Performance Analysis of Caches on POWER Architecture

Hemanth Ramesh
M.Eng Computer Engineering
Virginia Tech.
Blacksburg, VA
hemanthr@vt.edu

Sairam Ganti
M.Eng Computer Engineering
Virginia Tech.
Blacksburg, VA
sairamg@vt.edu

***Abstract*—The growing demand for higher computation power complemented with the power wall (along with the memory wall and ILP wall) shifted the computation industry towards designing multi-core processors. As more compute-intensive workloads are built, and it becomes increasingly challenging to meet the performance goals.**

In modern processors, caches play a crucial role in reducing the memory bottlenecks and helping us meet the performance goals. Today, we use caches at various levels and it has proven to be an efficient way to reduce latency. This work focuses on simulating POWER 8 and POWER 9 architectures and demonstrating the performance impact. We implement this on a gem5 simulator environment and run a custom benchmark based on Radix sorting and investigate the performance variations between POWER 8 and POWER 9 architectures. Additionally, we would provide an extensive study and determine the cache configurations at which the system performs the best.

***Index Terms*—Computer Architecture, gem5, Simulation, Caches, Radix Sort, POWER ISA, DRAM, Analysis**

I. Introduction

Every aspect of today's world is heavily reliant on computational devices. Primarily, the computation needs of large corporations and data centers are higher than ever and it is expected to grow more year by year. The end of Dennard's scaling has caused the computation industry to slowly move away from traditional uni-processor systems to multi-core processor systems. However, this multi-core processor systems suffered from performance bottlenecks due to the access latency of the DRAM memory, which led to the extensive use of faster SRAM-based caches. As the number of cores increased, various levels of caches were put in place to reduce latency, share data and also improve performance. In general, caches are meant to hold the copies of the main memory blocks which enable faster access to frequently accessed data.

Generally, higher levels of caches (closer to the CPU) have the fastest access time and this time increases as we go up the hierarchy. Even though caches reduce access latency in most cases, a cache miss could drastically increase the access time and degrade the performance. However, if a workload successfully exploits properties such as the Principle of Locality, it could greatly benefit from the caches. In the past two decades leading processor manufacturers such as Intel and AMD have largely adapted caches in their designs and developed several architectures targeting different application areas. Most importantly, with today's processors using virtual addressing, efficient cache design is a major concern. Several designs such as virtually indexed, physically tagged caches are developed today, that couples caches with Transfer Look-aside Buffers (TLBs) for higher performance gains.

This work primarily focuses on demonstrating performance variations that result from the introduction of level 3 cache for varied levels of configurations such as the cache size, associativity, block size, etc on POWER 8 and POWER 9 architectures. As a part of this work, we have developed POWER 8 system with 4 level caches and a POWER 9 system with 3 level caches on the gem5 simulator. To demonstrate the cache performance on both these architectures we have developed a custom benchmark that performs Radix sorting and multiplication on a 2-D array. The benchmark is designed to perform a radix sort of the 2-D array, row-wise starting from the first row. Once the sorting is completed, the benchmark performs a scalar multiplication on all the elements again starting from the first row. We adapted this strategy specifically to ensure that the data is brought in from the memory and placed in the cache one row at a time, row-wise, until all the rows are sorted. Following this, the multiplication causes the current data in the cache to be replaced with the new data from the memory (starting from the first row of the 2-D array). We provide more details on how the benchmark uses the cache in section X. We run this workload targeting level 3 cache with variations of cache capacities, associativity, block size, and for different DRAM technologies. Section 2 describes the implementation aspects of our work.

II. Related Work

It is highly beneficial to have full system simulations for architectures in tools like gem5 for various reasons. As observed in [11] gem5 simulations of various ISAs tend to provide the necessary data for an in-depth analysis of design, debugging of the design at an appreciable pace. The other extremes of a full-blown VHDL/RTL based simulation would be more accurate but would be slow and tedious for quick development and simulations based on QEMU etc, would be much faster but fail to provide with necessary insight into low level design. Also, these are not always cycle-accurate, which is an important gap that gem5 bridges.

This section also talks about some of the current research that is done in areas related to the L3 cache and the resulting performance benefits. [1] uses an HPC benchmark to demonstrate the performance effects of a shared L3 victim cache coupled with a cache replacement policy on Intel's Cascade Lake processor. [8] provide a detailed performance evaluation of the Cache compression systems at different cache levels using an execution-driven simulation environment. Recently, AMD released their next-generation Zen 3 core CPU which includes a redesigned octa-core complex, a new set of L3 cache solutions, and support for stacked AMD 3D V-Cache [9]. According to [10] several benchmarks including Gem5, libx264 4K Transcoding, 7-Zip, and more were run on this new technology. Only in Gem5 did 3D V-Cache make a significant difference to performance. The rest was barely noticeable, with around 5
These studies provide an emphasis on the fact that only a certain type of workload can reap the most benefits from a given architecture. In [12] the authors use gem5 in conjunction with NVMain to model and simulate hybrid multi-level cache hierarchies and provide functional verification along with the results of SPEC2006 benchmarks on the same. Similarly, in [14] the authors configure and calibrate multilevel caches by modifying the configurations and run benchmarks on the simulation and real-life hardware to compare performance. In [15] the authors go a step further and also change the cache coherence protocols to see their impact on performance.

III. Implementation

This section outlines the implementation aspects of our work. Here we specifically discuss the capabilities of the gem5 simulator at our disposal and the detailed walk-through of building a POWER 8 and POWER 9 systems. In this work, we will be developing python configuration scripts for modeling POWER 8 and POWER 9 systems. We focus on modeling a single CPU system and hence no explicit cache coherence protocol will be modeled. We will be using the Classic caches subsystem offered by gem5 in our work.

Initially we had to make several changes to the source files of gem5 to facilitate the functioning of L3 and L4 caches in the cache hierarchy model. In the *configs/common/CacheConfig.py* file we added additional objects to initiate the L3 and L4 caches for different kinds of CPUs. We added these configurations to all the available CPU models so that we can facilitate the proposed work in future work sections as well, but the most important part is adding the objects for the SimpleCPU configurations for the scope of this project. Then, in the Caches.py we add classes L3 Cache and L4 Cache which define the parameters for the respective caches. In the Options.py class provided by gem5, we pass the additional L3 and L4 level cache parameters, namely, their sizes, associativity and line sizes, with their respective default values, if and where necessary.

In the *"gem5\src"* folder we modify two files to aid the implementation of the cache hierarchies. Firstly, we edit the *BaseCPU.py* class in the *cpu* directory to include two new definitions *addThreeLevelCacheHierarchy* and *addFourLevel-CacheHierarchy* to initialize the connections between the CPU, various cache levels and ultimately the memory. Then we modify the *XBar.py* in the *mem* directory to include two classes L3XBar and L4XBar to include the 256 Bit crossbar that will function between the cache hierarchies and defined the width, latency and the snooping characteristics of the crossbars for each level.

After this we created a configuration file that creates the SimObject, which defines the system we are going to simulate. This System object will be the root of all other objects in our system. The System object contains various information such as physical memory ranges, clock and voltage domains, etc. The values that will be used for these variables are not defined in this report.
Next, we will make a script that contains the cache object that we are creating, which is basically an extension of the Cache SimObject. The Cache SimObject inherits from the BaseCache object that is available on gem5. This allows us the flexibility to define various parameters such as associativity, tag latency, etc. Our model consists of an L1 cache with two sub-classes, an L1 data cache and an L1 instruction cache which constitute the lowest level of the hierarchy. At higher levels, we will introduce a unified L2 and L3 cache. Once all the levels of caches are defined, we instantiate and connect the caches to the interconnects.
To connect the L1 cache, we will utilize the connectCPU function to connect the cache and CPU and connectBus function to connect the cache to a bus. These functions are defined separately for L1 DCache and L1 ICache. Next, we will add functions to connect the L2 Cache to the CPU-side and L3 Cache-side bus. Finally, we will add functions to connect the L3 Cache to CPU-side and Memory-side buses. Once the cache object script is created, we will import the names in this script to the namespace of our initially created configuration file (that contains our System) and connect all the components appropriately.

The above-described procedure allows us to model the complete POWER 9 system and POWER 8 system with only three levels of cache. To fully model the POWER 8 system, we will have to add the level 4 cache extension. To do this, we follow the similar procedure outline above, but instead of connecting the L3 cache directly to the memory bus we will model an L4 cache and connect the CPU side bus of the L4 cache to the L3 cache and finally, connect the memory side of the L4 cache to the memory bus. With these changes incorporated, we have a fully modelled POWER 8 system. Fig. 1 and 2 show the high-level view of POWER 8 and POWER 9 systems.

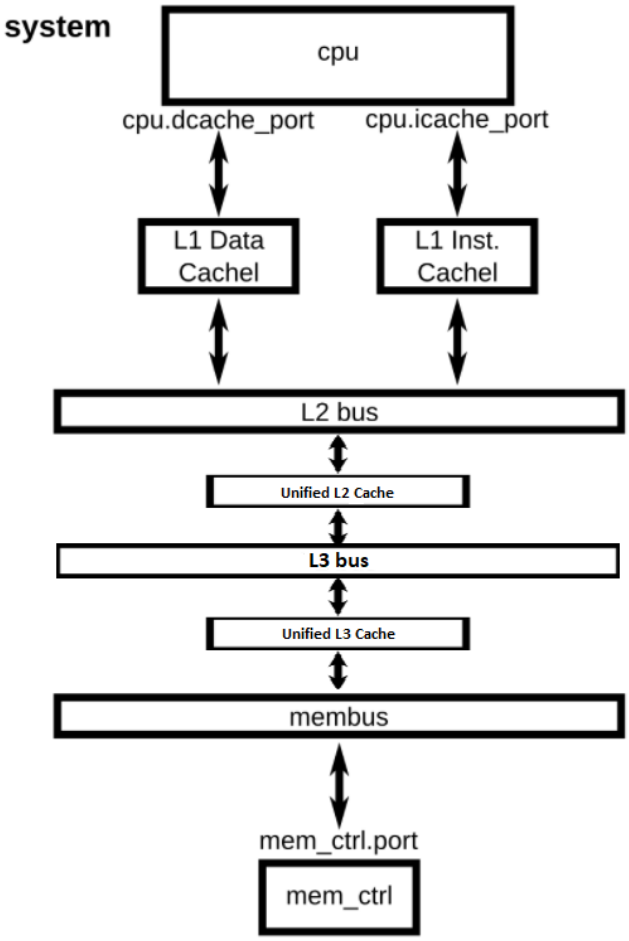

Fig. 1. POWER 9 design

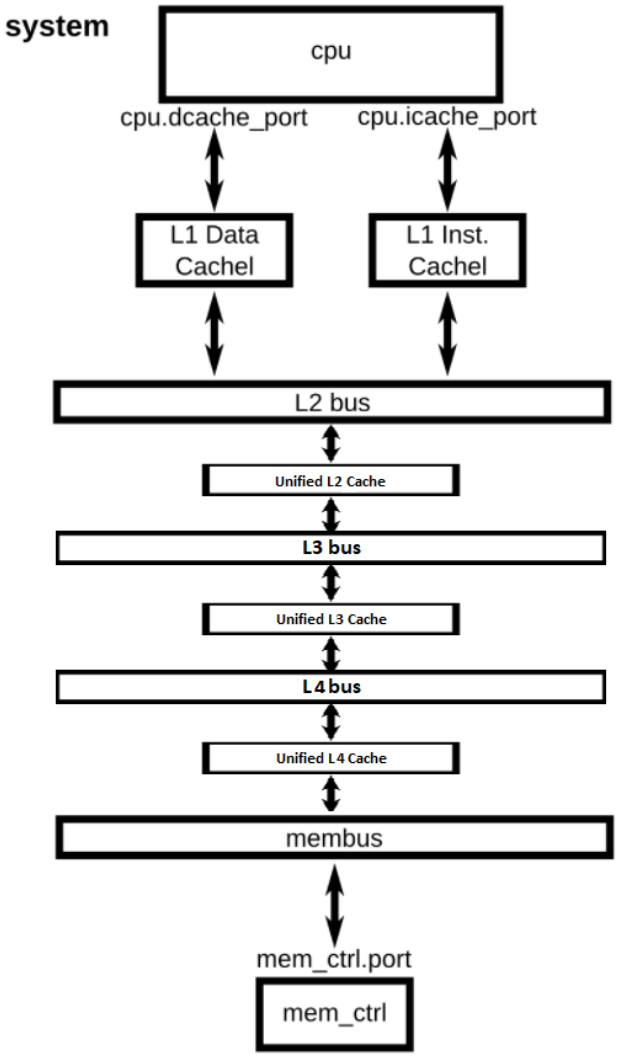

Fig. 2. POWER 8 design

IV. Benchmark

This section describes the Radix sort and multiplies benchmarking program that we developed to evaluate the performance of level 3 caches on POWER 8 and POWER 9 systems. The initial part of this section provides the background on the working of Radix sort and the later part describes the design of our benchmark and how it utilizes the cache.

A. Radix Sort

Radix sort is a non-comparative sorting algorithm, which avoids comparison by creating and distributing elements into buckets according to their radix (or the base). For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all the digits have been considered [source Wikipedia]. This sorting method is well suited to the data than can be sorted lexicographically, such as integers.

Radix sort is one of the most widely used non-comparison-based sortings. N keys can be Radix sorted by using two N key arrays along with a count array of size 2r which can hold integers of size n, where r is the radix. This method enables the data to be sorted with a constant number of passes over the data for a given length of the keys and the radix value. Consider an example, if we are sorting x bit integers with a radix of r then we need [x/r] iterations each with two passes over the source array. The first pass gathers the counts of the number of keys with each radix. The counts are then used to determine the offsets in the keys of each radix in the destination array. The second pass moves the source array to the destination array as per the offsets.

Radix sort can be implemented either to start at the most significant bit (MSB) or at the least significant bit (LSB). MSB sorts are better suited for sorting strings and fixed-length integers. If variable-length integers are sorted lexicographically then the numbers from 1 to 10 would be sorted as [1, 10, 2, 3, 4, 5, 6, 7, 8, 9]. Also, MSB sorts are not very stable the sample space contains duplicate keys. On the other hand, LSB radix sorts the short keys first and then the longer keys lexicographically. This sorting coincides with the normal ordering of the integer representation, unlike the MBS sort. This work uses LSB-based sorting.

B. Design

This benchmark performs a row-wise radix sorting on a configurable randomly generated 2-D array of dimension mxn. Each row consists of n elements bounded between 0 to 1000, which will be fed into the radix sorting function. Once the row is sorted, it will be written to the original matrix, this step is repeated till all the m rows are sorted, the benchmark program performs a scalar multiplication on all the elements row-wise starting from 0 through m. Sorting row-wise causes each row to be pulled into the cache and sorted iteratively based on the radix. Performing a row-wise scalar multiplication starting from the first row of the 2-D array after all the rows in the 2-D array are sorted further enhances the benchmarking program as this ensures the data is moved in and out of the caches constantly. Also, this cache activity increases with the growing dimensions of n and m.

V. Evaluation & Analysis

To evaluate our work we used the benchmark described in section 4. We evaluated level 3 cache performance with respect to various cache parameters such as associativity, cache capacity, and workloads. Additionally, we evaluated our system performance with different DRAM technologies such as DDR3 1600, DDR3 2133, LPDDR3 1600 and HMC 2500. The following part of this section discusses the cache performance as applied to various parameters described above. We configured the 2-D array to have 100 rows and 1000000 columns of integers, which gives us a total of about 380 MB of data (4bytes integer x 100 rows x 1000000 columns) which is over 60% more data than all the cache levels put together on POWER 8 and on POWER 9 it is around 200% more data can the total cache capacity.

A. Associativity

Set associative caches are extensively used in today's cache design. Here we will discuss the cache performance for various levels of associativity for the benchmark we have designed. We evaluated the performance of the level 3 cache on both POWER 8 and POWER 9 for associativity values, 1, 2, 4, 8 and 16 with cache level 3 cache size set to 96 MB on the POWER 8 system and 128MB on the POWER 9 system and the cache block is set a 64bytes. As it can be seen from Table 1, that the overall misses reduces as we increase the associativity. This is primarily because every time the radix sort function is called, it causes 8 integers from the array to be brought into the cache. For the lower values of associativity the data blocks maps to a smaller number of the cache blocks, with the worst case mapping being the direct mapped cache with only one block per memory block. This increases the conflict misses in the system and hence the performance degrades.
Also, it can seen that the miss rate on POWER 8 for 1 way and 2 way associativity is about three times higher than that on POWER 9, which can be attributed to the higher cache capacity on POWER 9 architecture.

Associativity	POWER8 Miss Rate	POWER9 Miss Rate
1 way	0.094447	0.037897
2 way	0.089647	0.037695
4 way	0.037762	0.037765
8 way	0.037622	0.0379
16 way	0.037622	0.037622

Table 1

B. Cache Capacity

This section describes the cache performance of POWER 8 and POWER 9 systems for varied levels of cache capacity ranging from 2MB to 128MB with an 8-way associative cache. As shown in Table 2, the overall cache misses with respect to level 3 cache reduces as we increase the cache size. In this specific case, where each row of the array can occupy 4Bytes x 1000000 = 3906KB of data when the cache value is lower, the data elements will be swapped in and out of the cache while sorting due to the limited capacity of the cache. In other words, we will have significantly higher capacity misses, which reduces as the cache size increases. Also, it can be seen that as the cache size increases, POWER 9 shows marginal improvement on the miss rates for our workload.

Cache size	POWER8 Miss Rate	POWER9 Miss Rate
2 MB	0.587095	0.588239
4 MB	0.163717	0.163724
8 MB	0.090271	0.090201
16 MB	0.090182	0.090112
32 MB	0.090004	0.089934
64 MB	0.089647	0.089578
128 MB	0.088915	0.088839

Table 2

C. Workloads

We also varied the benchmark to have different dimensions (rows x columns) as in Table 3 to observe the execution time on POWER 8 and POWER 9 systems. Table 3 summarises the execution times of the workloads of different dimensions. It can be seen that as the number of elements increase, the execution times also increases. However, it can be that the POWER 9 system with a larger level 3 cache show improved execution time as compared to POWER 8. This shows that having a larger low-level cache could positively impact the performance of a given workload as compared to having larger higher-level caches. This could be one of the reasons why IBM decided to discontinue using level 4 cache on POWER 9 systems.

Workload	POWER8 CPU Time	POWER9 CPU Time
5x1000	0.016261	0.0162
10x1000	0.033739	0.033662
5x10000	0.187521	0.187237
50x1000	0.173141	0.172929
10x10000	0.374553	0.374092
100x1000	0.34542	0.345039
5x100000	1.937796	1.935371
50x10000	1.874263	1.872341
10x100000	3.873406	3.869319
100x10000	3.748015	3.744295

Table 3

D. DRAM

We evaluated our benchmark on POWER 8 and POWER 9 systems with varying DRAM memory types. We considered four different types of memories, DDR3 1600MHz, DDR3 2133 MHz, LPDDR3 1600 MHz, and an HMC-type memory. The HMC2500 is an implementation of the Hybrid Memory Cube model defined in [18]. What we observe from the data is that the HMC2500 model which is the Hybrid memory cube model consumes the least amount of energy in terms of memory refresh and reads with its closest contender consuming at least 8x the energy in terms of reads.

However, of the three more conventional memory systems, Low Power-DDR3 consumes about half the energy required to keep the memory consistent(memory refreshes) and on memory reads. We observe the various trends that are followed for these parameters in figures 3 and 4. Another stark difference we noticed in terms of memory types is the metric "Warm-up tick" which is denoting the number of ticks it took to get the memory ready at startup. DDR3 memories took the shortest amount of time, being at least 2x faster than LPDDR3 and HMC memory types, we plot these values in Fig.5. We also noticed that the average bandwidth that is offered by different memories and memory controllers is fairly the same and we think this is because the memory implementations are able to accommodate the requests placed by our workloads. From the data we've gathered and analyzed, we see that when performance is the requirement, the standard DDR3 memories offer the highest performance of the bunch even though they consume more power. However, for applications where the performance-per-watt is more concerning, LPDDR3 memory is a more suitable option.

VI. Challenges Encountered

This section outlines the challenges faced and the actions that were taken during the course of this project. Our first step was to model the POWER 8 and POWER 9 system on the gem5 simulator. Developing the configuration scripts for POWER 8 and 9 systems involved modifying various gem5 system files. Our initial hurdle was defining the separate classes for coherent interconnects (XBar's) needed for level 3 and level 4 caches that allows us to configure various parameters such as response latency, snoop response latency,

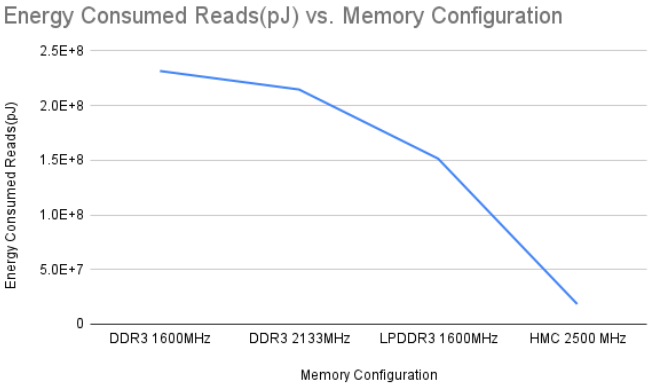

Fig. 3. Energy Consumed for the read operations vs Memory Type

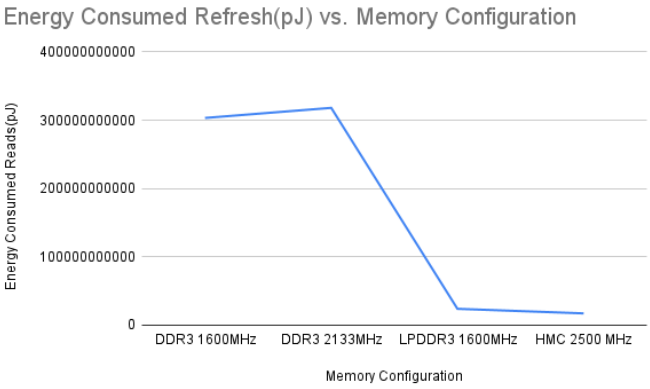

Fig. 4. Energy Consumed for the refresh operations vs Memory Type

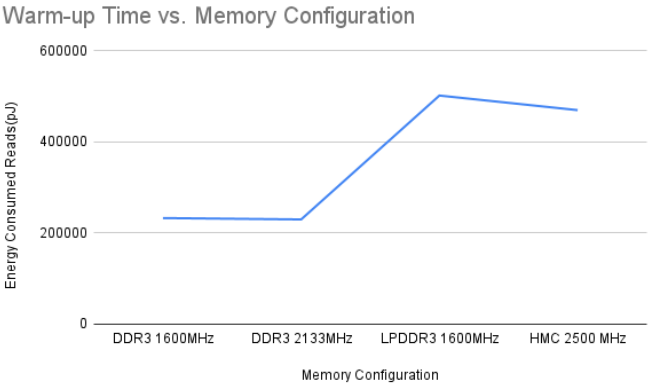

Fig. 5. Warm-up time for the memory to become active and available vs Memory Type

etc be set independent of the interconnects used by the other levels of caches. These classes are defined in the XBar.py file.

After the coherent interconnects were defined we had to actually implement the cache hierarchy in the BaseCPU.py file. This step took a significant amount of our time, initially

to identify the file itself (BaseCPU.py) and later to define the hierarchy.

Once the POWER 8 and POWER 9 systems were defined, we used the matrix multiplication binary that was given to us to perform the initial tests. For benchmarking, we intended to use the Parsec benchmarking suite initially, however, the gem5 simulator is set up in system call emulation mode with several undefined system calls. We cross-compiled parsec and recompiled gem5 by ignoring several system calls, but this approach ended in a kernel panic (this could be because of the several ignored system calls). Finally, we decided to develop our own benchmark, which primarily [16] helped us understand the requirements for the implementation.

Another challenge that we faced was related to Cache Line Sizes. We initially wanted to vary the cache line sizes across workloads and caches to identify and analyze the performance impact of the parameter. However, the SimpleCPU and BaseCPU implementations of gem5 did not allow us to pass the parameter for *cachelineSize* through our configuration files. We've later identified a potential solution to this and included it in our future work section.

Following this, we noticed that as we kept increasing our workload size to run the benchmark program on our cache implementations, gem5 simulations took too long to complete and would often crash the simulator before completing the run. We are yet to identify the related issue for this challenge and fix this and is left for future work on the project.

VII. Future Work

This section outlines the potential extension to the work presented in this paper. Our work primarily focused on gaining insights into the level 3 cache performance on the POWER architecture. However, in this work, we have modeled the CPU as an in-order execution unit. Hence, one immediate extension would be to design an out-of-order CPU that enables us to analyze various parameters such as execution time and speedup of the given workload.
Also, the current system runs in syscall emulation mode and supports only trivial system calls. One can implement some of the common syscalls such as access, getsockname,

There To facilitate the execution of real-world benchmarks, we could add definitions for some of the key system calls for the gem5 simulator running in system call emulation mode. Also, our simulated POWER systems use in-order execution CPU. To gain a more realistic understanding of the parameters such as execution time, speed up, etc an out-of-order CPU in full system mode can be implemented that closely resembles the actual POWER machines.

For the hurdle, we faced with the *cachelineSize* parameter that controls the block size of the caches we found that there are two potential solutions. We can hard-code the varying sizes at the build stage for POWER architecture and run multiple builds for workloads to simulate and analyze the differences. The second and more proper solution is to modify the gem5 source implementation to include an additional parameter to take as an option and determine the line size for simple and base CPU implementations. While we explored the second solution, we were constrained by the time we have to implement it and hence left it as future work for this project. The first solution, albeit feasible, requires a lot of time in terms of build and simulations to be viable within the time frames of this project.

VIII. Resources

We have compiled all the data from our simulations and tests and documented them in a Google Sheet for reference and included it in this section for reference. The link to the sheet is: https://tinyurl.com/3hrx8utu and this is shared with the Virginia Tech domain for users to see our statistics. Along with this, we have also uploaded all our stats.txt files for all the runs and the code with separate patches for L3 and L4 cache implementations and relevant scripts for POWER8 and POWER9 systems in a github repository whose URL is: https://github.com/hemanthr28/power_cache_hierarchy.git. This repo also consists of a readme file to walk through the setup and execution processes of our project.

IX. Conclusion

In our work, we simulated POWER 8 and POWER 9 architectures on the gem5 simulator and evaluated the performance variances caused by level 3 cache on these systems using a custom radix sorting and multiply benchmarking tool. Our results demonstrate that the POWER 9 system with a larger level 3 cache shows improvement in the overall performance (reduced execution time and cache miss rate) compared with the POWER 8 system.

References

[1] Christie L. Alappat, Johannes Hofmann, Georg Hager, Holger Fehske, Alan R. Bishop and Gerhard Wellein, "Understanding HPC Benchmark Performance on Intel Broadwell and Cascade Lake Processors", High Performance Computing: 35th International Conference, ISC High Performance 2020.

[2] Buchholz, Werner (1962). Planning a Computer System. p. 5.

[3] GEM5 Simulator: https://www.gem5.org/

[4] Aleksei Sorokin,Sergey Malkovsky, Georgiy Tsoy, Alexander Zatsarinnyy,Konstantin Volovich, "Comparative Performance Evaluation of Modern Heterogeneous High-Performance Computing Systems CPUs", Electronics 2020, 9(6), 1035.

[5] H. Q. Le, J. A. Van Norstrand, B. W. Thompto, J. E. Moreira, D. Q. Nguyen, D. Hrusecky, M. J. Genden, M. Kroener, "IBM POWER9 processor core".

[6] "IBM POWER9 processor core".

[7] Bin-feng QIAN, Li-min YAN, "The Research of the Inclusive Cache used in Multi-Core Processor", ICEPT-HDP 2008.,

[8] Keun Soo Yim, Jihong Kim, and Kern Koh, "Performance Analysis of On-Chip Cache and Main Memory Compression Systems for High-End Parallel Computers", Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '04, June 21-24, 2004.

[9] M. Evers, L. Barnes and M. Clark, "The AMD Next Generation Zen 3 Core," in IEEE Micro, doi: 10.1109/MM.2022.3152788.

[10] https://www.tomshardware.com/news/amd-3d-v-cache-benchmarks-mixed-results-milan-x-cpus

[11] Roelke, Alec, and Mircea R. Stan. "Risc5: Implementing the RISC-V ISA in gem5." First Workshop on Computer Architecture Research with RISC-V (CARRV). Vol. 7. No. 17. 2017.

[12] Asif Ali Khan, Fazal Hameed, and Jeronimo Castrillon. 2018. NVMain Extension for Multi-Level Cache Systems. In ¡i¿Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools¡/i¿ (¡i¿RAPIDO '18¡/i¿). Association for Computing Machinery, New York, NY, USA, Article 7, 1–6. DOI:https://doi.org/10.1145/3180665.3180672

[13] Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 34(4), 1-17.

[14] Weber, Taisy Silva, Basso, Pedro Martins, Cache Calibration for Accurate Simulation of Multi-core Systems. https://www.lume.ufrgs.br/handle/10183/235491

[15] N. B. Mallya, G. Patil and B. Raveendran, "Simulation based Performance Study of Cache Coherence Protocols," 2015 IEEE International Symposium on Nanoelectronic and Information Systems, 2015, pp. 125-130, doi: 10.1109/iNIS.2015.52.

[16] Anthony LaMarca and Richard E. Ladner, "The Influence of Caches on the Performance of Sorting", Journal of Algorithms, Volume 31, Issue 1, April 1999, Pages 66-104.

[17] István Z. Reguly, Abdoul-Kader Keita and Michael B. Giles, "Benchmarking the IBM Power8 processor", CASCON 2015 November 2-4, 2015, Toronto, Canada

[18] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2015. High performance AXI-4.0 based interconnect for extensible smart memory cubes. In Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition (DATE '15). EDA Consortium, San Jose, CA, USA, 1317–1322.