Multi-Core Simulations with Cache Coherent Protocols

Prakhar Gupta* Harshil Mital* Harsh Popat
{prakhar21550, harshil21050, harsh21048}@iiitd.ac.in

Abstract

The growing need for energy-efficient and high-performance systems in modern computing has led to a surge in the development of multi-core systems, with increasing popularity for chiplet-based architectures. Multi-core systems are turning out to be a promising solution for scaling core counts while addressing manufacturing and power challenges. This simulation study provides insights into single-core and multi-core simulations with various cache coherence protocols, architectural scalability, and performance in heterogeneous multi-chiplet environments. We employ MESI (Modified, Exclusive, Shared, Invalid) protocol as our intra-chiplet cache coherent protocol. We simulate the PARSEC benchmark with MESI 2 level protocol for a multi-core single chiplet architecture, followed by simulating a Mesh architecture with Garnet standalone protocol.

1. Introduction

The demand for energy-efficient and high-performance computing systems is rapidly increasing as modern applications and workloads become more complex. In response to this demand, multi-core systems have emerged as a promising solution, offering the ability to scale core counts and improve performance. One of the key innovations in this space is the use of chiplet-based architectures, which provide a modular approach to system design while addressing challenges related to manufacturing, power consumption, and performance. This report explores the simulation of single-core and multi-core systems, focusing on architectural scalability and performance in heterogeneous multi-chiplet environments.

Specifically, we investigate the impact of cache coherence protocols on system performance. The MESI (Modified, Exclusive, Shared, Invalid) protocol is employed as the intra-chiplet cache coherence protocol, enabling efficient data sharing and synchronization across cores. Our study begins by simulating a multi-core, single-chiplet system using the MESI 2-level protocol, followed by an exploration of a Mesh architecture with the Garnet standalone protocol. Followed by this, we attempted to also utilize Garnet along with MESI protocols to simulate a multi-chiplet multi-core architecture but found it difficult to be implemented. We also provide methods and steps to build the PARSEC benchmark, which was employed to test the single-chiplet multi-core MESI 2-level architecture, by exploring the variation across the number of cores in the architecture.

*Equal contribution

2. Methodology

This study investigates the impact of cache coherence protocols on the performance and scalability of multi-core systems in chiplet-based architectures. We focus on simulating systems using the MESI (Modified, Exclusive, Shared, Invalid) protocol for intra-chiplet cache coherence and evaluate the performance under different configurations.

The methodology begins with simulations of a multi-core, single-chiplet system using the MESI 2-level protocol. To test the architecture, we employ the PARSEC benchmark suite [2], exploring the variation in performance across different core counts. Following this, we attempt to implement and run a Mesh architecture using the Garnet standalone protocol, which manages communication between multiple chiplets. Although we attempted to integrate the MESI and Garnet protocols [1] for a multi-chiplet multi-core system, implementation challenges hindered successful execution. Instead, we ran experiments with single-chiplet multi-core architecture on MESI 2 level protocol, by varying the number of cores in the setup, and we used the `bodytrack` and `ferret` benchmarks of the PARSEC benchmark for running the test.

2.1. Methods Attempted

2.1.1 Building PARSEC disk image

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [2] benchmark suite is a collection of industry-relevant applications designed to evaluate the performance of multi-core processors. It includes workloads from diverse domains such as financial analytics, computer vision, video encoding, and scientific computing, making it a comprehensive tool for studying parallel computing.

In the context of gem5 [3], PARSEC benchmarks [2] are

used to simulate full-system workloads, allowing detailed evaluation of processor architectures, memory hierarchies, and cache coherency protocols. For running tests with the PARSEC benchmark, we needed to compile and build a disk image for PARSEC, and this was done with a series of steps. The built PARSEC disk image will be compatible with the gem5 simulator.

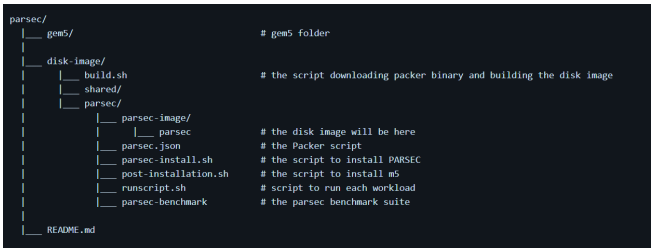

Figure 1. PARSEC directory tree

To build PARSEC with gem5 simulator [3], we follow the following steps presented here. Create a new directory termed `parsec` and replicate the directory structure as shown in 1 using the files provided here. The directory named `parsec-benchmark` can be cloned from here using the command -

```
git clone https://github.com/darchr/parsec-
    benchmark
```

Now, we need to clone gem5 [3] from the source and then build the m5 utility, which is ISA-agnostic. This can be done as follows -

```
git clone https://github.com/gem5/gem5
cd gem5/util/m5
scons build/x86/out/m5
```

Then, we are required to build the packer tool to create the disk image. This can be done by the following commands -

```
cd disk-image
./build.sh
```

Post running, we can find the disk-image in `parsec/parsec-image/parsec`. Once building the disk image is done, we can run an example script that is present within the gem5 repository at gem5/configs/example/gem5_library/x86-parsec-benchmarks.py.

However, this requires KVM for running the script, and hence, we provide an alternative script which uses O3CPU with X86 system here.

The script uses x86-linux-kernel-4.19.83 and x86-parsec, and the disk image as created from the above instructions. We need to compile and build gem5 using the `MESI Two Level` cache coherence protocol, which can be done using the following commands -

```
scons build/X86/gem5.opt -j<proc>
```

and here proc stands for the number of cores you wish to specify for a faster build. Once compiled, you may use the aforementioned config file to run the PARSEC benchmark programs using the following command:

```
cd gem5
build/X86/gem5.opt \
configs/example/gem5_library/x86-parsec-
    benchmarks.py \--benchmark <
    benchmark_program> --size <size>
```

Description of the two arguments, provided in the above command are:

–benchmark, which refers to one of 13 benchmark programs, provided in the PARSEC benchmark suite. These include `blackscholes, bodytrack, canneal, dedup, facesim, ferret, fluidanimate, freqmine, raytrace, streamcluster, swaptions, vips, x264.`

–size, which refers to the size of the workload to simulate. There are four valid choices for the same: `test, simsmall, simmedium and simlarge.`

2.1.2 Multi-Core Single-Chiplet

We simulate multi-core single-chiplet architecture by using MESI 2-level cache coherence protocol, which has an L1 cache private to each core, while the L2 cache is shared amongst all the cores. L1 cache is further split into instruction and data caches. As per information provided here, we can setup and build gem5 with MESI 2 level as follows-

```
git clone https://github.com/gem5/gem5
cd gem5
scons defconfig build/X86 build_opts/X86
scons setconfig build/X86
    RUBY_PROTOCOL_MESI_TWO_LEVEL=y
scons build/X86/gem5.opt -j<proc>
```

After building gem5 with MESI 2 Level protocol, we can run the script present here as per the following command -

```
build/X86/gem5.opt configs/example/
    gem5_library/x86-parsec-mesi2.py
```

2.1.3 Multi-Core Mesh Architecture with Garnet Standalone

Garnet standalone is a cache coherence protocol that is used to operate Garnet [1] in a standalone manner, and works in conjunction with Garnet Synthetic Traffic injector.
This protocol assumes a 1-level cache hierarchy. The role of the cache is to simply send messages from the CPU to the

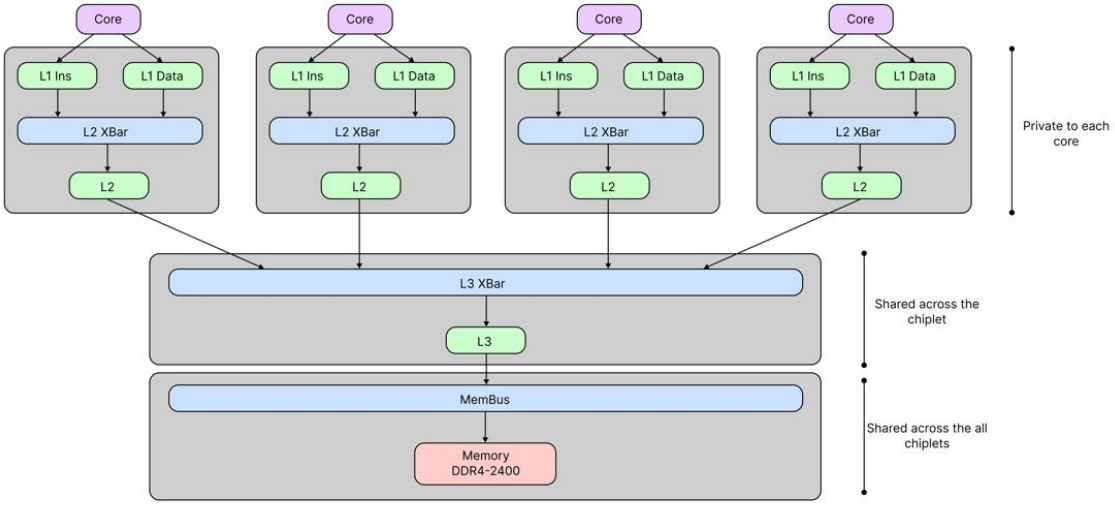

Figure 2. Multi-core Single Chiplet Design

appropriate directory (based on the address), in the appropriate virtual network (based on the message type). It does not track any state. The directory receives the messages from the caches but does not send any back. The goal of this protocol is to enable simulation/testing of just the interconnection network.

We implement Garnet Standalone [1] with Mesh XY, 16 CPUs, and 4 Mesh rows by following the given instructions here -

```
scons defconfig build/X86 build_opts/X86
scons setconfig build/X86
    RUBY_PROTOCOL_GARNET_STANDALONE=y
scons build/X86/gem5.debug -j<proc>
```

The above builds gem5 with the Garnet Standalone [1] protocol and the required script is available at configs/example/-garnet_synth_traffic.py, and we can run it as follows -

```
./build/X86/gem5.debug configs/example/
    garnet_synth_traffic.py  \
        --num-cpus=16 \
        --num-dirs=16 \
        --network=garnet \
        --topology=Mesh_XY \
        --mesh-rows=4  \
        --sim-cycles=1000 \
        --synthetic=uniform_random \
        --injectionrate=0.01
```

2.1.4 Multi-Core Multi-Chiplet Architecture

This setup comprises of a 4x4 core architecture which has 4 chiplets with each containing 4 cores. Each if these cores has a private L1 and L2 cache and there is a shared L3 cache across the chiplets. The 4 cores present on each chiplet are ”intra-connected” by a single chiplet and communicate via a simple network to ensure cache coherence across the chiplet. This is the same chiplet as referred in 2.2.2. Furthermore, the communication between the 4 chiplets handled by using garnet to ensure cache coherence across the them.

For the intra-chiplet communication, each private cache’s (L1 and L2) cache controller as well as the directory controller are connected a router for communicating to the other caches in the chiplet using a point to point topology. In this case these routers are just simple switches.

```
Router<-InternalLink->Cache-Controller
Router<-InternalLink->Directory-Controller
Router<-ExternalLink->Router
```

For the inter-chiplet communication, the L3 cache controllers and the directory controller contain garnet routers. Their connections are shown below.

```
Router<-InternalLink->L3-Cache-Controller
Router<-InternalLink->Directory-Controller
Router<-ExternalLink->Router
```

Experiment	Benchmark	Cores	Hits	Misses	Accesses	Miss Rate (%)	Hit Rate (%)
1	Bodytrack	2	180	42802	42982	99.68	0.32
2	Bodytrack	4	180	27076	27256	99.34	0.66
3	Bodytrack	8	133	37020	37153	99.64	0.36
4	Bodytrack	16	135	34493	34628	99.61	0.39
5	Ferret	2	182	34555	34737	99.47	0.53
6	Ferret	4	206	32462	32668	99.37	0.63
7	Ferret	8	133	25890	26023	99.59	0.41
8	Ferret	16	135	22702	22837	99.38	0.62

Table 1. Miss Rate and Hits Trends Across Cores for Bodytrack and Ferret.

The script for the above system has been created but, unfortunately, cannot be tested since it would require a coherence protocol to be written and compiled specifically for it. This means writing a .slicc file from scratch to define the individual states and transitions for the protocol and then getting it to compile successfully.

2.2. Experimental Setup

In this study, we utilized two benchmarks from the PARSEC suite—`bodytrack` and `ferret`—to evaluate the performance of a multi-core, single-chiplet system using the MESI 2-level protocol. The experiments were conducted by varying the number of cores in the system (2, 4, 8, and 16) to analyze the effect of core count on the L2 cache's miss and hit rates.

The primary objective was to observe how these cache performance metrics change as the number of cores increases within a single-chiplet architecture using the MESI protocol. To ensure the analysis remained benchmark-agnostic, both `bodytrack` and `ferret` were chosen, providing a diverse range of workload behaviours.

Due to lack of time, simulations were run for a duration of 3 minutes, and observations were made based on the data collected within this time frame, as the full simulation runs were not completed. This setup allowed for an initial assessment of the cache behaviour and its dependency on the number of cores in the system.

3. Results and analysis

3.1. Bodytrack Benchmark

The **Bodytrack benchmark** exhibits consistently high miss rates (>99%) across all core configurations, starting with **99.68% at 2 cores** and stabilizing around **99.61% at 16 cores**. The small hit rates (0.32% to 0.39%) indicate poor cache locality, with limited performance gains as cores scale. The slight improvement at 4 cores suggests better cache utilization, but contention increases beyond 4 cores, leading to diminishing returns.

3.2. Ferret Benchmark

The **Ferret benchmark** shows slightly better cache performance, with **miss rates ranging from 99.47% (2 cores)** to **99.38% (16 cores)** and corresponding hit rates from 0.53% to 0.62%. The improved performance compared to Bodytrack indicates better memory locality, allowing Ferret to scale more efficiently with increased cores.

3.3. Comparison and Recommendations

Ferret consistently outperforms Bodytrack in cache efficiency due to its more localized memory access patterns. Both benchmarks, however, suffer from contention in the shared L2 cache at higher core counts. To improve performance, we can Increase L2 cache size. Use cache partitioning to reduce contention.

In summary, Bodytrack's scattered access patterns lead to higher memory dependency, while Ferret's better locality enables slightly better scalability. Both workloads highlight the limitations of shared L2 caches in multi-core systems.

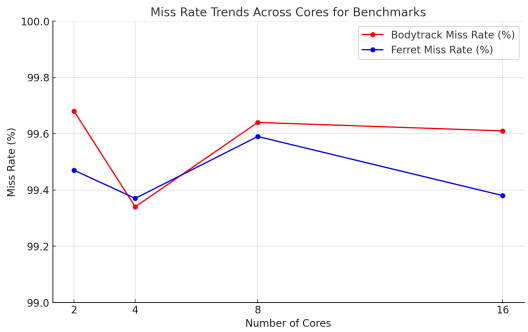

Figure 3. Miss Rate Trends Across Cores for Bodytrack and Ferret.

References

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 33–42, 2009. 1, 2, 3

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery. 1

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011. 1, 2