Hardware Support for Concurrent Detection of Multiple Concurrency Bugs on Fused CPU-GPU Architectures

Weihua Zhang, Shiqiang Yu, Haojun Wang, Zhuofang Dai, and Haibo Chen, *Senior Member, IEEE*

Abstract—Detecting concurrency bugs, such as data race, atomicity violation and order violation, is a cumbersome task for programmers. This situation is further being exacerbated due to the increasing number of cores in a single machine and the prevalence of threaded programming models. Unfortunately, many existing software-based approaches usually incur high runtime overhead or accuracy loss, while most hardware-based proposals usually focus on a specific type of bugs and thus are inflexible to detect a variety of concurrency bugs. In this paper, we propose Hydra, an approach that leverages massive parallelism and programmability of fused CPU-GPU architectures to simultaneously detect multiple concurrency bugs in threaded software, including data race, atomicity violation and order violation. Hydra extends contemporary fused CPU and GPU by introducing two modules: 1) a trace collecting module (TCM) that instruments and collects program behavior on CPU; 2) a trace preprocessing module (TPM) that processes and then transfers the traces to GPU for bug detection. Furthermore, Hydra exploits three optimizations to improve speed and accuracy, which includes: 1). using the bloom filter to filter out unnecessary traces; 2). avoiding eviction of shared traces; 3). comparing only last-write traces for shared data with the happens-before relation. Hydra incurs small hardware complexity and requires no changes to internal critical-path processor components such as cache and its coherence protocol, and is with about 1.1 percent hardware overhead under a 32-core configuration. Experimental results show that Hydra only introduces about 0.18 percent overhead on average for detecting one type of bugs and 0.46 percent overhead for simultaneously detecting multiple bugs, yet with the similar detectability of a heavyweight software bug detector (e.g., Helgrind).

Index Terms—Concurrency bug detection, fused CPU-GPU architecture

1 Introduction

The increasing number of cores in a single chip has made parallel programming a necessity to harness the abundant hardware resources. However, writing robust parallel software is notoriously hard, partly due to the pervasive yet hard-to-detect concurrency bugs. Such bugs, once manifest, can lead to catastrophic failures, causing not only economy loss, but also social impact [26].

Worse even, there are a variety of concurrency bug types, such as data race, atomicity violation and order violation. These bugs are usually hard to spot due to their non-deterministic nature. Hence, a number of approaches have been proposed to detect concurrency bugs. Generally, they can be divided into two categories: software-based approaches and hardware-based ones. Software-based approaches [14], [18], which instrument program code and analyze bug patterns in a software manner, may suffer from either large performance overhead or poor detection accuracy. In contrast, hardware-based approaches [4], [5], [6], [30], [31] result in better performance. However, they generally focus on only one specific type of bugs, which limits their flexibility to detect other types of bugs. As indicated in prior work [33], there may be more than one types of bugs hidden in the same program, even for some mature commercial software such as MySQL. As programmers usually have no idea about which type the bug is, using a different type of hardware detectors one by one is cumbersome and inflexible. It is thus demanding for a hardware proposal that simultaneously detects multiple types of bugs.

Currently, integrating CPUs and GPUs on a single chip [45], [46] has become increasingly popular, which opens new opportunity for bug detection. On one hand, GPU includes a lot of general-purpose computation resources, which are usually idle at debug time.[1] On the other hand, the mainstream dynamic concurrency bug detection algorithms are typically triggered by certain shared resource accesses, resulting in similar detection process. Furthermore, they usually have very good computation and data parallelism. Therefore, it is intuitive to exploit the massive computation resources and programmability of GPU for flexible concurrency bug detection.

In this paper, we propose a flexible and efficient GPU-assisted software concurrency bug detection mechanism, called Hydra, which leverages massive parallelism and programmability of GPU to simultaneously detect multiple

- *W. Zhang, S. Yu, H. Wang, and Z. Dai are with the Software School, Shanghai Key Laboratory of Data Science, and Parallel Processing Institute, Fudan University. E-mail: {zhangweihua, squ14, wanghaojun, dzf}@fudan.edu.cn.*
- *H. Chen is with the Institute of Parallel and Distributed Systems, Shanghai Jiaotong University. E-mail: haibochen@sjtu.edu.cn.*

Manuscript received 10 Mar. 2015; revised 30 Oct. 2015; accepted 12 Dec. 2015. Date of publication 24 Dec. 2015; date of current version 14 Sept. 2016. Recommended for acceptance by C. Metra.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2512860

1. The computation resource in GPU can be divided into general-purpose computation units and graphics processors. Except for 3D games, GPU only uses the graphics processors to handles displaying work. Therefore, the general-purpose units are basically idle and are available for other purposes even for basic desktop displaying.

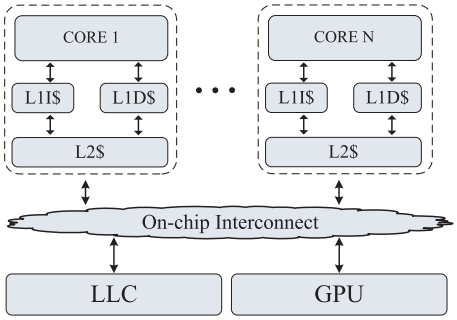


Fig. 1. Fused CPU and GPU architecture.

concurrency bugs. Hydra extends contemporary fused CPU and GPU architecture by introducing two modules to collect and transfer in-flight traces to GPU for bug detection, including: 1) a per-core trace collection module (TCM) that collects traces; 2) an added trace pre-processing module (TPM) between CPU and GPU that receives traces from TCM, stores the traces in a trace history buffer and passes the trace to GPU for detection.

Further, as concurrency bugs are only triggered by shared memory traces, Hydra exploits three optimization strategies to improve both detection speed and accuracy. First, Hydra leverages the bloom filter [29] to filter out traces of private memory accesses. Second, Hydra uses a feedback-directed approach that uses GPU execution information to boost the priority of certain shared traces in the history buffer to avoid being evicted. Finally, Hydra only compares last-write traces for the shared data with the happens-before (HB) relation in prior computation. Consequently, Hydra removes unnecessary traces in the history buffer, which brings benefits in both speed and accuracy.

By leveraging the programmability of GPU, Hydra is able to detect data race, atomicity violation and order violation, which accounts for almost all (about 97 percent) of the non-deadlock concurrency bugs [37]. For data race, Hydra uses the happens-before algorithm [21], which compares the timestamps of memory accesses to derive happens-before relations. Order violation detection is also based on the happens-before algorithm and the difference from data race detection is that order violation focuses on operation orders instead of orders of memory accesses (e.g., a file read operation should be after the corresponding file open operation) [36]. To detect atomicity violation, a typical approach (e.g., ColorSafe [31]) is assigning related variables in an atomicity region with the same color. Possible atomicity violations occur when two accesses in an atomic region with the same color are interleaved by at least one remote access with the same color. During detection, TCM collects the traces and TPM generates and stores the history information (e.g., traces, timestamps and colors), while GPU processes the history information and executes the kernels of the corresponding detecting algorithms. Hence, the detection logics executed on GPU can be easily reprogrammed, which makes it appealing to detect multiple bugs.

We have implemented a simulated version using Sniper [25] with GPGPU-Sim [38] integrated into its backend. Experimental results show that Hydra only introduces about 0.18 percent execution overhead on average for one type of bug detection at a time and 0.46 percent for simultaneous detection of the above three types of bugs under a 32-core configuration with only 1.1 percent hardware overhead on average. Furthermore, Hydra requires no modification to the critical execution paths in CPU, such as caches and coherence protocols.

In summary, this paper makes the following contributions:

- A case of reusing GPU for scalable and low-overhead concurrency bug detection.
- A flexible design for general-purpose concurrency bugs detection, including data race, atomicity violation, order violation, and simultaneous detection of multiform concurrency bugs.
- Three optimizations that significantly improve the detection efficiency and reduces performance overhead.
- A thorough evaluation showing the effectiveness and efficiency of Hydra.

The rest of the paper is organized as follows. We briefly introduce the basic GPU architecture and the concurrency bug detection and discuss related work in Section 2. We present the motivation and architecture design of Hydra in Sections 3 and 4. In Section 5, we evaluate the performance of Hydra. Finally, we draw a conclusion in Section 6.

2 Background and Related Work

To gain insight into using GPU for concurrency bug detection, we first briefly introduce the fused CPU and GPU architecture. Then, we present an overview of several bug detection algorithms and analyze the basic detection logic to uncover possible portions that are suitable to be mapped onto GPU.

2.1 Fused CPU and GPU Architecture

Typically, a GPU consists of hundreds of streaming-processor (SP) units. These SP units are grouped into streaming multiprocessors (SM). Each SM has a region of on-chip memory, which contains thousands of registers and tens of kilobytes shared memory. The on-chip memory is shared among SPs inside a SM and not visible outside SMs. Different SMs share data through off-chip global memory. Every 32 or 64 threads are grouped into a warp and multiple warps are assembled as a thread block. Each SM supports a few warps to hide memory latency. GPU provides good programmability with mature programming models such as CUDA [47] and OpenCL [42].

Due to the potentially superior performance and low power consumption, fused CPU-GPU architecture has become increasingly popular. Example architectures include Intel Ivy Bridge [45] and AMD Fusion [46]. As shown in Fig. 1, a CPU and a GPU are combined onto the same chip. They exchange data through an on-chip interconnect, which connects all on-chip modules, such as CPU cores, GPU and the last-level cache (LLC). In this paper, we will use such a design as the baseline architecture of Hydra, as it provides high bandwidth and low latency for communication between CPU and GPU.

2.2 Concurrency Bug Detection

Based on the analysis of Lu et al. [37], real-world concurrency bugs can be categorized into deadlock bugs and non-deadlock bugs, non-deadlock bugs occupy a majority of concurrency

bugs (more than 70 percent as analyzed in [37]). Non-deadlock bugs can be further divided into atomicity violation and order violation. These two kinds of bugs account for 97 percent of non-deadlock bugs. Hence, Hydra mainly targets at detecting these two types of bugs. In [37], data race is not classified as a bug pattern because there are benign races. However, Hydra also considers data race detection since a data race is prone to cause a concurrency bug [3].

Atomicity violation. Atomicity violation occurs when a code region of one thread is unserializably interleaved by another thread, and it accounts for about 65 percent of non-deadlock bugs. According to the number of variables involved in the code region, atomicity violation can be further divided into single-variable atomicity and multiple-variable atomicity violation. Lu et al. proposed AVIO to extract access interleaving invariant to detect single-variable atomicity violation [30]. Lucia et al. proposed the ColorSafe method to group related variables in an atomicity region with the same color for both kinds of atomicity violation detection [31]. Details of ColorSafe algorithm can be found in the Appendix.

Order violation. Order violation occurs when an order between two operations from different threads must be guaranteed, but programmers forget to enforce this order. This kind of bugs accounts for about one third of non-deadlock bugs. Order violation is different from atomicity violation because even if making critical regions atomic to each other, it can also manifest in some execution orders. Detection algorithms [36] based on the Happens-before algorithm are proposed for order violation bugs, details can also be found in the Appendix.

2.3 Related Work

While there have been a number of concurrency bug detection systems, Hydra mainly differs from prior efforts in its novel reuse of GPU hardware for general and flexible detection. In the following, we will discuss the close work to Hydra and briefly illustrate related GPU architecture.

Software-based detection. Software detectors can be categorized into static and dynamic ones, according to when bugs are being detected. There are also several efforts in accelerating software-based detectors.

Static detectors, such as RacerX [7], are generally based on static analysis of source code to detect data race. Due to the potential state space explosion problem, it is usually difficult for static detectors to scale to large programs. Further, they may generate an excessive amount of false positives.

Dynamic detectors detect bugs by constantly monitoring program execution and dynamically analyzing the runtime states, which usually has very few false alarms. However, to record and analyze the frequent memory accesses, software-based detectors usually involve significant runtime overhead. In addition, data race detectors for C/C++ can incur more than 30X overhead [14]. As another example, ConMem [36], an order violation detector, also brings about 16X slowdown for memory-intensive programs such as FFT.

To reduce the detection overhead, there are several efforts trying to reduce the runtime overhead by trading accuracy for speed. Examples include sampling-based approach [12], [18] and leveraging existing hardware assistance [8]. Such proposals have to make a balance between performance and accuracy. By leveraging scalar timestamps instead of vector ones, FastTrack [16], incurs about 8.5X slowdown on the execution time of Java programs. Another software approach is epoch outcome-based detection [13], which introduces a small amount of overhead, but leverages three times of CPU cores. Similarly, Wester et al. [40] rely on uniparallelism to accelerate two classic types of data race detectors, but at the cost of using four times the number of cores as the original application. Such an approach also relies on complex system software stack supporting record and re-execution.

Hardware-based detectors. Most hardware proposals require single-purposed changes to the processor internals, which limits their detectable bug types to only one [4], [5], [6], [30], [31], [32], [43]. Compared to existing proposals, Hydra makes a novel reuse of GPU for flexible concurrency bug detection, thus avoids changes to internal critical-path processor components such as cache and its coherence protocol. Leveraging the massive parallelism of GPU, Hydra can achieve better performance and scalability. Further, as shown in this paper, Hydra can be used to detect different types of bugs simultaneously.

Due to reusing GPU, Hydra is not only more flexible and scalable but also incurs less performance and hardware overhead. Early research efforts [5], [6] usually exploit cache coherence-based mechanisms, which incur relatively large space overhead (about 19 percent [6] and 12.5 percent [5] overhead of cache capacity). SigRace [4] is similar with Hydra in terms of hardware complexity, but it has higher runtime overhead (about 22 percent under a eight-core configuration) compared to Hydra. RADISH [32] requires less hardware overhead than Hydra due to reusing of the cache space to store metadata. However, Hydra achieves a better performance than it (RADISH incurs 0-2x runtime overhead). Similar to Hydra, LifeGuard [44] also aims at providing multiple-purpose hardware support, but for accelerating a wide range of instruction-grain monitoring tools.

KUDA [39] maps a software-based lock-set algorithm on GPU to detect data race only, which, however, still suffers from a large overhead (only about 3-14x speedup over sequential version). Hydra extends the fused CPU-GPU architecture with a novel set of hardware extensions to simultaneously detect multiple bugs with negligible overhead.

Fused CPU-GPU architectures. The fused CPU-GPU architecture also stimulates research interests in improving performance recently. Some prior efforts [9] use CPU to prefetch memory data to accelerate GPU application performance and vice versa. Hydra takes a different approach in reusing fused CPU-GPU architecture for concurrency bug detection.

3 Motivation

To gain insight into flexible concurrency bug detection with GPU, we further analyze the detection algorithms for these bugs. We find that such concurrency bugs are similar in manifestation. All of them are caused by some illegal shared resource accesses and there are significant similarities in their detection processes. The processes can be abstracted and divided into three steps: 1) trace collection; 2) trace preprocessing; and 3) bug detection. In the trace collection step,

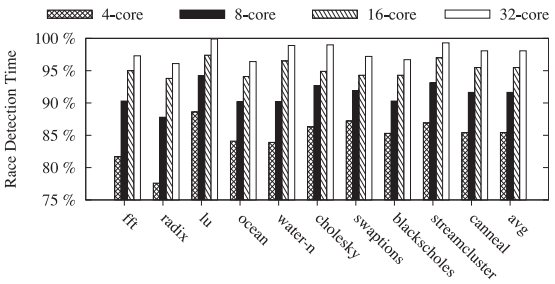


Fig. 2. Execution time percentage of race detection step.

memory, and related operation traces for each core are collected during the program execution. Then, each collected trace is attached with a timestamp or a color signature. In the detection step, the algorithms for different bugs, such as happens-before or ColorSafe, are applied to detect the bugs.

In the detection step of data race and order violation, every memory trace needs to be compared with the history traces and compute the happens-before relation. In atomicity violation detection, there is intensive intersection computation and each intersection is independent from others. Note that, in data race detection, as each access has to be compared with all other cores (or threads), the computation complexity of the detection algorithms inherently increases with the number of cores (or threads). This indicates that using on-chip CPU logics for concurrency bug detection may cause increasingly large overhead with the increasing number of cores. Thus, it is no surprise that prior hardware proposals usually focus on small-scale cores (e.g., 4 and 8-cores).

To validate this hypothesis, we use QEMU [27] to collect memory traces and implement a software-based happens-before race detector as an example. Then we use Intel VTune to analyze the hotspot in this detector. The results in Fig. 2 show that the detection step accounts for more than 85 percent of the whole execution time under four-core, 91 percent under eight-core, 95 percent under 16-core and 98 percent under 32-core configurations.[2]

Fortunately, the comparison or computation in different cores is completely independent (traces are only compared with history without modifying it and each trace is independent) and there are millions of traces in each thread, which means there is abundant fine-grained parallelism. Therefore, it is intuitive to map the computation in the detection stage on GPU.

4 Design of Hydra

The overall architecture of Hydra is shown in Fig. 3. Hydra adds two simple hardware modules on chip: trace collection module (Fig. 4) for each core and a global trace pre-processing module (Fig. 5) between CPU and GPU. Briefly speaking, TCM generates timestamp, collects traces in each CPU core, and sends them to TPM. When TCM collides with L2 memory request on the interconnect, it waits L2 to finish. TPM receives traces, maintains them in a history buffer and sends them to GPU for bug detection. In this section, we will first illustrate the basic workflow of Hydra and then describe some optimizations for performance and accuracy. Finally, we discuss how to virtualize Hydra to accommodate more threads than cores.

2. Prior study uses FastTrack [16] under eight-core shows less ratio of detection time [32]. This is because FastTrack uses scalar clock instead of vector clock for most traces, which may cause accuracy loss.

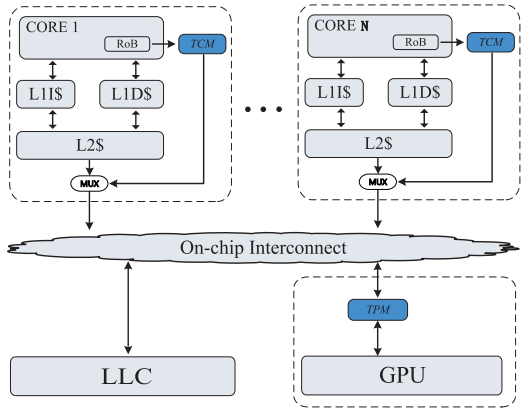


Fig. 3. Hydra overview.

4.1 Trace Collection and Stamp Generation

Trace collection. Hydra instruments code in similar ways as prior work [4], [32]. We use special instructions to label operations related to order violation detection. Hydra recognizes the special instructions or memory access instructions in Re-order Buffer (RoB). For RoB, only a path is required to forward ROB results to TCM. When an instruction is committed by RoB, it is also forwarded to TCM. TCM identifies the labeled instructions and memory access instructions, and stores them in its entries. Each trace consists of three elements: 1) instruction program counter (PC); 2) memory access address; 3) trace type to identify the operation type, such as memory operations or the operations used for detecting order violation.

Thread timestamp generation. Since timestamps for each thread are generated according to synchronization operations (e.g., program-level synchronization such as lock, unlock and barrier), we re-encapsulate synchronization library for timestamp generation as in other work [4], [32]. Each TCM contains a register for current timestamp, which is accessible to software and managed by the Hydra-aware synchronization library. The main changes to the library includes: (1) for each synchronization operation (e.g., lock), the library maintains extra fields in each *sync* variable to store some information (e.g., lock's timestamp and last-holder core) used for timestamp generation; (2) synchronization operations are re-encapsulated with timestamp generation function call through some extended instructions.

Hydra introduces two instructions, *hydra_off* and *hydra_on*, to disable and enable TCM trace collection, as shown in the following code snippet. *hydra_off* means a sync operation will begin. When Hydra encounters *hydra_off* in a core, TCM sends all remaining traces in current trace buffer to the TPM, because the timestamp will be changed and these traces belong to the current timestamp but not the upcoming one.

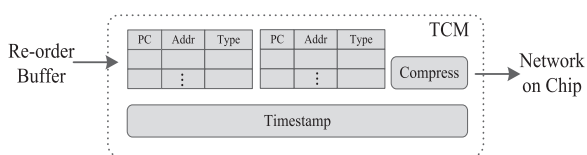


Fig. 4. TCM module.

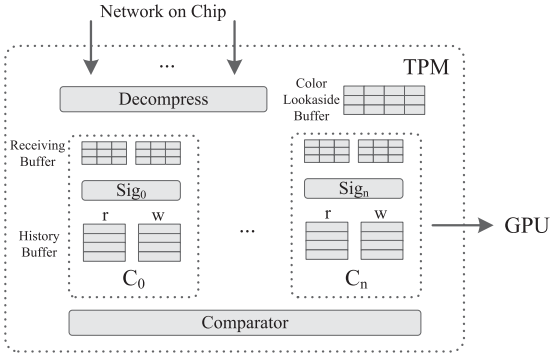


Fig. 5. TPM module.

Hydra maintains a register in TPM for each core to store its current timestamp for an executing thread. In the *UpdateTS* function (Line 4), current core's timestamp will be updated according to the *lock m*'s timestamp and its last lock holder core maintained in the library. *hydra_on* means a sync operation will finish. At this point, Hydra sends the current timestamp to the TPM.[3] In this way, Hydra prevents address pollution from instrumented code. Since synchronization operations are relatively infrequent [4], [32], the timestamp is also changed infrequently, especially compared with the trace collection.

```
1 LOCK ('{
2   hydra_off;
3   lock($m);
4   TS = UpdateTS(TS, $m);
5   hydra_on;
6 }')
```

Trace transmission. The collected traces will be transmitted to TPM for pre-processing via the network on chip (NoC). Other than the forced transmission by *hydra_off*, Hydra transfers traces in a batched manner when the collection buffer is full. To support parallel trace transmission and generation, Hydra uses a rotation buffer mechanism. If one buffer is being transmitted, the other continues to collect the incoming traces; if the other is also full, the CPU core has to be blocked until one buffer is available. Fortunately, CPU blocking due to rotation buffer rarely happens, as Hydra requires only a small amount of NoC bandwidth (Section 5.3.8).

To further reduce the transmission overhead, Hydra compresses traces into a *sending message* before sending. The compression idea is based on the access locality. Hydra selects a base address or PC and only sends offsets of traces. Messages sent from the same core need to arrive in the TPM in order, while messages from different cores can arrive in any order. Such a design will not influence the detection accuracy [4], as data race detection is based on the timestamps of the traces but not the arriving order of the traces.

4.2 Trace Pre-Processing

Trace pre-processing module mainly pre-processes the collected traces before they are transferred to GPU (Fig. 5). We use a centralized TPM here, as GPU has to access the history buffer during detection. The access speed and bandwidth will benefit much more from a centralized history buffer than a distributed one. However, in extremely large or distributed environment, the TPM can be distributed as well, which will be our future work.

When a trace arrives TPM, it will be decompressed and attached with detection signatures (e.g., timestamp or color). Then it will be put into a history buffer inside TPM for bug detection. The followings describe the related hardware mechanisms.

4.2.1 Trace Receiving Buffer (RB)

For large-scale designs (16-core, 32-core or more), only one receiving port to the NoC may become a bottleneck (causing traffic jam here). Hence, TPM uses different receiving ports for different core groups (e.g., one port for four or eight cores). To support multiple-bug detection on GPU, Hydra needs to attach several detection signatures according to different detection algorithms.

For data race and order violation, Hydra associates accessing timestamp for traces according to the saved timestamps in TPM. To maintain color information, Hydra follows the approach in ColorSafe [31]. A multilevel color table resides in memory and keeps the ColorID information at the desired granularity (word, line, page, etc.). To provide fast lookup, a Color Lookaside Buffer (CLB) in TPM directly caches coloring information from the Color Table. When a CLB miss occurs, TPM will fetch the entry from the multilevel color table in memory.

To process traces while receiving them, Hydra also adopts two rotation receiving buffers for each core similar to that in TCM, where one is for receiving and the other is for processing.

4.2.2 History Buffer (HB)

Hydra uses a history buffer to keep the received traces for latter detection by GPU. The history buffer consists of two parts: one is the trace buffer used in the happens-before algorithm for data race and order violation detection, while the other is the color signature buffer used in the ColorSafe algorithm for atomicity violation detection.

Trace buffer. Since the happens-before algorithm needs to compare the current trace's timestamp with those of previous traces (with the same address) to detect races, a trace buffer is used to record the previous traces. After the timestamp attachment step, a trace with the information of *address*, *type* and *timestamp* is inserted into the trace buffer.

To reduce address search time in the trace buffer, traces are maintained in different sub-buffers according to their core ID. The trace buffer is organized as a hash table shown in Fig. 6, each entry of which is a FIFO queue to store traces hashed to the same hash entry. To further filter out read-read trace comparison that is race-free in race detection, read and write trace histories are organized separately. TPM hashes traces to different read/write FIFO queue according to address and read/write types. When GPU compares the current trace to the trace buffer, it hashes the trace address to find the hashed buffer entry and then compares the trace to those in the corresponding read/write queue. Through such a strategy, GPU only needs to traverse

3. TCM does not attach the current timestamp to each trace to save space. Instead, TPM will use the received timestamp from TCM upon the *hydra_on* call to attach the accessing timestamp. This can avoid redundant timestamp transmission.

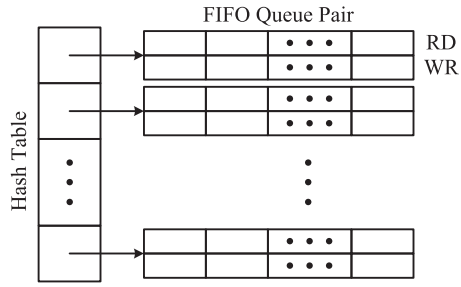


Fig. 6. Hashed trace buffer for each core.

the corresponding read/write queue (if the current trace is read, it only traverses write queue; otherwise, it traverses both queues) for trace comparison instead of traversing the entire trace buffer.

In order violation detection, Hydra reuses the trace buffer. As the operation types for order violation, such as open/close files or malloc/free, are more than those in memory access (write/read), TPM also hashes these operations based on their addresses for a unified hardware design. Although such a design would involve some unnecessary computation, it does not lead to any additional overhead because the number of instrumented operations related to order violation is much less compared to memory accesses in data race detection. When Hydra detects multiple bugs simultaneously, the traces for different bug detection will be mixed together. Otherwise, only the traces for one bug detection will be stored in the buffer.

Color signature buffer. TPM generates color signature as follows. The colors of memory traces is firstly encoded into a signature (local read, local write, remote read or remote write) using a bloom filter. After all signatures in an epoch are grouped together to form a history item, it is inserted into the color signature buffer. The color signature buffer is organized as shown in Fig. 7. In each core, TPM maintains four color signatures, representing 1) local read, 2) local write, 3) remote read, and 4) remote write, for the current epoch. Every four signatures are grouped together to form a history item.

4.2.3 History Buffer Access Synchronization

As TPM maintains history buffer in pre-processing stage and GPU reads it in detection stage, there may be races between TPM and GPU. While GPU reads an entry in the history buffer, its content may be updated by TPM, which will lead to fewer traces being compared on GPU. To guarantee the accuracy, Hydra uses a simple synchronization strategy between TPM and GPU. TPM processes incoming traces and inserts them into the history buffer. After that, GPU reads incoming traces into memory for bug detection. While the next processing request is coming, TPM checks whether GPU has completed its buffer reading. If so, new traces are inserted into the history buffer. Otherwise, TPM will be blocked until GPU finishes reading. Such a design can work well. To illustrate its efficiency, we collect the proportion of the conflicts between TPM and GPU to the number of GPU access to TPM, it is about 0.04 and 0.01 percent under a four-core and a 32-core configuration accordingly.

4.3 Concurrency Bug Detection on GPU

For data race detection, each trace is transferred to the buffer on GPU and GPU will compare its address and

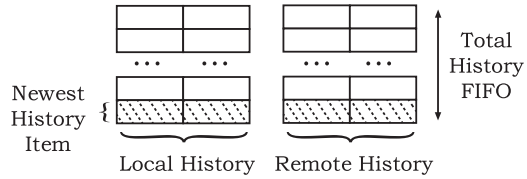


Fig. 7. Color signature buffer for each core.

timestamp to those in other cores' trace buffers to detect races. Here is a summary of the detection process:

- Each trace is hashed using its address to find the corresponding hashed trace buffer entry in all other cores. If the current trace is a read operation, it will be compared with the write queues. Otherwise, it will be compared with both the read queues and the write queues.
- The timestamp of the current trace will be compared with the recorded timestamp of the found trace. If there is no happens-before relation, these two memory accesses are identified as a race.

For order violation detection, TPM transfers all new operations to GPU and then GPU detects violations almost the same as it does in data race detection.

For atomicity violation detection, TPM transfers all new traces to GPU. Then GPU detects bugs in the following steps. First, it computes the intersection of colors in the current traces, the earliest local signature and all remote signatures between them (except for read-read-read cases). Then the computation is processed from the earliest local access to the latest access. If there is a non-empty intersection, an atomicity violation is detected.

Since the detection process has very good parallelism, Hydra offloads the whole process on GPU: each detection trace is attached to a hardware thread on the SM so that GPU can detect all traces in an epoch in parallel.

Detecting multiple bugs simultaneously. The basic detection algorithms of these three bugs are implemented as different GPU kernels. Since they are software kernels, Hydra can flexibly detect each of such bugs respectively. Moreover, Hydra can also detect these three types of bugs simultaneously. We use a tag to indicate whether Hydra is used to detect a certain type of bugs or detect all three types of bugs simultaneously. These three kernels are implemented in a detection process. As a result, unnecessary context switches are avoided. When the detection process begins, the kernel of the happens-before algorithm will be invoked first for the detection of data race and order violation, Then, the kernel of atomicity violation will be executed.

4.4 Optimizations for Hydra

Hydra is also designed with three optimizations to improve detection performance and accuracy. First, it uses a bloom filter to filter out private traces, which can remove unnecessary computation in bug detection. Second, it avoids shared traces eviction in history buffer as long as possible to improve accuracy. Finally, it only compares with last-write traces for shared data with happen-before relation in history buffer based on the feedback in GPU.

TABLE 1
Detection Overhead without Optimization

	Race	Atom-V	Order-V
4-Core	0%	0%	0%
8-Core	0%	0%	0%
16-Core	11.98%	0%	0%
32-Core	76.08%	0%	0%

4.4.1 Counted Bloom Filter Optimization

We measured the average execution overhead for the basic design of Hydra.[4] As shown in Table 1, atomicity violation or order violation is almost free of overhead. However, the happens-before algorithm for data race detection scales poorly with cores, due to its inherent super-linear computation nature [16]. As all traces have to be compared with traces in all other cores' trace buffers, the required computation will grow super-linearly [16] with the core number increasing. With a relatively large number of cores, the GPU resources may not be enough to finish the trace comparison in time. This may further block the execution of CPU cores. Because order violation detection only focuses on operations, which is much fewer than memory traces, the detection workload is much less than that of data race detection. Atomicity violation detection does not suffer from this problem because it considers all other cores as one remote core.

Bloom filter uses multiple hash functions to map an element into a bit vector, converting the expensive set operations to fast bitwise logic operations, which can be performed very efficiently in hardware with negligible overhead [4], [5], [31]. This kind of signature may cause false positives, as it is a superset of the encoded addresses. The false positives will not affect correctness, as the unfiltered traces will still be checked one by one in GPU. It never introduces false negatives and thus causes no accuracy loss in detection.

Besides, to allow the removal of an individual trace from the history buffer, Hydra uses a counting bloom filter [41]. As entries are inserted and removed from history buffer, their addresses are added and removed from the filter. With such a filter, before comparing the current trace to those in another core's trace buffer, Hydra first compares it with the corresponding signature to check whether the trace buffer contains the same address. If so, the current trace will be compared with those in the corresponding hashed trace buffer. Otherwise, the comparison will be skipped. Since most traces are private and most of them can be filtered out, a lot of computation on GPU can be avoided.

4.4.2 Avoiding Shared Traces Eviction

Due to the limited hardware structure, eviction of traces may lead to possible accuracy loss. However, in practice, a small amount of hardware is usually enough to detect the same bugs as software, since concurrency bugs mostly happen within a small window [34], [37]. To further mitigate the influence due to trace eviction, Hydra is integrated with two optimizations to make the history buffer to keep more shared traces and hold more traces.

More shared traces. In concurrency bug detection, only the shared traces are useful. To hold more shared traces, Hydra adds a shared tag for each entry to indicate whether the entry is shared or not. When GPU finishes a detection process, it returns the shared addresses to the TPM. TPM marks the corresponding traces in history buffer as shared. When there is no empty entry for a new trace, TPM evicts those non-shared entries first. In this way, Hydra maintains more useful shared traces in the trace buffer.

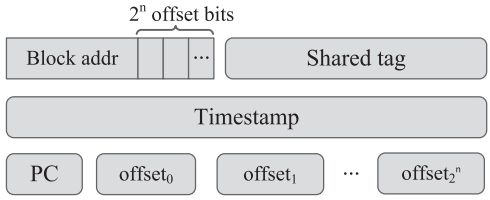


Fig. 8. Reorganized TPM trace buffer entry for mitigating trace eviction.

More traces. Hydra exploits access locality in a program to hold more traces. The trace buffer entry is reorganized as shown in Fig. 8. Instead of storing a complete address per entry, Hydra stores a block of addresses per entry. There is a *base block address* in the entry, which is the trace address except the least n significant bits. The remaining n bits have 2^n combinations so there is a structure containing 2^n offset bits in the entry. Each bit in the structure maps to a combination. For example, when $n = 3$, the *7th* is set to 1 to represent the last *3* bits is *"111"*. To store the corresponding pc for each address, the entry contains a *base PC* and 2^n PC offsets. Unlike the offset bit, each PC offset is a byte, indicating the offset (-128-127) relative to the *base PC*. Further, there are 2^n type bits indicating the address type ("0" for read, "1" for write or write and read).

In this way, each entry can contain a block of addresses. There are some corner cases for this optimization: 1). All the addresses in the block must share the same timestamp. If two addresses are mapped to the same block with different timestamps, they will be stored separately; 2). When the new trace's PC cannot be represented by offset (−128-127), the trace will be stored in another entry. The reorganized entry works well as the access locality.

4.4.3 Last-Write Awareness Optimization

Shared traces many exist in multiple threads. If there are happens-before relations among them, we can only compare a new trace's timestamp with the last-write trace to the same address [16]. The following is such an example. Supposing x and y are two memory operations to address A and y is a write operation. x happens before y ($x \mapsto y$), y will be the last-write operation to A. When a new memory operation to A (z) comes, if y happens before z ($y \mapsto z$), x happens before z ($x \mapsto z$) based on the transitivity feature of happen-before algorithm. If there are no happens-before relations between y and z, a race happens. As a result, last-write is enough for detection.

Therefore, Hydra introduces optimization in two folds: the first one is that for the local accesses within one thread, the succeeding traces happen after all local previous traces with the same address. Therefore, Hydra will replace previous traces to the same address in the buffer; the second one is that for accesses across threads, GPU will send derived happens-before relations in detection as feedback. Hydra resolves the relations and marks the last-write across threads. For other

4. Detailed experimental setup and benchmarks are in Section 5.

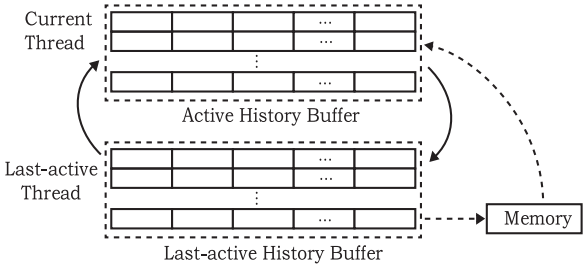


Fig. 9. Sub history buffer structure.

traces to the same address except the last-write, Hydra removes them from history buffer. By leveraging this feedback-directed strategy, Hydra improves both accuracy and speed.

4.5 Hydra Virtualization

Here, we consider how Hydra works on a real multi-threaded environment where threads may migrate and the number of threads may be larger than cores.

Thread migration. We first consider the case of thread migration only (the numbers of threads and cores are equal): 1). Hydra uses thread ID to identify threads in the program instead of the core ID. As a result, each thread ID represents each dimension in the timestamp. The buffers in TPM are indexed through thread ID instead of the core ID; 2). The timestamp in each TCM will be saved when the current thread is preempted. When the thread runs again, its timestamp will be restored according to the thread ID. Moreover, Hydra regards the switching as a synchronization operation, the remaining traces of the preempted thread will be sent to TPM. After that, the upcoming thread will send its timestamp and its thread ID to TPM; 3). A redirection table is added in the TPM. The table is the one-to-one mapping between thread ID and core ID. The mapping is built and updated when TPM receives thread ID from TCM (indicating core ID). It will ask TPM to insert the traces in receiving buffer to which thread's history buffer.

More threads than cores. Based on thread migration extension, Hydra is extended to support a larger number of threads than the number of cores. In this situation, the dimensions of timestamp and redirection table in TPM are set as large as the maximum range of thread ID. However, if we set the number of sub history buffers in TPM as many as the maximum range of thread ID, it would involve large space overhead. Similarly, if we set the number of sub history buffers the same as the number of cores, it would incur large overhead due to swapping sub history from memory when thread migration happens. To avoid unnecessary space and swapping overhead, we set the number of sub history buffers in TPM as twice as physical core number, as shown in Fig. 9. Besides, we keep the signature number for the bloom filters as the maximum range of thread ID. Every two sub history buffers are binding to one physical core. One sub history buffer is used to maintain the current active thread's history, called active history buffer. The other sub history buffer is used to maintain the last active thread's history, called last-active history buffer. The two sub buffers are rotated during processing. For the threads that are not active or last-active, their sub history buffers will be stored in memory. When these threads are active, Hydra switches their sub history buffers into TPM. To illustrate this more clearly, we use an example show in Fig. 9. If the current running thread a is preempted by thread b and b is the last-active thread, then the last-active history buffer become the active history buffer and the active history buffer become the last-active history buffer. If b is not the last-active history buffer, the history buffer of b is loaded from memory, the last-active history buffer is swapped into memory, and the buffer of a becomes the last-active history buffer.

Additionally, for all threads, their signatures are kept in TPM all the time. Therefore, the filter optimization can work well in this situation. When a new trace needs to compare with the traces in memory, the traces in the last active history buffer is swapped into memory and the corresponding traces are loaded into the last-active history buffer. Although a swap of history buffers needs thousands of cycles to finish, such a strategy is almost overhead-free according to our experimental results. The primary reason is that such a swap rarely happens with our three optimization techniques.

4.6 Hydra Configuration

We set some parameters of Hydra according to prior proposals or experimental results.

Collection buffer size and timestamp length. The size of collection buffer in TCM and receiving buffer in TPM is a tradeoff between performance and space overhead. We collect the performance data under different buffer sizes and get the best configuration. When the buffer size is 4, the CPU blocking overhead (both in TCM and TPM) is 0 percent for all benchmarks. When the buffer size reduces to 2, the overhead is about 2.75 percent. Hence, we set the collection buffer size as 4. The length of timestamp is the number of cores and each dimension is 20 bits, which is a popular configuration in prior designs [4], [6] and is enough in Hydra.

History buffer size. The lengths of hashed trace buffer and that of FIFO queues may influence not only the performance and space overhead, but also detection accuracy. To make a reasonable design tradeoff, we collect the data for the parameters of the hashed trace buffer. Based on the analysis, when hashed buffer size is 16, FIFO queue size is 16 and eviction block size is 8, the race detection accuracy can be guaranteed[5] by Hydra, i.e., finding the same amount of static races with that of Helgrind [48], which is a widely-used software tool for data race detection. Due to much fewer traces in order violation detection, this configuration also ensures detection accuracy. Therefore, we use this setting as our default configuration. Since the least significant bits of traces will be more distinct to find the same address. Therefore, we use the lower 3-to-6 bits as the hash function. Second, in atomicity violation detection, the color look-aside buffer size is the same as that in ColorSafe [31]. In theory, there could be possible accuracy loss due to limited hardware structure. In practice, a small amount of hardware is usually enough to detect the same bugs as software due to the fact that concurrency bugs mostly happen within a small window [34], [37].

5. Here, we mainly focus on the number of static race instead of the number of dynamic race instances. The reason is that many dynamic race instances are related to the same static race. Therefore, it is unnecessary to report all the dynamic instances.

TABLE 2
Injected Bugs Detected by Hydra

Bug Classification	Bug Description	% detected
Atomicity violation	Single variable	100%
Atomicity violation	Multiple variables	100%
Order violation	Access closed files	100%
Order violation	Invalid pointer dereference	100%

Filter configuration. We also collect the filter efficiency of different bloom filters. The efficiency is the percentage of filtered out traces. With an 8-to-256 bloom filter, more than 95 percent traces have been filtered out. When a much larger filter, such as 16-65536, is applied, only about 0.1 percent more traces can be filtered out. Moreover, it also involves more space overhead (16 KB per core). Therefore, we use a bloom filter with a 8-256 size as the default configuration.

5 Evaluation

In this section, we evaluate Hydra from the following two aspects using simulation: 1) bug detection capability; 2) time and space overhead.

5.1 Experimental Setup

We use Sniper[6] integrated with GPGPU-Sim in its memory back-end. The basic architecture of Hydra is a four-core Out-of-Order CPU integrated with a 240 SPU core GPU, which is a low-end configuration for a fused CPU and GPU architecture. The system uses the MESI coherence protocol, and is with four-way 32 KB private L1 caches, four-way 256 KB private L2 cache and eight-way 4 MB shared last level cache (all with 64 B lines). The latencies of L1, L2 and last level cache hit is set as 1, 20 and 40 cycles. In terms of our simulation model, the frequency rate between CPU and GPU is set as 4:1. The latency for *hydra_off* and *hydra_on* are 100 cycles. Modules are connected in a multistage network with a maximum bandwidth as 32 Bytes/Cycle.

To evaluate the scalability of Hydra, we also evaluate the configurations of eight-core, 16-core and 32-core. We use SPLASH2 [19], PARSEC [20] and some real-world applications for evaluation, which are widely used in prior bug detectors [4], [31], [32]. The programs in benchmarks include fft, radix, lu, ocean, water-n, cholesky, swaption, blackscholes, streamcluster and canneal. The real-world applications are pfscan, pbzip2, and aget.

5.2 Bug Detection Capability

To measure the race detection capability of Hydra, we use Helgrind [48], a well-known open-source data race detector, as the baseline. Our results show that Hydra is able to detect all bugs found by Helgrind, which are shown in Table 3. Lu et al. [37] present the definitions of atomicity violation and order violation and analyzed different reasons for such bugs. Based on their case studies, we inject one single-variable atomicity violation and one multiple variable atomicity violation and two order violations randomly in each benchmark. We classify the injected bugs according to their causes, as summarized in Table 2. Experimental results show that Hydra can detect all these bugs.

6. We have tried GEMS, which, however, cannot simulate beyond 24 cores.

5.3 Overhead Evaluation

To illustrate the efficiency of Hydra, we evaluate the time, bandwidth and hardware overhead respectively.

5.3.1 Time Overhead for Single Bug Detection

We collect the detection performance data under four-core, eight-core, 16-core and 32-core configurations respectively. By default, we run the same number of threads as cores, as done in prior evaluations. The data are shown in Fig. 10. As shown in the figure, the overheads for single bug detection are about 0, 0, 0.07 and 0.18 percent respectively. As Hydra is mostly overhead-free for atomicity violation and order violation detection (the overhead is less than 1 percent even in the worst case), all the maximum values are from data race detection. Thanks to powerful computation ability of GPU, Hydra is nearly overhead-free to detect a single type of bugs alone. It is also enough for the detection of atomicity violation and order violation. Although the computation load of data race detection increases super-linearly, the three optimizations can significantly improve the performance and mitigate the scalability problem.

5.3.2 Execution Overhead for Multiple-Bug Detection

We further evaluate the execution overhead for simultaneous detection of multiple bugs. As shown in Fig. 11, the overhead for simultaneous detection is only a little bit larger than single bug detection. Hydra incurs only 0.46 percent on average under the 32-core configuration.

5.3.3 Virtualization Overhead

By leveraging last-active history buffer, Hydra reduces the memory-swap rates to under 0.01 percent even when running 16 or 32 threads on four physical cores. The overhead of running 32 threads on 4 physical cores is only 0.93 percent.

5.3.4 Effectiveness of Optimizations

We also evaluate the efficiency of our three optimizations.

Counted bloom filter optimization. By leveraging counted bloom filter, we filter out 95 percent traces for the configuration in 4.6 and achieve nearly overhead free during detection (reducing overhead from 76.08 to 0.35 percent under 32-core).

Keeping more shared traces optimization. The optimization for keeping more shared traces results in that the shared traces eviction rate is under 0.01 percent on average. Further, mitigating trace buffer eviction by leveraging locality can save more precious on-chip space. Our evaluation show that it results in 5.11X more capacity for history buffer on average with only 1X more space overhead.

Last-write awareness optimization. For more than 49.32 percent shared traces detected in GPU, Hydra can figure out which one is last-write trace by the derived happens-before relation from GPU, which means large computation reduction. Therefore, Hydra achieves significant performance improvement (from 0.35 to 0.18 percent under 32-core).

5.3.5 GPU Scale Requirement and Utilization

The GPU used in experiments only contains a low-end configuration. A configuration with 400 SPUs is a modest

TABLE 3
Races Detected by Hydra and Helgrind

Benchmark	fft	radix	lu	ocean	water-n	cholesky	swaption	blackscholes	streamcluster	canneal	pfscan	pbzip2	aget	sum
Helgrind	0	0	0	1	0	3	0	0	62	0	10	6	0	82
Hydra	0	0	0	1	0	3	0	0	62	0	10	6	0	82

configuration in 2012. With such a trend, the GPU scale in a fused GPU and CPU architecture will continue to increase.

We also evaluated a relatively larger GPU configuration. The overhead for simultaneous bug detection can be further reduced to 0.15 and 0.10 percent for 320 and 400 SPUs accordingly shown in Fig. 12. Hence, a user can make a tradeoff between the incurred overhead and the dedicated SPUs. Therefore, it is reasonable to allocate some GPU cores for production-run bug detection while the remaining GPU cores continue to process general GPGPU applications.

Influence on fused CPU and GPU. In our current design, a fused CPU and GPU is exploited. Since CPU and GPU exchange data through on-chip interconnection with some hardware support, such a design involves very little overhead. To further illustrate the influence of fused CPU and GPU on performance, we also implemented a software version of Hydra based on QEMU and the detection part is also mapped on GPGPU. Experimental results show, the data communication will become the performance bottleneck, which leads to about 8X to15X additional overhead over native execution except the instrumentation overhead.

5.3.6 Benefit from HW and GPU

On one hand, software implementation suffers from prohibitively large instrumentation overhead to collect traces. For example, the software version of Hydra using QEMU mentioned in Section 2 incurs 46X for happen-before data race detection alone. The overhead largely comes from instrumentation. Even excluding the essential overhead with QEMU, the overhead would still be around 5X, which prevents this from being used in the production run.

On the other hand, the CPU implementation suffers from large detection computation workload. In the software version, GPU is about 10X faster than CPU does in detection. Moreover, KUDA [39], deploys a software-based lock-set algorithm on GPU to detect data race, showing that the ratio of the speedup of race detection is between 3.3X and 14.7X.

Worse even, a software version may incur much more overhead to detect the three types of bugs simultaneously.

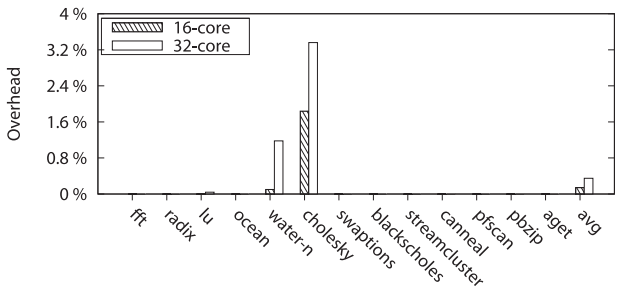


Fig. 10. Maximum overhead for single type of bug detection among data race, atomicity violation and order violation. Overhead free under four-core and eight-core configurations.

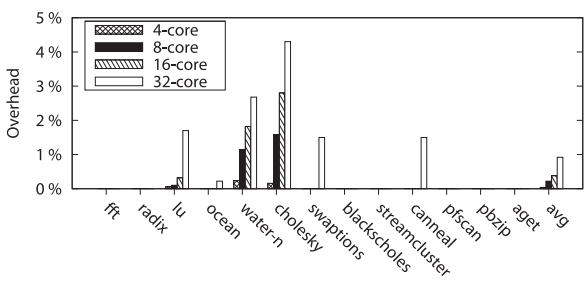


Fig. 11. Overhead of simultaneous multiple-bug detection.

5.3.7 Space Overhead

The space overhead of Hydra comes from TCM and TPM. We present the detailed analysis of four-core configuration as an example. After that, we present the overhead under eight-core, 16-core and 32-core configurations.

- Each core has a TCM. TCM contains a vector timestamp and two collection banks. Each bank has four entries and each entry is 83 bits. Therefore, it is 83 bytes for the collection buffer. The timestamp is 10 bytes under a four-core configuration. Hence, the overhead of each TCM is 93 bytes.
- There is a centralized TPM between CPU and GPU. Each core has a two-bank receiving buffer on TPM to receive traces from the corresponding TCM. Each bank can buffer up to four traces. To record trace and color signature history, each core consumes about 3,680 and 15,360 bytes, respectively. Furthermore, to reduce the address comparison in data race detection, each core maintains a 1,024-bit signature. In total, the size for each core inside TPM consumes about 20K bytes space overhead.

As a result, Hydra involves 93 bytes space for TCM. For the TPM on-chip, it requires less than 25K bytes per core. The total space overhead is less than 0.23 percent compared to the whole die area [35] under four-core configuration. The overhead under 8-core configuration is 0.28 percent. Similarly, the space overheads are about 0.74 and 1.1 percent for 16- and 32-core accordingly. Moreover, we also evaluated the area overhead in the logic simulation with Synopsys Design Compiler using CMOS technology, the

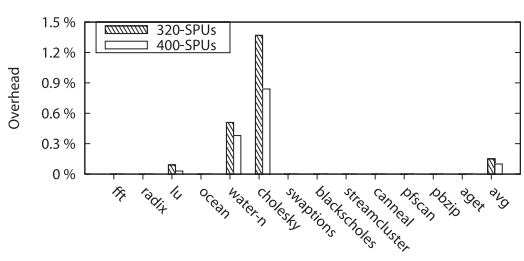


Fig. 12. Overhead of simultaneous multiple-bug detection with larger GPU Scale.

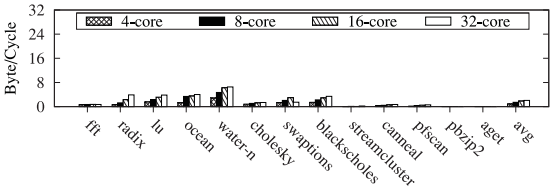


Fig. 13. Bandwidth consumed by Hydra.

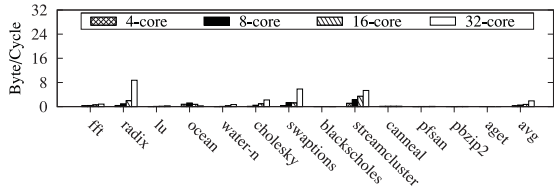


Fig. 14. Bandwidth consumed by LLC Miss.

clock frequency is set to 1 GHz. Based on the evaluations, the extra area overhead incurred is about 0.01 percent for Intel i7 core or Xeon Nehalem-EX core, and 0.09 percent for ARM Cortex-A9 core. Therefore, the space overhead of Hydra is reasonably small. Though adding virtualization support further introduces some hardware such as the mapping table and per-thread buffer, the overhead is negligible.

Hardware complexity. Though there are some other hardware extensions in Hydra, we argue that it leverages reasonable and well-understood technologies. First, the mechanisms to collect traces in processors have already been proposed in some previous work [4], [32]. Then for the hardware pre-processing logic in Hydra, the timestamp and signature generation logic has also been used in [5], [6], [31], [32]. The support for an external history buffer is also popular in hardware bug detection mechanisms [4], [31]. Therefore, our design should be relatively easy to implement by incorporating prior hardware proposals.

5.3.8 Bandwidth Overhead

To measure the bandwidth consumption of Hydra, we count how many bytes of traces are transmitted on the interconnect. Fig. 13 shows the transmitted data size per cycle for each benchmark. As the data shows, Hydra produces an average bandwidth overhead of about 0.86, 1.43, 1.87 and 2.06 bytes/cycle under 4-, 8-, 16- and 32-core configurations respectively, where the on-chip interconnect bandwidth is 32 bytes/cycle. In other words, the bandwidth overhead is less than 2.70, 4.48, 5.83 and 6.44 percent. Moreover, we evaluate the bandwidth with the compression mechanism we mentioned in Section 4.1. Even under a 32-core configuration, the bandwidth overhead is reduced to 3.01 percent on average and 7.58 percent as the maximum with an average compression rate of about 3.8. To further show that the bandwidth usage is reasonable, we count the bandwidth overhead occurred by last level cache miss of these benchmarks. The data is shown in Fig. 14. The bandwidth overhead between LLC and memory is about 0.26, 0.55, 0.75 and 1.91 bytes/cycle on average, less than 0.81, 1.72, 2.33 and 5.97 percent of the bandwidth.

Moreover, the link between TPM and GPU is not a bottleneck on Hydra. On fused CPU-GPU architecture, the on-chip bandwidth for GPU is about 250 bytes/cycle. We collected the data transferred to GPU for debugging. The bandwidth occupation for debugging only takes about 2 percent of GPU on-chip memory bandwidth [45], [46]. Therefore, the bandwidth consumption is modest.

6 Conclusion and Future Work

This paper proposed Hydra, a flexible and efficient GPU-assisted concurrency bug detector with low overhead in both space and time. Unlike prior work, Hydra exploited the massive parallelism and computation power of GPU to simultaneously detect multiple concurrency bugs. Experimental results showed that Hydra introduced small performance, space and bandwidth overhead, even under a 32-core configuration. In future, we plan to study the performance implication on extremely large-scale cores, extend Hydra to support more algorithms and study the energy efficiency of Hydra.

Appendix A
Bug Detection Algorithms

A.1 Data Race Detection

Hydra uses the happens-before algorithm [21] for data race detection, as done in many other systems [4], [6], [8], [18], [32], [40]. It will be our future work on extending Hydra to support the lockset algorithm [28]. Here, we briefly describe such algorithms.

Happens-Before Algorithm. The HB relation is formally defined as the least strict partial order on events, which can be described by the following three rules:

- HB1: $a \mapsto b$ if a and b are events from the same thread execution and a precedes b.
- HB2: $a \mapsto b$ if a and b are synchronization operations from different threads and the synchronization semantics infer that a precedes b.
- HB3: transitivity, if $a \mapsto b$ and $b \mapsto c$, then $a \mapsto c$.

According to the above rules, a data race is defined as two memory accesses (at least one is write) to the same address without a happens-before relation. Typical implementation partitions a program into epochs, which are separated by synchronization operations. Each thread maintains a vector timestamp, with each dimension in the vector timestamp representing the perceived epoch for each thread. A synchronization operation will cause a thread to update its own vector timestamp from the prior thread accessing the synchronization variable, and to increase the logic clock of the executing thread (representing a new epoch). Each variable also has a vector timestamp derived from the thread's vector timestamp, presenting when a thread accesses a variable. Upon each access, the timestamp for a shared memory access will be compared with accesses in all other threads to the same variable to see if a happens-before relation holds by comparing the vector timestamps. Fig. 15 shows an example of the happens-before relation between two threads. As the timestamp ([2, 1]) of "Wr" in thread 1 is larger than that for "Rd" ([1, 0]) in thread 0, there is a happens-before relation between the two accesses and thus no race is detected. The traces in each core will be compared with traces from other cores to check happens-before relation.

A.2 Atomicity Violation Detection

Atomicity violation is a popular concurrency bug, which accounts for about 65 percent of non-deadlock concurrency bugs [37]. Therefore, there are many proposals for atomicity

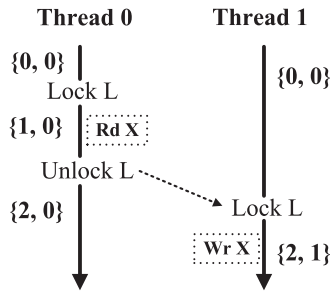


Fig. 15. Happens-before example.

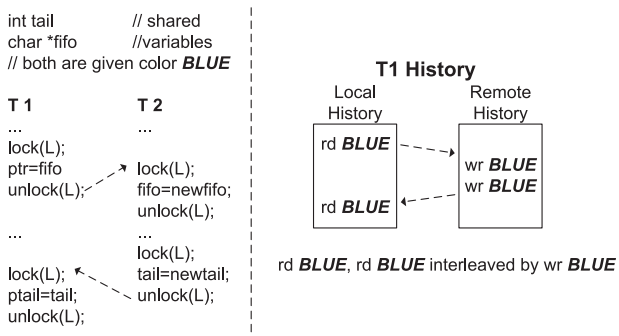


Fig. 16. ColorSafe example.

violation detection [30], [31]. Here, we choose ColorSafe [31] as our atomicity violation detection algorithm because it is efficient and can be used for multi-variable bug detection.

ColorSafe. ColorSafe assigns related variables in an atomicity region with the same color. Possible atomicity violation occurs when two accesses in an atomic region with the same color are interleaved by at least one remote access with the same color. The code in the left side of Fig. 16 shows an example of multiple variable atomicity violation. The code of T1 should be an atomic region and there should be no other write operations to variables *fifo* and *tail*. However, when the code is executed, the atomic region, read operations on variables *fifo* and *tail*, is broken by some remote write accesses to the same variables in thread T2. The dotted arrows in the figure denote the access order of this atomic violation. In the ColorSafe algorithm, related variables *fifo* and *tail* are both colored with *BLUE* before detection. The detection workflow is shown in the right side of Fig. 16. All accesses are inserted into their corresponding history buffers (local or remote) as they happen. When the last *rd BLUE* is inserted, ColorSafe detects that two *rd BLUE* accesses are interleaved by two remote *wr BLUE* accesses, which results in an atomicity violation.

A.3 Order Violation Detection

Order violation bugs account for about 32 percent of non-deadlock concurrency bugs [37]. Recently, researchers start paying more attention to order violation. The detection algorithms in [36] are based on happens-before algorithms. They analyze whether the order between operations obeys correct happens-before relation (specifications order). For example, operation "file open" should happen before operation "file access".

Acknowledgments

The authors are grateful to supports from the National High Technology Research and Development Program of China (No.2012AA010901), the National Natural Science Foundation of China (No. 61370081). They would like to thank all our anonymous reviewers for valuable feedback on the paper. Haibo Chen is the corresponding author.

References

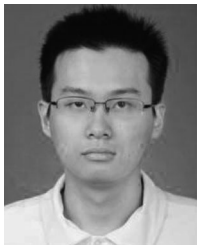
[1] M. Prvulovic and J. Torrellas, "ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes," in *Proc. 30th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 110–121.
[2] R. Huang, E. Halberg, and G. E. Suh, "Non-race concurrency bug detection through order-sensitive critical sections," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 655–666.
[3] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: Telling the difference with portend," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 185–198, 2012.
[4] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas, "SigRace: Signature-based data race detection," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 337–348, 2009.
[5] P. Zhou, R. Teodorescu, and Y. Zhou, "HARD: Hardware-assisted lockset-based race detection," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 121–132.
[6] M. Prvulovic, "CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 232–243.
[7] D. Engler and K. Ashcraft,"RacerX: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 237–252, 2003.
[8] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin, "Demand-driven software race detection using hardware performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 165–176, 2011.
[9] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on fused CPU-GPU architectures," in *Proc. IEEE 18th Int. Symp. High Perform. Comput. Archit.*, 2012, pp. 1–12.
[10] D. H. Woo and H. S. S. Lee, "COMPASS: A programmable data prefetcher using idle GPU shaders," *ACM SIGPLAN Notices*, vol. 45, no. 3, pp. 297–310, 2010.
[11] A. Nistor, D. Marinov, and J. Torrellas, "Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 541–552.
[12] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, vol. 10, pp. 1–16.
[13] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 369–384.
[14] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas, "Accurate and efficient filtering for the Intel thread checker race detector," in *Proc. 1st Workshop Architectural Syst. Support Improving Softw. Dependability*, 2006, pp. 34–41.
[15] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proc. Workshop Binary Instrumentation Appl.*, 2009, pp. 62–71.
[16] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 121–133, 2009.
[17] K. Sen, "Race directed random testing of concurrent programs," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 11–21, 2008.
[18] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 134–143, 2009.
[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 24–36, 1995.
[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.
[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[22] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 382–393.

[23] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.
[24] C. J. Mauer, M. D. Hill, and D. A. Wood,"Full-system timing-first simulation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 108–116, 2002.
[25] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, p. 52.
[26] K. Poulsen, "Software bug contributed to blackout," *Security Focus*, 2004.
[27] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Techn. Conf.*, FREENIX Track, 2005, pp. 41–46.
[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs." *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.
[29] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[30] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 37–48, 2006.
[31] B. Lucia, L. Ceze, and K. Strauss, "ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 222–233, 2010.
[32] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, "RADISH: Always-on sound and complete race detection in software and hardware," in *Proc. 39th Int. Symp. Comput. Archit.*, 2012, pp. 201–212.
[33] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 325–336, 2009.
[34] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations," in *Proc. 35th Int. Symp. Comput. Archit.*, 2008, pp. 277–288, .
[35] A. Branover, D. Foley, and M. Steinman, "AMD fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, Mar./Apr. 2012.
[36] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 179–192, 2010.
[37] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 329–339, 2008.
[38] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 163–174.
[39] U. C. Bekar, T. Elmas, S. Okur, and T. S. Kuda, "GPU accelerated split race checker," presented at the Workshop Determinism Correctness Parallel Programming, London, England, U.K., Mar. 2012.
[40] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Parallelizing data race detection," in *Proc. 18th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2013, pp. 27–38.
[41] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting Bloom filters," in *Proc. 14th Conf. Annu. Eur. Symp.*, 2006, pp. 684–695.
[42] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, OpenCL programming guide. Pearson Education, 2011.
[43] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iWatcher: Efficient architectural support for software debugging," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 224–235.
[44] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 377–388, 2008.
[45] (2016). [Online]. Available: http://en.wikipedia.org/wiki/Ivy_-Bridge_(microarchitecture)
[46] (2016). [Online]. Available: www.amd.com/us/products/technologies/fusion/Pages/fusion.aspx
[47] (2016). [Online]. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation
[48] (2015). [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html

Weihua Zhang received the PhD degree in computer science from Fudan University in 2007. He is currently an associate professor in the Parallel Processing Institute, Fudan University. His research interests are in compilers, computer architecture, parallelization, and systems software.

Shiqiang Yu is currently working toward the graduate degree in the Software School of Fudan University and in the Parallel Processing Institute. His work is related to computer architecture, CUDA Programming, parallel optimization, and so on.

Haojun Wang is currently working toward the graduate degree in Software School of Fudan University and in Parallel Processing Institute. His work is related to computer architecture, simulation, parallel optimization, and so on.



Zhuofang Dai is currently working toward the graduate degree in the Software School of Fudan University and in the Parallel Processing Institute. His work is related to computer architecture, simulation, parallel optimization, and so on.

Haibo Chen received the BSc and PhD degrees in computer science from Fudan University in 2004 and 2009, respectively. He is currently a professor in School of Software, Shanghai Jiao Tong University, doing research that improves the performance and dependability of computer systems. He is a senior member of the IEEE and the IEEE Computer Society.