Assignment 3: implement the MOESI protocol

Sam van Kampen

February 6, 2022

1 Introduction

In this assignment, we were asked to implement a multiprocessor system using the MOESI cache coherency protocol. Aside from the coherency protocol, the cache is the same as in the second assignment; it is 32 kibibytes in size, 8-way set-associative and uses a least-recently-used replacement policy. In this report, the implementation and results are described and given.

2 Implementation

As in the second assignment, the implementation is subdivided into a number of components: the CPU, the Cache, the Bus and the Memory.

2.1 CPU

The processor implementation is very simple; every cycle, an operation from the trace file is executed. If it is a read, `cpu_cache_if::cpu_read` is called, if it is a write, `cpu_cache_if::cpu_write` is called.

2.2 Cache

As before, the CPU can perform memory requests through the `cpu_cache_if` interface.

2.2.1 Reads

On a read hit, we only need to do statistics bookkeeping; no interesting cache behavior occurs.

On a read miss, we simply issue a read request. We then mark the cacheline as "pending exclusive"; this allows us to properly mark snooped reads as reading a shared cacheline. We wait for a response, which may be a response from the memory controller or another cache which has the cacheline in Modified or Owned state. While we're waiting, the cacheline might be invalidated, requiring us to issue the read again. If it has not been invalidated, we mark the line as no longer pending and optionally mark it as shared, if another processor has notified us that they also have this cacheline.

2.2.2 Writes

Writes are more complicated; on a write hit, if we are the only cache which has this cacheline, we can transition to the Modified state without any hassle. Otherwise we need to issue an invalidation request, which has complications (see the subsection on the bus). Write misses are simpler; they require us to first perform a *read with intention to modify*, which invalidates the cacheline in other caches while reading it into this one, and we can then mark the cacheline as modified and perform the write.

2.2.3 Snooping

The cache snoops the bus, looking for read requests, invalidation requests and reads with intention to modify.

2.3 Bus

The bus has changed substantially from the second assignment; instead of putting a request and waiting for a response being a single operation, it has been split up into three phases: `make_request`, which enqueues the request, `wait_for_response`, which waits for one response (can be called multiple times), and `end_request`, which marks the request as being finished. Additionally, the bus interface now has a method which marks a read request as returning a shared cacheline, telling the reading processor to mark the cacheline as shared in its local cache. Lastly, the bus keeps multiple invalidation requests to the same address from being pending at the same time; this safeguards against the case where a shared cacheline is written to by two processors, which invalidate each other's cachelines, causing (in the worst case) data loss, when the data in main memory is not up to date, or (in the best case) a performance degradation because both caches will have to do reads from main memory again.

A request on the bus is represented by the `BusRequest` type. This contains the data which is actually put on the bus, an event for when the request has been handled fully (including having been responded to) and a boolean which indicates whether the request has been started.

The data is represented by the `BusData` type. It contains a source (currently the processor IDs are used), a request ID (sequential, generated by the bus when a request is made), an operation (read, write, read done, write done), a priority (currently always 1) and a 32-bit data field.

Every negative clock edge, the bus puts either a request or a response on the bus, with responses being prioritized. In case a request is put on the bus, it is marked as started and its data is put on the `portBusReq` port. The only current bus slave (the memory) is also informed and asked to handle the request. The bus takes responses through its `respond` method.

2.4 Memory

The memory module is also straightforward; the memory simply takes requests from the bus (through implementing the `Bus_slave_if` interface) and sends out responses after 100 cycles.

3 Results

Unfortunately, my implementation was not able to generate results for all tracefiles; it hung on most of them.