Multicore Architectures

Embedded Systems Architecture

Technische Universität Berlin

Einsteinufer 17, 10587 Berlin, Germany

Appendix A:
Memory Architectures in Multiprocessing Systems

The most straightforward way to accelerate the execution of a sequential program is to extract multiple tasks from the program and run them simultaneously on multiple processors, which is called parallel computing. From the memory perspective, three major groups of memory architectures associated with multiprocessing systems can be identified: Shared Memory Architecture, Distributed Memory Architecture, and Distributed Shared Memory Architecture.

Generally **Shared Memory Architecture** refers to a large Bulk of memory can be accessed by multiple different processors in a multiprocessing computer system. In this category all the processors share a unique memory addressing space. All the memory accesses are directed to the large bulk of shared memory.

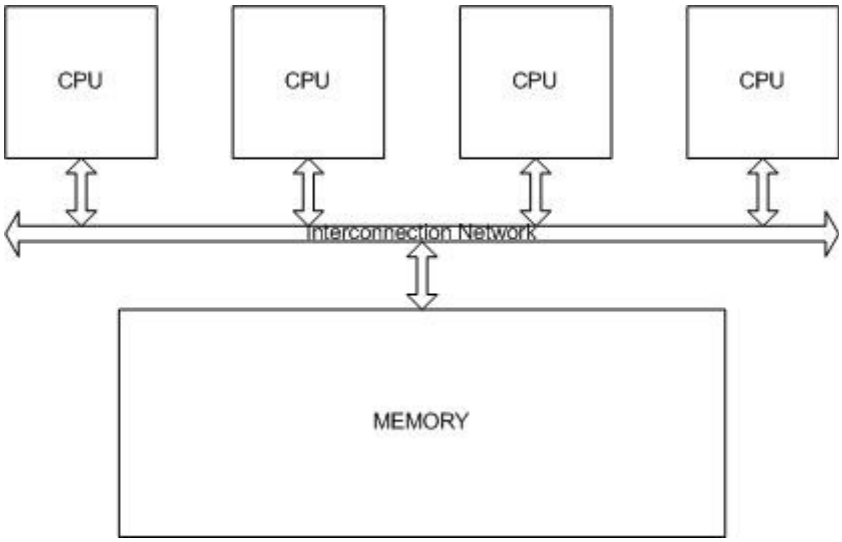

Figure 3 - Shared Memory Architecture

In a Multiprocessing **Distributed Memory System**, each processor has its own memory. The communication between different processors is carried out by massage passing across the interconnection network.

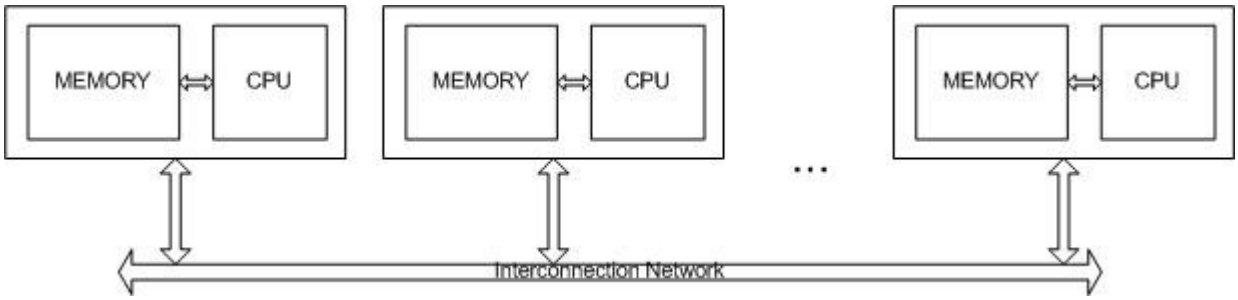

Figure 4 - Distributed Memory Architecture

The third group of memory organization is **Distributed Shared Memory Architecture**. In this group of multiprocessing system, all the processors still hold a common addressing space. However, the physical memory is distributed across the network, and each processor is associated with a bulk of local memory, which is associated with a part of the memory addressing space. Each processor could access all the memory modules across the network, meanwhile, the access to the local memory is much faster than the access to remote memory module.

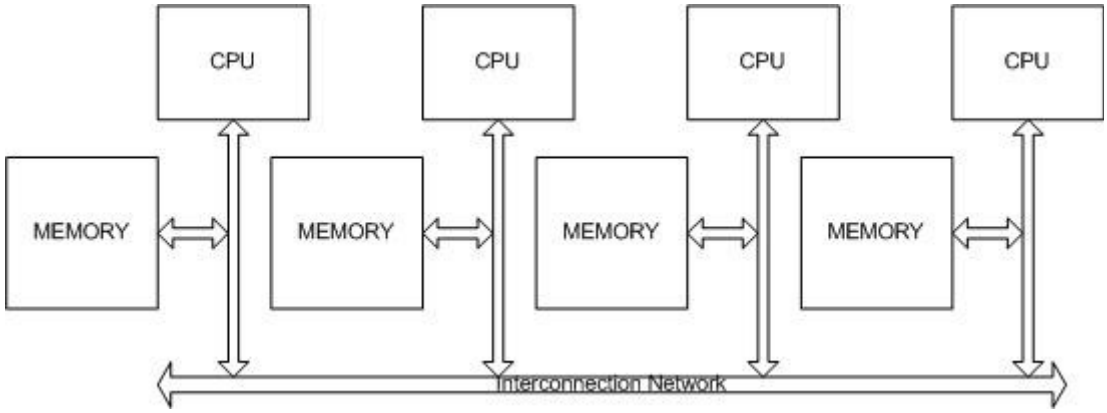

Figure 5 - Distributed Shared Memory Architecture

Cache Coherence

In this assignment we focus on the Shared Memory Architecture. As in an uniprocessor system, caches can help reducing the memory access delay. However, now each processor can read and write directly to local cache, as shown in Figure 4. Thus, when one processor writes to a memory location and its cache just stores it locally, a read of the same memory location on another processor can read a different value from its cache. Subsequently, this could lead to the wrong execution outcome of the program. To avoid this problem, cache coherence among different cache modules has to be implemented within the cache system.

Two major coherence protocols are the *snooping protocol* and *directory based protocol*. In the bus **snooping** technique each cache monitors every request on the bus and changes the state in its cache lines accordingly. This protocol requires every transaction in the network to be visible to all caches. Thus it is most commonly used with a bus implementation.

The other coherence protocol is a **Directory-based protocol**. In this protocol, directories are utilized to track data and memory requests in the memory system. The directories can be implemented distributed as well as centralized. In comparison with the snooping protocol, the directory based protocol is easier to use in network topologies other than a bus topology.

Those other network implementations are more scalable than normal bus. This protocol is generally used in distributed shared memory architecture.

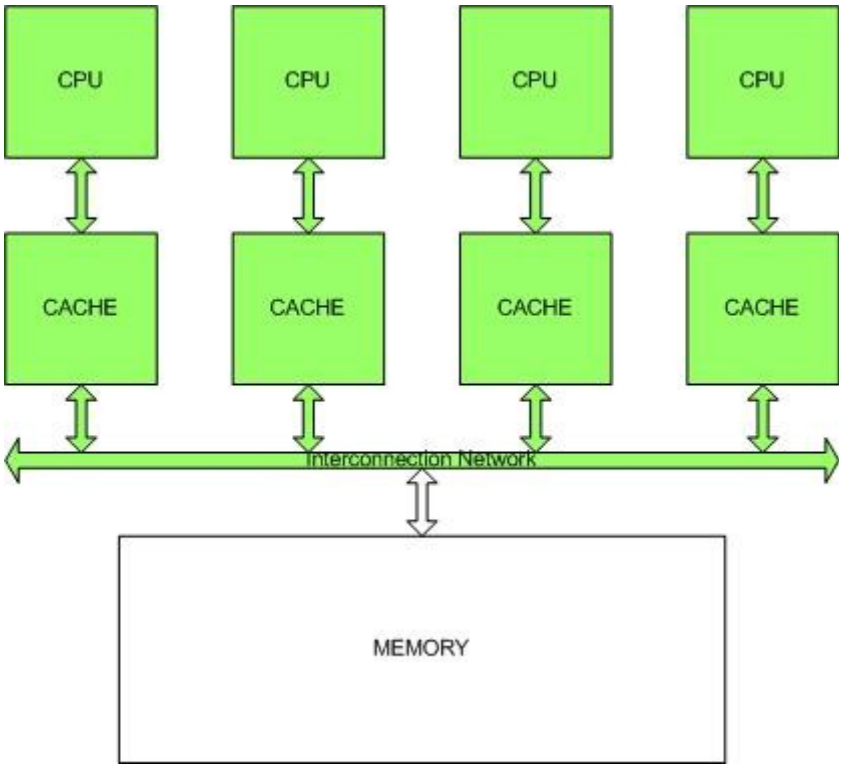

Figure 6 - Cache associated with different processors in Shared Memory system

From the cache line state transition perspective, most of the cache coherence protocol consists of a portion of five states, which are Modifed, Owned, Exclusive, Shared, and Invalid.

- *Invalid* state means that the current cache line does not hold valid data.
- *Shared* state represents the data is shared across the network and the value is the same as the value in the main memory.
- *Exclusive* state means that the data is the only copy in the cache system, the value is the same as the value in the main memory.
- *Owned* state means that the cache line holds the correct data and it could be shared, however at the same time, the value could be different from the value in the main memory. Only one cache can hold the data in Owned state.
- *Modified* state means that the cache line holds the most recent and correct data, which is different from the value in main memory.

One of the most commonly used cache coherence protocols are different versions of MESI protocol. In this version, the arcs in the transition diagrams are slightly different.

Appendix B:
Implementing a Bus

1. A bus could simply be represented by a group of signals.
   For instance, the address bus could be represented with
   ```
   sc_signal_rv<32> BusAddress;
   ```

   The corresponding port in cache module can be constructed as
   ```
   sc_inout_rv<32> Port_BusAddress;
   ```

   By connecting all the **Port_BusAddress** in cache with **BusAddress** signal, all the cache modules could read from or write to the address bus. It is important to notice that the main memory delay has to be handled at some places, probably in cache.

2. A bus can also be packed into a **Bus** class which implementing a bus interface **Bus_if**.

A simple bus interface is:

```
class Bus_if : public virtual sc_interface
{
public:
    virtual bool read(int addr) = 0;
    virtual bool write(int addr, int data) = 0;
};
```

An example of a bus class is:

```
class Bus : public Bus_if, public sc_module
{
public:
    // ports
    sc_in<bool>      Port_CLK;
    sc_signal_rv<32> Port_BusAddr;

public:
    SC_CTOR(Bus)
    {
        // Handle Port_CLK to simulate delay
        // Initialize some bus properties
    }

    virtual bool read(int addr)
    {
        // Bus might be in contention
        Port_BusAddr.read(addr);
        return true;
    };

    virtual bool write(int addr, int data)
    {
        // Handle contention if any
        Port_BusAddr.write(addr);
        // Data does not have to be handled in the simulation
        return true;
    }
}
```

Within the cache, a bus port with bus interface can be represented with
```
sc_port<Bus_if> Port_Bus;
```

The following statement in cache can put a read request on bus.

```
Port_Bus->read(address);
```

The above code is not complete, so please add more code to simulate your bus. You could add as many signals and ports as needed. Please notice that in task 3, the bus might also need to transfer some information about cache line state information, because if cache A wants to read data which is at Modified state in cache B, cache B has the obligation to write the data back to main memory before cache A can read from it.

Appendix C:
Helper Functions

Trace files

The TraceFile class is a data type that represents a file that contains traces of memory requests from a program's execution. Instances of this class can be used to read those traces from *trace files* in order to drive a memory simulation. A trace file might contain traces from multiple processors if it has been created from a multi-processor system. The file **`aca2009.h`** contains the declaration of the TraceFile class while the file **`aca2009.cpp`** contains its implementation. These two files must be part of programs that use the class. We also provide a structure to collect statistics from the simulation. Below is brief explanation of using the trace file and how to store the statistics to be displayed at the end of simulation.

Opening a trace file

A function named `init_tracefile` is provided which takes the commandline arguments and creates a `TraceFile` object from the tracefile specified in the first argument. The signature of this function is given as:

```
void init_tracefile(int* argc, char** argv[])
```

The created `TraceFile` object can then be accessed by the global pointer `tracefile_ptr`. This function also sets the global variable `num_cpus` to indicate how many CPU traces are present in the opened tracefile. Furthermore, as `argc` and `argv` are passed by reference, the function removes the first argument so that the rest of the arguments can be parsed afterwards.

Alternatively, the tracefile can be opened manually by creating a `TraceFile` object. The `TraceFile` class has only one public constructor with the following signature:

```
TraceFile(const char* filename);
```

The parameter **filename** is a C-style string with the name of the file. On object creation (either on stack or dynamically using **new**) the file will be opened. If an error occurred when opening the file, a **runtime_error** is thrown.

On object destruction the file will be closed, if it wasn't already. You can explicitly close a file and free up resources before the object is destructed by using the `close` member function.

Reading from a trace file

Reading a trace for a processor from the file can be achieved using the `next` member function (after the file has been opened). The signature of this function is the following:

```
bool next(uint32_t pid, Entry& e);
```

The parameter **pid** is the id/index of the processor for which the next trace-entry will be read and the **e** parameter is a reference to a structure of type `TraceFile::Entry` which will receive the entry read from the file. The function will return false if an error occurred. If there

is no event for the current time step, the function will return true, and the entry's type will be `ENTRY_TYPE_NOP`.
The `Entry` data type is a struct declared in the public interface of the class. Thus, when variables of this data type are declared, the name of the class must be prefixed with the name of the class (e.g. `TraceFile::Entry`).

If the file contains traces for more than one processor, then the reading of the traces does not need to be synchronized between processors. This means that each processor can call the `next` function individually. If a processor calls `next` when it already reached the end of its trace, it will keep reading `ENTRY_TYPE_NOP`.

The following code is an example of how the functions to handle tracefiles can be used:

```
// Open and set up the tracefile from cmdline arguments
init_tracefile(&argc, &argv);

TraceFile::Entry    tr_data;
Memory::Function    f;

// Loop until end of tracefile
while(!tracefile_ptr->eof())
{
  // Get the next action for the processor in the trace
  if(!tracefile_ptr->next(pid, tr_data))
  {
    cerr << "Error reading trace for CPU" << endl;
    break;
  }

  switch(tr_data.type)
  {
    case TraceFile::ENTRY_TYPE_READ:
      cout << "P" << pid << ": Read from " << tr_data.addr << endl;
      break;
    case TraceFile::ENTRY_TYPE_WRITE:
      cout << "P" << pid << ": Write to " << tr_data.addr << endl;
      break;
    case TraceFile::ENTRY_TYPE_NOP:
      break;
    default:
      cerr << "Error got invalid data from Trace" << endl;
      exit(0);
  }
}
```

The above code will read the trace file provided at command line using function `init_tracefile(&argc, &argv)` and will write all its trace entries to the standard output stream.

The `TraceFile` class also contains a number of member functions that return information about the file or its state. Those functions are shown in Table 1:

Function Signature	Operation
`bool eof() const;`	Returns **true** when all processors reached the end of its trace
`uint32_t get_proc_count() const;`	Returns the number of processors for which the file contains traces for.

Table 1: Member Functions

How the functions in Table 1 can be used is also demonstrated in the example code above.

Statistics

The **`aca2009.cpp`** file also comes with functions to gather and print statistics required for the simulator. These functions for the statistics are shown in the Table 2:

Function Signature	Operation
`void stats_init();`	Initialize internal data for the number of processors required in the trace file. Requires `num_cpus` to be set.
`void stats_cleanup();`	Free the internal statistic data
`void stats_print();`	Prints the statistics for each processor at the end of the simulation.
`void stats_writehit(uint32_t cpuid);`	Increments the writehit for the specified processor
`void stats_writemiss(uint32_t cpuid);`	Increments the writemiss for the specified processor
`void stats_readhit(uint32_t cpuid);`	Increments the readhit for the specified processor
`void stats_readmiss(uint32_t cpuid);`	Increments the readmiss for the specified processor

Table 2: Functions to calculate statistics

To explain the use of functions in Table 2 consider the code given below. The cache hit or miss is calculated randomly.

```
case TraceFile::ENTRY_TYPE_READ:
    f = Memory::FUNC_READ;
    if(rand()%2)
        stats_readhit(pid);
    else
        stats_readmiss(pid);
    break;
```