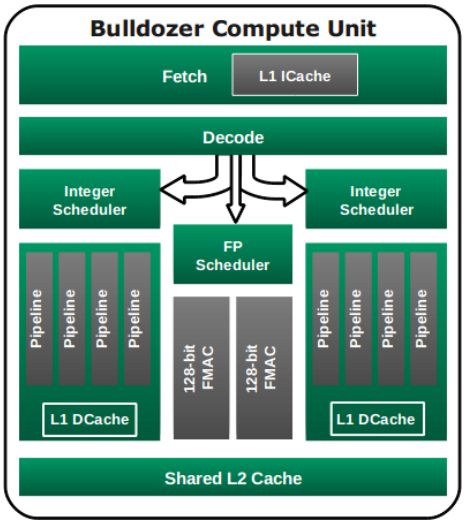
Shared Level-1 instruction-cache performance on AMD family 15h CPUs

Advanced Micro Devices, Inc.

December 2011

Introduction

AMD family 15h-class CPUs use a new micro-architecture that, under specific circumstances, leads to different performance characteristics compared to previous processor generations. This white paper describes the nature of instruction-cache cross-invalidations, symptoms and detection methods, as well as a number of provisions to ensure optimal performance in this micro-architecture.



AMD family 15h CPUs feature a compute unit architecture that unites two separate integer-execution units (called *cores*) with one wide floating-point unit (FPU) and a shared decoding front end. This shared front end includes the level-1 instruction cache (L1 ICache), in contrast to the L1 data cache, which is replicated per integer unit. By design, the L1 ICache does not allow physical aliases, that is, the same physical line cannot appear twice. If a second cache line with the same physical address is about to be stored in the cache array, the first one will be invalidated and thus removed from the cache. Under the following conditions, this leads to an increased number of cache misses, which decreases the overall performance.

All of the following conditions must be met for aliasing to occur:

1. A physical code page (not data) is mapped to different virtual addresses.
2. The virtual addresses must differ in bits 14-12.
3. Instructions from the same cache line of this code page must be requested by both cores of the same compute unit in a short time window (the two cores of one compute unit share the L1 ICache).

These requirements will be referenced as *Condition n* throughout this document. For more details, refer to the *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors* [1].

For clarification, any of the following will prevent increased L1I invalidations:

- The code runs on different compute units (compute units do not share one L1 ICache).
- Although pages contain the same code, they are not backed by the same physical page (duplicated buffers).
- Bits 14-12 are the same for both code pages executed on a compute unit. This is always true if code pages are aligned to 32KB or multiples thereof.

Impact on actual software performance

Operating systems

We have observed the invalidations on default installations of several **Linux** distributions with address-space layout randomization (ASLR) [2] turned on (default setting for most current distributions).

We could *not* reproduce it on systems running under **Windows**® because the Windows kernel virtually aligns code mappings to 64KB boundaries (*Condition 2* is not fulfilled).

Applications

If all three conditions apply, the actual performance impact depends on the runtime behavior of applications. If different code is executed concurrently on both cores, no impact will be observed. At the opposite extreme, the same code running in tight loops in both cores of a compute unit may render the L1 ICache ineffective because the invalidations will lead to continuous cache refills from the unified L2 cache. The L2 cache is not affected by the issue because it uses physical addresses for indexing, so aliases do not lead to conflicts.

The range of the performance impact can differ greatly based on the three conditions, from non-measurable under many scenarios to more-than-doubled run-time under rare conditions.

Affected setups

Because Windows is not affected, this paper describes scenarios, verification procedures, and solutions applicable only to Linux systems. Most of these may also apply to other Unixes.

The most prominent cause of instruction-cache invalidations is the use of dynamic libraries in combination with ASLR. Dynamic libraries are shared by multiple processes in a system. The Linux kernel holds only a single copy of each library in memory, mapping the pages multiple times into applications' address spaces.

ASLR causes the load addresses in each process to be different, thus fulfilling *Condition 1*. By default, the Linux kernel aligns those mappings on page boundaries (4KB). There is a 7 in 8 chance (or 87.5%) that two random mappings differ in bits 14-12 (*Condition 2)*.

Suppose two processes use the same library but have it mapped at a different address due to ASLR. If both processes run on the same compute unit and access the same function in the library, then aliasing conflicts are created, leading to an increase in cache evictions.

Single L1 ICache misses are not rare events and are expected to happen regularly under normal operation. Occasional misses due to aliasing conflicts usually do not stand out from the noise. If, however, long-running code executed in tight loops conflicts, the L1 misses dominate the instruction fetches, leading to a measurable impact.

This scenario does not apply to normal multi-threaded applications (pthread) because all threads share the same virtual address space and do not have differing mappings. Also not affected are processes that are created by the **`fork()`** system call alone, without using **`exec()`**. After the **`fork()`** call, address spaces in both processes remain unmodified, so *Condition 1* does not apply. A prominent example is the pre-fork mode of the Apache webserver [3].

When all three conditions apply, these applications may be affected:

- **Scripting languages like Python or Perl**: Some builds include the often-executed language interpreter loop in shared library code, so each program has this hot code loop loaded at a different virtual address. For scripts that strongly exercise the interpreter, the interpreter loop is a tight hot code location susceptible to increased evictions.
- **Applications that excessively use code in shared libraries**: Examples would be users of the math library, OpenSSL, or other compute-intensive code. Also, if applications put their own core computing code into a library (as multi-media codecs do), they can be affected. This applies only to multiple instances of the same binary; it does not apply if the application uses multi-threading or optimizes for performance scheduling on different compute units.
- Even code outside of libraries can be affected, **if the binary is built as a position-independent executable** (PIE) [4]. This, for instance, applies to network-facing server applications on Ubuntu [5].
- **In virtualized environments that use page merging** [6] [7]. Here, memory pages from different guests containing the same content will be merged into a single physical page by the hypervisor. Two single instances of a program running on distinct VMs using ASLR will be created with a different address-space layout. If both VMs are scheduled on one compute unit by the hypervisor and their code pages have been merged, they will now conflict with each other. VMware ESX since version 4.0 and Linux's KVM since kernel version 2.6.32 support page merging.

How to detect / Symptoms

This section describes typical symptoms of setups susceptible to excessive L1I invalidations and practical tests for verifying the cause. Applying one of the solutions in the following section can also be used for confirming L1 ICache aliasing issues. We use bash-shell syntax for the following examples.

L1 ICache aliasing occurs only with the clustered micro-architecture of processors of the AMD FX series or of the AMD Opteron™ 42xx/62xx classes. Other AMD CPUs of older generations are not affected.

The first recommended test makes direct use of a *dedicated performance counter*. Use this

test if you have access to performance counters on the target machine; alternatively, temporarily disabling ASLR can be used for verification.

- There is a dedicated performance counter measuring the L1 ICache invalidations. More details can be found in Chapter 3.15.4 *PMCx075 Cache Cross-invalidates* of the BKDG [1]. Using the unit mask 04h, you can restrict measurement to *L1-ICache-to-L1-ICache cross invalidations*. If you have the *perf* tool available in your Linux setup, you can run the following command in parallel with your workload in question:

  ```
  # perf stat -a -e instructions,r475 sleep 60
  ```

  This will use the system-wide collection mode of *perf* to count all L1 ICache cross-invalidations (along with the number of retired instructions for reference) for one minute.

  If you experience far more than one cache cross-invalidation per 1,000 retired instructions, you may encounter measurable drawbacks in performance. Please try one of the solutions discussed in the next section. If the invalidation count is lower, any unexpected performance problems are *not caused* by cache cross-invalidations.

  If you cannot use *perf*, you may use other tools using the performance counting unit of the CPU (e.g., *oprofile*). If you cannot specify arbitrary performance-measurement unit (PMU) event numbers, you may use the generic counter *L1 instruction cache load misses* as a first indication for excessive cache cross-invalidations.
- Turning off ASLR temporarily may help in most cases. ASLR assigns differing virtual addresses to a single physical page, leading to cache cross-invalidations.

  ```
  # sysctl -w kernel.randomize_va_space=0
  ```

  This command disables ASLR system-wide for newly created processes. If you see improved and constant performance afterwards, this is a strong (though not sufficient) indicator that cache cross-invalidations play a role.

  Do not forget to enable ASLR again after your test has finished. Also, disabling ASLR is not a permanent work-around because it decreases the whole system's security level. To enable ASLR again, execute:

  ```
  # sysctl -w kernel.randomize_va_space=2
  ```

Alternative tests:

- If you identify a generic slow-down in hot code paths in shared libraries using your favorite performance monitoring tool or profiler, this may be a hint that your workload is affected. You may want to explicitly look for the performance counter PMC0x75 or fall back to other tests described here.
- If the performance of your workload varies greatly over several runs, this is also an indication that cache cross-invalidations may play a role. Because there is a 1 in 8 (or 12.5%) chance that randomized mappings match in bits 14-12 and the actual scheduling may differ between runs, sometimes the performance is as expected. Use

one of the other checks to verify.

- Make sure your workload does not run on the same compute unit. You could either fixate the logical CPUs that your application is allowed to run on:

```
# taskset -c 1,3,5,7 $$
```

or you can take one core of each compute unit offline:

```
# for i in 1 3 5 7; do echo 0 > \
    sys/devices/system/cpu/cpu$i/online; done
```

Afterwards, start your workload (from the same shell if using taskset) and observe the performance difference. Take into account that the number of available processors is halved.

- Use the *prelink* tool [8] to fix libraries' load addresses. If, after a run with *prelink* enabled, the execution times are shorter and more stable, your setup may be affected. More details on using *prelink* are included in the Solutions section.

Solutions

If you verified with one of the methods described in the How to detect / Symptoms section that your setup is affected by excessive L1 ICache cross-invalidations, you can apply any of the following solutions to avoid the situation and restore expected performance.

1. Kernel patch

   All Linux kernel versions starting with 3.2-rc1 have a kernel patch that addresses the issue, basically fixing *Condition 2*. This patch will align all executable mappings on a 32K boundary to avoid bits 14-12 being different. This reliably prevents the issue and is the recommended solution (see the following discussion about interaction with *prelink*). Check back with your distribution vendor for kernel updates incorporating this patch. You can find the original commit here:

   http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=commit;h=dfb09f9b7ab03fd367740e541a5caf830ed56726

   Note that *prelink* (as discussed in the following sub-section) will render this kernel patch ineffective because the kernel will use the predetermined load addresses instead of generating (and aligning) them. This is not an issue in most of the cases (because *prelink* will also fix the issue); however, the virtualization scenario described above will not be fixed by the standard *prelink* if 32-bit guests are used. 64-bit libraries will be aligned to 2M boundaries by *prelink*, so even the virtualization scenario is safe here. A patch to address the 32-bit *prelink* issue is under development.

   If a kernel update is not feasible for you, try one of these other solutions.

2. Prelink

   The *prelink* tool will scan all the system's libraries and binaries and will generate a

conflict-free mapping of all symbols in the library files. The loader will then use these hints to load the libraries at a now-constant address, so all instances of all processes use the same virtual address for a given physical page. If you have the *prelink* package installed, you can either let *cron* do the prelink automatically or trigger it manually:

```
# prelink -a -R
```

This will handle all libraries and binaries in the default system paths (provided in /etc/prelink.conf) and will also use a (one-time) randomization to increase security. The randomization is used once to compute the permanent relocation addresses for the whole system and will lead to similar address-space layouts in all processes in a system.

Libraries in non-standard paths will not be covered by *prelink* out of the box. You have to explicitly call *prelink* with the respective path as a parameter. Take special care to also apply *prelink* to libraries that come bundled with benchmark suites.

3. Prelinking in 32-bit environments

   Prelinking in 32-bit environments with exec-shield support (e.g., RHEL 5 and 6) will lead to occasional collision caused by the vDSO page, mapped by the Linux kernel. Because the mapping address of the vDSO is not under the control of *prelink* but determined by the kernel at load time, it interferes the with address layout determined by *prelink* at its runtime. This leads to non-deterministic virtual mappings even on a single system with ASLR disabled. *Prelink* can be configured to create mappings that do not try to emulate the very compact exec-shield mapping from the kernel and will not exhibit the vDSO problem using the **`--no-exec-shield`** switch:

   ```
   # prelink -a –R --no-exec-shield
   ```

4. Statically linked libraries

   If it is feasible in your specific setup, you may consider statically linking libraries with hot code paths into your executable. This prevents different mappings for the same code in the same binary (unless built as PIE binary). This solution makes most sense for problems that are limited to few specific applications. Refer to your compiler documentation or development environment documentation for details on enabling static linking of certain libraries.

5. Temporary work-around: Turn ASLR off ***(NOT RECOMMENDED)***

   We observed that address-space layout randomization is the most prominent reason for excessive cache cross-invalidations. Therefore it is an easy solution to just turn off this setting, because that fixes the issue in most of the cases. However, this will compromise the whole system's security because the libraries' load addresses are now predictable and an attacker can use this knowledge to handcraft exploits. This increases the risk that your system may be compromised. For that reason, we do not recommend this solution. However, it may be feasible in controlled or offline environments, for test setups, or for benchmarking. To disable ASLR in a running

system, use this line:

```
# sysctl -w kernel.randomize_va_space=0
```

Please note that you have to restart running programs for the setting to take effect. To enable the feature again after finishing your experiments, use:

```
# sysctl -w kernel.randomize_va_space=2
```

There are distribution-specific ways to make this setting persistent.

Conclusion

This white paper discussed required software adaptations to make optimal use of new micro-architectural features in AMD's recent processor families. Minor adaptations in the mapping strategies for virtual memory of executable pages minimize required reloads of lines in the instruction cache and prevent excessive L1 ICache line reloads.

Bibliography

[1] AMD. (2011, Oct.) AMD Developer Guides & Manuals. [Online]. http://support.amd.com/us/Processor_TechDocs/42301.pdf

[2] (2011) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Address_space_layout_randomization

[3] Apache Webserver documentation. [Online]. http://httpd.apache.org/docs/2.0/mod/prefork.html

[4] (2011) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Position-independent_code

[5] (2011) Ubuntu Wiki. [Online]. https://wiki.ubuntu.com/Security/Features#pie

[6] (2011) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Kernel_SamePage_Merging

[7] VMWare Knowledge Base. [Online]. http://kb.vmware.com/kb/1021095

[8] (2011) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Prelink

CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.