RUNTIME PERFORMANCE OPTIMIZATION BLUEPRINT:

INTEL® ARCHITECTURE OPTIMIZATION WITH LAST BRANCH RECORD (LBR)

Authors: Suresh Srinivas, Uttam Pawar, Catalin Manciu, Gabriel Schulhof
October 6, 2020

Contact: suresh.srinivas@intel.com

CONTENTS

ABSTRACT

This document is a Runtime Optimization Blueprint illustrating how the performance and observability of runtimes can be improved by using Last Branch Record (LBR) on Intel® architecture. The intended audience for this document is runtime implementers, and customers/providers deploying runtimes at scale. In the Overview section, we introduce the problem that runtimes have with high Instruction Cache (I$) miss stalls (on average 12% of the CPU cycles are stalled across seven runtime workloads). In the Diagnosis section, we illustrate how to diagnose this problem using Performance Monitoring Unit (PMU) counters on Intel® architecture and sample tools. In the Solution section, we describe how to solve this problem. The Case Studies section details how this optimization improves performance and reduces I$ misses as well as Instruction TLB (ITLB) misses (up to 50%) in three applications in three environments. The last section summarizes the blueprint and provides a call to action for runtime developers/implementers.

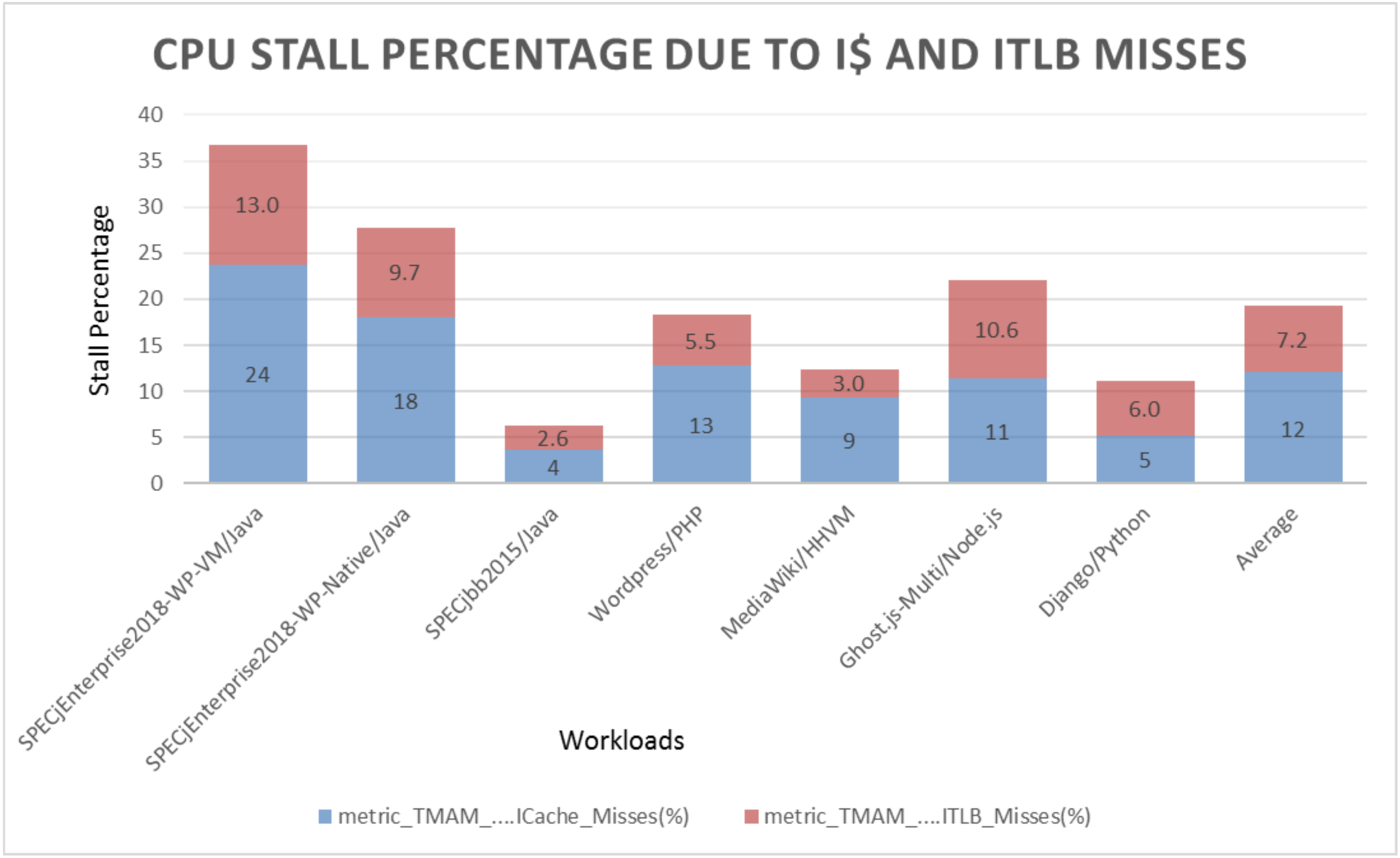
OVERVIEW

PROBLEM

Modern microprocessors have caches to avoid the cost of access from memory. They have independent caches for instruction and data and are usually organized into multiple levels (L1, L2, L3). For example, the current generation server platform Intel® Xeon® Platinum 8280 processor (formerly Cascade Lake) has a 32KB Level 1 Instruction (L1-I) cache, a 32KB Level 1 Data (L1-D) cache, a 1MB Unified Level 2 (L2) cache, and a 1.375 MB/core Level 3 cache or LLC (Last Level Cache). Despite these caches, runtime workloads have high miss stalls resulting from Instruction Cache (I$) and Instruction TLB (ITLB) misses.

Figure 1 shows the CPU stalls resulting from I$ and ITLB misses on an Intel® Xeon® Platinum 8180 processor across a range of runtime workloads. On average, 19% of the cycles are stalled on I$ and ITLB misses. Benchmarks such as SPECjbb2015* ((SPECjbb2015, n.d.)) have lower stalls (6.6%) compared to SPECjEnterprise* ((SPECjEnterprise, n.d.)), which has much higher stalls (37%). Among interpreted systems (PHP* and Python*), there is considerable difference in the stalls due to I$ misses. PHP running WordPress* has 18.5% stalls but Python running Django* has only 11% stalls. There is also considerable difference in running the same workload under a VM (37% stalls) vs running natively (27.7% stalls).

Figure 1: CPU Stalls due to I$ and ITLB in language runtimes on Intel Xeon Platinum 8180 Processor



Why do they have high I$ & ITLB misses and stalls? There are several reasons:

- Runtime Workloads have a large code footprint as well as poor locality.
- They are complex applications that lack a single hotspot, and have a large amount of code that is executed. For example, HHVM* has 133MB and the Node.js* v13 binary has about 40MB in the `.text` segment.
- The functions in the native code (which includes runtime helpers) are usually laid out without any consideration for their hotness. Furthermore, within a function, the cold and the hot basic blocks are intermixed.
- The JIT compiles methods on demand and the JITed code addresses on the heap are not organized to improve code locality.
- Any calls from the JITed methods to the native and vice versa are far calls and don't have good locality.
- Each instance of the runtime workload has its own copy of JITed code. This can result in duplication in the caches of the same JITed code (e.g. a core library method) across instances.

This results in high I-Cache (I$) and ITLB misses for these workloads. Improving code locality is a key performance enabler for these workloads. In this blueprint, we will demonstrate how the Last Branch Record can be used to improve code locality for the `.text` segment.

I$ AND STALLS

The Intel Optimization manual (Intel 64 and IA-32 Architectures Optimization Reference Manual) details the cache sizes and the updates to the instruction cache over the generations. For example, the Intel® microarchitecture code named Nehalem doubled the capacity of the L1 instruction cache (16KB to 32KB). Since the Intel® microarchitecture code name Haswell microarchitecture, the L1 instruction cache has stayed the same to meet latency requirements.

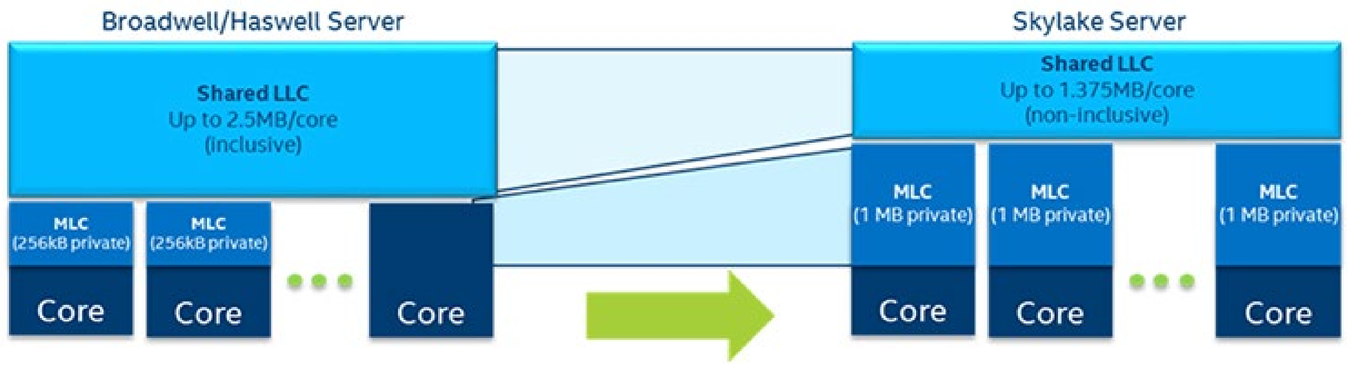


Figure 2: Comparing caches across Intel Xeon Processor server generations

In the previous server generation (Intel® microarchitecture code name Broadwell and Intel® microarchitecture code name Haswell), the mid-level (L2) cache (MLC) was 256 KB per core and the last level cache was a shared inclusive cache with 2.5 MB per core. In the Intel Xeon processor Scalable family (starting from the Intel® microarchitecture code name Skylake), the cache hierarchy has changed to provide a larger L2 of 1 MB per core and a smaller shared non-inclusive 1.375 MB LLC per core. Table 1 shows the cache sizes and associativity for the different levels.

L1I Cache	32 KiB/core 8-way set associative
L2 Cache aka MLC	1 MiB/core 16-way set associative
L3 Cache aka LLC	1.375 MiB/core 11-way set associative

Table 1: Cache sizes of the Intel Xeon Platinum 8180 (Intel® microarchitecture code name Skylake)

How are instructions brought into the caches? Demand code fetches to the L1-I are driven by the branch predictor. As the branch predictor is running ahead of the execution engine, demand misses (based on those predictions) into the L1-I are sent to the L2. The L1-I is filled when the L2 returns the requested cache lines. Demand requests sent to the L2 may trigger the L2 HW prefetchers (mechanisms that monitor patterns and issue prefetches) but these prefetch requests do not fill the L1.

DIAGNOSING THE PROBLEM

WHAT ARE I$ STALLS & I$ MISSES?

Intel has defined a Top-down Microarchitecture Analysis Method (TMAM) (Ahmad Yasin, 2014) that proposes a hierarchical execution cycles breakdown based on a set of new performance events. TMAM examines every instruction issue slot independently and is therefore able to provide an accurate slot-level breakdown.

One of the components of the Front-End Latency is the metric $ICache\_Miss_{stall}$. This metric represents the fraction of cycles the CPU was stalled due to instruction cache misses. On the Intel Xeon Scalable family of processors (formerly the Intel® microarchitecture code name Skylake), it can be computed through three PMU counters `ICACHE_64B.IFDATA_STALL, ICACHE_64B.IFDATA_STALL:c1:e1,` and `CPU_CLK_UNHALTED.THREAD` as shown in Equation 1.

Equation 1: Calculation of the ICache miss stall metric

$$
\begin{aligned}
a &= ICACHE\_16B.IFDATA\_STALL \\
b &= ICACHE\_16B.IFDATA\_STALL{:}c1{:}e1 \\
c &= CPU\_CLK\_UNHALTED.THREAD \\
ICache\_Miss_{stall} &= 100 * \frac{(a + 2 * b)}{c}
\end{aligned}
$$

Let us look at a concrete example. The WordPress*/PHP workload has an ICache_miss stall of 13% across the whole system. A sampling of the three counters along with *Equation 1* enables us to determine the percentage of I$ miss stalls as shown in *Table* 2. A **13% stall** because of I$ misses is significant for this workload.

Table 2: Calculating I$ miss stall for WordPress/PHP

ICACHE_16B.IFDATA_STALL	7325457
ICACHE_16B.IFDATA_STALL:c1:e1	876869
CPU_CLK_UNHALTED.THREAD	69103608
I$ Miss Stall %	13%

Measuring I$ miss stall is critical for determining if your workload on a runtime has an I$ performance issue.

In the Case Study section, we show that by improving the code layout of the `php-fpm` binary we can reduce both the I$ and the ITLB miss stalls. The Case Study section also describes improvements to other workloads such as Ghost.js and gcc compiler.

Another metric is the I$ misses in the different cache hierarchies. This metric is a normalization of the I$ misses against number of instructions. This metric is calculated using two PMU counters `<Cache-Hierarchy-Miss>`, and `INST_RETIRED.ANY` as described in Equation 2. There are distinct PMU counters for each of the cache levels, so Equation 2 shows the calculation for each PMU counter, respectively. In the case study section, we show how the optimizations reduce the I$ misses in each of the cache hierarchies.

Equation 2: Calculating ICache Misses Per Kilo Instruction (MPKI)

$$L1I\_Cache\_MPKI = 1000 * \left(\frac{L2\_RQSTS.ALL\_CODE\_RD}{INST\_RETIRED.ANY}\right)$$

$$L2\ Demand\ Code\_MPKI = 1000 * \left(\frac{L2\_RQSTS.CODE\_RD\_MISS}{INST\_RETIRED.ANY}\right)$$

$$LLC\ Code\_MPKI = 1000 * \left(\frac{UNC\_CHA\_TOR\_INSERTS.IA\_MISS{:}filter1 = 0x12CC0233}{INST\_RETIRED.ANY}\right)$$

TOOLS FOR MEASURING I$ MISS STALL

Intel has many tools to automate measuring I$ miss stalls, including VTune™ tools, EMON/EDP, and Linux* PMU tools. This Blueprint offers a convenient tool (`measure-perf-metric.sh`) based on `perf` to collect and derive various stall metrics on Intel Xeon Scalable processors. The tool is open-sourced and available for download at http://github.com/intel/iodlr. Figure 3 shows the command line to collect and derive I$ miss stalls for an application with process id=69772. The tool output shows the application has a **13.14%** ITLB miss stalls.

Figure 3: `measure-perf-metric.sh` tool usage for process id 69772 for 30 seconds

```
$ git clone http://github.com/intel/iodlr; export PATH=`pwd`/iodlr/tools/:$PATH
$ measure-perf-metric.sh -p 69772 -t 30 -m icache_miss_stalls
Initializing for metric: icache_miss_stall
Collect perf data for 30 seconds
perf stat -e cycles,instructions,icache_16b.ifdata_stall, icache_16b.ifdata_stall:c1:e1
--------------------------------------------------
Profile application with process id: 69772
--------------------------------------------------
Calculating metric for: icache_miss_stall

=================================================
Final icache_miss_stall metric
--------------------------------------------------
FORMULA: metric_ICache_Misses(%) = 100*((a+2*b)/c)
         where, a=icache_16b.ifdata_stall
                b=icache_16b.ifdata_stall:c1:e1
                c=cycles
=================================================
metric_ICache_Misses(%)=13.14
```

Use the command `measure-perf-metric.sh –h` to display help messages for using the tool. Refer to the `README.md` file, which describes how to add new metrics to the tool.

WHERE ARE THE I$ MISSES COMING FROM?

The next question is: Where are the I$ misses coming from? They could be coming from the `.text` segment of the runtime, JITed code, some other dynamic library of the runtime, or native libraries in the user code. Performance tools such as `perf` are able to determine the source of the I$ misses. The following figure shows an example of the tool in iodlr being used for this purpose on a Node.js workload. We can see from the highlighted row that most of the I$ misses are coming from the Node.js binary and after that from the JITed code (shown as perf-99903.map).

Figure 4: Using `measure-perf-metric.sh` with -r to determine where the ICache misses are coming from

```
$ measure-perf-metric.sh -p 99903 -r -t 30 -m icache_miss_stalls
Samples: 2K of event 'icache_16b.ifdata_stall', Event count (approx.): 5398008097
Overhead  Shared Object        Command
  61.58%  node.orig            node.orig
  21.90%  perf-99903.map       node.orig
  11.08%  [kernel.kallsyms]    node.orig
   3.15%  libc-2.27.so         node.orig
   1.56%  libpthread-2.27.so   node.orig
   0.63%  libstdc++.so.6.0.25  node.orig
   0.11%  [vdso]               node.orig
```

WHAT IS THE CALL STACK OF I$ MISSES?

The next question is what is the call stack of the I$ misses? That is, how do we determine the function call paths that resulted in the I$ misses? Intel provides a profiling mechanism called Last Branch Record (LBR) which can be used to determine the call stack and the control flow.

Last Branch Record (LBR) on Intel Architecture

Intel architecture CPUs have a feature called Last Branch Records (LBR) where the processor continuously logs branches. LBR captures the source and target of each retired branch (i.e., only the taken branches) in a rotating ring buffer. The number of entries in the buffer varies based on the Intel CPU. With LBR, we can sample the branches and build a control flow graph of the hot code paths. It has several different usages including determining call stacks, basic block frequencies, and being used for profile guided optimizations.

Using LBR to Determine Call Stacks

The CPU can keep track of the current call graph by logging every call and return into the LBR and treating them as a stack. **`perf`** has an option (**`--call-graph lbr`**) to use this mode like in the following Figure 5. LBR is usually limited in stack depth (either 8, 16, or 32 frames), so it may not be suitable for deep stacks. This feature is especially useful with Language Runtime JIT compilers that do not generate frame pointers.

Figure 5: Using –call-graph lbr with perf to determine the stack for l1i_miss event for Webtooling workload

```
$ perf record -o webtooling.node12.14.1.callgraph.l1i_miss.out --call-graph lbr  -e
frontend_retired.l1i_miss  -- <path>/node-v12.14.1/node --perf-basic-prof dist/cli.js
[ perf record: Woken up 112 times to write data ]
[ perf record: Captured and wrote 28.336 MB webtooling.node12.14.1.callgraph.l1i_miss.out
(26671 samples) ]

$ perf script -i webtooling.node12.14.1.callgraph.l1i_miss.out
node. 84653 3305591.869979: frontend_retired.l1i_miss:
            5555567dca80 Builtin:LoadIC_Megamorphic (/tmp/perf-84653.map)
            2a7254ffbc42 LazyCompile:*transpile /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:337156 (/tmp/perf-84653.map)
            2a7254ff3e36 LazyCompile:*scanPunctuator /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:375402 (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.map)
            555556840c43 Builtin:ArrayForEach (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.map)
            2a7254ff31a9 LazyCompile:*scanIdentifier /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:375762 (/tmp/perf-84653.map)
            2a7254bbd760 LazyCompile:*pp$8.readToken_eq_excl /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:66285 (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.map)
            2a7254bbd760 LazyCompile:*pp$8.readToken_eq_excl /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:66285 (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.map)
            2a7254bbd956 LazyCompile:*pp$8.readToken_eq_excl /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:66285 (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.map)
            2a7254ff31a9 LazyCompile:*scanIdentifier /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:375762 (/tmp/perf-84653.map)
            2a7254bbd760 LazyCompile:*pp$8.readToken_eq_excl /home/upawar/projects/web-tooling-
benchmark/dist/cli.js:66285 (/tmp/perf-84653.map)
            5555567ae602 Builtin:InterpreterEntryTrampoline (/tmp/perf-84653.
```

What can we do with this data? This gives us insight into the control flow of the workload. One technique we can use is to place the functions that call each other close together.

SOLUTION

On Linux and Windows* when the application is compiled and linked, no consideration is given to function placement. The linker places functions in the order the object files are specified on the command line. The order within each object file is driven by the placement of the function in the source code. This leads to poor locality of hot code as the functions are placed across the text region, negatively impacting the efficiency of the caches. Another related problem in language runtimes that have a JIT is that the JITed code is not laid out according to hotness. Furthermore, the addresses of the code in the JITed region and the .text region are in distinct spaces and locality is not maintained across the boundary.

DYNAMIC CODE LAYOUT FOR `.TEXT`

One way to reduce the I-Cache misses through software is to be intelligent about code placement, to use dynamic profiles to determine which functions should be placed close to each other, and to place the hot and cold parts of the function in distinct regions.

There are a few solutions on Linux for solving the I-Cache miss problem for the `.text` segment:

- **hfsort/Gold Linker**: hfsort (Ottoni & Bertrand, 2017) is a tool that generates an optimized list of hot functions from an input profile The input profile is gather using Linux perf. This optimized list is used along with the Gold Linker (ld.gold) to order the functions according to this list.
- **BOLT**: BOLT (Panchenko, Auler, Nell, Ottoni, & Guilherme, 2019), is a post-link optimizer built on top of the LLVM framework by Facebook. It uses LBR sample-based profiling and reduces I$ misses by improving the layout of both functions and basic blocks within a function and producing a new binary.
- **PGO or FDO**: Traditional feedback-directed optimization (FDO) is available in compilers from Intel, GCC, LLVM. GCC uses static instrumentation to collect edge and value profiles. GCC then uses execution profiles, consisting of basic block and edge frequency counts, to guide optimizations such as instruction scheduling, basic block

reordering, function splitting, and register allocation. The current method of feedback-directed optimization in GCC. involves the following steps

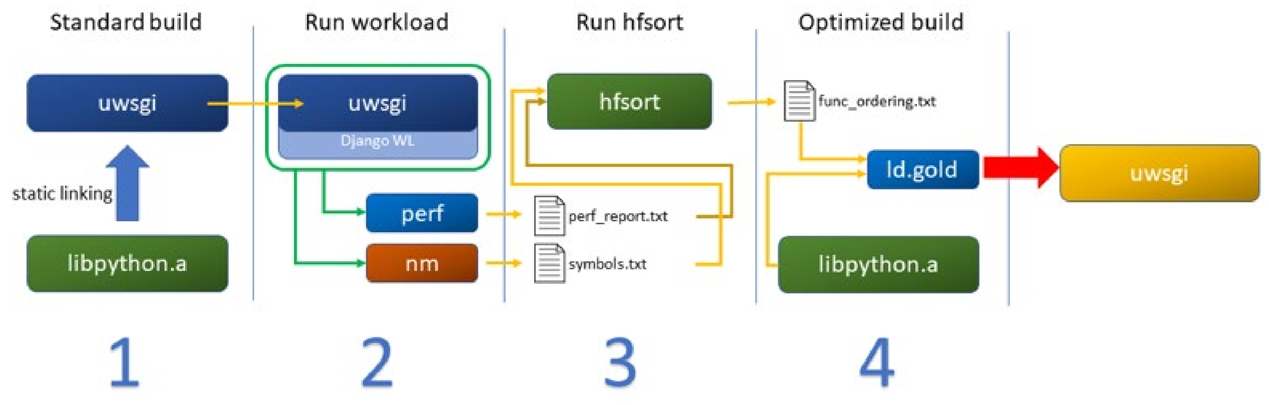
- Build an instrumented version of the program for edge and value profiling with `-fprofile-generate.`
- Run the instrumented version with representative training data to collect the execution profile.
- Build an optimized version of the program by using the collected execution profile (`-fprofile-use=` to guide the optimizations (FDO build).
- The GCC compiler supports function reordering via its link-time optimizer (LTO), which simply partitions functions into two sections (.text.hot and .text.unlikely) based on profiling data to segregate the hot and cold functions (via --freorder-functions).

Using LBR with hfsort/ld.gold

hfsort (Ottoni & Bertrand, 2017) is a tool that generates an optimized list of hot functions from an input profile. It was created by Facebook and is open-sourced under the hhvm project. Let us examine in detail how this works.

1. The application is built (usually the build flags need to be enhanced to maintain relocation or to put each function in a separate section).
2. The workload is run with `perf` to collect LBR samples (using `perf record -j any,u. --no-buffering -ag -e instructions:u`).
3. The profiles are fed to hfsort, which builds a weighted and directed call graph of the program. It clusters the functions in this graph and finally sorts these clusters into a function order list.
4. The application is relinked with the Gold Linker (ld.gold) using the function ordering list to generate a new optimized version of the application.

Figure 6: Profile-guided optimization flow with LBR and hfsort



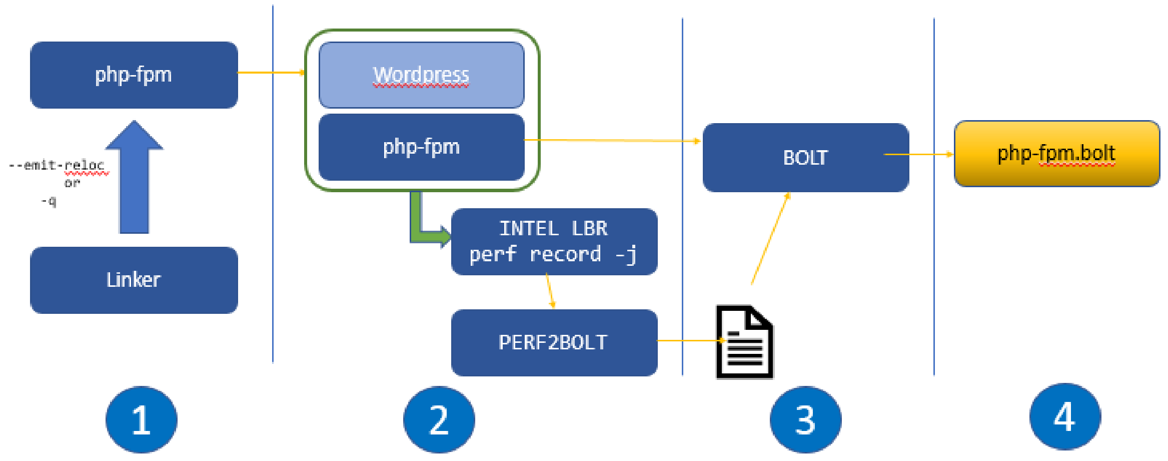
Using LBR with BOLT

BOLT is a post-link optimizer developed by Facebook as an LLVM tool. It achieves its improvement by optimizing the application's code layout based on the execution profile gathered through LBR profiling in perf. It operates on x86-64 binaries and produces a new x86-64 binary. An overview of the ideas implemented in BOLT along with a discussion of its potential and current results is available in the published CGO paper ((Maksim Panchenko, 2019) It is an open source project and can be downloaded from https://github.com/facebookincubator/BOLT

1. The application is built with relocations (through the linker flag -q or `--emit-reloc`).
2. The workload is run with perf to collect LBR samples (using `perf record -e cycles:u -j any,u`).

3. The perf data is read and aggregated into a profile that is used by BOLT.
4. The profiles and the original binary are fed to BOLT which produces a new optimized binary. The application is a new binary that BOLT has optimized by reordering blocks, functions, splitting functions etc.

Figure 7: Profile-guided optimization flow with LBR and BOLT



LIMITATIONS

There are several limitations to be aware of when using LBR and Dynamic Code Layout Optimization.

1. As with all Profile Guided Optimizations, the benefit that is obtained on the workload the profiles are obtained might not translate to other workloads (e.g. If php-fpm is trained with WordPress and gets a benefit for WordPress, it may not see a benefit for a different workload, such as MediaWiki*)
2. In several of the Dynamic Code Layout Optimizations, the source code needs to be recompiled or relinked. Moreover in the case of hfsort, the Gold Linker (`-Wl,-fuse-ld=gold`) needs to be used, which is not the default for either gcc or clang builds. BOLT does not work with gcc 8's default options. You need to use `-fno-reorder-blocks-and-partition` to disable a gcc 8 optimization that renders the binary unsupported by BOLT.
3. These optimizations are limited to the text segment in the binary. They usually don't work on shared libraries or JITed code. BOLT does support shared libraries. They also don't optimize across the text and the JITed address spaces.
4. In virtualized cloud environments LBR access will usually not be available. They are available on bare metal systems. LBR is a unique IA feature and is not available on other systems. Users can use PGO or BOLT with instrumentation when LBR feature is not supported. When LBR is not supported, you will likely get an error like this.

```
perf record -F 99 -a --call-graph lbr
Error:

PMU Hardware doesn't support sampling/overflow-interrupts.
```

CASE STUDY

This section details how this optimization helps performance and reduces I$ and ITLB misses in three workloads in three environments. The workloads are:

- WordPress, a content management system written in PHP paired with a MariaDB database.
- Building clang7.0 with gcc8.2 as shown in the CGO Paper artifact (Maksim Panchenko, 2019).
- Ghost (Ghost Team, n.d.), a fully open source, adaptable platform for building and running a modern online publication.

We conducted all the experiments on the Intel Xeon Gold 6248 processor (details of the platform and environment in the appendix). The specifications of the processor can be found at https://ark.intel.com/content/www/us/en/ark/products/199351/intel-xeon-gold-6248r-processor-35-75m-cache-3-00-ghz.html

WORDPRESS WORKLOAD WITH HFSORT

WordPress is a free and open-source content management system (CMS) written in PHP and paired with a MySQL* or MariaDB* database. This workload is set up as a WordPress deployment with database content and a concurrent client that drives the load. The performance metric is Throughput measured in transactions/second. This workload is used to demonstrate the performance of PHP.

We first ran the workload and collected performance stats with LBR (using `perf record -j any,u`) and then used the `hfsort` tool to create a hot functions list and then rebuilt PHP to place each function in a separate section (`-ffunction-sections -fdata-sections`) and relinked it with Gold Linker (`-Wl,-fuse-ld=gold`).

Table 3 shows a 2% performance improvement in the cycles. Both the Code misses from the Instruction Caches and the ITLB misses have reduced and the Front-end Stalls have reduced as well. There is a 1.56x improvement in the L1 ITLB misses due to frequent functions being placed in the same page.

Table 3: Comparing php with php with hfsort

Comparison	**PHP-ORIG**	**PHP-HFSORT**	**PHP-HFSORT/PHP-ORIG**
Average CPU utilization	92	92	1.00
Cycles per transaction	69722525	68553080	0.98
CPI	1.61	1.58	0.98
Path length (inst retired per txn)	43360610	43335505	1.00
Code Miss Breakdown			
L1-I code read misses per txn	2199447	2210473	1.01
L2 demand code misses per txn	548640	530067	0.97
LLC Code Read misses per txn	3907	3833	0.98
ITLB Breakdown			
ITLB misses per txn	27967	25258	0.90

L1 ITLB misses per txn hitting in L2 ITLB	96453	61858	0.64
L2 ITLB misses per txn	27967	25257	0.90
L2 ITLB misses per txn for 4K pages	27732	25046	0.90
L2 ITLB misses per txn for 2M pages	235	210	0.90
Front-end Stalls			
ITLB Miss Stall %	7.35	6.04	0.82
I$ Miss Stall %	13.64	13.08	0.96

GHOST.JS WORKLOAD WITH PROFILE GUIDED OPTIMIZATION (PGO)

Ghost is an open source blogging platform written in JavaScript* and running on Node.js. We created a workload that has a single instance Ghost.js running on Node.js as a server and uses Apache* Bench as a client to make requests. The performance is measured by a metric called Requests Per Second (RPS).

We configured Node.js with `--enable-pgo-generate` and ran the Ghost workload which generates the profile. Using the profile we collected we reconfigured with `--enable-pgo-use` and created a new Node.js binary.

The compiler performs several optimizations based on the profile. For example, one optimization is function inlining, where the frequently called function is inlined into the caller. This reduces the call overhead and the total number of instructions. Another optimization is to reorder code blocks based on branch targets. From these examples we can see that PGO reduces the number of instructions needed to do the given work. Table 4 shows the key metrics for the Profile Guided Optimization. From this we can see that PGO reduces the pathlength by 8% which is the primary reason for the 8% cycle improvement. With fewer instructions other key metrics such as the I$, ITLB misses reduce. However, we can also see that the front-end stalls don't reduce much with PGO.

Table 4: Key metrics for Ghost.js with and without PGO

Comparison	Baseline	PGO	PGO/Baseline
Cycles per transaction	26488097	24244355	0.92
CPI	0.86	0.86	0.99
Path length (inst retired per txn)	30669828	28341101	0.92
Code Miss Breakdown			
L1-I code read misses per txn	1084849	999092.91	0.92
L2 demand code misses per txn	385459.47	353600.13	0.92
LLC Code Read misses per txn	800.25	610.85	0.76
ITLB Breakdown			
ITLB misses per txn	15374.08	14185.41	0.92
L1 ITLB misses per txn hitting in L2 ITLB	45511.48	34159.59	0.75
L2 ITLB misses per txn	15373.97	14185.34	0.92
L2 ITLB misses per txn for 4K pages	15122.86	13952.24	0.92
L2 ITLB misses per txn for 2M pages	251.11	233.10	0.93
Front-end Stalls			
ITLB Miss Stall %	7.53	7.34	0.97
I$ Miss Stall %	11.94	11.47	0.96

GCC WITH BOLT

CLANG is a **compiler** front end for the C, C++, Objective-C and Objective-C++ programming languages, as well as the OpenMP, OpenCL™, RenderScript and CUDA frameworks. We are optimizing the gcc compiler toolchain to build the CLANG compiler with BOLT (binary optimizer) demonstrating the reduction in I$ and I$ misses. The performance is measured by an elapsed time as a metric.

We first built the baseline gcc 8.2.0 compiler. We then built a gcc compiler that could be processed by BOLT. BOLT requires relocation metadata to be added to the binaries (through the linker flag `-Wl,-q,-znow`). BOLT also requires disabling a gcc 8 optimization (through the flag `-fno-reorder-blocks-and-partition`). Next we collect the LBR data while compiling "gcc" itself (`perf record -e cycles:u -j any,u`). We convert the perf data into a profile suitable for BOLT. Using that profile and the input binary we create a gcc.bolt similar to Figure 7.

When we compare the gcc baseline build to the gcc BOLT build for compiling CLANG 7.0 as shown in Table 5 we see gcc.BOLT cycles have improved by 10% while the path length has stayed the same. The impact on the Front-end Stalls is significant. There is a 47% improvement in ITLB misses and a 51% improvement in I$ misses. On the code miss breakdown, there is improvement across all the cache levels. On the ITLB breakdown, there is a small increase in the L2 ITLB misses but a huge reduction (5x) in L1 ITLB misses.

Table 5: Improvement from using BOLT on gcc while building CLANG 7.0

Comparison	Baseline	BOLT	BOLT/Baseline
Cycles per transaction	31,695,471	28,580,233	0.90
CPI	1.45	1.31	0.91
Path length (inst retired per txn)	21,858,102	21,762,907	1.00
Code Miss Breakdown			
L1-I code read misses per txn	815,671	517,447	0.63
L2 demand code misses per txn	78,083	39,632	0.51
LLC Code Read misses per txn	1,636	1344	0.82
I-TLB Breakdown			
ITLB misses per txn	6,816	7,207	1.06
L1 ITLB misses per txn hitting in L2 ITLB	70,676	13,308	0.19
L2 ITLB misses per txn	6,799	7,206	1.06
L2 ITLB misses per txn for 4K pages	6,799	7,192	1.06
L2 ITLB misses per txn for 2M pages	16.38	14.05	0.86
Front-end Stalls			
ITLB Miss Stall %	6.51	3.48	0.53
I$ Miss Stall %	7.72	3.81	0.49

INTERACTION

LARGE PAGES & DYNAMIC CODE LAYOUT

In our previous Blueprint (Srinivas, 2020), we had demonstrated the benefits of using 2M pages. We wanted to see how the optimizations interact with each other.

We combined the PGO optimization with the large pages optimization as shown in Table 6. We see a 10% improvement with PGO and an additional 2% improvement from using large pages. It shows that the Large Page Optimization and the Dynamic Code Layout Optimization are complementary.

Table 6: Improvement from PGO and PGO+Large Pages for Ghost

	Baseline	PGO	PGO + Large Pages
Relative Performance	1.00	1.10	1.12

In Table 3 we saw that using HFSORT with PHP leads to a 2% improvement. We added Large Pages to it. We see the performance improves by an additional 1%. This is coming from the reduction in ITLB misses. However, while using large pages we see an additional 8% in LLC misses. This is due to a lack of sharing of the text that has been remapped to 2M across all the instances (in this experiment 96 php-fpm instances are running).

Table 7: PHP original vs PHP HFSort + large pages

Comparison	PHP-ORIG	PHP-HFSORT-LARGE	PHP-HFSORT-LARGE/PHP-ORIG
Average CPU utilization	92	92	1.00
Cycles per transaction	69722525	67674451	0.97
CPI	1.60	1.56	0.97
Path length (inst retired per txn)	43360610	43362506	1.00
Code Miss Breakdown			
L1-I code read misses per txn	2199447	2212530	1.01
L2 demand code misses per txn	548640	522884	0.95
LLC Code Read misses per txn	3907	4230	1.08
I-TLB Breakdown			
ITLB misses per txn	27968	18551	0.66
L1 ITLB misses per txn hitting in L2 ITLB	96453	30488	0.32
L2 ITLB misses per txn	27967	18558	0.66
L2 ITLB misses per txn for 4K pages	27732	18168	0.66
L2 ITLB misses per txn for 2M pages	235	390	1.66
Front-end Stalls			
ITLB Miss Stall %	7.35	4.39	0.60
I$ Miss Stall %	13.64	13.36	0.98

PGO AND BOLT TOGETHER

Table 5 showed 10% improvement in cycles when using BOLTed gcc on clang7. We see that the combination of using both BOLT and PGO together improves cycles by 12%. BOLT is able to provide an additional 2% improvement in cycles over the PGO binary. We see a 10% reduction in pathlength from the combination. The ITLB misses and ICache misses are reduced by 43% and 42% respectively. LLC Code Read misses do increase by 8% from PGO optimization.

Table 8: Improvement from PGO+BOLT to GCC to build CLANG

Comparison	Baseline	PGO+BOLT	PGO+BOLT/Baseline
Cycles per transaction	31,695,471	27,760,845	0.88
CPI	1.45	1.41	0.97
Path length (inst retired per txn)	21,858,102	19,667,442	0.90
Code Miss Breakdown			
L1-I code read misses per txn	815,671	542,913	0.67
L2 demand code misses per txn	78,083	58,598	0.75
LLC Code Read misses per txn	1,636	1,761	1.08
I-TLB Breakdown			
ITLB misses per txn	6,816	6,753	0.99
L1 ITLB misses per txn hitting in L2 ITLB	70,676	17,093	0.24
L2 ITLB misses per txn	6,799	6,752	0.99
L2 ITLB misses per txn for 4K pages	6,799	6,739	0.99
L2 ITLB misses per txn for 2M pages	16.38	13.42	0.82
Front-end Stalls			
ITLB misses %	6.51	3.70	0.57
ICache_misses %	7.72	4.44	0.58

SUMMARY

This Runtime Optimization Blueprint described the problem that runtimes have with high I$ miss stalls and discussed how to diagnose the problem as well as techniques (Dynamic Code Layout Optimization) to solve the problem. The three examples in the case study demonstrated that using LBR to direct code layout optimizations reduced the I$ and ITLB misses and improved performance up to 10%, improved ITLB misses up to 47%, and improved I$ misses up to 51%. We also demonstrated that the benefits that this blueprint provides are additive to the benefits provided in the earlier blueprint (Srinivas, 2020).

CALL TO ACTION

Our call to action is:

- Determine if your workload has I$ miss stalls in the `.text` segment.
- Use the suggested solutions to address the problem.
- Contribute any fixes or challenges you faced in integration.
- Partner with Intel to share your solution and effect on your workload with customers.

ACKNOWLEDGEMENTS

This document was substantially improved due to the efforts of the reviewers. We want to thank and acknowledge their effort. Several of our partners and customers provided valuable feedback. Thanks also to Doug Martin, and David Kinder for their tech writing, formatting, and staging.

REFERENCES

Ahmad Yasin, Intel Corporation. (2014). A Top-Down method for performance analysis and counters architecture. *In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS* , (pp. 35-44).

Ahmad, Y. (2017). *Performance Monitoring Event List*. Retrieved from 01.org: https://download.01.org/perfmon/SKX/

Aleksey Shipilev, Redhat. (2019, 03 03). *Transparent Huge Pages*. Retrieved from JVM Anatomy Quarks: https://shipilev.net/jvm/anatomy-quarks/2-transparent-huge-pages/

Ghost Team. (n.d.). *Ghost: The professional publishing platform*. Retrieved from Ghost Non Profit Web Site: https://ghost.org/

Google V8 JavaScript. (2019, August). *V8 JavaScript Engine*. Retrieved from V8 JavaScript Engine: https://v8.dev/

Google Web Tooling. (2019, August). *Web Tooling Benchmark*. Retrieved from Web Tooling Benchmark: https://github.com/v8/web-tooling-benchmark

Intel 64 and IA-32 Architectures Optimization Reference Manual. (n.d.). Retrieved from intel.com: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

Lavaee, R., Criswell, J., & Ding, C. (Oct 2018). Codestitcher: Inter-Procedural Basic Block Layout Optimization. *arXiv:1810.00905v1 [cs.PL].*

Maksim Panchenko, R. A. (2019, February 20). *BOLT: A Practical Binary Optimizer for Data Centers and Beyond*. Retrieved from Facebook Research: https://research.fb.com/publications/bolt-a-practical-binary-optimizer-for-data-centers-and-beyond/

NodeJS Foundation. (2019, August). *Node.js JavaScript Runtime*. Retrieved from Node.js JavaScript Runtime: https://nodejs.org/en

Ottoni, G., & Bertrand, M. (2017). Optimizing Function Placement for Large-Scale Data-Center Applications. *CGO 2017.*

Panchenko, M. (2017). *Building Binary Optimizer with LLVM*. Retrieved from LLVM.ORG: https://llvm.org/devmtg/2016-03/Presentations/BOLT_EuroLLVM_2016.pdf

Panchenko, M., Auler, R., Nell, B., Ottoni, & Guilherme. (n.d.). BOLT: A Practical Binary Optimizer for Datacenters and Beyond.

SPECjbb2015. (n.d.). *SPECjbb2015 Design Document*. Retrieved from SPEC - Standard Performance Evaluation Corporation: https://www.spec.org/jbb2015/docs/designdocument.pdf

SPECjEnterprise. (n.d.). *SPECjEnterpise 2018 Web Profile*. Retrieved from SPEC - Standard Performance Evaluation Corporation: https://www.spec.org/jEnterprise2018web/

Srinivas, S. (2020, March 20). *Runtime Performance Optimization*. Retrieved from Intel Software: https://software.intel.com/en-us/articles/runtime-performance-optimization-blueprint-intel-architecture-optimization-with-large-code

WikiMedia Foundation. (2019, August). *Mediawiki Software*. Retrieved from Mediawiki Software: http://mediawiki.org/wiki/MediaWiki

Xeon-D, I. (n.d.). *Xeon-D*. Retrieved from intel.com: https://www.intel.com/content/www/us/en/products/processors/xeon/d-processors.html

NOTICES AND DISCLAIMERS

TEST CONFIGURATION

System Details

System Info	
Manufacturer	Intel Corporation
Product Name	S2600WFT

System Info	
BIOS Version	SE5C620.86B.0X.02.0094.102720191711
OS	Ubuntu 19.10
Kernel	5.5.5
Microcode	0x500002c
MemTotal	385 GB
DDR Memory Config	12 x 32 GB 3200 MT/s DDR4

CPU Details

CPU Info	
Model Name	Intel® Xeon® Gold 6248R CPU @ 3.00GHz
Sockets	2
Core(s) per socket	24
Hyper-Threading Enabled	Yes
Total CPU(s)	96
NUMA Nodes	4
L1d Cache	1.5 MiB
L1i Cache	1.5 MiB
L2 Cache	48 MiB
L3 Cache	71.5 MiB
Prefetchers Enabled	DCU HW, DCU IP, L2 HW, L2 Adj.
Intel Turbo Boost Enabled	Yes

Power & Perf Policy	Performance
CPU Freq Driver	intel_pstate
CPU Freq Governor	Performance
Current CPU Freq MHz	3503
AVX2 Available	Yes
AVX512 Available	Yes
AVX512 Test	Passed

Vulnerability Status

Vulnerabilities	
CVE-2017-5753	OK (Mitigation: usercopy/swapgs barriers and __user pointer sanitization)
CVE-2017-5715	OK (Enhanced IBRS + IBPB are mitigating the vulnerability)
CVE-2017-5754	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-3640	OK (your CPU microcode mitigates the vulnerability)
CVE-2018-3639	OK (Mitigation: Speculative Store Bypass disabled via prctl and seccomp)

Vulnerabilities	
CVE-2018-3615	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-3620	OK (Not affected)
CVE-2018-3646	OK (your kernel reported your CPU model as not vulnerable)
CVE-2018-12126	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-12130	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-12127	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2019-11091	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2019-11135	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-12207	OK (this system is not running a hypervisor)