Snap & Replay: A new way to analyze uarch-scale performance bottlenecks for ML accelerators

Ioannis Zarkadas*
Columbia University
New York, NY, USA
iz2175@columbia.edu

Amanda Tomlinson*
University of California, San Diego
San Diego, CA, USA
actomlin@ucsd.edu

Asaf Cidon
Columbia University
New York, NY, USA
asaf.cidon@columbia.edu

Baris Kasikci
University of Washington
Seattle, WA, USA
baris@cs.washington.edu

Ofir Weisse
Google
USA
oweisse@google.com

Abstract

As models become larger, ML accelerators are a scarce resource whose performance must be continually optimized to improve efficiency. Existing performance analysis tools are coarse grained, fail to capture model performance at the machine-code level and often do not provide specific recommendations for optimizations. In addition, existing methodologies are hard to apply in Google's production environment, as they require hardware changes or recompilation. We present SnR, a fine-grained methodology for analyzing ML models at the machine-code level that provides actionable optimization suggestions. It requires no hardware changes and no recompilation.

Our core insight is to use a hardware-level simulator, an artifact of the hardware design process that we can re-purpose for performance analysis. Traditionally, these simulators are confidential tools used to improve hardware designs given representative software workloads. However, as a hyperscaler practicing hardware/software co-design, we are uniquely positioned to use them in the opposite direction: to optimize software given fixed hardware architectures. SnR captures traces from production deployments running on accelerators and replays them in a modified microarchitecture simulator to gain low-level insights into the model's performance. We implement SnR for our in-house accelerator (TPU) and used it to analyze the performance of several of our production LLMs, revealing several previously-unknown microarchitecture inefficiencies. Leveraging these insights, we optimize a common communication collective by up to 15% and reduce token generation latency by up to 4.1%.

ACM Reference Format:
Ioannis Zarkadas, Amanda Tomlinson, Asaf Cidon, Baris Kasikci, and Ofir Weisse. 2025. Snap & Replay: A new way to analyze uarch-scale performance bottlenecks for ML accelerators. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3772052.3772233

1 Introduction

As generative artificial intelligence (AI) is projected to become a trillion dollar market by 2032 [9], an increasing number of companies invest in developing ML accelerators. This environment has motivated many traditionally consumer-focused companies to develop their own hardware, giving rise to the hardware/software co-design paradigm. Hyperscalers like Google [28, 29], Meta [19], and Amazon [50, 51] now design custom ML chips alongside established semiconductor companies such as NVIDIA [13], AMD [4] and Intel [12]. Unlike traditional chip development cycles where semiconductor companies rely on representative applications from consumer companies to inform next-generation designs, co-design enables faster iterations and closer integration by allowing hardware and software teams to collaborate directly throughout the development process.

A complex ecosystem of tools have been built around these accelerators to support ML development. High-level frameworks like TensorFlow [2], JAX [20], and PyTorch [46] allow engineers to express machine learning models with simple APIs. These high-level models are then translated into ML intermediate representations like MLIR [33] or OpenXLA

*Both authors contributed equally to this research.

SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772233

StableHLO [49]. These mid-level representations provide a layer of indirection between high-level ML frameworks and machine-level code, allowing compilers to target custom hardware from a reduced set of intermediate representations. These portable mid-level representations are then compiled into the byte-code which runs on the ML accelerator. The development of each of these levels of abstraction requires a huge engineering effort, and inefficiencies introduced at any level can cause performance degradation for the model. The companies that offer generative AI services are often doing so at a massive scale (for example, the infrastructure to provide inference for Microsoft's Bing AI chatbot is estimated to cost $4 billion [58]), meaning that even a small degradation in performance can lead to large capital losses. Some companies such as DeepSeek are even implementing features directly in machine-code [55], showcasing the importance of low-level optimizations.

Because of the utmost importance of model performance, ML engineers need robust profiling and optimization tools. However, existing performance profiling tools fall short in analyzing low-level performance. Many tools (e.g., Nvidia NSight Systems [41], Tensorboard [22]) provide coarse-grained metrics on the high-level operations (HLOs) of the intermediate representation by leveraging the accelerator's performance monitoring unit (PMU). This is useful for revealing inefficient HLOs, but offers little visibility into the interaction between the code and hardware. Moreover, the granularity of metrics provided by the PMU is constrained by its buffer size.

Other tools focus on finding places in the code that cause hardware stalls by sampling program counters (e.g., Nvidia CUPTI [26], Intel VTune [26]). However, the stalled instruction is not always the root cause of the stall. For example, a matmul instruction might stall because its operands haven't arrived at the data cache yet due to a load instruction that was issued too late, but stall sampling will only point to the matmul rather than the problematic load. In addition, stall PC sampling requires specialized hardware support that is not universally available, including on our own accelerator. Another category of tools (e.g., NVBit [60], CUDAAdvisor [54], ValueExpert [62]) are based on binary instrumentation, which records information about every low-level instruction executed in the accelerator. While this information enables fine-grained analysis of the software, it changes the hardware utilization characteristics at runtime. Finally, instrumentation typically requires recompilation, which makes it hard to apply in a production setting where models and code are constantly changing.

Existing approaches such as PMU-based profiling, stall sampling, and software instrumentation are inadequate for our goals due to inherent limitations in granularity, accuracy and practicality. Simulation, however, offers a promising alternative, supported by a rich history and extensive literature for both CPUs [8, 10, 37] and GPUs [6, 21, 30, 31, 34, 61]. Traditionally, hardware companies have employed simulators to optimize hardware designs based on fixed software workloads. Conversely, simulations can also be leveraged to optimize software for given hardware designs. Yet, this potential is hindered because proprietary hardware simulators are typically kept confidential by hardware manufacturers to protect design secrets, while academic simulators fall short in accuracy due to incomplete knowledge of closed-source designs, making them unsuitable for effective performance debugging. In the contemporary era of software-hardware co-design, however, consumer companies increasingly design their own hardware accelerators, providing access to previously guarded hardware insights. This shift enables new profiling techniques.

To fill this gap, we present Snap & Replay (SnR), a novel methodology to analyze the microarchitectural efficiency of ML accelerators in the context of a hyperscalar datacenter. As shown in Figure 1, SnR enables a "record and replay" style of profiling by recognizing that a common artifact of the accelerator design process, a Golden Reference Model (GRM), can be repurposed as an Instruction Set Architecture (ISA) level simulator. This methodology allows the capture of fine-grained details for both the software (e.g., instruction dependencies, memory accesses) and the hardware (e.g., compute units utilization, memory stalls). SnR first uses a step debugger to capture traces from production deployments of our in-house ML models, then replays them in a modified ISA-level simulator. We then use the data produced from the ISA-simulator to build a series of performance analyses, and automatically suggest performance optimizations. As shown in table 1, SnR unlocks low-level performance insights while avoiding the pitfalls of existing approaches.

In this paper, we describe how we used SnR to optimize our large language models (LLMs). First, we describe how SnR can help identify and visualize inefficient memory transfers (DMAs). Second, we use SnR to analyze the utilization and fragmentation of our accelerator's data-cache and provide fine-grained instruction-by-instruction utilization information for various compute units of our accelerators, enabling our engineers to reason about how their code performs at the machine-code level. Third, we use SnR to analyze the dependencies of each instruction to automatically suggest optimizations via instruction reordering. Although our in-house LLMs are already highly optimized, SnR revealed several previously unknown inefficiencies that amounted to up to 4.1% of token generation time.

In summary, we make the following contributions:

Approach	No HW Change	HW Metrics	Granularity	No Recompilation	Overhead
PMU	Yes	Yes	HLO-Level	Yes	Low
Stall Sampling	No	Yes	Instruction-Level	Yes	Low
Instrumentation	Yes	No	Instruction-Level	Maybe	High
SnR	Yes	Yes	Instruction-Level	Yes	High

Table 1: Comparison of Performance Analysis Approaches

- We introduce SnR, a performance analysis framework that repurposes a microarchitectural simulator to optimize software performance for a *specific* accelerator architecture. In contrast to prior work that optimizes hardware for fixed software, our approach exploits hardware/software co-design, made possible by in-house TPU development, to specifically optimize software for existing hardware.
- We implement SnR for TPUs and construct several performance analyses based on the data we collect with it, which are very hard or impossible to implement with existing tools. SnR revealed several inefficiencies in our LLMs, even though they have been thoroughly optimized already.
- Using SnR's suggestions, we optimize a common communication collective (All-Gather) by 15% and decrease the generation time of our LLMs by up to 4.1%. These LLMs are deployed at huge scale internally and each percent improvement in model serving leads to significant savings in total cost.

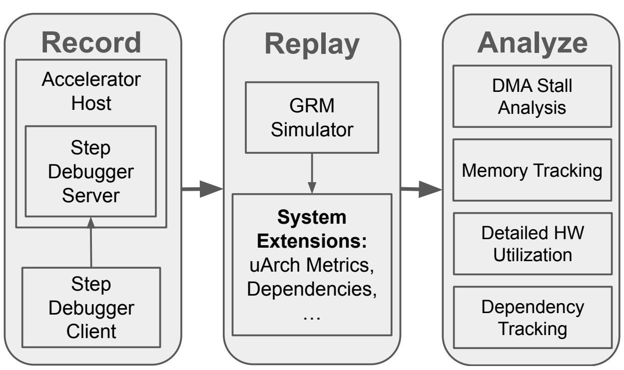


Figure 1: Overview of SnR.

2 Background and Related Work

In this section, we provide a brief overview of the ML software development landscape (§2.1). We then discuss the growing need for optimizing accelerator performance and how profilers are a crucial tool in doing so (§2.2). Finally, we compare to prior work in simulation (§2.3).

2.1 ML Software Stack

High-level frameworks like TensorFlow [2], JAX [20] and PyTorch [46] provide simple APIs for constructing ML models. High-level models are then translated into intermediate representations like MLIR [33] and OpenXLA StableHLO [47]. The intermediate representation is usually a computation graph consisting of higher level operations (HLOs), like "matmul" and "transpose". These mid-level representations provide a layer of indirection between the ML frameworks and ML compilers [11, 35, 45, 49]. The compiler then uses these representations to generate accelerator code (e.g., CUDA kernels, PTX, SASS) in a process known as *lowering*.

Given the very high costs of deploying and operating AI infrastructure, there is a lot of work on squeezing more performance out of existing accelerators. Performance experts focus on constructing faster kernels through careful orchestration of the hardware and especially memory, via techniques like FlashAttention and PagedAttention [16, 17, 32, 53]. Compiler engineers invent new frameworks that expose greater control of the hardware, like Triton [56] and Pallas [27]. And at a cluster level, resource managers like Pathways [7] take ML workload characteristics into account when making global scheduling decisions.

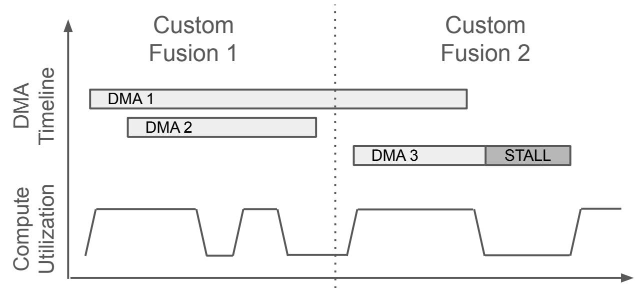
2.2 Profilers

To allow developers to debug and optimize their workloads, vendors and academics have developed a number of performance profiling tools.

Motivating Example. In Figure 2 we show a simplified example of an inefficiency which would be hard to detect with current profiling tools. Existing tools present performance information in a coarse-grained format, usually listing an operation (the HLO or kernel name, like "Custom Fusion 1") and any aggregated statistics of interest (like runtime, FLOPS, or utilized memory bandwidth). These tools show which operations have low performance, but do not provide the reasons for low performance or clues on how to increase performance. In contrast, a low-level optimization tool like SnR can offer deeper insight into each microarchitectural component of the accelerator. In our example, the course grained analysis shows that the second custom fusion HLO has lower than expected average FLOPS. We may also be able to see that hardware bandwidth utilization can be improved

Operation	Avg Time (us)	Avg FLOPS (GFLOP/s)	HBM BW (GiB/s)
Collective	10	2,000	80
Custom Fusion 1	20	900	450
Custom Fusion 2	30	100	300

(a) Coarse Grained



(b) Fine Grained

Figure 2: A toy example showing a coarse-grained vs fine-grained analysis. Many existing tools provide coarse-grained information at the kernel or HLO level, while deep optimization requires a fine-grained view.

which gives us some clue that the memory subsystem is responsible for the low performance. The fine grained analysis gives the information we need to fully analyze and fix the issue with our custom HLO. In Figure 2b we can see each memory transfer along with detailed compute utilization metrics. We can infer that the compute utilization is low because the hardware is stalling when issuing DMA 3. We can also see that to fix the issue, we need to issue the DMA earlier, preferably during or before the first custom fusion HLO (cross-HLO optimization). Finding opportunities for optimization across HLOs is especially difficult with coarse grained tools, as it is not clear how different operations interact from aggregated statistics. Here we showed a simple example of how a fine grained view into the microarchitectural utilization of the accelerator can make performance analysis much easier, and we now give an overview of existing profilers and their capabilities.

PMU. Vendors typically provide hardware support to record performance events, the performance monitoring unit (PMU), along with profilers that leverage it. The PMU has a number of registers (performance counters) which can record performance events such as cache hits, or collect statistics like cycles executed within a function. Profilers that use the PMU include Nsight Systems [40], nvprof [39], Intel VTune [26], AMD ROC-profiler [5] and Google Tensorboard [22]. More specifically, Nsight offers utilization metrics at the level of a CUDA kernel and Tensorboard at the level of a HLO. These tools are usually lightweight and rely on the PMU to provide performance metrics, which is constrained both in granularity, because of its limited buffer size, and in variety, as it typically allows recording a limited number of performance counters simultaneously [44].

Program counter sampling. Besides the PMU, some vendors (e.g., Nvidia) provide program counter (PC) sampling and stall attribution. PC sampling requires additional hardware support to sample the program counter during execution. If the code is stalling at the time of the sample, the hardware may also provide the reason for the stall (e.g., memory stall, synchronization stall). Tools like CUPTI [38], VTune [26], HPCToolkit [3] and DrGPU [23] use PC sampling to collect stack traces and their stall reasons, coalesce them and suggest optimization strategies. While this approach can be useful for finding opportunities to improve performance, it has a number of disadvantages. First, it requires additional hardware to sample running code and attribute stall reasons correctly, which is not available in all accelerators, including ours. Second, while PC sampling can pinpoint various inefficiencies like memory transfer stalls, it does not show the root cause instruction. For instance, a stalled instruction waiting for a memory transfer is the symptom, not the cause. Finally, it does not provide any information on how well the hardware is utilized, as it focuses on stalls and not on hardware utilization.

Instrumentation. Another category of tools use binary instrumentation to gain performance insights on a more microscopic level, albeit with great overhead. Binary instrumentation engines such as Nvidia NVBit [60], SASSI [43], Sanitizer API [42], Intel GTPin [25] and LLVM [36] can change code at a very low level, so that every instruction can be recorded. This approach is used to analyze the behavior of the software at very high detail. For example, VALUEEXPECT [62] traces every load and store instruction to discover inefficient patterns in the data, like hidden sparsity or repeated computation. CUDAAdvisor [54] traces memory accesses to reveal memory metrics such as reuse distance. While these approaches are useful to explore the inner workings of a program running on an accelerator, they offer no insight on how well the program is using the underlying hardware, as instrumenting the program totally changes its execution characteristics which renders the PMU useless. In addition, instrumentation typically requires re-compilation, which makes it hard to use in the context of a hyperscalar where compilation is complex, while code and dependencies change frequently.

Cross-cutting. Last but not least, some tools combine multiple techniques to provide a more detailed performance analysis. GPA [63] combines PC sampling with instrumentation to detect inefficient parts of the code, analyze their dependencies and suggest root causes. It is a powerful optimization tool but does not provide specific optimization suggestions (i.e. move this instruction here). In addition, it relies on hardware support for PC sampling, which is not available in every accelerator, including ours. NVidia NSight Compute [40] is the most comprehensive tool we know, analyzing CUDA kernels down to the PTX level, detecting instruction dependencies, and warning of inefficiencies. However, it analyzes only one kernel at a time, misses Nvidia SASS-level insights, and remains opaque due to its proprietary nature.

2.3 Simulation

Simulators have a rich history in both industry [18, 59] and academic literature [6, 8, 10, 21, 30, 31, 34, 37, 61]. They are an indispensable tool of the hardware design process and for microarchitectural research. First, low-level ISA simulators like gem5 [8, 37] for CPUs and GPGPU-Sim [6, 21] and AccelSim [30] for GPUs, have been used extensively to explore new hardware designs. Second, higher level simulators like Sniper [10] and Maestro [31] for CPUs and Accelergy [61] and Meta's Arcadia for GPUs [18] are a great tool to estimate end-to-end performance for different hardware architectures or system configurations. However, both of these approaches cannot produce low-level performance insights to optimize models on existing hardware. Low-level ISA simulators lack the fidelity needed to capture low-level microarchitectural inefficiencies, as they do not have access to the proprietary microarchitecture design, so they need to approximate it. On the other hand, high-level simulators estimate end-to-end performance with a much larger margin of error and cannot offer insight into how the specific hardware / software interaction is causing performance issues.

3 Design Requirements

To get the best performance out of an accelerator, it is important to have good visibility of how the software interacts with the hardware, at a fine granularity. Based on these observations, we pose the design requirements for SnR as a series of questions that an ideal low-level profiler should answer.

$\boxed{Q1}$ *Are there optimization opportunities inside and across HLO boundaries?* Higher level performance analysis tools only present aggregate metrics for each HLO, even though data dependencies often span HLO boundaries. Can a detailed analysis below this abstraction unlock new opportunities?

$\boxed{Q2}$ *What is the instantaneous utilization of individual microarchitectural units?* ML workloads require heavy matrix and vector multiplications and manipulations which need to be carefully orchestrated to achieve optimal use of the accelerator. Aggregated statistics can hide opportunities to fully utilize the accelerator hardware.

$\boxed{Q3}$ *Is our data-cache properly utilized to alleviate memory bottlenecks?* Data-caches help alleviate the memory bottleneck, and bridge the gap between relatively slow memory and incredibly fast compute. Efficient use of the data-cache is perhaps the most crucial element of achieving maximum performance.

$\boxed{Q4}$ *Can the tool provide actionable insights?* Performance analyses and visualizations can help users better understand how the machine-code interacts with the hardware and spot inefficiencies. Existing tools can often point to general causes, but cannot suggest specific actions to fix them.

In addition to these questions, we face several challenges in the context of a hyperscaler:

$\boxed{C1}$ *Use software, not hardware.* Techniques like PC sampling help in pinpointing inefficiencies, but they require hardware support that is not available in all accelerators, including ours. We want the solution to be implementable entirely in software, so that it can be applied immediately to our entire accelerator fleet.

$\boxed{C2}$ *Avoid recompilation.* Many tools that analyze performance at the lowest level commonly need model code to be compiled with special flags that instrument the resulting program. However, large ML models often have complex compilation pipelines that are slow and hard to modify. In addition, code and dependencies are constantly changing, making it harder to accurately recompile production models.

4 Design and Implementation

SnR consists of an *execution recorder*, a *replayer*, and an *analyzer* (Figure 1). The execution recorder (§4.1) uses the hardware step debugger to break in the middle of the model execution and record traces of the machine-instructions executing the ML model. The traces include the minimal architectural state and memory required to replay the execution from the mid-point of the model we started recording from. The replayer (§4.2) is a modified existing ISA-level simulator, which replays the execution trace and generates detailed raw microarchitectural metrics. Lastly, the analyzer (§4.3) takes the raw generated metrics and performs various analyses to pinpoint inefficiencies and provides a detailed view of the hardware utilization to the user.

SnR is entirely implementable in software, requiring only a step-debugger and a simulator, two pieces of software that are commonly co-developed with the accelerator ($\boxed{\text{C1}}$). Most importantly, SnR can be used to analyze any production model without requiring special recompilation ($\boxed{\text{C2}}$). We have incorporated SnR in our performance analysis workflow, which we describe in §4.4.

The rest of this section will focus on the design and implementation of the SnR system, while the next section (§5) will dive deeper into the analyses developed with SnR and how they answer the questions we posed.

4.1 Execution Recorder

The first step for SnR is to capture the code running on the accelerator, along with any architectural state required to replay that code. It is necessary to record the execution trace directly from the accelerator, rather than taking the compiler output, as the code can contain conditional executions and loops, and the contents of memory and registers are unknown at compile time.

An alternative to instruction traces might be to capture targeted performance traces using an instrumentation engine. However, this would require recompilation ($\boxed{\text{C2}}$), which we want to avoid as it is both challenging in our environment, and we want to profile our models with their production compilation settings. Instead, we take advantage of a step debugger to capture these traces.

Step debuggers. Step debuggers are a common software tool that is typically developed along with any accelerator. A step debugger uses hardware support to set breakpoints in the machine-code, which will pause the execution once the breakpoint address is hit. Users can then explore the contents of the memory, registers, or execute the next instructions one at a time (*single stepping*). Because it is such a fundamental tool, all major vendors that we are aware of offer a step debugger for their ML accelerators (e.g., NVidia CUDA-gdb [14], Cerebras CSDB [1]).

We require the step debugger to provide three simple functions: a `step` function to execute instructions one-by-one, `read_memory` and `read_register` functions to read memory and registers respectively. To use the recorder, the user needs to specify a breakpoint at a location of interest and how many instructions to record. Once the breakpoint is hit, the recorder uses the `step` functionality of the debugger to execute and record instructions one-by-one. A sketch of the recorder's logic is shown in Algorithm 1. We'll now briefly describe the intuition behind it.

Each instruction takes zero or more inputs from input registers, and can write to zero or more output registers, or modify a memory region on the accelerator (e.g., in the case of a DMA). In addition, some instructions (like DMAs) read directly from device memory. So, to accurately replay an instruction, we need to know the contents of these input registers and memory regions. To naively capture this information, we must first record the entire contents of all memory regions on the accelerator, as well as all registers that can be used as inputs to instructions. However, this can make the trace file very large. As an optimization, we recognize that we only need to store the contents of a register or memory region if those contents are *first* used as input (and not as output) by a traced instruction. This is important to avoid recording intermediate results, which will be reconstructed in the simulator and are unnecessary to capture. To avoid capturing this unnecessary information, the recorder maintains a set of each instruction's output registers and memory region addresses, and then only saves each instruction's input register contents or input memory region contents if those contents had not been modified by a previous instruction.

Algorithm 1 Execution Recorder Algorithm

$R \leftarrow \emptyset$ {Set of used registers}
$M \leftarrow \emptyset$ {Set of used memory regions}
for $instruction\_count = 1$ **to** N **do**
    Parse instruction input register IDs and memory region addresses.
    Save READ_REGISTER(r_i); $\forall$ input register $r_i \notin R$
    Save READ_MEMORY(m_i); $\forall$ input memory region $m_i \notin M$
    Parse instruction output register IDs and memory region addresses.
    $R \leftarrow R \cup r_o; \forall r_o \in$ Output register IDs
    $M \leftarrow M \cup m_o; \forall m_o \in$ Output memory regions
    Step instruction.
end for

4.2 Replayer

The second step in the SnR methodology is to replay the captured trace of the model in a simulator and capture detailed metrics about the underlying microarchitecture of the accelerator. This is realized by the *replayer*, a component based on an existing ISA-level simulator for the accelerator. Hardware simulators, often called Golden Reference Models (GRMs), are an artifact of the integrated circuit design process, where they help validate hardware design decisions before the final production [24, 48, 57]. The SnR replayer repurposes this artifact of the design process for performance analysis, by extending its capabilities to capture metrics. We add a component to the GRM simulator, the *performance tracker* which tracks metrics of interest. The performance tracker is passed as a dependency of the simulation, and registers callback functions with the modelled architectural components. On performance events, such as a memory read, the GRM executes the callback to the performance tracker, which records the event. When loading a trace, we first modify the memory and register state in the simulator to match the starting condition in our trace, then resume the execution of the recorded instructions. The performance tracker collects performance event information as the trace runs, which is finally saved to a metrics file for further analysis.

Implementing the replayer within an existing hardware simulator presents unique challenges due to the complexity and multithreaded nature of its large codebase. Simply overwriting memory or register states at arbitrary points could lead to invalid or inconsistent architectural states. To reliably load and replay traces, we introduce a new machine-code command specifically designed for trace loading. This command is implemented comprehensively across the simulator's parser and processor modules, allowing precise overwriting of instruction memory and relevant architectural states with trace data. To verify the correctness of our implementation, we cross-reference the simulator's execution sequence against the trace's instruction pointer sequence. Any divergence between these sequences signals a potential bug, ensuring robust and accurate trace replay.

4.3 Analyzer

The third step in the SnR methodology is to analyze the captured events and metrics from the replayer. Our analysis framework is extensible and can examine any microarchitectural component which is modeled in our GRM Simulator. We briefly describe three sample analyses here, with full detail and results in our evaluation (§5).

The focus of our first analysis is the DMA subsystem. Memory bandwidth is one of the most precious accelerator resources and for this reason we want to understand how the model interacts with system memory at a fine-grained level ($\boxed{\text{Q3}}$). Using the replayer's data, we construct a timeline view of all memory transfers along with their stalls. Users can leverage the information from this analysis to pinpoint inefficient DMAs. This analysis also reveals optimization opportunities across HLOs, as it is often hard to hide the DMA latency within a single HLO, but possible by looking further ($\boxed{\text{Q1}}$).

Our second analysis focuses on analyzing the fine grained compute and memory utilization. Providing instruction-by-instruction utilization information can help developers reason about resource availability at every step of execution ($\boxed{\text{Q2}}$) and debug low utilization of the hardware.

Finally we add an extension to SnR to track instruction dependencies of the machine-code code in each trace. SnR can track all accesses to registers and memory locations in the simulator, capturing the data dependencies across instructions. We can then use this dependency information to automatically suggest which instructions can be issued earlier ($\boxed{\text{Q4}}$). We elaborate on how these analyses can be combined and provide valuable insights in §5.

4.4 Workflow

Finally, we describe where SnR fits in the performance analysis workflow. This workflow is a result of our experience using SnR to analyze the performance of several ML models. It consists of 1) classic **profiling** to gather regions of interest, 2) **recording** a trace, 3) **replaying** the trace in the simulator, and lastly, 4) **analyzing and visualizing**. We first collect some initial profiling with our in house performance profiling tool, which has similar functionality to Tensorboard [22]. This gives us a coarse-grained overview of performance and where potential bottlenecks might be. We note the instruction addresses of any areas of interest to record with our execution recorder (§4.1). To record the execution trace, we launch the execution recorder on the accelerator host machine, which attaches to a local accelerator running the model and sets a breakpoint for the target instruction address. Once the breakpoint is hit, the recorder takes over and creates execution traces, which are then saved remotely. These traces are typically 100-600k instructions long, which corresponds to a couple of model layers. These traces then serve as the input to the replayer (§4.2), which replays the trace and outputs performance metrics. Finally, the raw metrics are fed into the analyzer to perform the various performance analyses.

Note that so far we have been recording models in a sandbox environment, as setting a breakpoint in a customer-facing model would introduce unacceptable latency. However there is no technical restriction that keeps us from attaching the recorder to an ML model running in production.

5 Evaluating Models With SnR

Armed with SnR, we can now find out the answers to our research questions (§3). First, SnR's DMA analysis (§5.2) reveals that our models frequently incur unnecessary stalls waiting for memory transfers to complete, because of ineffective DMA scheduling. Second, SnR's microarchitectural utilization analysis helps users debug issues that aggregate

Model	#Params	#Accel	Speedup
LLM-Small	<10b	N	1.0%
LLM-Medium	10b-100b	4N	4.1%
LLM-Big	>100b	16N	0.14%

Table 2: Model descriptions. We evaluate three important in-house LLMs.

metrics overlook, such as identifying the root cause of low compute utilization (§5.3.1). Finally, SnR enables the user to reason about possible optimizations, by providing detailed information about the data-cache utilization (§5.3) and the dependencies of each instruction (§5.4). By assembling a complete view of the system, SnR is even able to automatically suggest optimizations via instruction re-orderings.

5.1 Experimental Setup

We use SnR to investigate inefficient operations on three of our most important in-house LLMs, shown in Table 2. Each model is compiled with our in-house compiler using production configurations and deployed in the same topology as on production machines.

5.2 DMA Stall Analysis

We illustrate SnR's effectiveness by describing a set of optimizations found using SnR within the DMA subsystem of our accelerator. Memory size and bandwidth are critical for modern ML accelerators, especially as larger models become increasingly memory-bound. To maximize memory utilization, we use SnR to analyze DMA performance and uncover inefficiencies ($\boxed{\text{Q1}}$, $\boxed{\text{Q3}}$).

Background on DMAs. DMAs are a mechanism used by ML accelerators to transfer data between high-bandwidth memory (HBM) and the various caches. In our accelerator, DMAs are performed by an asynchronous DMA engine. They are controlled with two simple machine-code instructions as shown in Figure 3. ISSUE will start a memory transfer and WAIT will block execution (sometimes stalling) until the transfer completes. Figure 4 shows the lifetime of a DMA. Once issued, a delay occurs before the command reaches the DMA engine for processing and queuing. This delay, called base latency (T_b), is constant and non-accumulative: when multiple DMAs are issued in parallel, their base latencies are fulfilled simultaneously. Once the base latency is fulfilled, the DMA waits until it reaches the start of the engine queue and starts transferring data between the various memories. This delay is called the transfer latency, or T_t, and depends on the available bandwidth of the memories involved, the contention with other DMAs and the size of the transfer. Data transfer on a single link (source-destination pair) cannot be parallelized. Given the above, we can now examine the three scenarios that can happen for a DMA, as shown in Figure 4. The first case is when the WAIT instruction comes before the base latency is fulfilled. In that case, the DMA incurs stalls first because of the base latency, pictured as green, and then because of the transfer latency, pictured as purple. The second case is when the WAIT instruction comes after the base latency is fulfilled and the incurred stalls are only because of the transfer latency. The third case is when the WAIT comes after the DMA has completed. In that case, we say the DMA has "slack", as the WAIT can be moved earlier. Slack is denoted in gray with a cross pattern.

ML Model Machine Code

ISSUE ⟨dma settings⟩
⟨other instructions⟩
WAIT ⟨dma_id⟩
⟨instruction using transferred memory⟩

Figure 3: Example machine-code instructions for handling memory transfers in an ML model. The ISSUE command starts the DMA, while the WAIT command blocks until it is complete and is typically inserted close to the command that needs the memory. The compiler can hide the DMA latency by inserting instructions between ISSUE and WAIT.

The DMA subsystem is a common source of stalls because ISSUE and WAIT instructions can be difficult to properly schedule. In order to avoid incurring stalls, the compiler must insert enough instructions between each ISSUE and WAIT, so that it can hide the DMA's latency. This could be very difficult or impossible, especially within the boundaries of a single HLO. Transfer stalls are also difficult to predict since transfers may be predicated, exist across multiple chips, and share limited bandwidth resources. We differentiate between these two types of stalls in our analysis to help end users understand and remedy each scenario.

To construct the DMA analysis, SnR captures DMA ISSUE and DMA WAIT in the replayer as it replays the trace. Then, given these events and the specifications about the transfer speeds of various memories, SnR can simulate these transfers and flag which ones will stall the program. SnR then plots the transfers in a timeline plot, highlighting the stalled parts. The goal of this analysis is to visualize DMAs in an intuitive way and clearly show areas of potential improvements.

5.2.1 Collective Operations Optimization. Collective operations (like All-Gather, All-To-All, etc.) are fundamental operations for distributing a model across accelerators [15, 45, 52].

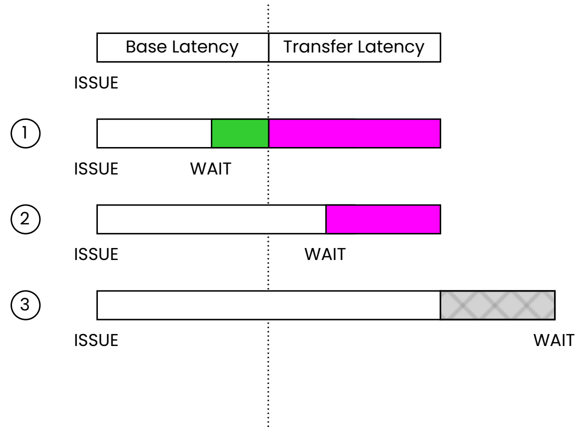


Figure 4: Lifetime of a DMA. A DMA is comprised of a base latency (constant) and a transfer latency (variable). DMAs begin with the ISSUE command. If the DMA has not completed by the time the WAIT command executes, the accelerator will stall until the DMA completes. We highlight three distinct scenarios. (1) WAIT comes before the base latency is fulfilled. The DMA incurs stalls first because of the base latency (green) and then because of the transfer latency (purple). (2) WAIT comes after the base latency but before the transfer latency is fulfilled. The DMA incurs stalls only because of the transfer latency (purple). (3) WAIT comes after the DMA finishes. The time between the DMA completion and the WAIT is slack (gray cross pattern).

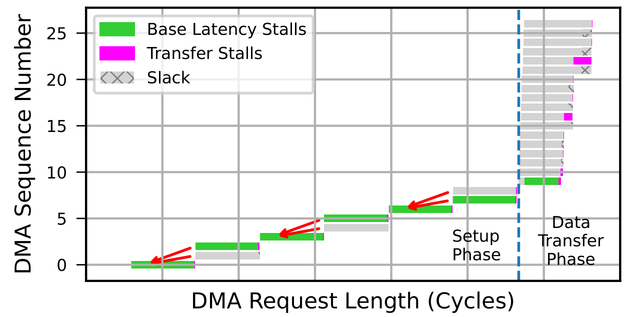


Figure 5: All-Gather DMA Pattern. Memory accesses are performed in two phases, the setup phase, and data transfer phase. Dependencies for the setup phase are shown with red arrows. These dependencies were manually discovered by reading the machine-code, a laborious process.

In this section, we describe our experience using SnR to profile and optimize an important communication collective, All-Gather, reducing its runtime by 15%.

We began our analysis by using our existing in-house state-of-the-art profiling tool to look at one of our most used models (LLM-Small). We observed that during generation for this model, the All-Gather operation was 13.3% of total runtime. Our existing tool also reported that roughly 40% of All-Gather execution time was spent stalling on DMAs. Beyond this information, we did not know what memory channels these stalls were coming from, what data was being transferred, or whether the stalls could be eliminated.

Using SnR, we capture a trace of 100,000 instructions for LLM-Small, and replay it on our modified ISA-simulator. The DMA stall analysis in Figure 5 shows most stalls happen in the first nine DMAs. These stalls are mainly due to base latency (shown in green), not transfer latency (shown in purple), as the data transferred is small. The transfers go from HBM to the data-cache. We identify two optimization strategies. The first strategy is to issue these DMAs in parallel, if possible, since base latencies can be parallelized in our DMA engine. Manual dependency analysis of the machine-code shows the DMAs form three groups. Each group has an initial DMA, followed by two dependent DMAs. Issuing the three initial DMAs in parallel, then the remaining six in parallel, would reduce stalls by a factor of three. The second strategy requires more knowledge about how the All-Gather is implemented. The first nine DMAs are part of the operation setup phase, in which the addresses of other nodes in the topology are loaded. These addresses are then used as destinations for the DMAs in the subsequent data transfer phase. Since these addresses typically use a small amount of memory, especially with smaller models, our insight is that they could be permanently pinned in the data-cache. This would completely remove the need for the setup phase DMAs, eliminating the stalls they incur. However in larger models, where topology sizes are larger, pinning the node addresses in memory may consume too much data-cache memory.

After concluding our analysis, we worked with internal developers to optimize the collective. Our internal compiler team opted to implement the second approach (pinning the node addresses in memory), because of its lower implementation complexity. The optimized code reduced the runtime of the All-Gather operation by 15%, as shown in Figure 6. In terms of end-to-end runtime, it improved generation latency by 4% in LLM-Medium and 1% in LLM-Small, as shown in Table 2. LLM-Big's improvement was a smaller 0.14%, highlighting the trade-off of the second approach. We are in the process of implementing our first approach (parallelizing setup phase DMAs), which is expected to perform better on larger models.

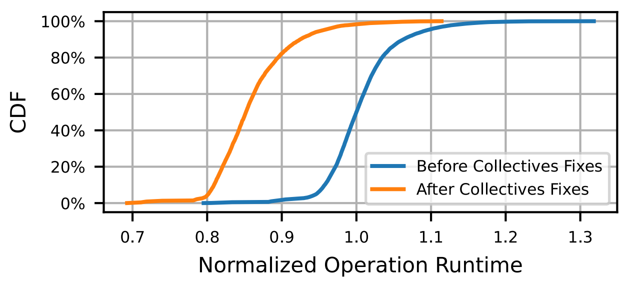


Figure 6: Reduction in runtime for the All-Gather collective operation after eliminating unnecessary DMAs. Runtime is normalized to the median value before optimization.

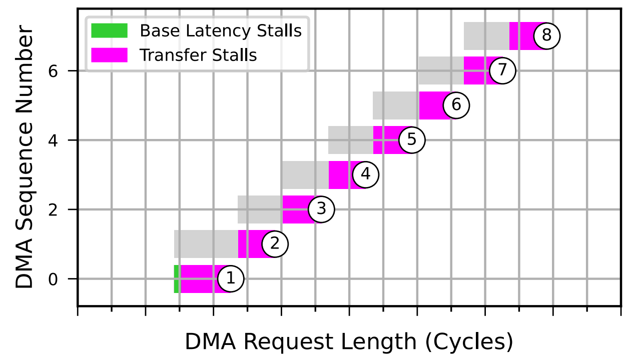
In conclusion, we used SnR to diagnose and optimize several important collectives operations in our in-house LLMs. Our existing state-of-the-art internal tools could only tell us that these operations were stalling, but could not provide the actionable insights that were obvious with SnR. Because these collectives operations are so crucial to LLM performance, our optimization lead to a realized improvement in end-to-end performance for our models. Every percent improvement in model serving leads to significant savings in total cost to run our models.

5.3 Detailed Microarchitectural Utilization

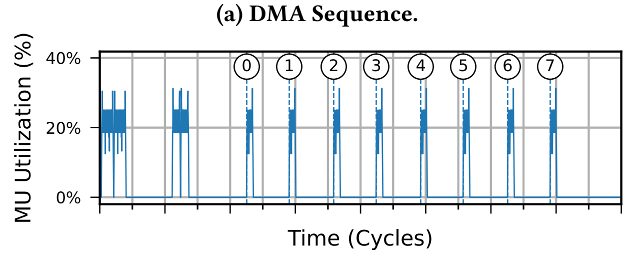
ML accelerators include an assortment of compute units (e.g., matrix unit, vector unit, scalar unit) and memories (e.g., HBM, data-cache). Underutilization in any of these units is an opportunity for further optimizations. Existing tools like Google's Tensorboard provide coarse-grained metrics about higher level functions. While an aggregated utilization is useful as a measure of performance, it does not give developers a full picture of how the accelerator is used or, more importantly, what could be changed. In this section, we show how SnR's detailed hardware utilization analysis enabled us to improve the runtime of a compute-heavy operation by 70%.

5.3.1 Compute Unit Utilization. To aid users in maximizing utilization of the compute units in an accelerator, we would like to provide them with a view of how the model interacts with the hardware at very high detail. Thanks to its use of a hardware simulator, SnR can reveal how well the model utilizes the compute units at single cycle granularity. To understand how this can assist users, we provide an example of debugging low utilization of the matrix multiplication unit (MU) in a real model[1].

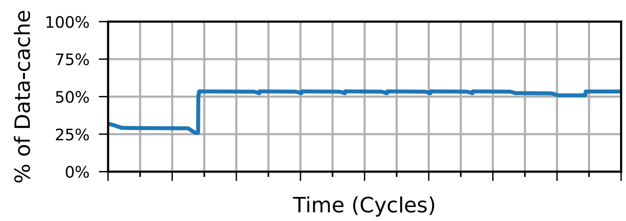
[1]Examples of matrix multiplication units include the MXU for Google TPUs and the TensorCore for Nvidia GPUs. For brevity, we'll just call it MU.



(a) DMA Sequence.



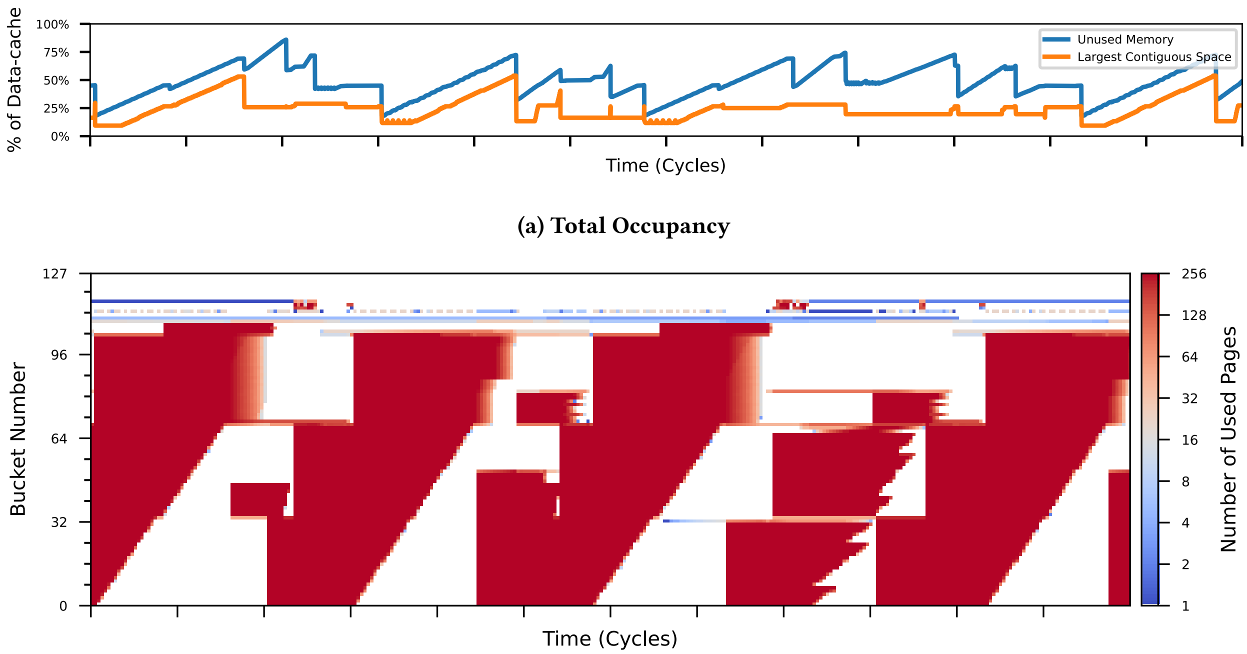
(b) Matrix Multiplication Utilization



(c) Data-cache Utilization

Figure 7: SnR can make it easy to diagnose low matrix multiplication utilization. We annotate the completion of each DMA, and the corresponding cycle on the MU utilization (Figure 7b). We see that the matrix multiplier is waiting on data transfers, resulting in low utilization. At the same time, the data-cache has enough is not fully utilized, so we might be able to prefetch more data.

Figure 7 shows SnR's analysis for a model section that exhibits low compute utilization. Our high level tool simply reports a low compute unit utilization, making it hard to dig deeper into the root cause. In contrast, SnR's analysis directly points out the problem. First, Figure 7b shows the utilization of the matrix multiplication unit, the main workhorse of the accelerator. We see an intermittent pattern of computation and data transfer, indicating that the problem is caused by some other component bottlenecking the flow to the MU. We can immediately spot the problem by using SnR to look at



(b) Fragmentation Heatmap

Figure 8: Figure 8a shows the percentage of data-cache space which is unused, and the percentage which is unused and contiguous. The largest contiguous region is consistently much smaller than the total unused portion. Figure 8b is the data-cache fragmentation analysis. The total memory space is divided into pages, where each read and write is tracked. Pages are then grouped into "buckets" of 256 pages. Each bucket is a horizontal line in the plot and its color denotes the number of used pages in the bucket for each cycle. We see considerable fragmentation of our data-cache, which is critical for performance.

the memory subsystem (Figure 7a). The high utilization coincide with DMA completions and the zero utilization spots coincide with the DMA stalls. The current DMA schedule is not feeding data to the matrix unit fast enough.

We see how an issue that was opaque before now becomes clear to the user, who can reason about optimizing it. A straightforward path from this finding would be to prefetch more aggressively before the chain of matrix operations begins. However, this requires reasoning about the availability of data-cache at that point in time. Because SnR also allows us to track this information, we can investigate data-cache utilization (plotted in Figure 7c), and we see that there is sufficient room in our data-cache to prefetch data. Finally, by manually analyzing the machine-code code, we conclude that these DMAs dependencies allow them to be issued earlier. With this information, we consulted with our internal compiler team to implement the prefetching, lowering operation runtime by 70% and increasing MU utilization.

5.3.2 Memory and Data-Cache Utilization. In our previous example, we showed how examining the overall utilization of the data-cache can help users make decisions about data prefetching. However, SnR can analyze the data-cache at a much finer grained level, enabling developers to not only look at overall utilization but also cycle by cycle memory usage and fragmentation.

Background on the data-cache. The data-cache is a relatively small, fast cache that stores vectors that can quickly be loaded into the matrix multiplication units. The data-cache bridges the slow HBM with the fast compute capability of the accelerator, making it crucial to model performance. The data-cache and memory allocations are static and managed by the compiler during the compilation process. However, because of predication and loops, it is hard to reason about the exact data-cache use at each point of the program. Moreover, a challenge with these memory allocation techniques is fragmentation. It is not enough to have a large on-chip memory, there also must be a contiguous region available as a target for memory transfer.

By using a hardware simulator, SnR can precisely track usage of the data-cache by tracking reads and writes on data-cache pages. A page is counted as "used" if it is read by an instruction after being written. If a page is brought in from HBM to the data-cache and the page is not read in subsequent instructions then that page is tracked as "unused". We are only able to track these categories as long as our simulation runs, so theoretically if a page is read after the simulation ends, our analysis would miss that read. However, our simulations typically run for hundreds of thousands of cycles. A memory transfer should be read within that period, and if a page is transferred but not used for tens or hundreds of thousands of cycles, that is an inefficiency in the model.

The data-cache fragmentation analysis is shown for a production LLM in Figure 8. The total data-cache space is divided into 128 "blocks", each of which contain 256 pages. The heatmap information shows how many pages were used within each block. In Figure 8a, we show both the percentage of unused memory, and the percentage of memory occupied by the largest contiguous region. The total portion of unused memory varies between approximately 20% and 80%, however the largest contiguous unused space is consistently less than this. In this trace, the median unused space is 47% of the total space, and the median contiguous block is only 25% of the total space. These results show that the data-cache space can be used more efficiently.

Beyond model developers, this analysis provides compiler engineers with a unique insight into data-cache utilization at the lowest level. Memory allocation is done by the compiler and relies on finely-tuned heuristics. Compiler engineers often evaluate memory allocation techniques on models that quickly become outdated and don't have a way to gain insight into the way these heuristics are affecting the latest production models. SnR offers a new way of debugging these complex compiler subsystems that wasn't possible before.

5.4 Dependency Analysis

In the previous examples, we demonstrated how SnR's fine-grained analysis can identify inefficiencies. However, it's difficult to differentiate between poor performance which is actually unavoidable due to dependencies, and poor performance which can be eliminated through alternative schedules. So far, we tackled the issue by manually going through the machine-code and discovering dependencies, which is a labor intensive and error prone process. Automating the discovery of these dependencies would not only alleviate the need for users to manually analyze complex machine-code code but also enable SnR to autonomously propose optimizations through alternative instruction scheduling. In this section, we present how SnR implements dependency tracking and leverages this information to enhance the effectiveness of our existing analyses.

There are two ways to do dependency analysis: static and dynamic. While static analysis can be done by the compiler, a static implementation will be missing key information about runtime behavior, such as predication, which limits its usefulness. SnR implements dynamic dependency analysis, using the GRM simulator to discover instruction dependencies as it replays the trace. Instruction dependencies can be registers (e.g., for arithmetic instructions), data-cache locations (e.g., for load/store instructions) or even HBM locations (e.g., for DMA instructions). To discover instruction dependencies, SnR traces all accesses to registers, data-cache and HBM in the simulator. When an instruction reads from a register or piece of memory that was previously written by another instruction, then those instructions have a dependency. With this information, SnR knows the exact dependency graph between instructions.

One use case for this analysis is to reason about rescheduling DMAs. The way DMAs are typically issued is to load some information about the DMA metadata from HBM. This metadata is put in registers and may be transformed before being used as input to the DMA instruction. These transformations are typically lightweight and they occur close to the DMA issue instruction. If we only track when a DMA instruction's immediate input registers as dependencies, then in many cases the dependencies will be flagged as "fulfilled" immediately preceding the DMA. However, our insight is that as long as any transformations on the DMA metadata are lightweight, we can move the DMA ISSUE and any transformation instructions as early as when the data is ready in the data-cache. We call this method of dependency accounting "relaxed", illustrated in Figure 9. In the conservative model, an instruction depends on its input registers. In the relaxed model, an instruction depends on the DMA that brought its inputs, propagated through dependencies, to memory. For DMA instructions, it is better to use the relaxed model to reason about their dependencies.

Using the generated dependency graph, SnR can display this information to the user, allowing them to reason about alternative schedules that eliminate stalls by reordering instructions. We illustrate the power of SnR's dependency tracking by applying it to the example shown above, where we had to manually discover dependencies by reading machine-code code. Figure 10 shows the DMA analysis with dependency information overlayed. Dependencies are visualized as "backtails" for each DMA, indicating how far back we can start it based on the relaxed model. The user can see that all DMAs can be issued earlier in time and thus prefetching is a viable optimization strategy.

Autonomous Optimization Suggestions. By combining the various analyses presented so far, SnR can reason about inefficient DMAs, check if alternative schedules are possible

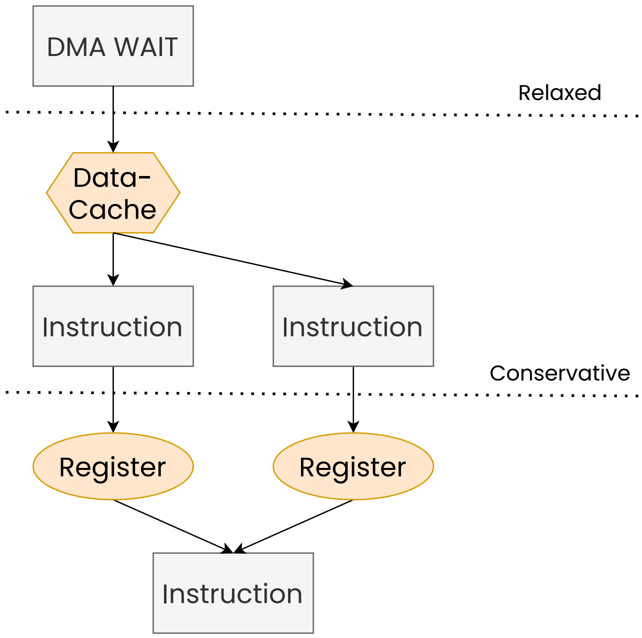


Figure 9: Different models of dependency tracking. In the conservative models, an instruction depends on the immediate instructions that shape its input. In the relaxed model, dependencies are propagated until the dependent memory is in the data-cache.

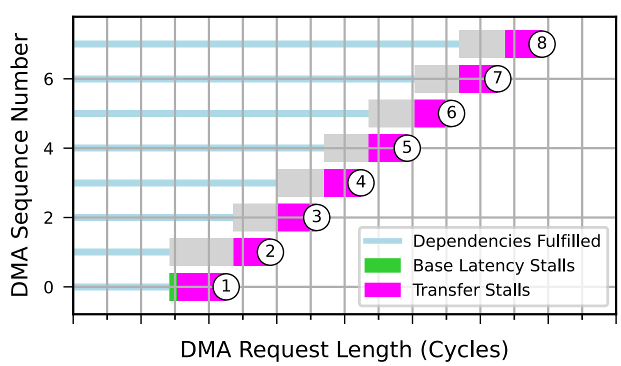


Figure 10: Example dependency analysis with SnR, showing the DMA analysis of section 5.3.1 with dependency information overlayed as "backtails". Thanks to that, the user can immediately conclude the DMAs can be issued earlier, without needing to manually analyze the machine-code.

and flag them for further investigation. An outline of this logic is shown in Algorithm 2. We examine each DMA separately. If the DMA does not have stalls, we have nothing to suggest. Otherwise, we first check how far we can push the DMA based on its dependencies. Specifically, we would like to push it back at least as many instructions as stall cycles, to eliminate those stalls. Second, we check if the data-cache has enough contiguous memory available to accommodate the DMA, using the analysis from Section 5.3. If these two conditions are true, then we suggest a DMA reordering to eliminate stalls. This algorithm successfully discovers and suggests the optimizations from the two examples we showed.

Algorithm 2 Automatic optimization suggestion algorithm

for each DMA operation **do**
    if not *stalled* **then**
        continue
    end if
    push_limit ← DMA dependency distance
    memory_available ← (contiguous data-cache ≥ *dma* size)
    if push_limit > stall_duration **and** memory_available **then**
        Suggest reordering
    end if
end for

In conclusion, dependency analysis enables SnR to both aid the user in reasoning about what optimizations are possible, as well as automatically suggest them.

5.5 SnR's Overhead

Since SnR traces every instruction in an ML model using a step debugger, the overhead can be high. Thus, we do not use SnR in production but in a sandbox where temporary performance degradation is acceptable. A key concern is whether SnR captures enough of the model's execution or zooms in too narrowly, missing context.

Currently, SnR records 400 us of model execution, covering multiple layers. The end-to-end process takes around 2 minutes, but can easily be reduced to under a minute. Given the model's self-similarity, capturing a few layers across different modes (i.e. prefill, decode) provides sufficient performance insights. Overall, SnR's overhead is manageable, with potential for further optimization.

6 Conclusion

The exploding demand for ML accelerators mandates squeezing the most out of the existing infrastructure. SnR is a novel approach for analyzing performance by bridging the gap between high level semantics, machine code, and micro-architecture - revealing new opportunities in an already highly-optimized environment. Other hyperscalers can reproduce SnR approach on their custom accelerators, given fairly basic prerequisites: a step debugger and an ISA simulator. The yields of optimizing large ML workloads will enable developing more sophisticated models while significantly reducing both capital and power requirements.

References

[1] 2023. Debugging with Cerebras SDK. https://sdk.cerebras.net/debug/debugging#csdb-debugger. Accessed: 2024-10-29.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 685–701.

[4] Inc. Advanced Micro Devices. 2024. AMD Official Website. https://www.amd.com Accessed: 2024-11-26.

[5] AMD 2024. *AMD ROCm Profiler*. AMD. https://rocm.docs.amd.com/projects/rocprofiler/en/docs-5.0.2/ Accessed on: October 15, 2024.

[6] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. doi:10.1109/ISPASS.2009.4919648

[7] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 430–449. https://proceedings.mlsys.org/paper_files/paper/2022/file/37385144cac01dff38247ab11c119e3c-Paper.pdf

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. doi:10.1145/2024716.2024718

[9] Bloomberg Intelligence. [n. d.]. Generative AI Races Toward $1.3 Trillion in Revenue by 2032. https://www.bloomberg.com/professional/insights/data/generative-ai-races-toward-1-3-trillion-in-revenue-by-2032/. Accessed: 2024-11-22.

[10] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2063384.2063454

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[12] Intel Corporation. 2024. Intel Official Website. https://www.intel.com Accessed: 2024-11-26.

[13] NVIDIA Corporation. 2024. NVIDIA Official Website. https://www.nvidia.com Accessed: 2024-11-26.

[14] NVIDIA Corporation. 2025. CUDA-GDB. https://developer.nvidia.com/cuda-gdb Accessed: 2025-02-14.

[15] NVIDIA Corporation. 2025. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl Accessed: 2025-01-21.

[16] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=mZn2Xyh9Ec

[17] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 16344–16359. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf

[18] Facebook Engineering. 2023. Arcadia: End-to-End AI System Performance Simulator. https://engineering.fb.com/2023/09/07/data-infrastructure/arcadia-end-to-end-ai-system-performance-simulator/.

[19] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Aredestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammed Al-Sanabani, Ana Zehtabioskuie, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Harikrishna Reddy, Prahlad Venkatapuram, and Alexis Bjorlin. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 80, 13 pages. doi:10.1145/3579371.3589348

[20] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. https://mlsys.org/Conferences/doc/2018/146.pdf

[21] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 407–420. doi:10.1109/MICRO.2007.30

[22] Google 2024. *Google TensorBoard*. Google. https://www.tensorflow.org/tensorboard Accessed on: October 15, 2024.

[23] Yueming Hao, Nikhil Jain, Rob Van der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. 2023. DrGPU: A Top-Down Profiler for GPU Applications. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (Coimbra, Portugal) *(ICPE '23)*. Association for Computing Machinery, New York, NY, USA, 43–53. doi:10.1145/3578244.3583736

[24] Ze He and Xiaowen Chen. 2017. Design and implementation of high-speed configurable ECC co-processor. In *2017 IEEE 12th International Conference on ASIC (ASICON)*. 734–737. doi:10.1109/ASICON.2017.8252580

[25] Intel Corporation 2024. *Intel GTPin: Graphics Program Instrumentation.* Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/tool/gtpin.html Accessed on: October 15, 2024.

[26] Intel Corporation 2024. *Intel VTune Profiler.* Intel Corporation. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html Accessed on: October 15, 2024.

[27] JAX Developers. [n. d.]. Pallas: A Kernel Language for JAX. https://jax.readthedocs.io/en/latest/pallas/index.html. Accessed: 2024-11-22.

[28] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. doi:10.1145/3579371.3589350

[29] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. https://arxiv.org/pdf/1704.04760.pdf

[30] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-sim: an extensible simulation framework for validated GPU modeling. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 473–486. doi:10.1109/ISCA45697.2020.00047

[31] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29. doi:10.1109/MM.2020.2985963

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[33] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[34] Jonathan Lew, Deval A. Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. 2019. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 151–152. doi:10.1109/ISPASS.2019.00028

[35] Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. 2022. TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment. *IEEE Micro* 42, 5 (Sept. 2022), 9–16. doi:10.1109/MM.2022.3178068

[36] LLVM Foundation 2024. *LLVM Project.* LLVM Foundation. https://llvm.org Accessed on: October 15, 2024.

[37] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[38] NVIDIA 2020. *NVIDIA CUPTI.* NVIDIA. https://developer.nvidia.com/cupti Accessed on: October 15, 2024.

[39] NVIDIA 2020. *NVIDIA nvprof.* NVIDIA. https://docs.nvidia.com/cuda/profiler-users-guide/ Accessed on: October 15, 2024.

[40] NVIDIA 2024. *NVIDIA Nsight Compute.* NVIDIA. https://developer.nvidia.com/nsight-compute Accessed on: October 15, 2024.

[41] NVIDIA 2024. *NVIDIA Nsight Systems.* NVIDIA. https://developer.nvidia.com/nsight-systems Accessed on: October 15, 2024.

[42] NVIDIA Corporation 2024. *NVIDIA Compute Sanitizer API.* NVIDIA Corporation. https://docs.nvidia.com/compute-sanitizer/SanitizerApiGuide/index.html Accessed on: October 15, 2024.

[43] NVIDIA Corporation 2024. *SASSI: A Low-Level GPU Instrumentation Framework.* NVIDIA Corporation. https://github.com/NVlabs/SASSI Accessed on: October 15, 2024.

[44] NVIDIA Corporation. 2025. CUPTI Documentation - Multi-Pass Collection. https://docs.nvidia.com/cupti/main/main.html#multi-pass-collection Accessed: 2025-02-17.

[45] OpenXLA Project. [n. d.]. OpenXLA: An Open Ecosystem for Machine Learning Infrastructure. https://openxla.org/. Accessed: 2024-11-22.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library.* Curran Associates Inc., Red Hook, NY, USA.

[47] OpenXLA Project. [n. d.]. StableHLO: A Portability Layer for ML Frameworks and Compilers. https://openxla.org/stablehlo. Accessed: 2024-11-22.

[48] Hemant Rotithor. 2000. Postsilicon validation methodology for microprocessors. *IEEE Design & Test of Computers* 17, 04 (2000), 77–88.

[49] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[50] Amazon Web Services. [n. d.]. AWS Inferentia. https://aws.amazon.com/ai/machine-learning/inferentia Accessed: 2024-11-22.

[51] Amazon Web Services. [n. d.]. AWS Trainium. https://aws.amazon.com/ai/machine-learning/trainium Accessed: 2024-11-22.

[52] Amazon Web Services. 2025. Collective Communication. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-features/collective-communication.html Accessed: 2025-01-21.

[53] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems.* https://openreview.net/forum?id=tVConYid20

[54] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. CUDAAdvisor: LLVM-based runtime profiling for modern GPUs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 214–227. doi:10.1145/3168831

[55] Anton Shilov. 2025. DeepSeek's AI breakthrough bypasses industry-standard CUDA for some functions, uses Nvidia's assembly-like PTX programming instead. https://www.tomshardware.com/tech-industry/artificial-intelligence/deepseeks-ai-breakthrough-bypasses-industry-standard-cuda-uses-assembly-like-ptx-programming-instead Accessed: 2025-02-17.

[56] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[57] Aakash Tyagi, Addison Crump, Ahmad-Reza Sadeghi, Garrett Persyn, Jeyavijayan Rajendran, Patrick Jauernig, and Rahul Kande. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. arXiv:2201.09941 [cs.CR] https://arxiv.org/abs/2201.09941

[58] Jonathan Vanian and Kif Leswing. 2023. ChatGPT and Generative AI Are Booming, but the Costs Can Be Extraordinary. *CNBC* (2023). https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html Published: Mar 13, 2023, Updated: Apr 17, 2023, Accessed: 2025-02-10.

[59] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 868–880. doi:10.1109/HPCA51647.2021.00077

[60] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. doi:10.1145/3352460.3358307

[61] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, David Z. Pan (Ed.). ACM, 1–8. doi:10.1109/ICCAD45719.2019.8942149

[62] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: exploring value patterns in GPU-accelerated applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 171–185. doi:10.1145/3503222.3507708

[63] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU Performance Advisor Based on Instruction Sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 115–125. doi:10.1109/CGO51591.2021.9370339