Cache Coherency in Itanium® Processor Software

By Thomas Wolfgang Burger

May 2005
**Itanium® Processor Cache Coherency**

The Itanium® architecture requires that software maintain coherence between stored instructions and cached instructions. Software is required to follow the architected sequence methodology if the instruction image is modified through the use of self-modifying code (SMC) and cross-modifying code (XMC).

Testing at Intel with the Intel Itanium processor, codenamed "Montecito," revealed that some software that performs self/cross-modifying types of code or updates code images does not meet the requirements in the *Intel Itanium Architecture Software Developer’s Manual*. While such cases are not common, you should ensure that when S/XMC is implemented your code performs the proper subsequent cache flushing, synchronization, and serialization.

**Itanium Processor Architectural Requirements**

Montecito cache implementation and size are different than on the Itanium 2 processor. Current releases of Itanium processors are tolerant of some deviations from the architectural requirements around S/XMC. The Itanium 2 processor is unable to expose these deviations because the Itanium 2 L1 D&I caches are 16KB and the L2 cache is unified. Montecito has separate L2 caches for data (256K) and instructions (1M) that are not forced to be coherent to each other by the hardware. A binary that deviates from the architectural requirements may run perfectly on the Itanium 2 processor but fail on Montecito uni-processor (UP), dual-processor (DP), and multi-processor (MP). Incorrectly designed S/XMC code will cause unexpected results. A worst case example is silent data corruption, which is nearly untraceable because there is no immediate signature.

```plaintext
st [L1] = Data  // store into local instruction stream
fc.i L1        // flush stale datum from instruction/data caches
;;;           // require instruction boundary between fc.i and
// sync.i       // synchronized
sync,.i       // ensure local and remote data/instr caches are
;;;           // require instruction boundary between sync.i and
// srlz.i       // srlz.i
srlz.i        // serialize to ensure sync has been observed by the local
;;;           // ensure subsequent instructions observe
// processor   // modified memory
L1: target    // instruction modified.
```

*Figure 1.0 – Example of Proper SMC Architected Sequence Methodology. See Additional References section for details.*
Who Should Take Notice?

Developers do not typically encounter instruction cache coherency problems due to non-conforming self-modifying code. Most likely to be affected are:

- Virtual machine developers
- Languages and tools developers (compilers, interpreters [static and just-in-time], debuggers, binary optimizers)
- OS developers

Is Your Software Deviating from the Architectural Requirements?

Review your code if any of the following situations apply:

- The code updates an image location before execution
- A thread updates an image location that another thread may execute
- Code images are copied from one location to another before execution
- The code image is decompressed and/or decrypted before execution

The following scenarios are typical examples of self/cross-modifying code or code modifying its own image:

- JIT type of compiler – dynamic code generators
- Loader’s image relocation and image patching by loader and debugger
- Programmed I/O for paging of text pages such as network drivers and slow I/O drivers
- Copying and massaging text pages. For example: OS copy-on-write or the decryption/decompression of text pages
- I/O buffering and caching used in an OS I/O manager, file system cache managers and device drivers
- Application loaders (user applications that load other code for execution)
- Dynamic languages such as Perl* and Python* that generate code without using a common language runtime or Java Virtual Machine*
- Tools that perform dynamic binary instrumentation/analysis

How to Examine Your Software

Follow these two steps, which are described in more detail in the next section:

1. Determine if your code contains S/XMC cases:
   a. Use the dynamic and static analysis methods described below.
   b. If there are no S/XMC cases, then there is no risk in the code.
2. If you find S/XMC cases, examine code sections that deal with cache coherency to make sure that you followed architected sequence methodology guidelines.
Step 1: Determine if Your Code Contains S/XMC Cases

The following is not an exhaustive study of analysis methods but provides general and common examples.

Examining Windows* Programs

Windows C code programs running on Itanium processors with S/XMC cases will call the `VirtualAlloc`, `VirtualAllocEx`, `VirtualProtect` and/or `VirtualProtectEx` functions.

Examining Linux* Programs

Linux C code programs running on Itanium processors with S/XMC cases will call the `mmap` and/or `mprotect` functions.

Analysis Tools

Common and ubiquitous compiler and development tools can be used to detect “write and execute” pages in executables or code images. The following examples are for Windows and Linux using C compilers.

Static Analysis Tools

Windows: Using `link` Switches or `dumpbin` on Windows

Use `LINK /dump /headers [executable file name]` to display section headers of executable binaries. Examine executable files that have sections with both execute and write flags.

The `DUMPBIN` utility may be a better choice than `LINK` for examining calls made in the executables. The `DUMPBIN` utility, included with the 32-bit version of Microsoft Visual C++, combines the abilities of the `LINK`, `LIB`, and `EXEHDR` utilities. `DUMPBIN` is able to provide information about the API calls, format and symbols in executable, library, and DLL files.

Linux: Using `objdump` on Linux

Use `OBJDUMP --h` to look at section headers.

Use `OBJDUMP -o` to display all the sections and each section’s protection in a binary image. Examine sections that are marked with RWX (read, write and execute) permissions.
Dynamic Analysis

Windows: Using WINDBG on Windows

Launch the target application from the Windows debugger and use the conditional breakpoint features to identify the enabling of execute privilege of any page during run-time using the following Windows APIs:

- `VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);`
- `VirtualAllocEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);`
- `VirtualProtect(LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect);`
- `VirtualProtectEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect);`

By setting a conditional breakpoint on these four entry points you can evaluate whether execute privilege is specified in the page protection parameter. If bits 3-7 of the page protection have a non-zero value, the execute privilege of the target page will be enabled. The following example sets a conditional breakpoint at `VirtualAlloc` in `kernel32.dll`:

```
bp kernel32!VirtualAlloc " j @r35 & 0xF0 '.echo VirtualAlloc; r r35; k; g' ; 'g' 
```

Linux: Using STRACE on Linux

Use `STRACE -e trace=mmap, mprotect application_name <app_arguments>` to find out if an application is changing the protections of certain pages dynamically to execute code on the fly.

A Sample Linux Tool to Capture the Dynamic Case

Use the following sample program to develop your own tools to detect potential problems caused by improper use of protected execute or protected write. In this sample `LD_PRELOAD` intercepts calls to the `mmap` and `mprotect` functions.
```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <sys/mman.h>

#ifndef USE_GNU
#define RTLD_NEXT ((void *)-1l)
#endif

static void * (*mmap_ptr)(void *start, size_t length, int prot, int flags, int fd, off_t offset) = 0;
static int (*mprotect_ptr)(void *addr, size_t len, int prot) = 0;
static FILE *log_fp = 0;

static void init_dll();

void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
{
    if(!mmap_ptr)
        init_dll();
    if((prot & (PROT_EXEC|PROT_WRITE)) != 0)
        fprintf(log_fp, "Found a call to mmap with PROT_EXEC and PROT_WRITE\n");
    return (*mmap_ptr)(start, length, prot, flags, fd, offset);
}

int mprotect(void *addr, size_t len, int prot)
{
    if(!mprotect_ptr)
        init_dll();
    if((prot & (PROT_EXEC|PROT_WRITE)) != 0)
        fprintf(log_fp, "Found a call to mprotect with PROT_EXEC and PROT_WRITE\n");
    return (*mprotect_ptr)(addr, len, prot);
}

static void init_dll()
{
    static int first = 1;
    char *log_name = NULL;
    if(!first)
        return;
    first = 0;
    if((log_name = getenv("SMC_LOG_FILE")) != 0)
    {
        // To make the dump to the file unbuffered
        if((log_fp = fopen(log_name, "w")) != NULL)
            setbuf(log_fp, NULL);
    }
    if(log_fp == NULL)
        log_fp = stderr;
    mmap_ptr = (void * (*)(void *start, size_t length, int prot, int flags, int fd, off_t offset))dlsym(RTLD_NEXT, "mmap");
    mprotect_ptr = (int (*)(void *addr, size_t len, int prot))dlsym(RTLD_NEXT, "mprotect");
}
```

Figure 2.0 – Source for the LD_PRELOAD solution.
To build a library from the source:

```
> gcc smc.c -shared -o libsmc.so -ldl
```

To run the application to be analyzed with this library use the `LD_PRELOAD` environment variable with a shell script (in this example called `check_smc.sh`):

```bash
#!/bin/sh
export LD_PRELOAD=<full path to libsmc.so>
exec $*
> check_smc.sh <app> [params]
```

The default output looks like this:

Found a call to mmap with PROT_EXEC and PROT_WRITE

This output is dumped to the standard error. To send the output to a file, set the name of the file in environment variable SMC_LOG_FILE.

This solution does not handle static executables or binaries with static calls to `mmap` or `mprotect`. In these cases intercepting the syscalls using `ptrace()` is effective.

**Step 2: Ensure Your Software is Compliant with Itanium Processor Architectural Requirements**

Ensure compliance by following the instructions in the Intel Itanium Architecture Software Developer's Manual, Volume 2 System Architecture, Revision 2.1, Section 2.5.1, Page 2:404.

If you write applications, use the operating system API, if available, rather than using the architected code sequences for cache coherence. For example, Windows provides `FlushInstructionCache` API for this purpose.
Additional References

- fc details can be found on page 3:55 of Intel® Itanium® Architecture Software Developer’s Manual Volume 3 Instruction Set Reference, Revision 2.1
- st details can be found on page 3:219 of Intel® Itanium® Architecture Software Developer’s Manual Volume 3 Instruction Set Reference, Revision 2.1
- Sync details can be found on page 3:226 of Intel® Itanium® Architecture Software Developer’s Manual Volume 3 Instruction Set Reference, Revision 2.1
- Sriz details can be found on Page 3:217 of Intel® Itanium® Architecture Software Developer’s Manual Volume 3 Instruction Set Reference, Revision 2.1
- SDM Vol2 page 2:404 section 2.5 (Updating Code Images)

About the Author

Thomas Wolfgang Burger is the owner of Thomas Wolfgang Burger Consulting. He has been a consultant, instructor, analyst, and applications developer since 1978. He can be reached at twburger@shaw.ca