Speculative Invalidation of Caches

We found a bug in the InvisiSpec's Gem5 implementation where speculative loads can still perform evictions, and thus affect the final state of the cache. An attacker could exploit this through a Prime+Probe-style attack to determine information about addresses which are only accessed speculatively.

Here is an assembly file which can reproduce this vulnerability:
https://github.com/XsPFTdot7Hy3/InvisiSpec-vulnerabilities/blob/main/spec-eviction-example.s

You can compile it with
```
gcc -nostdlib -no-pie  spec-eviction-example.s -o spec-eviction-example
```
then run it with
```
build/X86/gem5.opt configs/example/se.py --cpu-type=DerivO3CPU --l1d_size=128kB
--l1i_size=128kB --l2cache --l1d_assoc=8 --num-cpu=1 --ruby --needsTSO=0
--scheme=FuturisticSafeInvisibleSpec -c spec-eviction-example
```

If you dump out the L1 cache tags, the address 4096 will be present in the final cache tags if and only if `SECRET` is set to 1, even though the value of `SECRET` does not affect any non-speculative loads.

The core of the issue (in `src/mem/protocol/MESI_Two_Level-L1cache.sm` and `src/mem/protocol/MESI_Two_Level-L2cache.sm`) is that the code for L1 replacement is triggered when the cache-set is full, regardless of the request type (speculative or non-speculative). So SPEC_LD requests in the case of the L1 caches, and GETSPEC requests in the case of the L2 caches also trigger evictions of valid cache lines, leaking information via evictions. Instead, the eviction should be postponed until the Expose request is issued. We believe that the following patch should resolve this issue:

```
diff --git a/src/mem/protocol/MESI_Two_Level-L1cache.sm b/src/mem/protocol/MESI_Two_Level-L1cache.sm
index 7b13563d01..4d075a9a10 100644
--- a/src/mem/protocol/MESI_Two_Level-L1cache.sm
+++ b/src/mem/protocol/MESI_Two_Level-L1cache.sm
@@ -510,9 +510,10 @@ machine(MachineType:L1Cache, "MESI Directory L1 Cache CMP")
                     L1Icache_entry, TBEs[in_msg.LineAddress]);
             }

-            if (L1Dcache.cacheAvail(in_msg.LineAddress)) {
+            if (L1Dcache.cacheAvail(in_msg.LineAddress) || in_msg.Type == RubyRequestType:SPEC_LD)
{
             // L1 does't have the line, but we have space for it
             // in the L1 let's see if the L2 has it.
+            // Or for speculative accesses, do not replace existing cache entry until Expose is
issued.
             trigger(mandatory_request_type_to_event(in_msg.Type), in_msg.LineAddress,
                     L1Dcache_entry, TBEs[in_msg.LineAddress]);
             } else {
diff --git a/src/mem/protocol/MESI_Two_Level-L2cache.sm b/src/mem/protocol/MESI_Two_Level-L2cache.sm
index a3df8a59ce..ad0419e7da 100644
--- a/src/mem/protocol/MESI_Two_Level-L2cache.sm
```

```
+++ b/src/mem/protocol/MESI_Two_Level-L2cache.sm
@@ -382,8 +382,9 @@ machine(MachineType:L2Cache, "MESI Directory L2 Cache CMP")
                                              in_msg.Requestor, cache_entry),
               in_msg.addr, cache_entry, tbe);
        } else {
-       if (L2cache.cacheAvail(in_msg.addr)) {
+       if (L2cache.cacheAvail(in_msg.addr) || in_msg.Type == CoherenceRequestType:GETSPEC) {
               // L2 does't have the line, but we have space for it in the L2
+              // Or for speculative accesses, do not replace existing cache entry until Expose is
issued
               trigger(L1Cache_request_type_to_event(in_msg.Type, in_msg.addr,
                                              in_msg.Requestor, cache_entry),
                     in_msg.addr, cache_entry, tbe);
```

Speculative-Interference Attack Observable from the Same-Core.

There is an additional issue with InvisiSpec's vulnerability to Speculative interference attacks. Previously, such attacks required a reference load to be from another thread, and the attacker to be a multi-threaded attacker. We observe that this attack works even in a 1-core setting. Below, we provide a simple example with two asms, one for a secret=0 (test_case_input1.asm), and another for a secret=1 ((test_case_input2.asm). Only the inputs in registers are different in the two asms, the program itself is the same.

We have a reference load in the same program at the very end, which is a non-speculative cache hit. We observe that the reference load has a long latency if there is MSHR interference or short latency if there is no MSHR interference, based on if there is a speculative miss OR not. The crux of the issue is that:

- a speculative load based can hit in the spec-buffer OR miss in the spec-buffer AND miss in the caches.
- Based on whether there is a miss OR a hit, cause contention or not on the MSHRs.
- This can cause delays on an Expose of a Non-Speculative Load from the same program or no delays.
- If the Expose is delayed, it occupies the head of the queue in the cache-controller, and delays all subsequent loads (including our reference cache hit at the end of the program)

We provide a pair of example ASMs highlighting the vulnerability below:

- Test Case A path: "mshr_contention/test_case_input1.asm"
- Test Case B path: "mshr_contention/test_case_input2.asm"
- link1.ld linker script located at: "mshr_contention/link1.ld"

To compile the test case:

```
# Assemble the asm
as -g -mmnemonic=intel -msyntax=intel $RUN_ASM -o
./out/test_case_$RUN_NAME.o;
objcopy --remove-section .note.gnu.property ./out/test_case_$RUN_NAME.o;
```

```
# Link it with our provided link1.ld
ld ./out/test_case_$RUN_NAME.o -o ./out/test_case_$RUN_NAME.out -T
./link1.ld;
```

Compiled binary will be at “./out/test_case_$RUN_NAME.out”

Please run these while setting the MSHRs in Controller.py::number_of_TBEs to 2, from 256. Then run the compiled binary and observe the timing measured by RDTSCP instructions at the end of the ASM.