S. Debray, S. Watterson, and K. DeBosschere. Alto: A link-time optimizer for the Compaq Alpha. *Software: Practice and Experience*, 21(1):67–101, 2001.

[18] K. Pettis and R. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[19] V. Ramasamy and R. Hundt. Dynamic binary instrumentation on IA-64. In *Proceedings of the First EPIC Workshop*, Dec. 2001.

[20] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, August 1997.

[21] G. Sander. *VCG Visualization of Compiler Graphs User Documentation V.1.30.* Universitat des Saarlandes, Feb 1995.

[22] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[23] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 02 Conference on Programming Language Design and Implementation*, pages 210–221, 2002.

[24] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.