Looking into the Black Box: Monitoring Computer Architecture Simulations in Real-Time with AkitaRTM

Ali Mosallaei
Electrical and Computer Engineering Dept.
University of Michigan
Ann Arbor, USA
alimos@umich.edu

Katherine E. Isaacs
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, USA
kisaacs@sci.utah.edu

Yifan Sun
Dept. of Computer Science
William & Mary
Williamsburg, USA
ysun25@wm.edu

***Abstract*—Computer architecture simulators are essential for validating novel chip designs. However, they often provide little transparency during execution. This opaqueness limits the ability of users to identify issues during the simulation, leading to both wasted computational and human time. We address these issues by providing an intuitive user experience during simulation execution. Particularly, we reveal the status of executions and allow users to control the execution of a computer architecture simulator through AkitaRTM, an interactive web-based tool for real-time monitoring of computer architecture simulations. We based its design on the design workflow inefficiencies experienced by computer architects when using simulations. We demonstrate AkitaRTM's utility through two case studies, the second leading to a patch in the simulator. Additionally, we conducted a user study with computer architects, aiming to validate AkitaRTM. We found that, in addition to solving the observed problems, AkitaRTM also provided an educational benefit by making simulators more transparent to users. Based on these findings, we reflect upon the design of AkitaRTM and provide guidance for future human-centered tools in this space.**

***Index Terms*—computer architecture simulation, real-time monitoring, debugging, user study**

I. Introduction

Computer architecture simulators play a pivotal role in developing and optimizing modern computer chips and hardware [32]. These tools provide a virtual environment for architects and developers to analyze, design, and test various hardware configurations and instruction sets without the need for physical prototypes. By facilitating detailed performance analysis and design space exploration, these simulators enable more efficient and cost-effective development cycles.

A major problem of computer architecture simulators, as discussed in prior work across the community [5], [26], [27], [31], [44], is slow simulation speed. As cycle-level simulators are usually millions of times slower than real hardware, simulating one second of execution can easily last days, weeks, or years [7], [39], [44]. This slowdown factor significantly limits the capability of computer architecture simulators, slowing down scientific innovation.

The most of the work was done while Ali Mosallaei was a research intern at William & Mary.

Existing methodologies mainly focus on "finishing the useful simulations faster." Research along these lines includes parallel simulation [34], [39] and sampling-based methods [5], [10], [27], [31]. However, little work solves the problem of how we can terminate problematic simulations early. We believe developing a systematic method that enables "fail early, fail fast" simulation can significantly accelerate the pace of computer architecture research by reducing researchers' wait time and freeing their computational resources for more fruitful investigations.

We consider the problem of not being able to "fail early, fail fast" to be rooted in the limited interactivity provided by computer architecture simulators. Computer architecture simulators typically work in a manner where users provide some input, wait for a long time (possibly days to weeks) for results to be calculated, and collect and analyze the results. It is inconvenient at best and impossible at worst for users to examine or control the simulation while it is running. These long cycles of waiting and analysis lead to excessively long turnaround times in research and development and can decelerate scientific discovery and engineering innovation.

One way to shorten this development cycle is to allow users to analyze partial simulation results while it is running. Yet, existing methods that enable user interaction with simulators are often constrained. These applications usually dump large amounts of information to the command-line interface [8], [12], [21], causing problems of having too much and, at times, too little useful information for the architect to digest in one sitting. As simulation developers cannot predict what information users need at any one time, it may be tempting for the developer to dump unnecessary data into the terminal, overwhelming users. However, even if the amount of data is large, the output may still not include the information required by the user. Solving this problem requires tools to provide a method for users to look into the metaphorical "black box" of the simulation at run time.

We pioneer the work of developing a formal real-time monitoring tool by implementing it as a "plugin" of a popular GPU simulator, MGPUSim [39]. MGPUSim is an AMD-based

GPU simulator that supports multi-GPU simulations. We selected MGPUSim because it provides a friendly programming interface and a drop-in plugin system that allows us to easily build a modular monitoring tool. However, the lessons learned from building this tool are not constrained to MGPUSim; they are generally applicable to a wide range of computer architecture simulators or even simulation tools in general.

The design and development of AkitaRTM has been a multi-year process, primarily influenced by interactions with other computer architecture researchers to understand their pain points. Based on frequent, informal interviews with computer architects, we carefully chose and designed features within AkitaRTM. After developing each feature, we collect feedback and additional user needs for new features, as well as edits to already implemented features. This continual process, being the basis of success for this tool, is iterative and based on challenges that continue to evolve in hardware development. AkitaRTM is both developed and released in an open-sourced manner under the permissive MIT license. The link will be revealed after the review process.

We validate the design of AkitaRTM through case studies (section V) and a user study with computer architects (section VI). In the case studies, we show how AkitaRTM helps conduct a performance analysis on both the simulation and the hardware and how it helped debug a hang in the simulator. To further understand how AkitaRTM can help computer architects, we perform a user study with six computer architecture researchers. We then discuss and reflect on these studies and the design, generating guidelines for future designs of monitoring tools (section VIII).

Our primary contributions are:

- summarizing the needs in computer architecture simulator monitoring (section III and section VIII)
- validating the implementation of AkitaRTM, a monitoring tool supporting those needs (section IV), and
- advocating for applying human-centered design methods in computer architecture research.

II. Background and Related Work

MGPUSim [39] is a GPU simulator that simulates OpenCL workloads running on AMD GPUs. In MGPUSim, groups of hardware circuits (e.g., computing cores, cache units) are organized as *Components*, which are specially defined structs within the program. Components can only communicate by exchanging messages. The isolation of components suggests that we can develop interfaces that allow users to monitor individual components.

MGPUSim's choice of the Go programming language necessitated that AkitaRTM also be developed in Go. Our experience with Go has highlighted several significant advantages over more traditional languages such as C or C++. Firstly, Go's design is inherently suited for cloud services, making the development of web servers using Go a seamless process. Additionally, Go incorporates *pprof* [16] as an inbuilt library, enabling efficient profiling of the program while it runs. Lastly, Go's straightforward approach to integrating third-party libraries greatly simplifies the process of implementing features like serialization of simulation status, which reduces the complexity typically associated with such tasks.

Apart from MGPUSim, extensive research has been conducted on other GPU simulators, such as GPGPUSim [7] (including its newer iteration, AccelSim [22]), the gem5 GPU model [18], NVArchSim [44], MacSim [23], and Multi2Sim [43]. In the broader realm of general computer architecture research, there is ongoing development and use of simulators for CPUs [10], memory systems [24], [45], [47], and domain-specific accelerators [30], [33]. Among these simulator projects, a few, such as gem5 [8], [28] and SST [34], show promise in creating frameworks that integrate simulator models developed by the community. Despite the proliferation of projects in computer architecture simulators, a notable gap exists in providing solutions for in-situ (during simulation execution) analysis.

Simulator researchers have reduced simulator execution time through methods such as parallel simulation and sampling-based simulation. Parallel computer architecture simulation [11], [34], [35], [39] has been demonstrated to be able to improve simulation performance from a few times to hundreds of times. Meanwhile, sampling-based simulation [6], [10], [27], [31], [36] usually can yield a much higher speedup by skipping repeated simulation segments, but also suffer risks of having higher error. Unlike these solutions, AkitaRTM employs a human-in-the-loop method that attempts to finish problematic simulations early rather than accelerating them. AkitaRTM can be combined with parallel and sampling-based methods to further reduce turnaround time.

Most existing simulators [8], [34], [43] have some type of capability of showing real-time data to users, e.g., dumping data periodically. However, these methods are usually ad-hoc and cannot meet users' needs. Expanding our search to related fields, gate-level simulators [38] usually allow users to monitor the logical levels of specific nodes in the circuit. The high-performance computing (HPC) domain also commonly engages real-time monitoring tools [1], [37], focusing on monitoring the performance of critical HPC applications. We consider multiple lessons from these tools: 1) aggregating data from different sources is critical; 2) interactivity can significantly enhance the capability of monitoring tools; and 3) monitoring tools are highly domain-specific, meaning existing tools cannot be directly applied to the computer architecture domain. The gap in effective monitoring tools within the computer architecture domain, coupled with the potential benefits identified from related fields, underscores the need for a dedicated tool.

III. Determining Monitoring Needs

Writing all of the possible raw data available during a simulation does not provide architects with the information and insights they need to steer or prematurely end a simulation. Instead, it is necessary to curate which data is provided to users and to transform the data into a format that architects can use. To identify data and information needs, we iteratively considered and prioritized features needed during monitoring. We describe the five common, highest-priority features identified below. These features guided our design and should be considered for other simulation monitoring tools.

To identify these features, we first started with our own needs based on firsthand experience in simulator development and use. As we gained traction among a circle of close collaborators, we documented needs and feedback from them through informal interviews, using our evolving prototype as a probe to help elicit and concretize the tasks they perform while using simulators and the inefficiencies, limitations, and frustrations in their existing workflows.

We frame the identified features as *tasks*, labeled T1 through T5. It should be noted that our focus is not on the conventional use of computer architecture simulators, i.e., analyzing and comparing performance through standard simulation methods. Instead, we consider the full life cycle of developing and using computer architecture simulators. This exploration underscores their importance in the broader context of computer architecture development.

T1: Predicting how long a simulation will take. How long any given simulation will take is difficult to predict. Users face a dilemma in terms of whether to wait for a simulation to finish or abort it and spend the resources elsewhere. They need assistance in determining simulation progress so they can accurately predict and decide.

This problem is exacerbated by the fact that simulations greatly vary in the time to finish executing depending on the hardware and software being simulated [5], [44]. The range of this variance can be from seconds to centuries. In the worst case, they face the Halting Problem, not knowing if the simulation will eventually finish [29] due to simulator bugs. As users modify simulators in an ad-hoc way, they will seldom be confident about whether a long-running simulation is free from hangs, another problem described later in this section.

T2: Monitoring a simulator's resource utilization. Our informal interviews revealed that computer architecture researchers often use command line tools such as *top* to monitor CPU and memory utilization when they start a batch of simulations. This monitoring is crucial for checking a simulation's health. For example, if simulations running simultaneously on a single computer surpass the physical memory limit, they might experience unnecessary performance slowdowns. Similarly, if a simulation shows unusually low resource usage, it could be an indication of a problem, like a simulation hang or the simulation blocking on disk IO.

We found that general system activity monitoring tools like *top* fall short of meeting these specific requirements. Their generic design is not tailored for detailed monitoring in this context. For example, when several simulations are running on one computer, each involving multiple threads, it becomes exceedingly challenging to discern the memory consumption of each individual simulation.

T3: Identifying causes of simulation hangs. Hanging is a common simulator bug where all the hardware components under simulation have no way to make forward progress due to a deadlock. For example, when two simulation components wait for messages from each other, they cannot move on.

Addressing hanging bugs presents a significant challenge with conventional break-line debugging techniques. One key difficulty is that these bugs can originate from any part of the simulated hardware, necessitating a thorough understanding of the hardware configuration to pinpoint the error's location. Furthermore, traditional break-line debuggers are designed to operate on actively running programs; however, a hanging simulation ceases code execution, thus rendering these debuggers ineffective. This complexity often leaves developers struggling to identify the specific line of code responsible for the hang. Resolving such an issue can demand an extensive investment of time, ranging from two to three full working days to even weeks. Consequently, there is a pressing demand among architects for a tool that can streamline and simplify this troubleshooting process.

T4: Profiling simulation performance. As suggested by T1-T3, the performance of the simulator greatly affects the research and development workflow of computer architects. When adding features to the simulator, they might accidentally also introduce performance bugs. When they notice their simulator running slowly, they often use the command line tool `pprof` [16] to collect timing information about the various aspects of the simulator. From our interviews and experience, we note that invoking `pprof` in this way typically requires re-familiarizing themselves with `pprof` and looking up relevant commands and input parameters. This added chore, while not particularly difficult, is disruptive to their workflow enough that some avoid profiling as long as they can. This hesitancy to interrupt their workflow and re-familiarize themselves with an external tool can lead to performance issues in the long run. When the computer architect does the performance analysis, the feature causing the problem may already be deeply integrated into the simulator and hard to change. Computer architects developing simulators need lighter-weight access to simulation performance data.

T5: Analyzing hardware performance bottlenecks. The ultimate goal of using a computer architecture simulator is to identify and understand hardware performance bottlenecks [40]. Existing tools [40] require complex, labor-intensive, and post hoc analysis to pinpoint bottlenecks. Real-time monitoring offers an advantage by providing access to more detailed data in the moment, which is not feasible for full-trace recordings due to their prohibitive storage demands. However, real-time monitoring is limited to analyzing data from a brief period preceding the current moment. This disparity in data availability necessitates the development of new views and tools. Once the users find a performance bottleneck, they may change hardware parameters to test if the bottlenecks persist. Such a workflow could reduce turn-around times by providing quick feedback.

A key point to consider is that users often initiate hundreds to thousands of simulations, each with varying configurations and benchmark combinations, raising questions about the

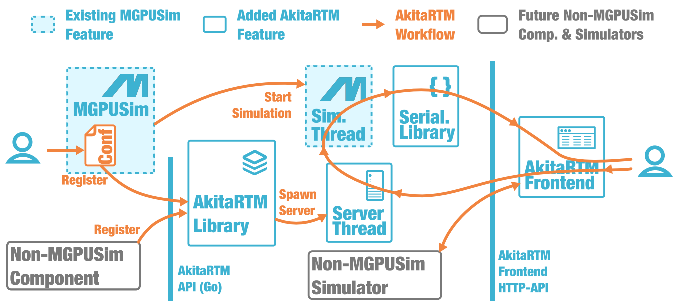


Fig. 1. An overview of the software components involved in running AkitaRTM. We highlight the standard APIs provided by AkitaRTM, which support new hardware components developed by users or even simulators other than MGPUSim.

feasibility of problem identification. However, this concern is mitigated by several factors: 1) Our focus primarily lies on enhancing the development phase, where the number of simulations executed is relatively limited. 2) The tool's user-friendly design should not require the users to spend a long time on each simulation. If the users check a subset of simulations more likely to suffer from problems, they will likely find common problems. 3) Should users be unable to detect problems early, the tool's minimal performance overhead ensures that reverting to traditional execution times will not impose a significant additional burden on the users.

IV. Design of AkitaRTM

We design AkitaRTM (Figure 2) to enhance the run-time interactivity of computer architecture simulators based on the tasks we identified previously. The resulting design is a multi-view web-based, real-time monitoring interface.

A. Technical Overview

An overview of the components and their interactions is shown in Figure 1. Central to AkitaRTM is a plugin (AkitaRTM library) to MGPUSim. To provide a configuration to MGPUSim, the user will need to invoke the AkitaRTM library to register hardware components under simulation. In this process, AkitaRTM also provides APIs for users to define the progress bars they want to show in the interface. By default, we show the progress of GPU kernels in terms of how many blocks have completed execution.

Upon initiating an MGPUSim program, AkitaRTM activates a server thread (backend) that is designed to handle incoming HTTP requests, effectively transforming any MGPUSim simulation into a web server. At the beginning of an MGPUSim simulation, a URL is displayed on the terminal, enabling users to easily access the server by clicking the provided link. This server is responsible for delivering the static webpage files (frontend) that users can navigate using their web browser. The frontend, in turn, retrieves data from the backend via HTTP requests. The backend of this system is developed in Go, while the frontend utilizes HTML, CSS, and TypeScript. Leveraging standard web technologies for the frontend allows AkitaRTM to facilitate remote monitoring and collaborative efforts.

For simulation monitoring requests generated from a user's interaction with the frontend, the server thread will take a snapshot of the corresponding component of the running simulation and serialize the component information to the frontend. The backend also provides auxiliary APIs that can, for example, query current simulation time, profile simulation, pause/resume simulation, update progress bars, and list buffer (hardware buffers under simulation) levels.

B. API Design

AkitaRTM provides two sets of standard APIs (see Figure 1), the Go API for adding new simulator components and the HTTP API to allow plugging in other simulators.

The Go API is small and lightweight. The `RegisterComponent` function starts the monitoring of a component. It uses reflection to discover buffers (for the bottleneck analysis) and fields (for simulation monitoring) of these components. Reflection eliminates the need to modify existing code and for users to manually select fields to monitor. Moreover, as the interface design of AkitaRTM highlights generality, adding a new component does not require designing a new view. Other functions include `RegisterEngine`, to link the simulation engine that manages simulation progress, and `{Create|Update|Destroy}ProgressBar`, to supply data for progress bars.

Simulators written in another language can still use AkitaRTM by implementing the Go API to call the HTTP API. Implementing the Go API requires only 12 functions, as described above. To use the HTTP API, developers need to integrate a web server that handles HTTP requests and can serialize the field values. Simulators written in C or C++ have mature web-serving [14] and JSON serialization [17], [19] libraries available to aid in this task. Other requirements, such as pausing/restarting simulation, reporting resource utilization, and updating progress bars, should be relatively trivial to implement. Since most simulators [8], [34] use a "Component-Port" paradigm, there is no need for a major paradigm change, and hence, the technical difficulty and the required work are manageable.

C. View Design

AkitaRTM is composed of several views as a dashboard to support the multiple tasks we identified. We defined the layout of the interface to be semi-flexible: Most regions in the interface serve predefined purposes, with the right side column being reconfigurable to show different content. The dividers can be draggable so that users can resize the regions.

1) Simulation Monitoring Views: The following views were designed to support monitoring of the simulation's health.

Resource utilization monitoring. As identified in task **T2**, computer architects frequently check CPU and memory usage as an indicator of how well the simulation is running. AkitaRTM displays this information directly (Figure 2 A), removing the need for additional user steps like running an external program like `top` and searching through all the

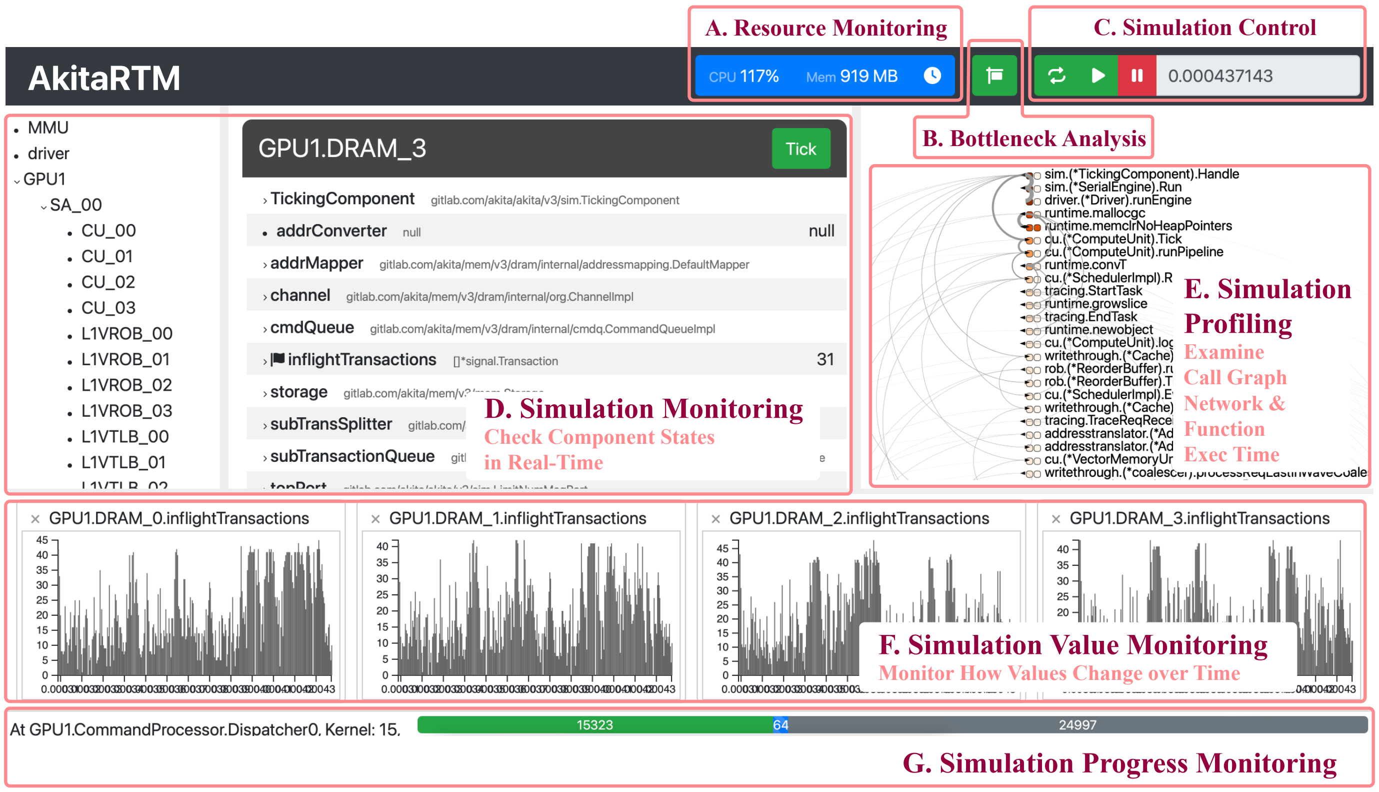

A. Resource Monitoring
B. Bottleneck Analysis
C. Simulation Control
AkitaRTM
CPU 117%
Mem 919 MB
0.000437143
D. Simulation Monitoring
Check Component States in Real-Time
E. Simulation Profiling
Examine Call Graph Network & Function Exec Time
F. Simulation Value Monitoring
Monitor How Values Change over Time
G. Simulation Progress Monitoring
GPU1.DRAM_3
Tick
GPU1.DRAM_0.inflightTransactions
GPU1.DRAM_1.inflightTransactions
GPU1.DRAM_2.inflightTransactions
GPU1.DRAM_3.inflightTransactions
At GPU1.CommandProcessor.Dispatcher0, Kernel: 15,
15323
64
24997

Fig. 2. Screenshot of AkitaRTM with labels for each major view. Other than the A. Resource Monitoring and E. Simulator Profiling, all views focus on examining the properties of the system under simulation.

processes running on a system. This view is always present and self-refreshing, permitting real-time monitoring.

Simulation controls. The simulation controls (Figure 2 C) allow computer architects to pause the simulation to inspect other views or to simply release CPU resources to a different simulation without aborting the current one.

The controls also show the progress in terms of simulation time. Should this value stop updating, it may indicate a hang (task **T3**). The speed at which it updates also indicates simulation performance in the absence of hangs (task **T4**) and can be used to estimate how long the simulation will take in real time (task **T1**).

Simulation profiling. The functionality of profiling the simulator itself (not the workload under simulation) is built-in to AkitaRTM (task **T4**).

Profiling data is captured using pprof [16] at a configurable interval at the granularity of function calls. The top-N (N is decided by pprof) functions that take the longest execution time are sent to the webpage to be visualized. These functions are then shown in the right panel (Figure 2 E) with arcs showing their calling dependencies. We choose a vertical arc diagram [25] to promote the visual search of time-consuming functions (red) over the topology of function calls. Identifying the most time-consuming functions is the most frequent sub-task for performance debugging. Following function calls can be helpful, but not always.

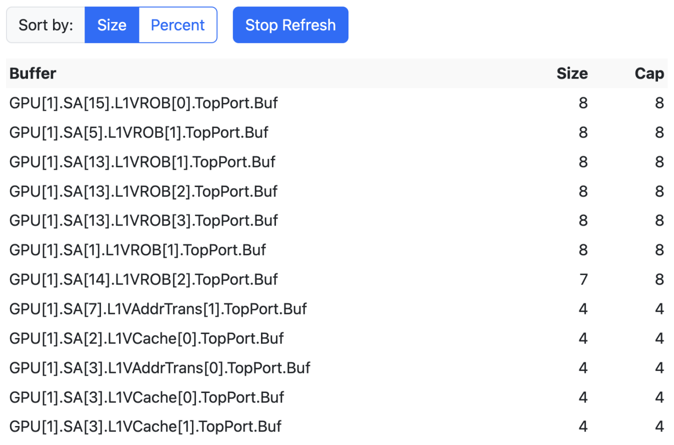
Each row in the diagram is one function. Two color-coded squares are displayed on the left of the function, showing its "self time" (time spent in only itself) and "total time" (all time taken, including by calls to other functions). These two properties are standard for performance profiling.

Arrows represent which function calls another, with arrow thickness representing time spent. Thus, the most time-consuming calls, which are most interesting to performance, are the most salient.

Simulation progress monitoring. At the bottom of the AkitaRTM interface is a region for progress bars, supporting task **T1**. Each bar is labeled (left) and has three segments, green, blue, and gray, which represent finished, currently executing, and not started tasks, respectively. For example, we show the number of thread blocks that are completed/ongoing/not-started to represent the progress of the kernel. Developers can call AkiteRTM's API to specify other progress bars (e.g., number of algorithm iterations and number of bytes copied in a memory copy operation). So far, we always require all three categories (completed/ongoing/no-started) of task counts. We will add support for unpredictable counts of not-started tasks when we observe the need.

2) Hardware Performance Analysis View: In addition to the views focusing on the simulation health, AkitaRTM supports the initial analysis of the simulation output.

Simulation status monitoring. Beyond the initial health checks of CPU and memory usage, AkitaRTM allows architects to dive deeper into the status of simulation components



Buffer	Size	Cap
GPU[1].SA[15].L1VROB[0].TopPort.Buf	8	8
GPU[1].SA[5].L1VROB[1].TopPort.Buf	8	8
GPU[1].SA[13].L1VROB[1].TopPort.Buf	8	8
GPU[1].SA[13].L1VROB[2].TopPort.Buf	8	8
GPU[1].SA[13].L1VROB[3].TopPort.Buf	8	8
GPU[1].SA[1].L1VROB[1].TopPort.Buf	8	8
GPU[1].SA[14].L1VROB[2].TopPort.Buf	7	8
GPU[1].SA[7].L1VAddrTrans[1].TopPort.Buf	4	4
GPU[1].SA[2].L1VCache[0].TopPort.Buf	4	4
GPU[1].SA[3].L1VAddrTrans[0].TopPort.Buf	4	4
GPU[1].SA[3].L1VCache[0].TopPort.Buf	4	4
GPU[1].SA[3].L1VCache[1].TopPort.Buf	4	4

Fig. 3. Showing the buffer analyzer as a table of the most occupied buffers. If the "flag" button is clicked, this table replaces the profiling visualization in the right panel of the interface shown in Figure 2 E. In this example, the Level 1 Cache's Reorder Buffer (L1VROB) is likely to be related to the performance bottleneck.

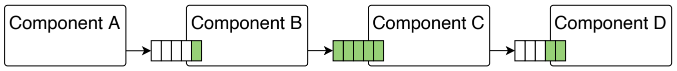


Fig. 4. A simplified demonstration showing why buffer fullness is an easy indicator of performance bottleneck. In this example, Component C is likely to be the performance bottleneck.

and even variables as well as controls to possibly debug either simulation performance issues (task **T4**) or simulated hardware performance issues (task **T5**).

The left-most panel (Figure 2 D) provides on-demand information regarding simulation internals. Combined with the simulation controls (Figure 2 C), which allow pausing the simulation, a computer architect can check any value within the simulation at any point in the simulation.

To help users navigate the large amount of possible data for inspection, the monitoring view displays a hierarchical view of the hardware components being simulated. This view can be expanded or collapsed to see further sub-components. Selecting a component or sub-component will show details of properties (variables) in the components, such as their name, type, and value (or multiple values if they are container properties like lists and dictionaries) if applicable. This functionality is similar to `gdb`-like debuggers [15], but made more accessible and convenient through the hierarchical presentation.

Bottleneck analysis. Bottlenecks in the hardware are often related to buffer capacities. AkitaRTM provides insight into buffer levels during the simulation through a buffer panel (Figure 3). The panel replaces the profiling panel when clicking the green "post" icon. This view helps computer architects understand simulated hardware performance (task **T5**).

To understand why buffer fullness can identify performance bottlenecks, we consider a chain with four components (see Figure 4) where each component delegates tasks to the next-level component. Figure 4 reasonably represents how hardware components are connected like caches in a memory hierarchy. Here, Component B does not have a full buffer, suggesting it can convert requests from A to requests to C at a reasonable rate. Similarly, D does not have a full buffer, suggesting that D can fulfill requests from C. Therefore, the only reason the whole chain of components cannot process requests faster lies in Component C. In actual simulations, the connections can be more complex, and hence, a formal performance analysis is still necessary [40]. However, the buffer-fullness-based analysis provides a lightweight method to identify potential bottlenecks.

The buffer panel lists the most occupied buffers, suggesting potential bottleneck points. When this panel is first shown, a snapshot of all the buffers in the simulation is taken and sent to the website frontend. Users can select to sort the buffers by their fullness percentage or by the number of elements stored in the buffer. Being repeatedly placed at the top of the list strongly suggests that a component is a bottleneck. If the user determines that the component is not supposed to be a performance bottleneck, the simulation can terminate early to allow updating the configuration.

Simulation value monitoring. While the Simulation Monitoring Panel (Figure 2 D) and the Buffer Panel (Figure 2 E) show instantaneous status, AkitaRTM also supports tracking individual values of the hardware under simulation over time. The simulation value monitoring view (Figure 2 F) plots up to five individual values over time.

The monitoring plots support a wide range of data types. The plot uses the variable's value for numerical types (e.g., integer, floating point numbers). For containers such as lists and dictionaries, the plot shows the container sizes, which usually represent meaningful values such as the number of transactions in a set of Missing Status Holding Register (MSHR) or the number of transactions processed by a cache.

The simulation value monitoring is achieved by periodically sending status lookup requests to the simulator to query the value. We designed it to keep only the most recent 300 data points, considering that the client's memory is usually limited.

This detailed monitoring allows users to monitor the changing fullness of a buffer of interest. In case of performance problems, users can then perform more targeted post-hoc analysis, essentially starting with a "smaller haystack" for needle-in-a-haystack performance problems.

V. Case Studies

We describe two case studies, performed by one of the authors, demonstrating real-world use cases for AkitaRTM and the tasks and features we identify.

A. Case Study 1: Performance Analysis with AkitaRTM

This case study uses the Image-to-Column Conversion (im2col) benchmark [46]. The im2col algorithm is a critical building block for Convolutional Neural Networks (CNNs) as it converts a 2D image convolution operation into matrix multiplications. We set the problem parameters with 24 $\times$ 24 images with six feature map channels. The batch size is 640.

This specific simulation was configured to run on a multi-chip module GPU [4] with four chiplets. Each chiplet is equivalent to an AMD Radeon R9 Nano GPU, which is the default configuration of MGPUSim. Specifically, each chiplet has 64 compute units (CUs), 16KB dedicated L1 cache per CU, 2MB shared L2 cache, and 4GB HBM memory.

While the selected problem and hardware configuration may not cover all the cases, the goal of the case study is to demonstrate the capability of AkitaRTM. In this case study, we monitor the simulation and locate possible bottlenecks that may occur in such an intensive workload.

Initial simulation assessment. The first step in the case study is to ensure the simulation was successfully started. The progress bar (see Figure 2 G) and timer (see Figure 2 C) on AkitaRTM verified that the simulation had begun progressing. Smoothly moving progress bars and the updating timer suggest that the simulation progresses appropriately.

Bottleneck identification. After the normal execution status is confirmed, the next step is to determine if there are performance bottlenecks and identify them. To do this, the bottleneck analyzer was repeatedly refreshed, and it was noticed that the L1-VROB (Level 1 Vector Re-Order Buffer) component had a consistently high size-to-capacity (e.g., 8:8) ratio, indicating that the system is not able to process memory request going through the ROB fast enough.

To further investigate this hypothesis, we monitor the value of the size of the buffer over time to observe how it changes. Using the Component Selection panel and the location provided in the Analyzer, the component information for an L1-VROB was obtained in the middle panel. Navigating to the buffer metric and clicking on the flag icon next to the "size" value of the buffer, launched a time graph (see Figure 5 (c)), allowing us to monitor the size over time. With a given capacity of 8, the cache exhibited constant fullness with no dips in size. The fact that there is no variance in size demonstrates that the L1-VROB is constantly filled. We can confidently conclude this is related to a performance bottleneck.

Considering the nature of a reorder buffer, we know it does not have a concept of "processing speed" in the traditional sense, as inserting and removing transactions from such a buffer is almost immediate. We hypothesize this performance bottleneck is due to either the reorder buffer not being large enough or the other components that process memory requests not being fast enough.

To test the first hypothesis, we first check how many transactions are there in the reorder buffer. Note that this is a different value from the number of transactions in the ROB's top port. As shown in Figure 5 (b), the top port holds transactions to be inserted into the reorder buffer. To check the number of transactions currently being handled by the reorder buffer, we use the Component Selection Panel, find the reorder buffer, and click the flag icon next to the `transactions` field. The monitoring graph is shown in the top-left figure in Figure 5 (d).

The number of transactions in the reorder buffer fluctuates between 70 and 130. Even without checking the capacity specified in the configuration, we know the buffer size is not likely to be fully exploited as the value does not stay at a certain level. If the reorder buffer is full, the value will stay at the capability for a prolonged duration.

Next, we investigate the second hypothesis by looking at other components. Using the Component Selection Panel to go further down the hierarchy (see Figure 5 (a), we monitor the transactions of other related hardware components, including the Address Translator, L1 Cache, and the RDMA engine (see Figure 5 (d)).

The Address Translator and the L1 Cache show different patterns. The Address Translator is not likely to be a performance bottleneck because we can see high peaks turning flat within a short duration, suggesting it has a reasonable processing speed. The L1 cache shows an alarming pattern of being constantly maxed out at 16 transactions. This typical pattern shows that the component is limited by specific resources (MSHR in this case).

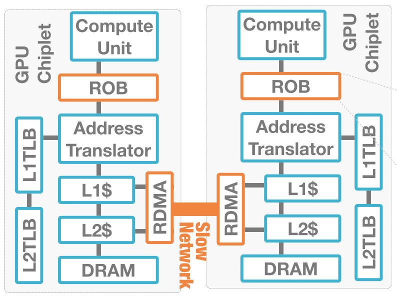
Moreover, as we check the RDMA engine, we find that the number of transactions is also at an alarmingly high level (about 1000 transactions). These are inflight transactions gathered from the L1 caches and waiting for a remote GPU chiplet to provide the data (or write acknowledgment). The high number suggests that the RDMA (or the network connecting the RDMAs) will likely be a performance bottleneck.

B. Case Study 2: Debug a Hanging Issue

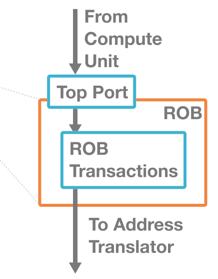
In this case study, we demonstrate how to debug a hang. This case study is also performed by one of the authors who knows MGPUSim well. The bug is an actual bug in MGPUSim, discovered while using AkitaRTM, and the solution has since been merged into the MGPUSim open-source repository. As we will see, debugging a hanging issue requires using most parts of AkitaRTM together.

We describe the problem first to help readers follow the case study, but we did not know the problem before the case study was performed. The problem originates in the L2 cache implementation. The L2 cache in MGPUSim has a write buffer that temporarily buffers evicted data to be written to the DRAM. The data that is fetched from the DRAM also go through the write buffer before they can be written to the local storage. So, at a certain time of simulation, the local storage wants to send a transaction to the write buffer to evict. Meanwhile, the write buffer wants to send a piece of fetched data to the local storage. However, since the local storage cannot free up the eviction transaction, it cannot take the fetched data, causing deadlock.

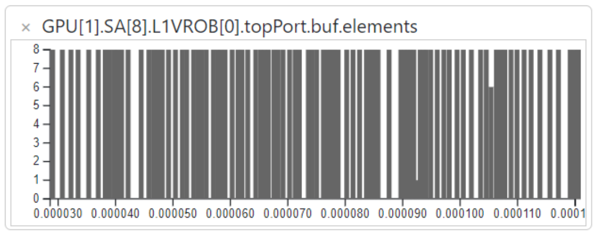
Starting the simulation. Before the debugging session starts, we have already experienced the issue that the simulation never finishes. Therefore, we suspect it is a hang due to a bug somewhere and take the standard procedure to solve the hanging bug. Since we are dealing with a hang, we start the simulation with a GDB-style debugger (`dlv` [41] for Go). Starting the simulation with a debugger does not change AkitaRTM's appearance.



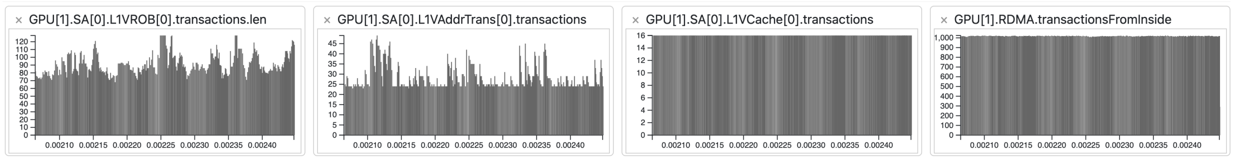
(a) Memory hierarchy in MGPUSim



(b) The local structure of the reorder buffer (ROB)



(c) Monitoring the buffer level of the ROB's top port suggests that it is full for an extended period.



(d) Monitoring other components in the memory hierarchy suggesting that the RDMA is causing performance issues as there is constantly a large number of transactions.

Fig. 5. The case study that demonstrate performance analysis of the Image-to-Column (im2col) workload running on a 4-chiplet GPU hardware. In (c), we monitor the buffer fullness of the reorder buffer's (ROB) top port's buffer, identifying performance bottlenecks related to the components that drain the ROB. By monitoring the transaction level of connecting components (see (d)), we notice the high transaction level in the RDMA engine, confirming that the processing capability (caused by the slow network) of the RDMA engine is the root cause of the performance issue.

Confirming the hang. Once the simulation started, we observed the simulation progress smoothly for some time. When the hang occurs, we observe that 1) the progress bars (Figure 2 G) stop moving, 2) the time in the Simulation Control (Figure 2 C) stops changing, and 3) the CPU usage (Figure 2 A) falls to a level significantly less than 100%. Once these states last for a few seconds, we are confident that the simulation has entered a hang state.

Identify hanging components. The Bottleneck Analyzer is our best friend in identifying hanging components. We click the Bottleneck Analysis button to show the buffer level. If the simulation is completed successfully, every component should be idle, and all the buffers should be empty. If there is any content in a buffer, we know the buffer owner cannot proceed to process the requests. In this particular situation, we observed that L1 caches, L2 caches, and DRAM controllers have buffer contents. More components may have buffer contents than the actually problematic components, which are caused by buffer backpressure. We thus proceed to investigate further to rule out components as the possible cause on the way to identifying the true cause.

Identify the cause of the hang. Without AkitaRTM, a programmer would typically need to set some breakpoints, based on their experience, and restart the simulation, hoping that the breakpoint they set can capture the hanging issue. However, given the large search space, it usually takes a long time and some luck before the best breakpoint is set. With AkitaRTM, programmers do not need to restart the simulation and can solve the problem within the current context.

Using the buffer levels, we gained a list of potential components that cause the hang. We can process them type by type (L1 caches are one type, L2 caches are another type) to see if they are problematic. We first set a breakpoint at the first line of code in the `Tick` function (most of the components in MGPUSim have such functions). We then locate the component in the Simulation Monitoring Panel (Figure 2 D). Since the component is not making forward progress, MGPUSim puts it to sleep. We need to wake it up by scheduling a Tick event in the next cycle. To do so, we may click the *Tick* button Tick, which is directly under the Resource Monitoring Panel (Figure 2 A). In case the whole simulation is not running, we would also need to use the "Kick Start" button in the Simulation Control Panel (Figure 2 C) to continue with the simulation.

Since the breakpoint is set, the execution will stop at the first line of code of the Tick function. We can then execute line by line to see why the component cannot make progress. Using this method, we determine that the L1 cache and DRAM controllers cannot make progress because they cannot send messages to L2 caches. In L2 caches, we discover that the local storage is not receiving or sending out transactions. The whole process of identifying this bug took about 1.5 hours. We estimate this time to be much shorter than our debugging process without AkitaRTM.

Overall, the design of the "Tick" button facilitates debugging the cause of hanging problems. This design supports cycle-based simulation but does not yet support fully event-driven simulation. To provide analogous functionality for event-driven simulators, we could add a dropdown menu for all the events the component can handle and change the "Tick" button to a "Schedule" button to schedule the selected event.

C. Beyond the case studies

Throughout the case studies provided, we have demonstrated how a majority of the elements of AkitaRTM can be utilized to solve real-world problems. Other elements, such as the play/pause/jumpstart (Figure 2 C), were not highly utilized in these case studies but are used in regular practice. Specifically, the simulator controls allow for "slowing down time" in the simulator to try to catch specific instances of component ticks. The value monitoring tool (Figure 2 D) allows for time-series visualization of task execution, and thus, events like filled buffers may be more visible. The profiling section (Figure 2 E) allows for identifying which function call hierarchy may be problematic. We believe all elements of AkitaRTM are important to getting the full picture of a simulation.

VI. Evaluation

The above case studies gave an expert-focused view of the possibilities of AkitaRTM. To broaden our understanding of the effectiveness of AkitaRTM across possible users, we conducted a user study with six computer architecture researchers. As a qualitative study, our goal is not to draw statistical conclusions but to derive meaningful themes and patterns from members of the target audience and inform further research and development. Qualitative studies require fewer participants than quantitative studies, as the depth of data from each participant is the more significant aspect than its numeracy [9]. This study was approved by the Institutional Review Board at our institution.

A. Method

We conducted evaluation sessions with participants to better understand the efficacy of AkitaRTM.

Format. Each evaluation session was conducted individually with each participant through a virtual conferencing platform (e.g., Zoom). The study consisted of 5 distinct parts:

1) Participants were given a demonstration of AkitaRTM with MGPUSim running the Image-to-Column Conversion (im2col) benchmark, the same as used in the first case study.
2) A simple simulation was provided to them using the "Finite Impulse Response" (FIR) benchmark. The participant used AkitaRTM while sharing their screen. No specific performance issues or complexities were added to this trial. The purpose was for the participant to become more comfortable with AkitaRTM and ask questions about it.
3) While still sharing their screen, participants were given a problematic "Image 2 Column" simulation (with multiple bottlenecks and performance issues) to observe whether they could identify these issues. We did not answer any questions they had about AkitaRTM. We expected participants to identify bottlenecks at the ROB level and to mention that these issues may cascade down to other connected components.
4) A brief semi-structured interview was conducted regarding their experience with AkitaRTM.
5) A short post-study survey was administered to gauge the success of AkitaRTM across pre-defined criteria.

Participants Recruitment. Participants were recruited using convenience sampling (i.e., using participants "convenient" to researchers) methods by reaching out directly to individuals via online messaging systems (e.g., Slack) in academia and with ties to the field of computer architecture. Of these, the participants who accepted the invitation and appeared in this study are colleagues of or affiliated in some way with one or more of the authors of this paper. Participants were not paid for their time. We refer to participants by code, PT{1-6}.

Among our participants, 3 are currently Ph.D. students (PT2, PT3, PT4), and 3 are undergraduate students who actively conduct computer architecture research (PT1, PT5, PT6). Four participants (PT2, PT3, PT5, PT6) had prior experience with AkitaRTM before participating in our study.

Analysis Method. The online meetings were recorded, and transcripts were generated. The first and the last authors open-coded the transcript together to find themes, referring to the recordings when the transcripts were unclear. Findings from this analysis are presented in the Results section.

B. Results

We first describe overall feedback and task performance in our results summary. Then, we discuss observations, notions, and points of view derived during open coding. Finally, we present the findings of the follow-up survey.

1) Summary: The overall response to AkitaRTM was positive among all study participants. During the evaluation sessions, the feature used the most by all participants was the Bottleneck Analyzer, while the least used feature was the Profiling panel. Concerning both the design and informativeness of AkitaRTM, users mentioned many successes of the tool while highlighting important critiques that can be learned and iterated upon for future releases.

Regarding whether the participants were successful in identifying bottlenecks in the third part of the sessions, PT3, PT4, and PT5 all identified proper bottlenecks in the system during the third portion of the study through primarily a mix of the Bottleneck Analyzer and Time Charts. They can identify the problem in the ROB and the RDMA engine.

2) Themes derived in open coding: We discuss findings derived during open coding of the evaluation sessions, including comments made during the demonstration, during participant use of AkitaRTM, and during the semi-structured interview. These have coalesced into themes regarding what AkitaRTM is: (1) a companion, (2) a different perspective, and most interestingly, (3) a learning tool, as well as a discussion of how AkitaRTM can be improved for new users, similarly identified through the evaluation sessions.

AkitaRTM is a companion. This theme demonstrates that users engaged with AkitaRTM in a way that matched the initial impetus for creating AkitaRTM. Despite being given such a complex tool without much time (i.e., multiple days or weeks) to learn about it, all participants had fluidity in navigating the tool quickly on their own without much guidance.

The word "companion" is meant to underscore the importance of AkitaRTM not being a standalone tool; it provides an avenue for in-situ debugging (as demonstrated in the case study above). PT2 and PT3 both had more experience in computer architecture development than other participants and thus were able to take the fullest advantage of all of the features AkitaRTM had to offer. Instead of solely relying on our demonstration to guide their exploration, they clicked on multiple different components in the Component Selection panel while analyzing the different values included on a component in which a bottleneck could occur.

AkitaRTM is a different perspective. On a similar notion to the theme of companionship, we strived to develop AkitaRTM with the idea of a differing perspective in mind. While this primarily came in the form of being able to define locations of bottlenecks in real time, such a requirement was ultimately deemed the most useful aspect of the tool by participants. For example, PT2 mentioned that they would use AkitaRTM, in reference to any computer architecture simulations they run, to see where the bottleneck is...to see why [they] are achieving a low or high bandwidth [in the component buffers]". PT5 mentioned that AkitaRTM provided a "useful list of all the components...there are a lot of components so [the tools] allow [them] to find bottlenecks and other problems". Post-hoc analysis software has the detriment of not being able to accurately display times when a simulation may have hanged or may not even be accessible if a simulation crashes completely mid-execution. Participants enjoyed being able to see the events and component values leading up to bottlenecks and problematic simulation portions. PT4 was even able to observe the simulation during the third part of the study hang properly and used the opportunity to use the tool and gain more insight into where the hang might occur. Post-hoc analysis methods display a lot of information that can be sorted through, but it is hard to pinpoint an exact location, as specific data points may easily be overlooked. The fact that AkitaRTM is able to embody this theme well plays to the strength of the design of such a real-time tool.

AkitaRTM is a learning tool. The most unexpected theme that arose from participants (specifically PT1 and PT6, who are undergraduate researchers) was how AkitaRTM helped not-as-experienced members learn more about computer architecture. While the prior two themes highlight AkitaRTM as a way for experienced computer architects to gain insights into their current simulations, this theme highlights the potential the design has in being an educational tool, something we did not anticipate.

While PT1 and PT6 were unable to successfully identify the issues in the problematic simulation provided, they each gained insight into how the GPU was executing the simulation. PT6 spent most of the study duration asking questions about the inner workings of the simulation and monitoring tools, as well as drawing hierarchical connections between all of the GPU's components. They mentioned towards the end of the study how "the comparison between [the time] plots here are very helpful in finding links between components." These experiences solidified that the design choices for AkitaRTM are robust and helpful enough to be used to grasp even initial concepts about the hardware the simulation is being run on. Even for other participants who were more versed in computer architecture, the tool prompted discussions about how the low-level hardware executing the simulations worked.

Questions:	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1. AkitaRTM is easy to learn				3	3
2. Progress bars are helpful				2	4
3. Component details are helpful			1	1	4
4. Time graphs are helpful				1	5
5. I can identify perf. issues			1	2	3
6. The profiling tool is helpful		1	1		4

Fig. 6. Distribution of responses to post-study survey statements. Question 4 had the highest average score (4.8), while Question 6 had the lowest (4.2).

AkitaRTM can be improved with guidance for new users. While sentiment towards AkitaRTM was largely positive, the data generated by our user study also suggests several areas for improvement. By far, the most voiced sentiment was that AkitaRTM is hard to get started on *individually*, as participants noted that unlike in the study, most potential users are not given a demonstration of how to use a given software and must learn it on their own. Though arguably a noticeable flaw, the current concept for AkitaRTM was geared at providing a design that can support the tasks we identified. Future iterations of AkitaRTM may want to focus on the interface's discoverability and accessibility.

The other critiques were similar but focused on more individually explainable UI elements. PT1, PT3, PT4, and PT6 all suggested that some of the buttons have more intuitive icons. Especially with the Bottleneck Analyzer, participants noted that the button was too much like the value monitoring button (the flag that opens the time graph) and did not relate much to the function of the button. Similarly, PT4 wished for more "flow between all the features" or indicators of cohesiveness between the tools. These concerns further point to the need for more guidance for individual first-time users.

C. Post-Study Survey

In the follow-up survey, participants were asked to rank six statements on a scale of 1 (low) to 5 (high), correlating to how much they agreed with the statement. The statements were:

1) AkitaRTM is easy to learn.
2) The progress bar helped me understand if the simulation is making progress and have a brief estimation about when the simulation will finish
3) The component details allow me to check critical statuses of the simulation
4) The component time graphs helped me trace how some parameters change over time
5) I can identify the performance issues of the software or hardware under simulation
6) The integrated profiling tools allow me to find the performance bottlenecks of the simulator.

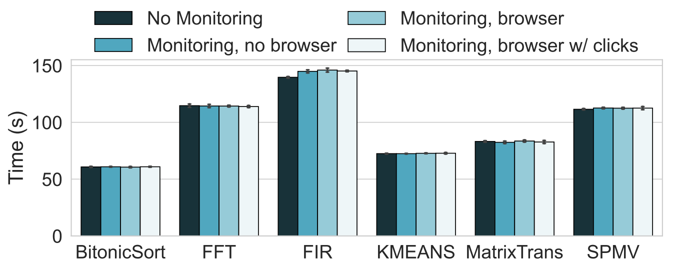


Fig. 7. Comparing the execution times of simulations with and without AkitaRTM. The performance overhead of AkitaRTM is negligible.

Figure 6 shows the distribution of the responses. Given a sample size of six completed surveys, the average response between all statements was 4.5, with an average standard deviation of 0.77. Most notably, question (3) had the highest response average of 4.8. While the sample size is small, these results suggest the viability of AkitaRTM as a tool. We were unable to follow up with the participant who disagreed with the profiling tool being helpful due to the survey being anonymous.

VII. Performance Evaluation

We assessed AkitaRTM's performance by simulating six benchmarks from MGPUSim, selecting problem sizes that fully engage all cores in MGPUSim's default configuration. Our evaluation encompasses four distinct scenarios: 1) Absence of monitoring, where the monitor is not activated; 2) Monitoring enabled without a browser interface, meaning the monitor operates in the background without handling HTTP requests; 3) Monitoring with a passive browser interface, where the monitor and browser window are active but without user interaction, allowing only time and progress indicators to update; and 4) Active monitoring with simulated user interactions, where the tool is in full operation, and elements within the component list receive automated clicks at one-second intervals via JavaScript to mimic regular user engagement with AkitaRTM. Each benchmark and scenario combination was executed five times to get the average performance.

Overall, we see no major performance overhead when using AkitaRTM (see Figure 7). The highest performance overhead is 3.7% observed in the FIR benchmark. For a few other benchmarks, the performance overhead is within the noise range. The low overhead also suggests that when many simulations run parallel on one machine, AkitaRTM will not slow the overall simulation process. We attribute the low overhead to three design choices: 1) AkitaRTM only performs actions on-demand to respond to frontend requests. When there are no requests, AkitaRTM's code will not run. 2) AkitaRTM serializes in fine granularity. AkitaRTM only serializes one component or value on each front-end request. Fine serialization granularity also avoids the requirement for global synchronization. And 3) AkitaRTM runs in a dedicated thread, parallel to the main simulation thread. Therefore, its execution will minimally interfere with the main simulation thread execution.

VIII. Discussion

We contend that advancing computer architecture research necessitates a human-centered approach, in addition to technical perspectives. The true effectiveness of these tools is realized only when they are designed with the user's understanding in mind. By developing tools that are intuitive and accessible to researchers, we can significantly enhance their ability to make informed decisions. In the following section, we summarize our key learnings, which may have broader applicability in developing future computer architecture tools.

Many small steps forward are a big step forward. AkitaRTM can be considered a data aggregation and visualization tool. It collects real-time data from a simulation and displays the data in an easy-to-process way on the interface.

While most of the features in AkitaRTM can be achieved by other methods, AkitaRTM reduces barriers and makes difficult tasks easier. For example, the progress bar can be printed in the terminal, and the simulation monitoring feature can be achieved with a regular breakpoint debugger. However, by simplifying all the processes, AkitaRTM significantly improves the user experience and workflow. Our collaborators told us they started to establish a habit of profiling the simulator. Previously, they would only profile the simulator every half a year, but now, because the feature is so accessible, they profile it daily to weekly, and performance bugs are solved as new features are added. While GDB can be used to debug simulation hangs, developers need to repeatedly restart the simulation and recreate the hanging scenario, spending days to weeks. AkitaRTM allows trigger component ticking, recreating the site in seconds, thereby saving developers significant time or engineering work so that they can focus more on research.

Focusing on side workflows. Existing workflow task analysis methods [13] mainly focus on the main tasks. While some methods [2], [48] add more emphasis on the context, the main workflow is still the leading factor that determines the design rationale. For computer architecture design, the main task for architects usually focuses on identifying performance bottlenecks and comparing two designs. However, beyond the main tasks, the workflow of computer architecture researchers includes many other items, e.g., simulator debugging and ensuring the simulator performance. While addressing their main needs is important and needs formal solutions [20], [40], our experience in AkitaRTM suggests the importance of developing tools for users' side workflow.

AkitaRTM is a tool that focuses on the side workflow. This nature determines the unique design decisions made within AkitaRTM. For example, since there is no single most important task in the side workflow, we take a dashboard-like [42] design by laying out all the content in one interface. Also, the first priority of designing the performance analysis feature is to provide quick, real-time feedback rather than the accuracy of identifying performance bottlenecks. Overall, the design of AkitaRTM may inspire future designs of similar tools that focus on facilitating the side workflow of users.

Considering human-centered methods in the tool-design process. Computer architecture tools are designed to be used by human researchers. While the technical perspective is crucial, human factors may also be considered. Recent research has provided more tools that bridge simulation tools with human users [3], [40]. Our research suggests that the following aspects may be considered in future tool-designing projects: 1) a user-based requirement collection processes, 2) task definitions that consider users' needs, and 3) user-based evaluation, quantitative or qualitative.

Limitations of AkitaRTM. AkitaRTM offers the possibility of opening the black box of computer architecture simulators. However, as one of the first tools of its kind, there are still several limitations.

One major limitation is rooted in the nature of AkitaRTM, where all the data are collected and displayed in real-time. For example, the bottleneck analyzer can only hint at the location of bottlenecks but cannot give a more definitive answer. Should this limitation become a high-priority barrier to users, we could address it by incorporating more data and more views (e.g., real-time achieved throughput of ports, graph-based view hardware connections) in the future.

Another set of limitations is caused by the learning curve and the requirement of knowledge of the hardware. We observe that users often struggle with understanding how hardware components are connected and require help finding the connecting components. AkitaRTM is designed for expert users, especially those who design the simulated system and thus intimately know the configuration. However, we believe having a more intuitive interface (e.g., showing a map of how components are connected) would significantly reduce the difficulty and improve the usability of AkitaRTM.

IX. Conclusion

Looking inside the "black box" of computer architecture simulators and monitoring real-time execution status helps accelerate the research process. Enhancing the usability and interoperability of such tools cannot be addressed solely from a technical perspective but should also be from a human-computer interaction perspective. This paper proposes an initial attempt to address the opaqueness of simulators that benefit users with an easy-to-use interface.

AkitaRTM is a monitoring tool for a popular GPU architecture simulator, MGPUSim. AkitaRTM supports monitoring the hardware resources and simulation progress, debugging simulation hanging issues, identifying performance bottlenecks, and profiling the simulator itself. AkitaRTM has been demonstrated to be helpful in computer architecture design, thanks in part to our identification of essential tasks performed by computer architecture researchers. We see AkitaRTM as an essential step towards building explainable computer architecture (XCA) by helping computer architecture researchers to better understand the intermediate states of simulations. AkitaRTM can potentially avoid asking simulator users to wait for the simulation to fully complete by terminating simulations early to make necessary modifications. Moreover, the design of AkitaRTM sheds light on future simulator development to make simulators humanly understandable through real-time monitoring tools.

Acknowledgment

We thank all the anonymous reviewers who have provided constructive feedback leading to the enhanced quality of this paper. We also thank our participants for testing AkitaRTM and telling us their opinion on this tool. This work is supported by the National Science Foundation (NSF) under award CNS-2234400, CNS-2234401, OAC-2246035, and CCF-2402804. This work is also supported in part by AMD.

References

[1] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi, S. Monk, J. Ogden, M. Rajan, and J. Stevenson, "Continuous whole-system monitoring toward rapid understanding of production hpc applications and systems," *Parallel Computing*, vol. 58, pp. 90–106, 2016.

[2] J. Annett, "Hierarchical task analysis," *Handbook of cognitive task design*, vol. 2, pp. 17–35, 2003.

[3] A. Ariel, W. W. Fung, A. E. Turner, and T. M. Aamodt, "Visualizing complex dynamics in many-core accelerator architectures," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 164–174.

[4] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 320–332, 2017.

[5] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 724–737. [Online]. Available: https://doi.org/10.1145/3466752.3480100

[6] ——, "Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 724–737.

[7] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 163–174.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[9] R. Budiu, "Why 5 participants are okay in a qualitative study, but not in a quantitative one," 2021. [Online]. Available: https://www.nngroup.com/articles/5-test-users-qual-quant/

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[11] J. Cubero-Cascante, N. Zurstraßen, J. Nöller, R. Leupers, and J. M. Joseph, "parti-gem5: gem5's timing mode parallelised," in *International Conference on Embedded Computer Systems*. Springer, 2023, pp. 177–192.

[12] S. Deublein, B. Eckl, J. Stoll, S. V. Lishchuk, G. Guevara-Carrion, C. W. Glass, T. Merker, M. Bernreuther, H. Hasse, and J. Vrabec, "ms2: A molecular simulation tool for thermodynamic properties," *Computer Physics Communications*, vol. 182, no. 11, pp. 2350–2367, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465511002025

[13] D. Diaper and N. Stanton, "The handbook of task analysis for human-computer interaction," 2003.

[14] O. C. Eidheim, "Simple-web-server: a fast and flexible http/1.1 c++ client and server library," *Journal of Open Source Software*, vol. 4, no. 40, p. 1592, 2019.

[15] A. Freeman, "Using the go tools," in *Pro Go*. Springer, 2022, pp. 35–57.

[16] Google, "google/pprof: pprof is a tool for visualization and analysis of profiling data." [Online]. Available: https://github.com/google/pprof

[17] K. Grochowski, M. Breiter, and R. Nowak, "Serialization in object-oriented programming languages," in *Introduction to data science and machine learning*. IntechOpen, 2019.

[18] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 608–619.

[19] F. Gündling, "Simple c++ serialization & reflection." [Online]. Available: https://github.com/felixguendling/cista

[20] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization." *EuroVis (STARs)*, 2014.

[21] T. Issariyakul and E. Hossain, "Introduction to network simulator 2 (ns2)," in *Introduction to network simulator NS2*. Springer, 2009, pp. 1–18.

[22] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[23] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[24] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[25] J. Kosakowska and M. Schmidmeier, "Arc diagram varieties," *Contemporary Mathematics series of the AMS*, vol. 607, pp. 205–224, 2014.

[26] Y. Li, Y. Sun, and A. Jog, "Path forward beyond simulators: Fast and accurate gpu execution time prediction for dnn workloads," 2023.

[27] C. Liu, Y. Sun, and T. E. Carlson, "Photon: A fine-grained sampled simulation methodology for gpu workloads," 2023.

[28] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[29] S. Lucas, "The origins of the halting problem," *Journal of Logical and Algebraic Methods in Programming*, vol. 121, p. 100687, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235222082100050X

[30] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 201–213.

[31] M. Naderan-Tahan, H. SeyyedAghaei, and L. Eeckhout, "Sieve: Stratified gpu-compute workload sampling," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 224–234.

[32] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, no. 6, pp. 4–12, 2015.

[33] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 81–92.

[34] A. F. Rodrigues, G. R. Voskuilen, S. D. Hammond, and K. S. Hemmert, "Structural simulation toolkit (sst)." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[35] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[36] H. SeyyedAghaei, M. Naderan-Tahan, and L. Eeckhout, "Gpu scale-model simulation," 2024.

[37] H. Sharifi, O. Aaziz, and J. Cook, "Monitoring hpc applications in the production environment," in *Proceedings of the 2Nd Workshop on Parallel Programming for Analytics Applications*, 2015, pp. 39–47.

[38] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[39] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 197–209.

[40] Y. Sun, Y. Zhang, A. Mosallaei, M. D. Shah, C. Dunne, and D. Kaeli, "Daisen: A framework for visualizing detailed gpu execution," in *Computer Graphics Forum*, vol. 40, no. 3. Wiley Online Library, 2021, pp. 239–250.

[41] The DELVE developers, "DELVE: A debugger for the go programming language," 2023. [Online]. Available: https://github.com/go-delve/delve

[42] R. Toasa, M. Maximiano, C. Reis, and D. Guevara, "Data visualization techniques for real-time information—a custom and dynamic dashboard for analyzing surveys' results," in *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, 2018, pp. 1–7.

[43] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 335–344.

[44] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for speed: Experiences building a trustworthy system-level gpu simulator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 868–880.

[45] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim: a memory system simulator," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.

[46] H. Wang and C. Ma, "An optimization of im2col, an important method of cnns, based on continuous address access," in *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. IEEE, 2021, pp. 314–320.

[47] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 496–508.

[48] Y. Zhang, K. Chanana, and C. Dunne, "Idmvis: Temporal event sequence visualization for type 1 diabetes treatment decision support," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 512–522, 2018.