Computer Architecture Explorer: An Educational Tool for Computer Engineering Students

Submitted To

Mattan Erez
Electrical and Computer Engineering Department
University of Texas at Austin

Prepared By

Ben Endara
Tianfang Guo
Cordell Mazzetti
Jackson Schilling

ECE464H Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Spring 2024

CONTENTS

TABLES

FIGURES

EXECUTIVE SUMMARY

The goal of this project was to build an educational tool for students taking the course *ECE460N Computer Architecture* that will assist them as they learn material throughout the semester. There were several steps to accomplish this goal. First, we identified a large set of parameters, which were essential to the design of any computer architecture; we then considered a range of reasonable changes that could be made to these parameters, such as different memory organizational patterns, different types of microarchitecture, different branch predictors, etc. Secondly, we took this set of parameters, and simulated each of their performance using a simulator. Then the performance data was stored in a database for use in the user interface tool, which displays the stored data for students to access.

Our system design includes three major subsystems: the simulator, database and data management, and website for the user interface. The simulator includes the parameters being simulated, the benchmarks those parameters are run against, and the simulator itself. The database consists of the database itself and data management for how the data is input from the simulator, digested for storage into the database, and the formatting and code to be output to the website. The website includes the integration from the database to frontend website code as well as the user interface for students to interact with in addition to graphical data representations.

The parameters to vary within the tool need to cover as many different configurations as possible while also not overwhelming the user with too many options to allow the user to gain a meaningful understanding of the effects of the different parameters on the performance metrics. Additionally, the total number of configurations grows exponentially with the number of parameters. Since each simulation takes a non-trivial amount of time to complete, this limits the total number of parameters that could have been chosen. With these restraints in mind, seven parameters were identified, and reasonable values for each parameter have also been selected. Again, the number of values for each parameter must be limited to reduce simulation count, but also chosen carefully to cover a wide enough range of values such that they will have a measurable impact on the results.

To effectively demonstrate the performance differences a robust benchmark is required. We used SPEC CPU 2017 as our benchmark suite. Due to simulation time constraints, only one of the benchmarks could be used. Any singular benchmark would take too long to simulate, and so the benchmark had to be broken down into smaller chunks called simulation points, or SimPoints. These SimPoints were generated using Valgrind to record basic block vectors for each interval of 100 million instructions in the benchmark, then the SimPoint tool was used to find one interval which best represented the entire benchmark. This 100 million instruction interval was determined to be the best overall representation of SPEC CPU 2017 while also being small enough to not exceed the simulation time limits for this project.

The gem5 simulator was used to gather performance data for each simulation configuration. Since the SimPoint only specifies the point in the execution at which to start simulating, the state of the simulation must be saved at that point and then restored for each simulation. The script which was used to configure gem5 for this task had to take the SimPoint information as well as the benchmark binary and inputs as arguments. Once the checkpoint had been created, detailed

simulations could be run using the gem5 O3 CPU model. The script for these runs had to take the aforementioned SimPoint information as well as the checkpoint and all of the parameters for the simulation.

The database storage functionality was required to have the ability to digest data from the simulator, store the data using a specified data schema into the database itself, allow access to the data from the database, and be fully integrated into the front-end code of the website through the user interface. All code is in Python and the database storage service is through MongoDB. This allowed ease of coding use and data manipulation using libraries such as pandas and numpy in addition to integration with MongoDB and the front-end code. The amount of space we used in the database did not exceed the free plan available with MongoDB and the speed of access times are also not important to users.

The first element inside the website is the front-end server, which handles rendering the web pages that the user interacts with. Requests sent from the front-end to the DBaaS are sent through a React component that will handle network failures and retries. The UI contained in the front-end code has multiple pages for the user to gain a better understanding of the computer architecture content that our site pertains to as well as systems for the user to pull simulated performance metrics for a set of input parameters. It also contains multiple ways to compare that data as you vary sets of parameters, such as how a specific performance metric changes as you increment or decrement a system parameter.

The front-end is composed of multiple key libraries, frameworks, and environments. React was chosen as the structure for site over alternatives like Flutter, Angular, and jQuery because of its simplicity and familiarity. Chart.js, React-toastify, and Styled-components are all additional libraries that will help generate our graphs, send notifications, and reduce the clutter of our react components respectively.

The DBaaS server is the server that handles data and communication with the database. It verifies that requests to specific URLs return properly serialized, formatted information that the front-end can utilize. The DBaaS is a service provided by MongoDB which is also hosting our database. Additionally, it allows the definition of functions that can be performed on the database and queried from the front-end. We interact with this DBaaS using Realm-web which is a lightweight package that provides connection functionality to Mongo services.

Many of the parameters changed throughout the implementation process due to unforeseen limitations in both time available to run simulations and in what parameters could be feasibly changed using the gem5 standard library. Originally, there were around twice as many planned parameters, with many more values that were planned for simulation. However, each additional parameter multiplies the total number of simulations by its number of values, causing an exponential increase in simulation time as each parameter is added. Additionally, the limitations within the gem5 standard library made it unfeasibly difficult to use certain other parameters, as to do so major modifications would have to be made to the standard library to expose those variables.

The benchmark was originally going to be SPEC CPU 2006, however we changed to the 2017 version since that was the version that we had access to through our faculty mentor. Additionally, we had originally planned to run at least three of the benchmarks within SPEC, but again due to simulation time constraints we had to use only one of the benchmarks, 505.mcf_r. For similar reasons only one checkpoint within that benchmark was used.

For the database, there was only one change made between the planning and implementation phases of the project. Originally, the database was designed to hold data in a mixed hierarchy data schema, where certain data points would be prioritized over others for ease of searchability when being called by the website. This proved to be unnecessary, as the website could easily parse whole dictionaries of data in the form of stored documents for the simulator outputs. Thus, the data outputs were stored as dictionaries, after being scraped for only needed data, and then were easily accessed by the website.

When initially creating the website, it was designed with a front-end and back-end server that was connected to the database. Given the simplicity of the query-based nature of the website, the backend server was deemed unnecessary as the front-end server could query directly to the public database. Originally, the website was designed so that students could submit a few parameters they wanted to keep static, and one they wanted to sweep over to observe how performance changes. However, it became apparent that graphing the higher dimensional graphs required to represent multi-variable states would become too complex and outside the capabilities of the graphing library used on this website. Due to the complexity of simulation, it was uncertain which parameters and metrics would be used by the end of the project. During implementation, it was decided that the website would be configurable such that it would recursively build out its inputs, allowing for rapid reconfiguration.

Simulator testing was completed by executing the procedure for creating and running checkpoints and simulations and confirming that the results given were reasonable. Database testing was outlined as including three key parts: access efficiency, correct data storage, and correct data reads. All three have been successfully tested and have shown positive results for the system in its final state. Website testing was successful, allowing the website to maintain a low level of code length while maintaining quality. The new recursive structure is capable of dynamically handling the input type of nested values depending on the value of previously selected inputs which was not possible with the previously static structure.

The simulator took significantly longer to get working than originally anticipated, this was due to complications with limited documentation and difficulties in using gem5 to create checkpoints and restore from them.

Although most of the output data management code had been written and tested, there was a significant gap in time before actual output data was received for processing. However, once data was received, the code was able to be run and within a few hours, the database had been populated, ready for use by the website. Due to the complexity of the data simulation, the website was unable to be built statically for a set of expected inputs which required a significant amount of flexible, dynamic design to handle any range of inputs. The result is a very extensible, quickly-modifiable site that can be changed to handle all types of data from a single config file.

In terms of possible ethical problems with our design, there is the potential issue of data posted on the website not allowing students to learn the correct information they might be looking for from the project, thus either invalidating the learning goals or worse, teaching the wrong things.

We have done our best to make sure that all simulation data is accurate and is presented in an intuitive way to students so that any misunderstandings or misinformation are minimized, if not eliminated.

Past the current iteration of the project, multiple recommendations can be made for further improvements, changes, or even experiments to run. For the simulator, more parameter and benchmark options could be used. For the database, a different document hierarchy or database service could be explored. Lastly, for the website, different visual aids might be possible or more helpful for students.

Overall, the Computer Architecture Explorer was split into three separate main subsystems: the simulator, the database, and the website. The gem5 simulator ran benchmark code to generate performance data for the computer architecture using various parameters, the performance metrics and specific settings were stored in a database, and lastly the website allows students to pull different scenarios from the database and view them in a user interface. The gem5 simulator was chosen over a custom design, MongoDB was used for the database service with panda and numpy for data crunching and sorting, and the website uses React, NodeJS, Chart.js, Realm-web, Styled-Component, and AWS.

1.0 INTRODUCTION

This document outlines how we approached building an intuitive computer architecture exploration tool for students taking Computer Architecture and those who are also looking to learn how changing system parameters affects performance metrics. The following content discusses our design problem, design solution, final implementation, testing and evaluations performed, time and cost considerations, safety and ethical aspects of the design where applicable, and future work and recommendations for further improvements. As an overall summary of the design, the tool had its data generated using the gem5 open-source, processor simulator and displayed through a free website running off of NodeJS, Django, and MongoDB.

2.0 DESIGN PROBLEM STATEMENT

The goal of this project was to build an educational tool for students taking the course *ECE460N Computer Architecture* that will assist them as they learn material throughout the semester. There were several steps to accomplish this goal. First, we identified a large set of parameters, which were essential to the design of any computer architecture; we then considered a range of reasonable changes that could be made to these parameters, such as different memory organizational patterns, different types of microarchitecture, different branch predictors, etc. Secondly, we took this set of parameters, and simulated each of their performance using a simulator. Then the performance data was stored in a database for use in the user interface tool, which displays the stored data for students to access.

2.1 Specifications

In order to be successful, the tool that we created must be a useful tool for students learning computer architecture. The parameters that we chose must be relevant to the course and be recognizable to students in the class. The parameters must also cover as much of the class as possible, so that students will be able to reinforce what they have learned from any part of the class. The performance parameters that the tool shows should be recognizable to students in the class and be able to clearly demonstrate the differences between various architecture configurations. The database needs to be fast enough to serve the data without any noticeable lag for the user. All data accessed from the database must be accurate to the correct simulated parameters requested. The tool should be easy to access for all UT ECE students. The interface

should be intuitive and provide a good user experience. The tool should also include explanations of the various parameters and performance data shown.

2.2 Background Information

Computer architecture is a complex field, and a basic understanding of it is required to understand this project. This section explains some of the concepts that are related to the areas of computer architecture that this project will cover.

2.2.1 Memory

In modern computer systems, roughly 90% of the hardware is made up of various forms of memory technologies. This system of memories is organized into an optimized hierarchy by studying the various trade-offs between capacity, speed, and cost. This memory hierarchy is generally broken down into three tiers: cache, main memory, and secondary storage.

The cache is the fastest and closest physically to the processor, as it is integrated as a part of the CPU package. Caches are extremely fast but this technology (SRAM) is also extremely costly, therefore caches are usually broken down into their own hierarchy (L1, L2, L3) to further optimize capacity and speed. Most systems have a tiny but extremely fast L1 cache that is the closest to each core of the CPU, this level is sometimes broken down further into an instruction cache and a data cache, to further optimize performance. The next level is the L2 cache, which is bigger than the L1 cache, and in turn it is slower. The final level is the L3 cache, which is a large cache that is usually shared between all cores in the CPU, with each core “owning” a piece of the whole L3 cache. Changes to the sizes of the cache hierarchy, as well as internally organizations and designs of specific levels have a dramatic impact on performances, and this level of memory has perhaps the largest effect on overall system performance.

Main memory, more commonly known as RAM, is the next level of the memory hierarchy. RAM uses DRAM technology, which is several magnitudes slower than the SRAM used in caches. However, this decrease in performance is accompanied by a large increase in memory area efficiency, and a decrease in manufacturing cost. This is the reason that modern computers often have tens of megabytes of cache, but many gigabytes or even terabytes of RAM. The purpose of

RAM is to fetch data from the much slower secondary storage devices and have it ready “on standby” to be loaded into the cache. However, RAM does not have the capacity to simultaneously hold a large amount of data from secondary storage, which is why various hardware and software techniques exist to counteract this fact. One such method is virtual memory, which will not be discussed as it is outside the scope of this project.

Secondary storage devices, most commonly HDD and SSD devices, are the lowest tier of the memory hierarchy. These devices provide very large capacities (many terabytes) at the cost of being extremely slow. Thus, the CPU will never access these devices directly, as doing so will have catastrophic impacts on performance. Rather, data will be forwarded to RAM, where it waits to be cached. Secondary storage is still a crucial part of the memory hierarchy, as it provides long-term data storage.

2.2.2 Pipelining

Modern CPUs are built upon three key concepts: Pipelining, Branch Prediction, and Out of Order Execution. These three techniques work in tandem to maximize core utilization and performance, thereby dramatically increasing throughput.

A pipeline is the fundamental structure of all modern CPUS. Each instruction is processed in discrete, isolated stages by dedicated hardware. This allows for instruction level parallelism whereby multiple instructions exist within the pipeline simultaneously, all in different stages of processing. This allows for a greater overall throughput. Pipelines are generally divided into 5 broad stages: Fetch, Decode, Execute, Memory Access, and Writeback. These stages may be further broken down into different stages, and they themselves pipelined, to achieve even higher parallelism and utilization. However, in general, the way a pipeline is built, how many stages it has, and various details in the design such as bypassing directly impact instruction processing speed and throughput.

One major flaw of pipelines is the fact that all software can have situations that would cause the pipeline to stall, meaning although there are stages that are empty, no instructions can be put into

them to be processed due to various dependencies from previous instructions that have yet to reach the end of the pipeline and resolve. Branch prediction and out-of-order execution seek to remedy this flaw. Normally, when a pipeline encounters a branch, the system must stall until the branch resolves before fetching the next instruction, effectively wasting a number of clock cycles equal to the pipeline depth. However, if a pipeline employs a branch predictor, the pipeline can immediately fetch the next instruction without having to wait for the previous branch to resolve. There are numerous branch prediction algorithms, each with their own performance and cost trade-offs, and this choice has an effect on system performance, as any branch mispredictions will have large impacts on the throughput, since the pipeline must be flushed out, and instructions re-fetched. It should be noted that branch predictors only solve one problem, which are stalls caused by a branch dependency. There are other types of dependencies, most of which are caused by instructions stalling to wait for data which is not available until previous instructions are resolved. Out-of-order execution (O3E) solves this problem. O3E buffers all instructions and only allows instructions to enter the execute stage when all its dependencies are resolved. This allows instructions that don't depend on previous instructions to skip over instructions that have unresolved dependencies, effectively eliminating stalls. Hence, this method will execute instructions out of order from the original order that they were fetched in. The obvious issue with this approach is that the software expects its program to execute in order. The reorder buffer mitigates this issue by buffering instructions until their appropriate commit time; This maintains the illusion of a von Neumann computing model while enjoying the performance of an out of order execution model. Along a similar vein, processors often have more physical registers than architectural registers. This enables techniques such as register renaming, which allows the processor to dynamically map architectural registers (ISA) to a larger number of physical registers (microarchitecture). This in turn enables out of order execution, maximizing the use of functional units and reducing stalls.

Functional units (FUs) are another large factor in the performance of a specific microarchitecture, since they are what actually executes instructions during the execute stage. The number of functional units in a microarchitecture significantly influences its overall performance by providing a ceiling for the pipeline throughput and determining to which degree instruction level parallelism can be exploited. Intuitively a larger amount of FUs allow the

processor to execute more instructions in parallel, increasing throughput; inversely a smaller number of FUs will have a negative impact on the overall throughput of a processor. It should be noted that at a certain threshold there will be significant diminished returns for adding more FUs, as the processor's front end will struggle to take advantage of all the additional hardware. In fact this is a problem that many modern processors run into, thus techniques such as simultaneous multithreading are employed to mitigate this problem.

Modern CPUs have been implementing increasingly complex pipelines that effectively utilize branch prediction and O3E, as well as other concepts such as superscalar and super-pipelines to create highly sophisticated systems that enhance parallelism and resource utilization, resulting in more efficient and faster instruction processing. The various specific implementations of these features greatly impact the performance of the system.

2.3 Conclusion

In conclusion, modern computer systems are extremely complex and sophisticated, and employ various elaborate systems and techniques to gain performance for various trade-offs. Any change to these specifications can have deep impacts on system performance, and specific combinations of these parameters may work well together whereas other combinations are fundamentally incompatible. We hoped to capture these characteristics and relationships within our simulation performance data.

3.0 DESIGN SOLUTION

This section discusses multiple aspects of our system that are required for it to function and deliver the product that we set out to create. We will discuss its design concept, risks, and testing plan. Each of these has multiple subsections detailing them, identifying our findings when researching and preparing as well as key insights regarding the future and continued development of each project aspect.

3.1 Design Concept

Our system design includes three major subsystems: the simulator, database and data management, and website for the user interface. The simulator includes the parameters being

simulated, the benchmarks those parameters are run against, and the simulator itself. The database consists of the database itself and data management for how the data is input from the simulator, digested for storage into the database, and the formatting and code to be output to the website. The website includes the integration from the database to frontend website code as well as the user interface for students to interact with in addition to graphical data representations. Figure 1, below, describes the system block diagram to show the flow of data between the three major subsystems. The following sections also describe the project in more detail on an individual subsystem level.

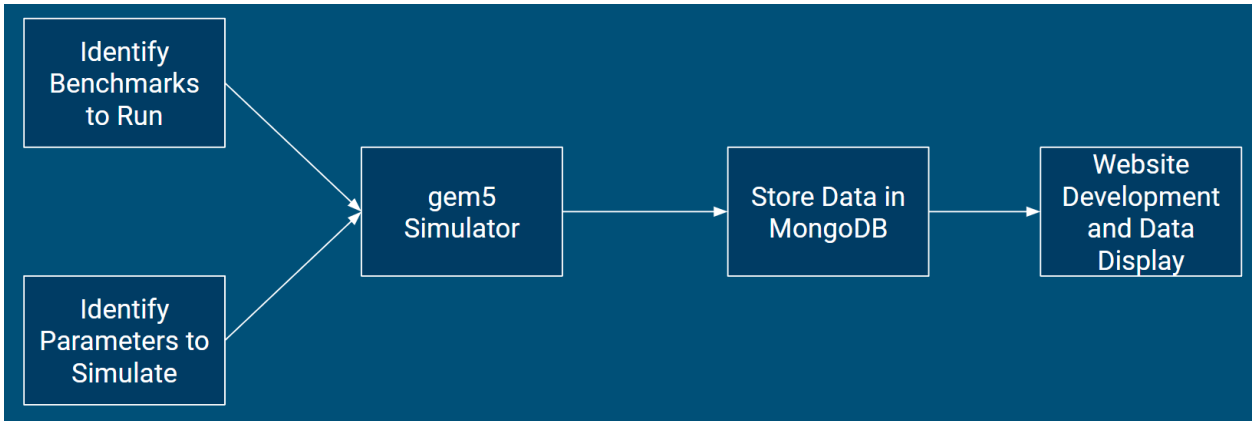

Figure 1. System Block Diagram

3.1.1 Parameters

The parameters to vary within the tool need to cover as many different configurations as possible while also not overwhelming the user with too many options to allow the user to gain a meaningful understanding of the effects of the different parameters on the performance metrics. Additionally, the total number of configurations grows exponentially with the number of parameters. Since each simulation takes a non-trivial amount of time to complete, this limits the total number of parameters that could have been chosen. With these restraints in mind, seven parameters were identified, and reasonable values for each parameter have also been selected. Again, the number of values for each parameter must be limited to reduce simulation count, but also chosen carefully to cover a wide enough range of values such that they will have a measurable impact on the results. The parameters that were chosen are summarized in Table 1.

Parameter Name	Parameter Values		
L1 Data Cache Size	8 kB	32 kB	128 kB
L2 Cache Size	128 kB	512 kB	2048 kB
Branch Predictor	2-Bit Local	TAGE	Perceptron
Reorder Buffer Size	32	128	512
Physical Register Count	128	256	512
ALU Count	1	4	16
Multiplier/Divider Count	1	4	16

Table 1. Simulation Parameter List

3.1.2 Benchmarks

To effectively demonstrate the performance differences a robust benchmark is required. We used SPEC CPU 2017 [1] as our benchmark suite. It provides 10 INT (integer) benchmarks, each of which has both a SPECspeed benchmark and a SPECrate benchmark. The SPECrate benchmarks are more suited for single threaded testing, so they were selected. Due to simulation time constraints, only one of the benchmarks could be used. Therefore, the 505.mcf_r benchmark was chosen since it is a part of the representative subset of SPECrate INT that is recommended in "Experiments with SPEC CPU 2017" [2]. Any singular benchmark would take too long to simulate, and so the benchmark had to be broken down into smaller chunks called simulation points, or SimPoints, as described in "Automatically Characterizing Large Scale Program Behavior" [3]. These SimPoints were generated using Valgrind [4] to record basic block vectors for each interval of 100 million instructions in the benchmark, then the SimPoint tool [5] was used to find one interval which best represented the entire benchmark. This 100 million instruction interval was determined to be the best overall representation of SPEC CPU 2017 while also being small enough to not exceed the simulation time limits for this project.

3.1.3 Simulator

The gem5 simulator [6] was used to gather performance data for each simulation configuration. Since the SimPoint only specifies the point in the execution at which to start simulating, the state of the simulation must be saved at that point and then restored for each simulation. This saved state, or checkpoint, was created using gem5 in a fast-forward mode using the simple atomic CPU model. The script which was used to configure gem5 for this task had to take the SimPoint information as well as the benchmark binary and inputs as arguments, and is in Appendix A. Once the checkpoint had been created, detailed simulations could be run using the gem5 O3 CPU model. The script for these runs had to take the aforementioned SimPoint information as well as the checkpoint and all of the parameters for the simulation, and is in Appendix B. Finally, the GNU parallel tool [7] was used to run all 2,187 combinations of the simulation parameters on a modern 16 thread desktop machine, which took roughly 24 hours to complete.

3.1.4 Database

The database storage functionality was required to have the ability to digest data from the simulator, store the data using a specified data schema into the database itself, allow access to the data from the database, and be fully integrated into the front-end code of the website through the user interface. All code is in Python and the database storage service is through MongoDB. This allowed ease of coding use and data manipulation using libraries such as pandas and numpy in addition to integration with MongoDB and the front-end code. We had experience from previous coursework, ECE 461L (Software Engineering and Design Laboratory), using MongoDB through Python and integrating it into a front-end website, allowing users to interact with an interface to call data from the database and have it displayed. Thus, the choice of MongoDB as our database service was easy and reliable. In addition, MongoDB allowed for free use of a shared server, and thus, cost nothing from our project budget. The amount of space we used in the database did not exceed the free plan available with MongoDB and the speed of access times are also not important to users.

A python script was used to slim down the simulator output data to a more digestible format for the database, in addition to removing any excess data not useful to the user themselves. This code crawled through each simulation output file, took the parameters for each run, the specific

requested output data, and the label for the run, and created a dictionary in the script. This dictionary was then used to input the data as needed into the document-style format present in MongoDB.

3.1.5 Website

The website can be broken down into two chunks: the Front-end and the Database-as-a-service (DBaaS). Their connectivity and properties can be seen in Figure 2 below.

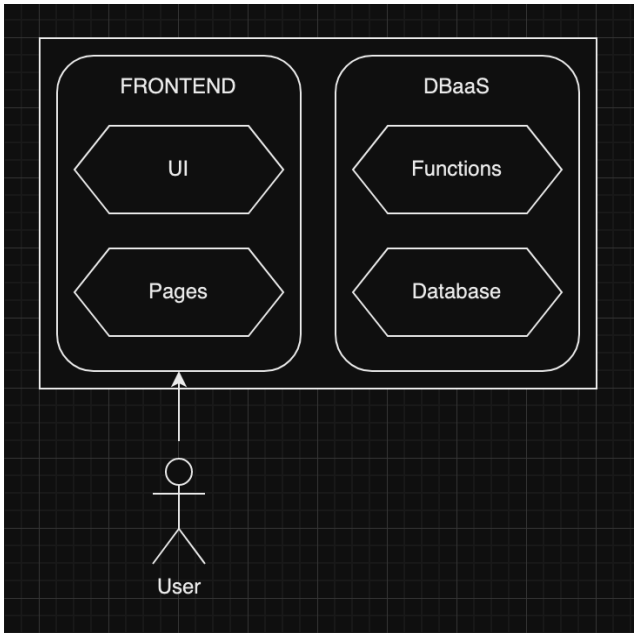

Figure 2. Website Server and Database Relation

The website as a subsystem is the most user-facing element, so it must be intuitive and fast for it to not bottleneck the rest of the work that goes into this project.

The first element inside this subsystem is the front-end server. This is the server that handles rendering the web pages that the user interacts with. Requests sent from the front-end to the DBaaS are sent through a React component that will handle network failures and retries. The UI contained in the front-end code has multiple pages for the user to gain a better understanding of the computer architecture content that our site pertains to as well as systems for the user to pull simulated performance metrics for a set of input parameters. It also contains multiple ways to

compare that data as you vary sets of parameters, such as how a specific performance metric changes as you increment or decrement a system parameter.

The front-end is composed of multiple key libraries, frameworks, and environments. React was chosen as the structure for site over alternatives like Flutter, Angular, and jQuery because of its simplicity and familiarity. Chart.js, React-toastify, and Styled-components are all additional libraries that will help generate our graphs, send notifications, and reduce the clutter of our react components respectively. Without using these free, open-source libraries, we could still construct the website, but it would be significantly more time-consuming and of lower quality.

The DBaaS server is the server that handles data and communication with the database. It verifies that requests to specific URLs return properly serialized, formatted information that the front-end can utilize. The DBaaS is a service provided by MongoDB which is also hosting our database. Additionally, it allows the definition of functions that can be performed on the database and queried from the front-end. We interact with this DBaaS using Realm-web which is a lightweight package that provides connection functionality to Mongo services.

Combined together, they allow for an elegant, compartmentalized server network that will be able to handle fetching the simulation data from the database and presenting it very easily with minimal computation. Our servers are hosted on AWS.

4.0 DESIGN IMPLEMENTATION

Throughout the implementation phase of the project, multiple changes were made that were unforeseen in the planning phase. Such changes included the parameters and benchmark for the simulator team, a change of the database data hierarchy structure for the database team, and changes to the back-end and visualization for the website team. These changes are detailed in the following sections.

4.1 Simulator

Many of the parameters changed throughout the implementation process due to unforeseen limitations in both time available to run simulations and in what parameters could be feasibly changed using the gem5 standard library. Originally, there were around twice as many planned parameters, with many more values that were planned for simulation. However, each additional parameter multiplies the total number of simulations by its number of values, causing an exponential increase in simulation time as each parameter is added. Additionally, the limitations within the gem5 standard library made it unfeasibly difficult to use certain other parameters, as to do so major modifications would have to be made to the standard library to expose those variables.

The benchmark was originally going to be SPEC CPU 2006, however we changed to the 2017 version since that was the version that we had access to through our faculty mentor. Additionally, we had originally planned to run at least three of the benchmarks within SPEC, but again due to simulation time constraints we had to use only one of the benchmarks, 505.mcf_r. For similar reasons only one checkpoint within that benchmark was used.

4.2 Database

For the database, there was only one change made between the planning and implementation phases of the project. Originally, the database was designed to hold data in a mixed hierarchy data schema, where certain data points would be prioritized over others for ease of searchability when being called by the website. This proved to be unnecessary, as the website could easily parse whole dictionaries of data in the form of stored documents for the simulator outputs. Thus, the data outputs were stored as dictionaries, after being scraped for only needed data, and then were easily accessed by the website. This turned out to be just as, if not more, efficient than the initial plan, as the stored data for each run was not significant enough to have slow downs in access times.

4.3 Website

When initially creating the website, it was designed with a front-end and back-end server that was connected to the database. Given the simplicity of the query-based nature of the website, the

backend server was deemed unnecessary as the front-end server could query directly to the public database. Removing the back-end server drastically decreased resource overhead and total network complexity. This also simplifies the hosting process and cost by requiring only the front-end server to be hosted on AWS.

Originally, the website was designed so that students could submit a few parameters they wanted to keep static, and one they wanted to sweep over to observe how performance changes. However, it became apparent that graphing the higher dimensional graphs required to represent multi-variable states would become too complex and outside the capabilities of the graphing library used on this website. This feature was reduced to a static state who had one parameter that was swept over.

Due to the complexity of simulation, it was uncertain which parameters and metrics would be used by the end of the project. During implementation, it was decided that the website would be configurable such that it would recursively build out its inputs. This allows for rapid reconfiguration, so that if the entire structure of the simulation including the parameters and metrics it generates change, the website could be augmented to function with this new data in a matter of minutes.

5.0 TEST AND EVALUATION

The test and evaluation section provides ample detail on the processes used to detail correct system functionality during the design and implementation phases of the project. This includes the methods used, results found, analysis performed, and any recommendations or actions taken on behalf of those findings.

5.1 Method

The project is split into three main subsystems, consisting of the simulator, database and data management, and website. During the last semester, test plans were laid out in the System Design Report, and further updated at the start of the current semester.

5.1.1 Simulator

SPEC 2017 was chosen as the suite of benchmarks to run in the simulator. These benchmarks are too large to be run entirely within the simulator, and so each benchmark must be split into smaller chunks called intervals. Then, a representative sample of intervals called simpoints are chosen using SimPoint based on a basic block vector for each interval. A basic block is a chunk of instructions with exactly one entry point and one exit point, and an interval's basic block vector counts the number of times that each basic block within a program was entered during that interval. To test the process of simulating one of these simpoints for a single benchmark, first the 605.mcf_s benchmark was chosen as it subjectively seemed simpler than the other benchmarks. Next, that benchmark was configured and built for the target ISA and OS, which is x86 Linux. The benchmark executable was then run with a tool such as Pin or Valgrind to generate the basic block vectors, which were then fed into SimPoint to select the simpoints. The gem5 simulator was built and a script will be written to simulate the benchmark using a CPU model without accurate timing in syscall emulation mode. This initial simulation will create checkpoints, which are saved states of the simulation, for each simpoint. Finally, one of the checkpoints will be simulated with gem5 using a different script and run using a timing accurate model to conclude the test.

5.1.2 Database

In the System Design Report, database testing was outlined as including three key parts: access efficiency, correct data storage, and correct data reads.

Access efficiency has been optimized by storing data in full documents. Each run of the simulator produces a set of output data and the parameters used in the simulation, which is then stored into MongoDB in a document style form. By having the website access the requested data through a document read method, parameters can be easily searched for, found, and read. This has been tested extensively using sample data stored in the database and then accessing it from the frontend website code.

Correct data storage has also been tested, where Python code can be used to place sample outputs into MongoDB. These outputs can be checked for accuracy by using the MongoDB Atlas tool, where testers are able to view the stored data in their correct document style format.

Correct data reads have been tested, where the website frontend code is able to access the accurate stored sample data from MongoDB and correctly show it as an output in code.

5.1.3 Website

The website is to be tested iteratively and methodically by focusing on user experience such that the design of the website is friendly and intuitive to the user. This included questions such as how aspects of the parameters and performance metrics should be displayed, how should the user be able to manage these simulated runs of their systems, how should the user get around the website, etc.

In addition to the experience of the website, the website's functionality was tested with unit testing and active edge case testing throughout development in addition to methodical design practices which were iteratively improved through refactoring. This included connecting the user to the database, creating, fetching, updating, and destroying documents, as well as how the code should be rendered. Animations, sizing, local storage, and other components of the code that were not easily unit-testable were actively tested throughout production.

5.2 Results

For each subsystem, relevant testing has yielded varying results, including timing issues, incompatibilities, and errors as well as correct and efficient results. These results are given in more detail in the following sections.

5.2.1 Simulator

The benchmark was successfully built and run on a laptop natively to confirm that it worked. Although Pin was unable to generate basic block vectors for the benchmark as it was incompatible with the newer versions of the Linux kernel, Valgrind was successful in generating the vectors. SimPoint was successfully built, and it generated simpoints for the benchmark based on the basic block vectors from Valgrind. The gem5 simulator was built successfully and a simple test simulation was able to be run using system emulation mode. Furthermore, a test run of SPEC2017's gcc_r benchmark was started successfully in full system mode with a modified sample script. However, since this script does not utilize simpoints/checkpoints, and due to the

fact that gem5's full system mode is 10,000-100,000x slower than bare metal; this benchmark would take more than 130 days to finish simulating (best case scenario) [8].

$$10{,}000 * 20\text{min} / 60\text{ min} / 24\text{hr} = 138.888\ldots \text{ days}$$

Therefore, as determined before, it is indeed necessary to run benchmarks using the simpoint methodology. By using the simpoint methodology, one simulation run including a warm up phase is reduced down to under 10 minutes.

5.2.2 Database

For database storage of simulator outputs, document style data hierarchy was found to be the optimal solution for ease of access by the frontend website code. Initial testing for data correctness is shown below in Figure 3. The top example is from when the database was first created and being tested for data input, while the bottom example is more representative of the document style hierarchy that will be used in the final product. Data outputs were easily inserted in the database by Python code and correctly read by the website frontend code during testing.

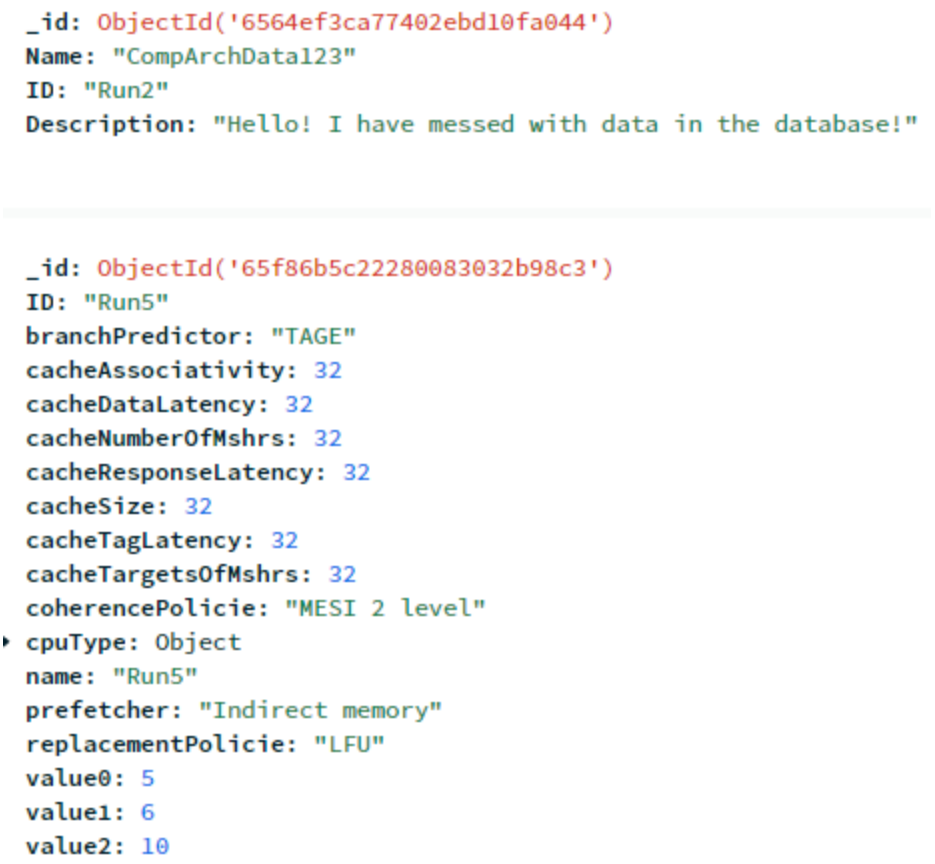

Figure 3. Data Hierarchy and Correctness Testing Example

5.2.3 Website

The website has been improving rapidly due to repeated and thorough testing and iteration of design. The code to connect, fetch, filter, and direct query database documents has grown more capable while reducing code length. The rendering code has been refactored to construct the input and graphing of the simulations recursively and through mapping so that there is very little repetition and clutter in the code. The website has maintained a very simple structure and has reduced in size and complexity over time as the backend server was eliminated from the process with our group instead opting to utilize MongoDB's Atlas database as a service which gives us direct query functionality without the need to run a second server. Authentication to our site is anonymous and requires no accounts, we have opted for the states of each user's respective accounts to be stored in local storage so that their last activity is locally accessible allowing for continuous work between sessions. Additionally, users can save, export, import, and delete simulation sets to organize their work more effectively.

5.3 Analysis

In the following section, we break down the results gathered from testing done for each subsystem, including the simulator, database, and website.

5.3.1 Simulator

All of the results up to the initial run of gem5 were completed, and so this portion of the test was a success. There is some concern over any potential differences in the way that Valgrind counts instructions vs gem5. For example, if Valgrind counts instructions within a system call, and gem5 does not, then this would cause the simpoints to be incorrect for use with gem5. This can be validated by making sure that total instruction count in gem5 matches or at least roughly matches the total count reported by Valgrind. Since the interval size was chosen to be 100 million instructions, a difference of less than 1 million should indicate that the intervals are not getting changed too significantly.

The issue which prevented the test from being completed before the writing of this report occurred when it was realized that an assumption about the SPEC benchmarks being single executables with no other dependencies was false. Most of the benchmarks must be passed input files via command line arguments, and that capability is not demonstrated by the example gem5 script, which can only take a single binary and run it without any command line arguments.

Another current issue is how to integrate simpoints into the script that is running gem5. The gem5 documentation is unclear about how to feed a pre generated set of simpoint files into an existing script such that it will take checkpoints dynamically.

5.3.2 Database

Testing showed successful results for correctly storing data, reading correct data to the website, and reading the data quickly. Of the available data hierarchies, the document style data hierarchy was found to be the best way to move forward with fast and efficient accesses for the website and thus, users.

5.3.3 Website

The website has improved dramatically through this iterative design and testing process. It maintains a low level of code length while increasing in quality. More functionality is continuing to be built out to maximize the capability of the continually refactored and improved code. The new recursive structure is capable of dynamically handling the input type of nested values depending on the value of previously selected inputs which was not possible with the previously static structure. This will allow for the user to dynamically assign filtered inputs for database queries. The user is able to quickly augment the parameters displayed as well as change how the graph looks.

The rest of the site is of high quality as it is highly dynamic and adjustable. A majority of its functionality can be augmented from changing configuration JSON objects in one file. This includes which values are displayed as well as how the inputs are presented. Ultimately, this will also include the output of the graphs.

6.0 TIME AND COST CONSIDERATIONS

For the simulator, database, and website, there were some setbacks in terms of expected project deadlines due to various factors. These time and cost considerations are discussed for each subsystem below at length.

6.1 Simulator

The gem5 simulator took considerably longer to learn how to use than was originally expected. Specifically, there was extremely limited documentation on the checkpoint creation and restoration processes, as well as how to customize various parameters the team wished to experiment on. As a result of this unfortunately several parameters were cut from the list as the way to customize them were never found. These problems held the team back several weeks behind schedule. Once the checkpoint generation script and the simulation script were created and tested, there was another unforeseen issue that caused delay. It was initially unclear how all of the simulations would be run in parallel; however, the GNU parallel tool was found to be capable of running all of the simulations. Once the tool was learned, progress on gathering simulation results continued smoothly.

6.2 Database

With simulator output data being held up in a lot of cases due to unforeseen roadblocks by the simulator team, the database was not able to receive and process data until later than expected. Although most of the output data management code had been written and tested, there was a significant gap in time before actual output data was received for processing. However, once data was received, the code was able to be run and within a few hours, the database had been populated, ready for use by the website. Apart from this setback, no other issues with the database in terms of time or cost were present.

6.3 Website

Due to the complexity of the data simulation, the website was unable to be built statically for a set of expected inputs which required a significant amount of flexible, dynamic design to handle any range of inputs. The result is a very extensible, quickly-modifiable site that can be changed to handle all types of data from a single config file. Fortunately, since the server utilizes only a frontend server, very little data transfer, and low query complexity, the incurred cost will likely sit well beneath the free tier of AWS and Mongo's DBaaS.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

In terms of safety considerations to our design, there are none that come to mind. The project is based entirely online with no personal or private information needed to access it. No bodily harm can come from any aspect of our project. In terms of ethical aspects, there is the potential issue of data posted on the website not allowing students to learn the correct information they might be looking for from the project, thus either invalidating the learning goals or worse, teaching the wrong things. We have done our best to make sure that all simulation data is accurate and is presented in an intuitive way to students so that any misunderstandings or misinformation are minimized, if not eliminated. We aim to teach students using our website, and do so correctly and with valid information and data.

8.0 RECOMMENDATIONS

Past the current iteration of the project, multiple recommendations can be made for further improvements, changes, or even experiments to run. For the simulator, more parameter and benchmark options could be used. For the database, a different document hierarchy or database service could be explored. Lastly, for the website, different visual aids might be possible or more helpful for students.

8.1 Simulator

More of the SPEC 2017 benchmarks could be used to broaden the results and make them more representative of actual workloads. This would significantly increase simulation time so a more powerful computer would be needed to do this. Access to a supercomputer would allow for more simulations to be run in parallel and would therefore enable such an improvement. More parameters could also be added; however, doing so would increase simulation time exponentially such that many of the combinations would need to be ignored. This could be done by creating a limited number of basic system configurations. Then additional parameters could be varied while the rest of the system is in one of the basic configurations. This would require the website interface to be modified to remove the ability to choose any arbitrary system configuration and instead choose some number of parameters to vary for a selected basic configuration.

8.2 Database

For the database, some different explorations of data hierarchies could be explored for future iterations of the project. In our current version, each simulation run is stored as a whole document, with each data point having no importance over another, simply stored in a dictionary. For future versions, slimming of data stored or even having data stored under sections could be possible, thus making searches and accesses faster and possibly more efficient from the website side. In addition, a different database hosting service could be used. If increased data storage is needed, MongoDB will no longer be able to work for the current version, and another database or a paid version will need to be explored.

8.3 Website

For further improvements upon the current system, detailed guides and lessons can be added to help the user understand both the tool and the systems taught more effectively. The website's components easily have this capability and it would take few technical changes to achieve. Additionally, more complex visual aids could be added with post-processing on the data inside the database such as percentiles, direct comparisons between minimum and maximums, and other processes which are more difficult to calculate or query than direct parameter searches.

9.0 CONCLUSION

Overall, the Computer Architecture Explorer was split into three separate main subsystems: the simulator, the database, and the website. The gem5 simulator ran benchmark code to generate performance data for the computer architecture using various parameters, the performance metrics and specific settings were stored in a database, and lastly the website allows students to pull different scenarios from the database and view them in a user interface. The gem5 simulator was chosen over a custom design, MongoDB was used for the database service with panda and numpy for data crunching and sorting, and the website uses React, NodeJS, Chart.js, Realm-web, Styled-Component, and AWS.

Several problems were encountered throughout development, the most prominent and detrimental of which was the unexpected lack of documentation for the gem5 simulator's more in depth functionality and characteristics. This has led to the unfortunate decision to cut back several important parameters within the initial list that was to be explored, and thus negatively impacted the core functionality of the website by an undesirably large amount. However, that is not to say the project is incomplete, many interesting interactions are waiting to be explored by users within the dataset produced by the remaining list of parameters. Overall the Computer Architecture Explorer website should be a useful and fun way for future students and all other interested parties to explore and understand the various intricacies and nuances in a computer's architecture.

REFERENCES

[1] *SPEC CPU*. (2017), SPEC. Accessed: Mar. 29, 2024. [Online]. Available: https://www.spec.org/cpu2017/

[2] S. Song, Q. Wu, S. Flolid, et. all, "Experiments with SPEC CPU 2017: Similarity, Balance, Phase Behavior, and SimPoints," Laboratory for Computer Architecture, Univ. Texas, Austin. TR-180515-01.

[3] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, "Automatically Characterizing Large Scale Program Behavior," Department of Computer Science and Engineering, Univ. California, San Diego.

[4] *Valgrind*. (2023). Accessed: Mar. 29, 2024. [Online]. Available: https://valgrind.org/docs/manual/bbv-manual.html

[5] *SimPoint*. (2006), UC San Diego. Accessed: Mar. 29, 2024. [Online]. Available: https://cseweb.ucsd.edu/~calder/simpoint/

[6] *gem5*. (2024). Accessed: Mar. 29, 2024. [Online]. Available: https://github.com/gem5/gem5/

[7] *Parallel.* (2010). Accessed: Apr. 21, 2024. [Online]. Available: https://www.gnu.org/software/parallel/

[8] J. Lowe-Power. "SPEC CPU 2017 taking days to simulate." The Mail Archive. https://www.mail-archive.com/gem5-users@gem5.org/msg20961.html (accessed Mar. 29. 2024)

APPENDIX A – CHECKPOINT GENERATION SCRIPT

APPENDIX A – CHECKPOINT GENERATION SCRIPT

```
import argparse
from pathlib import Path

from gem5.components.boards.simple_board import SimpleBoard
from gem5.components.cachehierarchies.classic.no_cache import NoCache
from gem5.components.memory.single_channel import SingleChannelDDR3_1600
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.isas import ISA
from gem5.resources.resource import (
    BinaryResource,
    SimpointResource,
)
from gem5.resources.workload import Workload
from gem5.simulate.exit_event import ExitEvent
from gem5.simulate.exit_event_generators import save_checkpoint_generator
from gem5.simulate.simulator import Simulator
from gem5.utils.requires import requires

requires(isa_required=ISA.X86)

parser = argparse.ArgumentParser(
    description="Generates checkpoints from simpoints"
)

parser.add_argument(
        "--binary",
        type=str,
        required=True,
        help="The binary to simulate.",
)

parser.add_argument(
        "--arguments",
        type=str,
        required=False,
        default="",
        help="The arguments to the binary for simulation."
)

parser.add_argument(
        "--simpoints",
        type=str,
        required=True,
        help="The simpoint file.",
)

parser.add_argument(
        "--weights",
        type=str,
        required=True,
        help="The weights file.",
)

parser.add_argument(
        "--interval",
        type=int,
        required=False,
        default=100_000_000,
        help="The size of an interval in instructions.",
)

parser.add_argument(
        "--warmup",
```

```
        type=int,
        required=False,
        default=1_000_000,
        help="The size of the warmup in instructions.",
)

parser.add_argument(
    "--checkpoint-dir",
    type=str,
    required=True,
    help="The directory to store the checkpoints.",
)

args = parser.parse_args()

cache_hierarchy = NoCache()

memory = SingleChannelDDR3_1600(size="2GB")

processor = SimpleProcessor(
    cpu_type=CPUTypes.ATOMIC,
    isa=ISA.X86,
    num_cores=1,
)

board = SimpleBoard(
    clk_freq="3GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)

def parse_simpoint_file(path):
        with open(path, "r") as file:
                return [line.split()[0] for line in file.readlines()]

simpoints = [int(e) for e in parse_simpoint_file(args.simpoints)]
weights = [float(e) for e in parse_simpoint_file(args.weights)]

board.set_se_simpoint_workload(
    binary=BinaryResource(
        local_path=args.binary,
        architecture=ISA.X86,
    ),
    arguments=args.arguments.split(),
    simpoint=SimpointResource(
        simpoint_interval=args.interval,
        simpoint_list=simpoints,
        weight_list=weights,
        warmup_interval=args.warmup,
    ),
)

dir = Path(args.checkpoint_dir)

simulator = Simulator(
    board=board,
    on_exit_event={
        # using the SimPoints event generator in the standard library to take
        # checkpoints
        ExitEvent.SIMPOINT_BEGIN: save_checkpoint_generator(dir)
    },
)

simulator.run()
```

APPENDIX B – SIMULATION SCRIPT

APPENDIX B – SIMULATION SCRIPT

```
import argparse
from pathlib import Path

from m5.stats import (
        dump,
        reset,
)

from m5.objects.BranchPredictor import (
        LocalBP,
        BiModeBP,
        TAGE,
        MultiperspectivePerceptron8KB,
)

from gem5.components.boards.simple_board import SimpleBoard
from gem5.components.cachehierarchies.classic.private_l1_private_l2_cache_hierarchy import (
        PrivateL1PrivateL2CacheHierarchy,
)
from gem5.components.memory import DualChannelDDR4_2400
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.isas import ISA
from gem5.resources.resource import (
        BinaryResource,
        SimpointResource,
        CheckpointResource,
)
from gem5.resources.workload import Workload
from gem5.simulate.exit_event import ExitEvent
from gem5.simulate.simulator import Simulator
from gem5.utils.requires import requires

requires(isa_required=ISA.X86)

parser = argparse.ArgumentParser(
        description="Simulates one checkpoint"
)

parser.add_argument(
        "--binary",
        type=str,
        required=True,
        help="The binary to simulate.",
)

parser.add_argument(
        "--arguments",
        type=str,
        required=False,
        default="",
        help="The arguments to the binary for simulation."
)

parser.add_argument(
        "--simpoints",
        type=str,
        required=True,
        help="The simpoint file.",
)

parser.add_argument(
        "--weights",
        type=str,
        required=True,
```

```
        help="The weights file.",
)

parser.add_argument(
        "--interval",
        type=int,
        required=False,
        default=100_000_000,
        help="The size of an interval in instructions.",
)

parser.add_argument(
        "--warmup",
        type=int,
        required=False,
        default=1_000_000,
        help="The size of the warmup in instructions.",
)

parser.add_argument(
        "--checkpoint",
        type=str,
        required=True,
        help="The path to the checkpoint. Expects a directory with pmem file and cpt file.",
)

parser.add_argument(
        "--l1size",
        type=str,
        required=False,
        default="32kB",
        help="Size of the L1 data cache.",
)

parser.add_argument(
        "--l2size",
        type=str,
        required=False,
        default="512kB",
        help="Size of the L2 cache.",
)

parser.add_argument(
        "--pred",
        type=str,
        required=False,
        default="perceptron",
        help="Branch Predictor. Can be 'local', 'bimode', 'tage', or 'perceptron'.",
)

parser.add_argument(
        "--robsize",
        type=int,
        required=False,
        default=128,
        help="Size of the reorder buffer.",
)

parser.add_argument(
        "--regcount",
        type=int,
        required=False,
        default=128,
        help="Number of physical registers.",
)

parser.add_argument(
```

```
        "--alucount",
        type=int,
        required=False,
        default=4,
        help="Number of ALUs",
)

parser.add_argument(
        "--mdcount",
        type=int,
        required=False,
        default=4,
        help="Number of multiplier/divider units.",
)

args = parser.parse_args()

# The cache hierarchy can be different from the cache hierarchy used in taking
# the checkpoints
cache_hierarchy = PrivateL1PrivateL2CacheHierarchy(
        l1d_size=args.l1size,
        l1i_size="64kB",
        l2_size=args.l2size,
)

# The memory structure can be different from the memory structure used in
# taking the checkpoints, but the size of the memory must be maintained
memory = DualChannelDDR4_2400(size="2GB")

processor = SimpleProcessor(
        cpu_type=CPUTypes.O3,
        isa=ISA.X86,
        num_cores=1,
)

def pred_arg_to_obj(arg):
        if arg == "local":
                return LocalBP()
        elif arg == "bimode":
                return BiModeBP()
        elif arg == "tage":
                return TAGE()
        elif arg == "perceptron":
                return MultiperspectivePerceptron8KB()
        else:
                assert False

processor.cores[0].core.branchPred.value = pred_arg_to_obj(args.pred)
processor.cores[0].core.numROBEntries.value = args.robsize
processor.cores[0].core.numPhysIntRegs.value = args.regcount
processor.cores[0].core.fuPool.FUList[0].count = args.alucount
processor.cores[0].core.fuPool.FUList[1].count = args.mdcount

board = SimpleBoard(
        clk_freq="3GHz",
        processor=processor,
        memory=memory,
        cache_hierarchy=cache_hierarchy,
)

def parse_simpoint_file(path):
        with open(path, "r") as file:
                return [line.split()[0] for line in file.readlines()]

simpoints = [int(e) for e in parse_simpoint_file(args.simpoints)]
weights = [float(e) for e in parse_simpoint_file(args.weights)]
```

```
board.set_se_simpoint_workload(
        binary=BinaryResource(
                local_path=args.binary,
                architecture=ISA.X86,
        ),
        arguments=args.arguments.split(),
        simpoint=SimpointResource(
                simpoint_interval=args.interval,
                simpoint_list=simpoints,
                weight_list=weights,
                warmup_interval=args.warmup,
        ),
        checkpoint=CheckpointResource(
                local_path=args.checkpoint,
        ),
)

def max_inst():
        warmed_up = False
        while True:
                if warmed_up:
                        print("end of SimPoint interval")
                        yield True
                else:
                        print("end of warmup, starting to simulate SimPoint")
                        warmed_up = True
                        # Schedule a MAX_INSTS exit event during the simulation
                        simulator.schedule_max_insts(
                                board.get_simpoint().get_simpoint_interval()
                        )
                        #dump()
                        reset()
                        yield False


simulator = Simulator(
        board=board,
        on_exit_event={ExitEvent.MAX_INSTS: max_inst()},
)

# Find warmup interval for the supplied checkpoint
def checkpoint_inst_count(checkpoint):
        with open(checkpoint + "/m5.cpt", "r") as file:
                for line in file.readlines():
                        parsed_line = line.split("=")
                        if parsed_line[0] == "instCnt":
                                return int(parsed_line[1])

def simpoint_idx(checkpoint):
        checkpoint_start_inst = checkpoint_inst_count(checkpoint)
        simpoint_start_insts = board.get_simpoint().get_simpoint_start_insts()
        min_diff = -1
        min_idx = -1
        for idx, simpoint_start_inst in enumerate(simpoint_start_insts):
                diff = abs(simpoint_start_inst - checkpoint_start_inst)
                if min_diff == -1 or diff < min_diff:
                        min_diff = diff
                        min_idx = idx
        assert min_idx > 0 and min_diff > 0 and min_diff < args.interval / 2
        return min_idx

simulator.schedule_max_insts(
        board.get_simpoint().get_warmup_list()[simpoint_idx(args.checkpoint)]
)
simulator
```