RUNTIME PERFORMANCE OPTIMIZATION BLUEPRINT:

INTEL® ARCHITECTURE OPTIMIZATION WITH LARGE CODE PAGES

Authors: Suresh Srinivas, Uttam Pawar, Dunni Aribuki, Catalin Manciu, Gabriel Schulhof, Aravinda Prasad

Contact: suresh.srinivas@intel.com

CONTENTS

ABSTRACT

This document is a Runtime Optimization Blueprint illustrating how the performance of runtimes can be improved by using large code pages. The intended audience is runtime implementers and customers and providers deploying runtimes at scale. In the Overview section, we introduce the problem that runtimes have with high *Instruction Translation Lookaside Buffer* (ITLB) miss stalls (on average 7% of the CPU cycles are stalled across seven commonly used runtimes). In the Diagnosis section, we illustrate how to diagnose this problem using Performance Monitoring Unit (PMU) on Intel® architecture processors counters and sample tools. In the Solution section, we provide an Intel reference implementation as well as other approaches to solve this problem. The Solution Integration section describes how to integrate the reference implementation in runtimes. The Case Studies section details how this optimization improves performance and reduces 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.

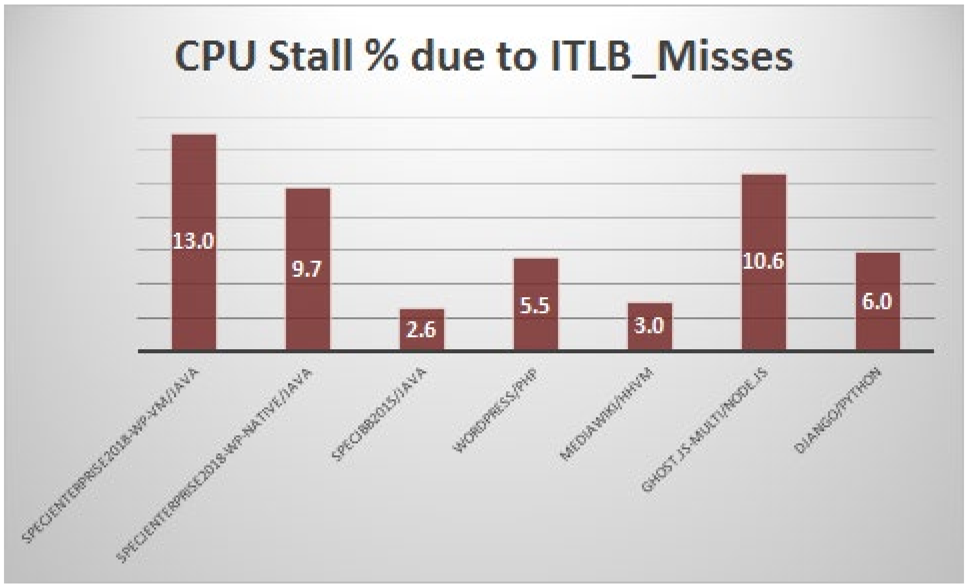
OVERVIEW

PROBLEM

Modern microprocessors support multiple page sizes for program code. For example, the current generation server platform Intel® Xeon® 8280 processor (formerly Cascade Lake) supports 4 KB, 2 MB, 4 MB pages for instructions and 4 KB, 2 MB, 4 MB, and 1 GB for data. Intel platforms have supported 4 KB, and 2 MB pages for instructions as far back as 2011 in Intel Xeon E5 processor (formerly Intel® microarchitecture code name Ivy Bridge). Nevertheless, most programs use only one page size, which is the default of 4 KB. On Linux*, all applications are loaded into 4 KB memory pages by default. When we examine performance bottlenecks for workloads on language runtimes, we find high stalls due to ITLB misses. This is largely due to the runtimes using only 4 KB pages for instructions.

Figure 1 shows the CPU stalls resulting from ITLB misses on anIntel Xeon 8180 processor across a range of runtime workloads. On average, 7% of the cycles are stalled on ITLB misses. Benchmarks such as SPECjbb2015* (SPECjbb2015, n.d.) have low ITLB stalls (2.6%) compared to SPECjEnterprise* (SPECjEnterprise, n.d.) which has high ITLB stalls (13%).

Figure 1: ITLB Miss Stalls in Language Runtimes on Intel Xeon 8180 Processor



ITLBS AND STALLS

Intel CPUs have a Translation Lookaside Buffer (TLB), which stores the most recently used page-directory and page-table entries. TLBs speed up memory accesses when paging is enabled, by reducing the number of memory accesses that are required to read the page tables stored in system memory.

The TLBs are divided into the following groups:

- Instruction TLBs for 4KB pages
- Data TLBs for 4KB pages
- Instruction TLBs for large pages (2MB, 4MB pages)
- Data TLBs for large pages (2MB, 4MB, or 1GB pages)

On the Intel Xeon Platinum 8180 (Intel® microarchitecture code name Skylake), each CPU TLB consists of dedicated L1 TLB for instruction cache (ITLB). (For details, refer to the (Intel 64 and IA-32 Architectures Optimization Reference Manual).) Additionally, there is a unified L2 Second Level TLB (STLB) which is shared across both Data and Instructions, as shown below.

- TLBs:
    - ITLB
        - 4 KB page translations:
            - 128 entries; 8-way set associative
            - Dynamic partitioning
        - 2 MB / 4 MB page translations:
            - 8 entries per thread; fully associative
            - Duplicated for each thread
    - STLB
        - 4 KB + 2 MB page translations:
            - 1536 entries; 12-way set associative. fixed partition

When the CPU does not find an entry in the ITLB, it has to do a page walk and populate the entry. A miss in the L1 (first level) ITLBs results in a very small penalty that can usually be hidden by the Out of Order (OOO) execution. A miss in the STLB results in the page walker being invoked -- this penalty can be noticeable in the execution. (Intel 64 and IA-32 Architectures Optimization Reference Manual) During this process, the CPU is stalled. The following table lists the TLB sizes across different Intel product generations.

Table 1: Core TLB Structure Size and Organization across Multiple Intel Product Generations

	Intel® microarchitecture code name Sandy Bridge/Intel® microarchitecture code name Ivy Bridge	**Haswell/Intel® microarchitecture code name Broadwell**	**Intel® microarchitecture code name Skylake/CascadeLake**
L1 Instruction TLB	4K – 128, 4-way 2M/4M – 8/thread	4K – 128,4 way 2M/4M – 8/thread	4K – 128, 8 way 2M/4M – 8/thread
L1 Data TLB	4K – 64, 4-way 2M/4M – 32 – 4-way 1G: 4, 4-way	4K – 64, 4-way 2M/4M – 32 – 4-way 1G: 4, 4-way	4K – 64, 4-way 2M/4M – 32 – 4-way 1G: 4, 4-way

	Intel® microarchitecture code name Sandy Bridge/Intel® microarchitecture code name Ivy Bridge	Haswell/Intel® microarchitecture code name Broadwell	Intel® microarchitecture code name Skylake/CascadeLake
L2 (Unified) STLB	4K – 512, 4-way	4K+2M shared: Haswell: 1024, 8-way Broadwell: 1536, 6-way 1G: 16, 4-way	4K+2M shared: 1536, 12-way 1G: 16, 4-way

From Table 1 we can see that 2M page entries are shared in the L2 Unified TLB from Haswell onwards.

LARGE PAGES

Both Windows* and Linux allow server applications to establish large-page memory regions. Using large 2MB pages, 20MB of memory can be mapped with just 10 pages; whereas with 4KB pages, 5120 pages are required. This means fewer TLB entries are needed, in turn reducing the number of TLB misses. Large pages can be used for code, data, or both. Large pages for data are good to try if your workload has large heap. The blueprint described here focuses on using large pages for code.

DIAGNOSING THE PROBLEM

ARE ITLB MISSES A PROBLEM?

Intel has defined a Top-down Micro-architecture Analysis Method (TMAM) (Ahmad Yasin, Intel Corporation, 2014), which 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 ITLB miss stall. This metric represents the fraction of cycles the CPU was stalled due to instruction TLB misses. On the Intel Xeon Scalable family of processors (formerly Intel® microarchitecture code name Skylake), ITLB miss stall can be computed through two PMU counters, `ICACHE_64B.IFTAG_STALL` and `CPU_CLK_UNHALTED.THREAD`, using Equation 1.

Equation 1: Calculation of the ITLB Stall Metric

$$ITLB\_Miss_{stall} = 100 * (\frac{ICACHE\_64B.IFTAG\_STALL}{CPU\_CLK\_UNHALTED.THREAD})$$

Let us look at a concrete example. The `Ghost.js` workload has an ITLB_miss stall of 10.6% when run in cluster mode across the whole system. A sampling of these two counters along with Equation 1 enables us to determine the percentage of ITLB_miss stall. A 10.6% stall due to ITLB misses is significant for this workload.

Table 2: Calculating ITLB Miss Stall for Ghost.js

CPU_CLK_UNHALTED.THREAD	69838983
ICACHE_64B.IFTAG_STALL	7412534
ITLB Miss Stall %	10.6

Measuring ITLB miss stall is critical to determine if your workload on a runtime has an ITLB performance issue.

In the Case Study section, we show that even while running in single instance mode, `Ghost.js` has a 6.47% stall due to ITLB misses. When large pages are implemented, the performance improves by 5% and the ITLB misses are reduced by 30% and the ITLB Miss Stall is reduced from 6.47% to 2.87% .

Another key metric is the ITLB Misses Per Kilo Instructions (MPKI). This metric is a normalization of the ITLB misses against number of instructions, and it allows comparison between different systems. This metric is calculated using two PMU counters `ITLB_MISSES.WALK_COMPLETED`, and `INST_RETIRED.ANY` as described in Equation 2. There are distinct PMU counters for large pages and 4KB pages, so Equation 2 shows the calculation for each PMU counter, respectively.

Equation 2: Calculating ITLB MPKI

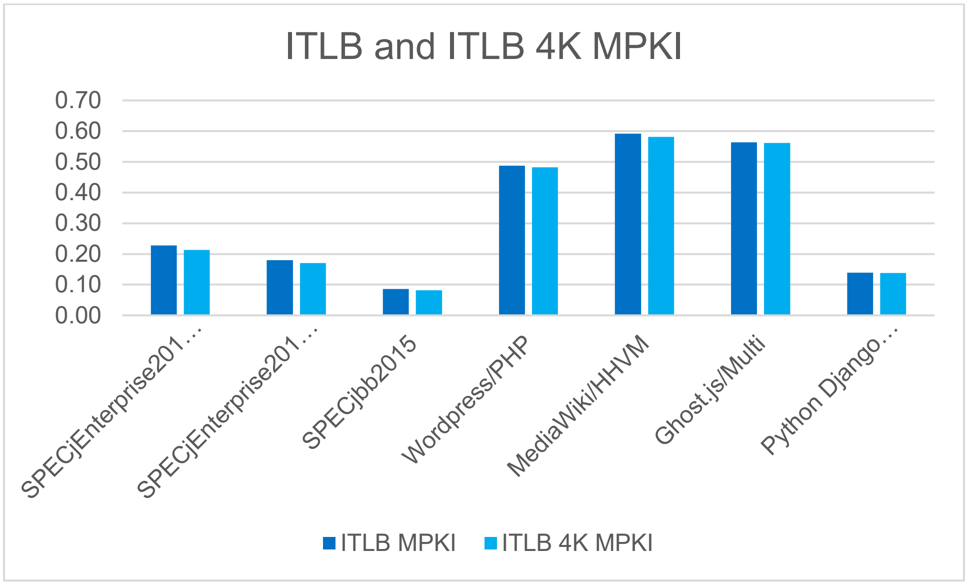
$$ITLB\_MPKI = 1000 * \left(\frac{ITLB\_MISSES.WALK\_COMPLETED}{INST\_RETIRED.ANY}\right)$$

$$ITLB\_4K\_MPKI = 1000 * \left(\frac{ITLB\_MISSES.WALK\_COMPLETED\_4K}{INST\_RETIRED.ANY}\right)$$

$$ITLB\_2M\_4M\_MPKI = 1000 * \left(\frac{ITLB\_MISSES.WALK\_COMPLETED\_2M\_4M}{INST\_RETIRED.ANY}\right)$$

Upon calculating the MPKI for the runtime workloads in Figure 2, we find that the ITLB MPKI and the ITLB 4K MPKI are very close to each other across the workloads. We can thus infer that most of the misses are from 4KB page walks. Another observation is that the benchmarks have lower ITLB MPKI than large real-world software, which means that optimization decisions made on benchmarks might not translate to open source software.

Figure 2: ITLB and ITLB 4K MPKI Across Runtime Workloads



Having ITLB MPKI also enables us to do comparisons across different systems and workloads. Table 3 compiles the ITLB MPKI across various workloads published in (Ottoni & Bertrand, 2017) and (Lavaee, Criswell, & Ding, Oct 2018). We can observe that there isn't a direct correlation of binary size to ITLB MPKI. Some smaller binaries such as MySQL* have one of the largest ITLB MPKI. When multiple threads are active, the ITLB MPKI almost doubles for both `Ghost.js` (single instance vs multi instance) and Clang (-j1 vs -j4). The ITLB MPKI is much lower on newer servers (Intel Xeon 8180 processors) as compared to older generation servers (Intel Xeon E5 processors).

Table 3: ITLB MPKI and Executable Sizes across Various Workloads

Workload	text (MB)	ITLB MPKI	System Details
AdIndexer	186	0.48	Dual 2.8 GHzIntel Xeon E5-2680 v2 (Intel® microarchitecture code name Ivy Bridge) server platform, with 10 cores and 25 MB LLC per processor
HHVM	133	1.28	
Multifeed	199	0.40	
TAO	70	3.08	
MySQL	15	9.35	Two dual core Intel® Core™ i5-4278U (Haswell) processors running at 2.60 GHz. 32 KB instruction cache and a 256 KB second level unified cache private to each core. Both caches are 8-way set associative. The last level cache is 3 MB, 12-way set associative, and is shared by all core
Clang –j4	50	2.23	
Clang –j1	50	1.01	
Firefox	81	1.54	
Apache* PHP (w opcache)	16	0.33	
Apache PHP (w/o opcache)	16	0.96	
Python	2	0.19	
SPECjEnterprise2018-WP-VM		0.23	Intel Xeon Platinum 8180 (Intel® microarchitecture code name Skylake) with 112 cores @ 2.5 GHz (except MediaWiki/HHVM which is on a SKX-D with 18 cores)
SPECjEnterprise2018-WP-Native		0.18	
SPECjbb2015		0.09	
Wordpress/PHP		0.49	
MediaWiki/HHVM		0.59	
Ghost.js/Multi		0.56	
Ghost.js/Single		0.23	
Python Django (Instagram)		0.14	

HOW DO YOU MEASURE THE ITLB MISS STALL?

Intel has a number of tools to automate measuring ITLB miss stalls, including VTune™, 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 ITLB miss stalls for an application with process id=69772. The tool output shows the application has a 3.09% ITLB miss stall.

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 itlb_stalls
Initializing for metric: ITLB_stalls
Collect perf data for 30 seconds
perf stat -e cycles,instructions,icache_64b.iftag_stall
--------------------------------------------------
Profile application with process id: 69772
--------------------------------------------------
Calculating metric for: ITLB_stalls

=================================================
Final ITLB_stalls metric
--------------------------------------------------
FORMULA: metric_ITLB_Misses(%) = 100*(a/b)
         where, a=icache_64b.iftag_stall
                b=cycles
=================================================
metric_ITLB_Misses(%)=3.09
```

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 ITLB MISSES COMING FROM?

The next question is where are the ITLB Misses coming from? They could be coming from the `.text` segment of the runtime, JITted code, some other dynamic library of the runtime, or native libraries in the user code. Performance tools such as `perf`, are required to determine where the ITLB misses are coming from.

In the case of `Ghost.js` that we examined earlier, most of the ITLB misses are coming from the `.text` segment of the Node.js [Node.js] binary. We find this to be the case for several other Node.js workloads. Using the current release of node.js (v12.8.0) and the `measure-perf-metric.sh` tool, we can determine it for a Node.js workload. Figure 4 shows that 65.23% of the stalls are in the node binary. The `'-r'` option to `measure-perf-metric.sh` uses `perf record` underneath to record the location in the source code that is causing the `ITLB_stalls.`

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

```
$ measure-perf-metric.sh -p 58448 -r -t 20 -m ITLB_stalls
Samples: 77K of event 'icache_64b.iftag_stall', Event count (approx.): 1558
Overhead  Shared Object        Command
  65.23%  node                 node
  20.69%  perf-56817.map       node
   4.53%  libc-2.27.so         node
```

On Linux and Windows systems, applications are loaded into memory into 4KB pages, which is the default on most systems. One way to reduce the ITLB misses is to use the larger page size, which has two benefits. The first benefit is fewer translations are required leading to fewer page walks. The second benefit is less space is used in the cache for storing translations, causing more space to be available for the application code. Some older systems, like Intel Xeon E5-2680 v2 processors (Intel® microarchitecture code name Ivy Bridge), have only eight huge-page ITLB entries that are organized in a single level, so mapping all the text to large pages could cause a regression. However onIntel Xeon Platinum 8180 processors (Intel® microarchitecture code name Skylake), the STLB is shared by both 4KB and 2MB pages and has 1536 entries.

SOLUTION

LINUX AND LARGE PAGES

On Linux OS, there are two ways of using large pages in an application:

- **Explicit Huge Pages (hugetlbfs)**. Part of the system memory is exposed as a file system that applications can `mmap` from. You can check the system through `cat /proc/meminfo` and see if lines like `HugePages_Total` are present.
- **Transparent Huge Pages (THP)**. Linux also offers Transparent Hugepage Support, which manages large pages automatically and is transparent for applications. The application can tell Linux to use large-pages-backed memory through `madvise`. You can check the system through `cat /sys/kernel/mm/transparent_hugepage/enabled`. If the values are `always` or `madvise`, then THP is available for your application. With `madvise`, THP is enabled only inside `MADV_HUGEPAGE` regions. Figure 5 shows how you can check your distribution for THP.

Figure 5: Commands for Checking Linux Distribution for THP

```
% cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
% cat /sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise [madvise] never
```

LARGE PAGES FOR .TEXT

There are a few solutions on Linux for solving the ITLB miss issue for `.text` segments:

- **Linking runtime with libhugetlbfs**: There are a number of support utilities and a library packaged collectively as libhugetlbfs. The library provides support for automatically backing text, data, heap, and shared memory segments with huge pages. This relies on explicit huge pages that the system administrator has to manage using tools like hugeadm.
- **Using the Intel Reference Implementation**: The reference implementations have both a C and a C++ module that automates the process using Transparent Huge Pages. A couple of API calls described below may be invoked at the beginning of the runtime to map a subset of the applications `.text` segment to 2MB pages.
- **Using an explicit option or flag in the runtime**: The Node.js runtime has an implementation that is exposed using --enable-largepages=on when you run Node.js. The PHP runtime has a flag that can be added to the .ini file. For details, see: https://www.php.net/manual/en/opcache.configuration.php

REFERENCE CODE

This Blueprint offers a reference implementation that enables an application to utilize large pages for its execution. The open source reference implementation is available for download at http://github.com/intel/iodlr. We provide both a C and C++ implementation.

The following is a high-level description of the reference implementation and its APIs.

1. Find the `.text` region in memory.
   a. Examine the `/proc/self/maps` to determine the currently mapped .text region and obtain the start and end addresses.
   b. Modify the start address to point to the very beginning of the `.text` segment.
   c. Align the start and end addresses to large page boundaries.
2. Move the .text region to large pages.
   a. Map a temporary area and copy the original code there.
   b. Use `mmap` using the start address with MAP_FIXED so we get exactly the same virtual address.
   c. Use `madvise` with MADV_HUGE_PAGE to use anonymous 2MB pages.
   d. If successful, copy the code from the temporary area and `unmap` it.

There are five API calls provided in the reference implementation as shown in Figure 6. Since the initial release we have added the ability to map DSO's

Figure 6: API Calls provided by the Intel Reference Implementation

```
/* Performs a platform-dependent check to determine whether it is possible to map
   to large pages and stores the result of the check in result. */
map_status IsLargePagesEnabled(bool* result);
```

```
/* Attempts to map an application's .text region to large pages.
If the region is not aligned to 2 MiB then the portion of the page that lies below
the first multiple of 2 MiB remains mapped to small pages. Likewise, if the region
```

```
does not end at an address that is a multiple of 2 MiB, the remainder of the region
will remain mapped to small pages. The portion in-between will be mapped to large
pages. */
map_status MapStaticCodeToLargePages();

/* Retrieves an address range from the process' maps file associated with a DSO
   whose name matches lib_regex and attempts to map it to large pages */
map_status MapDSOToLargePages(const char* lib_regex);

/* Attempts to map the given address range to large pages. */
map_status MapStaticCodeRangeToLargePages(void* from, void* to);

/* A string containing the textual error message. The string is owned by the
implementation and must not be freed. */
const char* MapStatusStr(map_status status, bool fulltext);
```

LARGE PAGES FOR THE HEAP

The Just-In-Time (JIT) compiler compiles methods on demand and the memory for the JITted code is allocated from the heap and subject to garbage collection.

The runtime can allocate heap on large pages using `mmap` with the flags argument set as `MAP_HUGETLB` (available since Linux 2.6.32) or `MAP_HUGE_2MB/ MAP_HUGE_1GB` (available since Linux 3.8). Alternatively, the heap region can be set to use transparent huge pages on Linux by using the `madvise` system call with `MADV_HUGE_PAGES`. When using `madvise`, the runtime must check that transparent_hugepage is set appropriately in the OS as either `madvise` or `always`, and not set to `never`.

The Java* VM has several options for mapping the Java heap with large pages (Aleksey Shipilev, Redhat, 2019). Since the JITted code is also on the heap, it allocates both the code and the data to large pages.

`-XX:+UseHugeTLBFS` mmaps Java* heap into hugetlbfs, which should be prepared separately.

`-XX:+UseTransparentHugePages` madvise-s that Java heap should use THP.

SOLUTION INTEGRATION

Integrating the solution into a new runtime requires the following changes:

1. Follow the style guide of the runtime and update the reference code.
2. Determine in the runtime where to make the API calls to remap the `.text` segment.
3. Change the build to link with the new files/library.
4. Provide a build time or runtime option to turn on this feature.

V8 INTEGRATION WITH THE REFERENCE IMPLEMENTATION

V8 (Google V8 JavaScript, 2019) is the Google* open source high-performance JavaScript* engine, written in C++. We integrated Intel's large pages reference implementation with V8 using the steps described above.

Here are the specific steps that we used to integrate the reference implementation within V8:

1. Check out, configure, and build v8 from https://v8.dev/docs/build-gn
2. Add call to `MapStaticCodeToLargePages()` at the beginning of `Shell::Main() in d8.cc`. Include `huge_page.h` in the source file.
3. Generate build files with the command:
   `gn gen out/foo –args=’is_debug=false target_cpu="x64" is_clang=false’`
4. Update the following build files:

   a. Update out/foo/obj/d8.ninja
      Add `–Ipath/to/huge_page.h` to include_dirs variable
      Add -Wl,-T path/to/ld.implicit.script to ldflags variable
   b. Update out/foo/toolchain.ninja
      Add path/to/libhuge_page.a –lstdc++ to link command, before –Wl,–endgroup

5. Compile V8 with the command:
   `ninja -C out/bar/ d8`

JAVA JVM INTEGRATION WITH THE REFERENCE IMPLEMENTATION

OpenJDK* is a free and open-source implementation of the Java Platform, Standard Edition written in a combination of C and C++. We determined that Java executable unlike v8 or nodejs is a thin ‘C’ wrapper that uses dlopen to load the libjvm.

We integrated Intel's C large pages reference implementation with OpenJDK. Here are the specific steps that we used to integrate the reference implementation

1. Check out, configure, and build OpenJDK using instructions at http://cr.openjdk.java.net/~ihse/demo-new-build-readme/common/doc/building.html
2. Modify the code in `src/java.base/unix/native/libjli/java_md_solinux.c` to load libjvm.so into 2M pages
   - Use the API to check if LargePages is supported
   - Use the API `MapDSOToLargePages(const char* lib_regex)` to load libjvm.so into 2M pages
3. Compile and Rebuild the Java wrapper.

LIMITATIONS

There are several limitations to be aware of when using large pages:

- Fragmentation is an issue that is introduced when using large pages. If there's insufficient contiguous memory to assemble the large page, the operating system tries to reorganize the memory to satisfy the large page request, which can result in longer latencies. This can be mitigated by allocating large pages explicitly ahead of time. The reference code does not have support for explicit huge pages.

- Another issue is the additional execution time it takes to perform the algorithm in the Intel reference code. For short running programs, it adds additional execution time and might result in a slowdown rather than a speedup.
- Tools like perf are no longer able to follow the symbols after the `.text` is remapped (Figure 7) and the perf output will not have the symbols. You will need to provide the static symbols to perf in `/tmp/perf-PID.map` at startup.

Figure 7: perf output will not have the proper symbols after Large Page Mapping

```
1.35%  node  perf-12142.map          [.] 0x0000562eb19e86aa
1.19%  node  perf-12142.map           [.] 0x0000562eb19e8803
0.71%  node perf-12142.map            [.] 0x0000562eb19e891f
```

CASE STUDY

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

- Ghost (Ghost Team, n.d.), a fully open source, adaptable platform for building and running a modern online publication.
- Web Tooling (Google Web Tooling, 2019), a suite designed to measure JavaScript-related workloads.
- MediaWiki* (WikiMedia Foundation, 2019), a free and open-source wiki engine written in PHP.

This case study uses data running on the Intel Xeon Platinum 8180 processor (Intel® microarchitecture code name Skylake) for Ghost.js and Web Tooling and uses the Intel Xeon D-2100 (Xeon-D, n.d.) processor for MediaWiki to showcase the benefits of large pages. The last case study demonstrates how to use Visualization tools to identify patterns in the data.

GHOST.JS WORKLOAD

Ghost is an open source blogging platform written in JavaScript and running on Node.js. We created a workload that has a single instance of 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).

Intel developed and contributed the 2MB for Code PR which is now merged into Node.js master. You can turn on large pages at runtime with the switch --enable-largepages=on in recent builds of Node.js. In older ones. you can enable large pages by building with `--use-largepages.` You can then compare the default build of Node.js with a build configured to use large pages.

Table 4 shows the key metrics with Node.js and Node.js with large pages. RPS improved by 5%. The stalls due to the ITLB misses were reduced by 56%. The ITLB MPKI improved by 57%. The gains are mostly coming from the ITLB walk reduction which was reduced by 55%. Note that the number of 2MB walks increased due to the use of 2MB pages.

Table 4: Key Metrics for Ghost.js With and Without Large Pages

Metric	Node.js with large pages	Node.js without large pages (default)	large pages/default
Throughput: Requests Per Second (RPS)	134.32	127.23	1.05
metric_ITLB_Misses(%)	2.87	6.47	0.44
metric cycles per txn	30048182	31426008	0.95
metric instructions per txn	46703106	47153431	0.99
ITLB_MISSES.WALK_COMPLETED	36799580	82504511	0.45
ITLB_MISSES.WALK_COMPLETED_2M_4M	938969	250959	3.74
ITLB_MISSES.WALK_COMPLETED_4K	35842004	82166894	0.44
metric ITLB MPKI	0.098	0.230	0.43
metric ITLB 4K MPKI	0.096	0.229	0.43
metric ITLB 2M_4M MPKI	0.002	0.0007	3.55

WEB TOOLING WORKLOAD

Node Version

Clear Linux* OS distributes Node 10.16.0 compiled with large pages support. We installed official Node 10.16.0 (which does not have large page support) on Ubuntu* 18.04 so we can compare the same version. Ubuntu only comes with Node 8.10.0 as part of the apt repository.

Web Tooling

Web Tooling is a suite designed to measure the JavaScript-related workloads commonly used by web developers, such as the core workloads in popular tools like Babel or TypeScript. It has a number of sub-components and reports a throughput score.

Comparing Clear Linux* OS and Ubuntu*

Table 5 shows the key metrics when running the Web Tooling workload. Although we observed small improvements in the throughput and cycles on Clear Linux when compared to Ubuntu, Clear Linux reduced the ITLB miss stall by 59%, the

ITLB MPKI by 51%, and the 4KB MPKI by 52%. This is due to Clear Linux distributing Node.js compiled with the `--use-largepages` option. The throughput isn't impacted significantly, since the ITLB stalls were not as significant to begin with.

Table 5: Key Metrics for Web Tooling across Clear Linux and Ubuntu 18.04

Metric	Clear Linux	Ubuntu 18.04	Clear Linux/Ubuntu
Throughput	10.91	10.80	1.01
metric ITLB_Misses (%)	0.91	2.21	0.41
metric cycles per txn	531,821,553.08	537,631,089.06	0.98
metric instructions per txn	836,955,459.53	879,834,649.97	0.95
ITLB_MISSES.WALK_COMPLETED	8,145,008	17,230,161	0.47
ITLB_MISSES.WALK_COMPLETED_4K	7,909,985	17,070,769	0.46
ITLB_MISSES.WALK_COMPLETED_2M_4M	241,215	142,356	1.69
metric ITLB MPKI	0.0298	0.0604	0.49
metric ITLB 4K MPKI	0.0289	0.0598	0.48
metric ITLB 2M_4M MPKI	0.0009	0.0005	1.76

MEDIAWIKI WORKLOAD

MediaWiki is a free and open-source wiki engine written in PHP. We used HHVM 3.25 to execute MediaWiki. HHVM maps the hot text pages to 2MB pages (Ottoni & Bertrand, 2017) and uses both the 4 KB and 2 MB pages. HHVM provides command line options `-vEval.MaxHotTextHugePages` and `-vEval.MapTCHuge` to enable large pages for the hot text pages and the Translation Cache pages (which holds the JIT generated code). In addition, it relies on code ordering to reduce the TLB misses. Table 6 shows the improvement in metrics with large pages. There is a reduction of 16% for the ITLB miss stalls, a 29% reduction for the overall walks completed, and a 66% reduction in the hits to the shared TLBs. Intel® microarchitecture code name Skylake introduced new precise frontend events (e.g., FRONTEND_RETIRED.ITLB_MISS Counts retired Instructions that experienced ITLB (Instruction TLB) true miss) and we can see that all those are lower with Large Pages with STLB_MISS reducing by 23%.

Table 6: Key Metrics for MediaWiki Workload on HHVM

Metric	Large Pages	No Large Pages	Large/No Large
metric_ITLB_Misses(%)	2.86	3.42	0.84
metric cycles per txn	44339066	44372158	0.99
metric instructions per txn	39129166	39168226	0.99
ITLB_MISSES.WALK_COMPLETED	7735.06	10850.57	0.71
ITLB_MISSES.WALK_COMPLETED_2M_4M	281.39	243.89	1.15
ITLB_MISSES.WALK_COMPLETED_4K	7453.67	10606.68	0.70
FRONTEND_RETIRED.ITLB_MISS	160403.87	162401.23	0.99
FRONTEND_RETIRED.L1I_MISS	160403.87	162401.23	0.99
FRONTEND_RETIRED.L2_MISS	19566.98	20075.91	0.97
FRONTEND_RETIRED.STLB_MISS	3820.17	4961.98	0.77
metric ITLB MPKI	0.2426	0.351	0.69
metric ITLB 4K MPKI	0.2310	0.341	0.67
metric ITLB 2M_4M MPKI	0.0116	0.009	1.18

VISUALIZATION OF BENEFITS

Precise Events

The PMU has Precise Event-Based Sampling (PEBS) which reports precise information such as the Instruction address of cache misses. On the Intel Xeon Platinum 8180 (Intel® microarchitecture code name Skylake), the PEBS events support additional Front-end events, which are hardest to locate in the source code. Two of them are for ITLB Misses as shown in the following table

Table 7: Precise Frontend events for ITLB Misses

Event	**Description.**
FRONTEND_RETIRED.ITLB_MISS (1st level)	Retired Instructions following a true ITLB miss
FRONTEND_RETIRED.STLB_MISS (2nd level)	Retired Instructions following an ITLB & STLB miss (2nd level)

Visualizing Precise ITLB Miss

We can use Linux `perf` to record the frontend_retired.ITLB_miss event and visualize the events with FlameScope an Open Source tool from Netflix. FlameScope is a visualization tool for exploring time ranges heatmaps & FlameGraphs. FlameScope starts by visualizing the input data as an interactive subsecond-offset heat map

Figure 8: Using `perf record` with -e `frontend_retired.ITLB_miss` to determine ITLB Misses & running `perf script` to obtain data for importing into FlameScope.

```
$ perf record -o /tmp/webtooling.node12.14.1.callgraph.ITLB_miss.orig.out.b -b  -e
frontend_retired.ITLB_miss -- /home/dslo/ssuresh1/node-v12.14.1/node.orig --perf-
basic-prof dist/cli.js
[ perf record: Woken up 8 times to write data ]
[ perf record: Captured and wrote 1.944 MB
/tmp/webtooling.node12.14.1.callgraph.ITLB_miss.orig.out.b (1539 samples) ]

$ perf script -i /tmp/webtooling.node12.14.1.callgraph.ITLB_miss.orig.out --header >
webtooling.node12.14.1.ITLB_miss.orig
```

The output of the `perf` script can be imported into FlameScope and we can visualize the ITLB Misses. We can see that some portions of the workload have much more ITLB misses than others. When we compare Figure 9 and Figure 10 we can see that the heatmap is much sparser for the ITLB misses when we are using Large Pages in Node.js.

Figure 9: Using FlameScope to visualize the ITLB misses heatmap from the WebTooling workload.

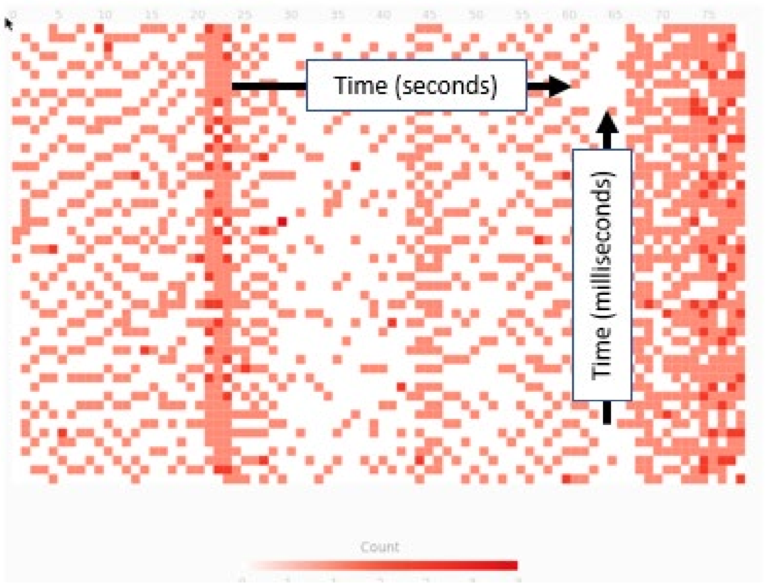
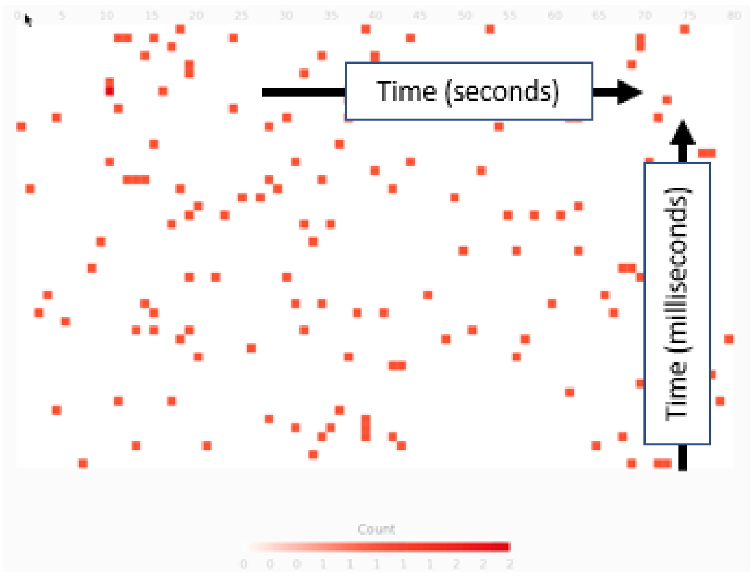


Figure 10: Using FlameScope to visualize the ITLB misses heatmap from the WebTooling workload when run with Large Pages.



SUMMARY

This Runtime Optimization Blueprint described the problem that runtimes have with high ITLB miss stalls, and discussed how to diagnose the problem as well as techniques and a reference implementation to solve the problem. A case study showed the benefits of integrating the solution into a new runtime. The three examples in the case study demonstrated that the use of 2M pages has the potential to improve ITLB Miss Stalls by 43%, ITLB Walks by 45%, and ITLB MPKI by 46%.

CALL TO ACTION

Our Call to Action is:

- Determine if your workload on a runtime has ITLB miss stalls in the `.text` segment or in dynamic library.
- Augment the runtime with the Intel Reference solution to solve the problem.
- Contribute any code changes or challenges you faced in integration.
- Contribute your integration to a runtime following their open source methods.
- Partner with Intel to share your solution and effect on your workload with customers.

ACKNOWLEDGEMENTS

The document 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, including Guilherme Ottoni (Facebook), Vijay Sundaresan (IBM), Kingsum Chow (Alibaba), and Toon Verwaest (Google). Several Intel engineers provided detailed feedback, including Sandhya Viswanathan, Milind Damle, Suleyman Sair, Vish Viswanathan, Andres Mejia, Hirally Santiago, Andrew Brown, Graham Whaley, and Sree Subramoney. Thanks also to Mary Camp and Deb Taylor 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].*

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/

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	DSLOHost011
Manufacturer	Intel Corporation
Product Name	S2600WFT
BIOS Version	SE5C620.86B.0X.01.0115.012820180604
OS	Ubuntu 18.04.3 LTS
Kernel	4.15.0-58-generic
Microcode	0x200005e

CPU Info

Model Name	Intel® Xeon® Platinum 8180 CPU @ 2.50GHz
Sockets	2
Hyper-Threading Enabled	yes
Total CPU(s)	112
NUMA Nodes	2
NUMA cpulist	0-27,56-83 :: 28-55,84-111
L1d Cache	32K
L1i Cache	32K
L2 Cache	1024K
L3 Cache	39424K
Prefetchers Enabled	DCU HW, DCU IP, L2 HW, L2 Adj.
Turbo Enabled	true
Power & Perf Policy	Balanced
CPU Freq Driver	intel_pstate
CPU Freq Governor	powersave
Current CPU Freq MHz	1000
Intel® Advanced Vector Extensions 2 (Intel® AVX2)Available	true
Intel® Advanced Vector Extensions 512 (Intel® AVX-512)Available	true
Intel® AVX512 Test	Passed
PPIN (CPU0)	c6aa1d2bcbba4d86

Kernel Vulnerability Status

Vulnerabilities	DSLOHost011
CVE-2017-5753	OK (Mitigation: usercopy/swapgs barriers and __user pointer sanitization)
CVE-2017-5715	OK (Full retpoline + IBPB are mitigating the vulnerability)
CVE-2017-5754	OK (Mitigation: PTI)
CVE-2018-3640	OK (your CPU microcode mitigates the vulnerability)
CVE-2018-3639	OK (Mitigation: Speculative Store Bypass disabled via prctl and seccomp)
CVE-2018-3615	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-3620	OK (Mitigation: PTE Inversion)
CVE-2018-3646	OK (this system is not running a hypervisor)
CVE-2018-12126	OK (Mitigation: Clear CPU buffers; SMT vulnerable)
CVE-2018-12130	OK (Mitigation: Clear CPU buffers; SMT vulnerable)
CVE-2018-12127	OK (Mitigation: Clear CPU buffers; SMT vulnerable)
CVE-2019-11091	OK (Mitigation: Clear CPU buffers; SMT vulnerable)