

COL718 Assignment 2 Report

Sayam Sethi (2019CS10399)

Rishi Sarraf (2019CS10393)

October 2022

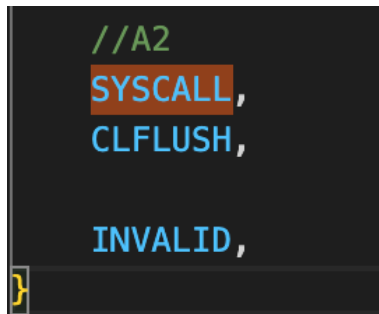
Contents

1	Implementation	1
1.1	System calls (syscall)	3
1.2	Clflush	4
2	Statistics	5
2.1	Simulation observations:	5

1 Implementation

We implemented the required functionality for system calls and clflush in the existing Tejas codebase. New classes have been defined wherever necessary. Most of our implementation has used the pre-existing pattern of code by making suitable changes. A few functions have been defined to handle execution of the new instruction types. The structure of implementation is similar for both instruction types:

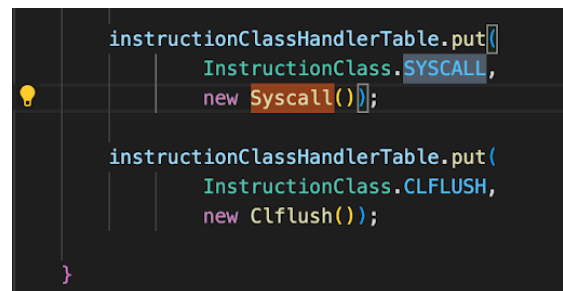
1. New instruction types SYSCALL and CLFLUSH are defined in `InstructionClass.java`
2. New handler classes `Syscall` and `Clflush` are defined in files `Syscall.java` and `Clflush.java` for syscall and clflush (respectively).
3. These instruction classes are added to `InstructionClassTable` and `InstructionClassHandlerTable`, these would be needed in the parsing of trace file.



```
//A2
SYSCALL,
CLFLUSH,

INVALID,
```

Figure 1: `InstructionClass.java`



```
instructionClassHandlerTable.put(
    InstructionClass.SYSCALL,
    new Syscall());

instructionClassHandlerTable.put(
    InstructionClass.CLFLUSH,
    new Clflush());
}
```

Figure 2: `InstructionClassTable.java`

```

syscall,
clflush,
//TODO Software interrupt can also be modelled as a far jump
interrupt,
no_of_types,
//sync type for causality check
sync
}

```

Figure 4: OperationType.java

```

}
case syscall : {
    return FunctionalUnitType.memory;
}
case clflush : {
    return FunctionalUnitType.memory;
}
}

```

Figure 5: OpTypeToFUTypeMapping.java

```

instructionClassTable.put(key: "syscall", InstructionClass.SYSCALL);
instructionClassTable.put(key: "clflush", InstructionClass.CLFLUSH);
}

```

Figure 3: InstructionClassTable.java

4. New operation types `syscall` and `clflush` are added in `OperationType.java`
5. These new operation types are mapped to `FunctionalUnitType.memory` in `OpTypeToFUTypeMapping.java` as they are memory based operations.
6. **Handling in IW** These instructions are issued in `issueMemoryInstruction()` in the file `IWEntry.java`
7. **Commit** Execution occurs similar to `store` instructions. They are executed at commit time, i.e., when they are at the top of ROB. Hence, in `performCommits()` in `ReorderBuffer.java`, they are handled in `if` blocks with condition evaluating `firstOpType == OperationType.syscall`

```

associatedROBEntry.setIssued(issued: true);
if(opType == OperationType.store || opType == OperationType.syscall || opType == OperationType.clflush)
{
    // these are issued at commit stage

    associatedROBEntry.setExecuted(executed: true);
    associatedROBEntry.setWriteBackDone1(isWriteBackDone1: true);
    associatedROBEntry.setWriteBackDone2(isWriteBackDone2: true);
}

```

Figure 6: IWWrite.java

8. New request types `Tlb_Flush` and `Cache_Line_Flush` are added in `RequestType.java` for creating events corresponding to execution of these instructions.

```

Cache_Hit, Cache_Miss, Cache_Line_Flush,
Tlb_Miss_Response, Tlb_Flush, Send_Migrate_Block,
}

```

Figure 7: RequestType.java

9. Visa Handlers `Syscall` and `Clflush` are defined in files `Syscall.java` and `Clflush.java` (in directory `visaHandler`) for `syscall` and `clflush` (respectively).

```

case syscall:
    return syscall;

case clflush:
    return clflush;

```

Figure 8: VisaHandler.java

```

syscall = new Syscall();
clflush = new Clflush();
}

```

Figure 9: VisaHandler.java

10. These handlers are instantiated in VisaHandler.java.
11. These are used to parse the address arguments and help with creation of the tail of instruction packets.

The above flow explains where in code their execution function can be called. In Tejas, execute steps are event-driven. We thus need to create ‘events’ with appropriate arguments in order to call the desired execute functions.

1.1 System calls (syscall)

An event based mechanism is used to handle the execution of `syscall`. An request type called `Tlb.Flush` has been defined and an event with this request type is added to the event queue in `performCommits()` function defined in `ReorderBuffer.java`. The event handler of TLB is responsible for calling the `flush()` function defined in `TLB.java`. The flush function just instantiates a new hashtable thus ‘flushing’ the TLB effectively.

```

if (firstOpType == OperationType.syscall)
{
    execEngine.getCoreMemorySystem().getTLB().sendEvent(new TLBFlushEvent(
        core.getEventQueue(), eventTime: 0, this,
        execEngine.getCoreMemorySystem().getTLB(), RequestType.Tlb_Flush,
        first));
    execEngine.getCoreMemorySystem().getTLB().sendEvent(new TLBFlushEvent(
        core.getEventQueue(), eventTime: 0, this,
        execEngine.getCoreMemorySystem().getTLB(), RequestType.Tlb_Flush,
        first));
}

```

Figure 10: ReorderBuffer.java

```

public void handleEvent(EventQueue eventQ, Event event)
{
    if(event.getRequestType()==RequestType.Tlb_Miss_Response) {
        long pageId = ((AddressCarryingEvent)event).getAddress();
        addTLBEntry(pageId);
    } else if (event.getRequestType() == RequestType.Tlb_Flush) {
        flush();
    } else {
        misc.Error.showErrorAndExit("Invalid event sent to TLB : " + event);
    }
}

```

Figure 11: TLB.java

```

public void flush()
{
    TLBuffer = new Hashtable<Long, TLBEntry>(TLBSize);
}

```

Figure 12: TLB.java

1.2 Clflush

A similar mechanism is used to handle the execution of `clflush`. An request type called `Cache_Line_Flush` has been defined and an event with this request type is added to the event queue in `performCommits()` function defined in `ReorderBuffer.java`. The event handler of Cache is responsible for calling the `handleCacheLineFlush()` function defined in `Cache.java`. This function recursively sends events invalidating all cache levels and eventually writes to memory.

```

if (firstOpType == OperationType.clflush)
{
    Cache l1Cache = execEngine.getCoreMemorySystem().getL1Cache();
    l1Cache.sendEvent(new AddressCarryingEvent(l1Cache.getEventQueue(), eventTime: 0, this, l1Cache,
        RequestType.Cache_Line_Flush, first.getInstruction().getSourceOperandMemValue()));

    // same for icache
    Cache iCache = execEngine.getCoreMemorySystem().getICache();
    iCache.sendEvent(new AddressCarryingEvent(iCache.getEventQueue(), eventTime: 0, this, iCache,
        RequestType.Cache_Line_Flush, first.getInstruction().getSourceOperandMemValue()));
}

```

Figure 13: ReorderBuffer.java

```

case Cache_Line_Flush: {
    handleCacheLineFlush(addr);
    break;
}

```

Figure 14: Cache.java

```

protected void handleCacheLineFlush(long addr) {
    CacheLine cacheLine = this.access(addr);
    if (cacheLine != null) {
        if (cacheLine.isModified()) {
            if (writePolicy == WritePolicy.WRITE_BACK)
                sendRequestToNextLevel(addr, RequestType.Cache_Write);
            sendRequestToNextLevel(addr, RequestType.Cache_Line_Flush);
        }
        if (mycoherence != null) {
            AddressCarryingEvent evictEvent = mycoherence.evictedFromCoherentCache(addr, this);
            mshr.addToMSHR(evictEvent);
        } else
            sendEvent(new AddressCarryingEvent(getEventQueue(), eventTime: 0, this, this, RequestType.EvictCacheLine, addr));
    }
}

```

Figure 15: Cache.java

2 Statistics

The following table summarises the simulation statistics for different scenerios, i.e., with and without flushing TLB and Cachelines.

Quantity	noflush	tlbflush	clflush	tlbclflush
IPC(micro-ops)	0.2574	0.2522	0.2574	0.2522
iTLB hits	78860	78514	78860	78514
iTLB misses	474	815	474	815
dTLB hits	29456	29225	29455	29224
dTLB misses	132	361	132	361
L1 hits	25447	25466	25444	25463
L1 misses	1439	1433	1440	1434
L2 hits	6799	6808	6799	6808
L2 misses	1541	1540	1541	1540
L3 hits	432	430	432	430
L3 misses	1314	1314	1314	1314
I1 hits	74342	74340	74342	74340
I1 misses	4986	4979	4986	4979
Directory Access EvictionFromCoherentCache	41	41	42	42
Coherence energy	42277.33	43149.6216	42278.01	43150.3008

Table 1: Simulation Statistics Comparison.

2.1 Simulation observations:

1. TLB flush decreases the IPC, whereas CL Flush doesn't. This is because TLB misses increase significantly (almost 2 fold) and in our simulation. CL Flush doesn't significantly change cache hit/miss rates.
2. TLB flush increases the iTLB and dTLB misses (thus decreasing the hits). This is an expected behaviour, flushing the TLB will lead to TLB misses. CL Flush has no impact on TLB stats.
3. CL Flush decreases L1 hits and increases L1 misses. It does not, however, change the stats at other levels.
4. CL Flush increases directory accesses due to eviction from coherent caches.
5. CL Flush and TLB Flush increase coherence energy.
6. TLB Flush and CL Flush seem to independently change coherence energy and the effect of (TLB+CL) Flush is the sum of individual flush effects.