SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5554-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SCW63240.2024.00030

Facilitating Bug Detection for OpenMP Offloading Applications

Lechen Yu
College of Computing
Georgia Institute of Technology
Atlanta, USA
lechen.yu@gatech.edu

Feiyang Jin
College of Computing
Georgia Institute of Technology
Atlanta, USA
fjin35@gatech.edu

Joachim Jenke
IT Center
RWTH Aachen University
Aachen, Germany
jenke@itc.rwth-aachen.de

Vivek Sarkar
College of Computing
Georgia Institute of Technology
Atlanta, USA
vsarkar@gatech.edu

***Abstract*—The paper introduces ARBALEST-VEC, a redesigned dynamic analysis tool for detecting data inconsistencies in OpenMP offloading applications, specifically targeting GPU environments. Building upon the shortcomings of its predecessor, ARBALEST, the new tool incorporates several improvements. ARBALEST-VEC is implemented on a more recent version of LLVM, addressing issues such as inaccurate data movement modeling and insufficient debugging information in ARBALEST. It introduces a new OpenMP Tool interface (OMPT) event, `device_mem`, to accurately capture data movements, thus reducing runtime overhead and enhancing debugging accuracy. The tool also leverages additional debug information available in the LLVM toolchain to generate more detailed and user-friendly bug reports. Furthermore, ARBALEST-VEC employs a dedicated shadow memory and vectorized dynamic analysis using SIMD instructions to improve the performance and precision of data inconsistency detection. Evaluations demonstrate that ARBALEST-VEC offers improved accuracy, usability, and performance over ARBALEST, with lower time overhead and more predictable memory usage, making it more suitable for real-world OpenMP programs.**

***Index Terms*—OpenMP, Data Inconsistency, Dynamic Analysis, SIMD**

I. INTRODUCTION

In the past decade, OpenMP has become a popular intra-node parallel framework. To cater to the general availability of graph processing units (GPUs), OpenMP has introduced *device directives* into the specification. These derivative-based, hardware-agnostic constructs mitigate the difficulty of GPU programming. Error-prone programming efforts, such as calculating the boundary of a parallelized for-loop for each thread, are delegated to the compiler and OpenMP runtime. Unfortunately, using device directives cannot avoid all types of bugs. For example, programming errors in device directives may result in incorrect data movements between the host and GPU, and therefore a variable may have inconsistent instances on different devices [1], [2]. In this paper, we refer to such errors as *data inconsistencies*.

There has been a handful of prior work focusing on the detection of data inconsistencies. ARBALEST [3] is a dynamic analysis tool built on top of Archer [4], the state-of-the-art OpenMP race detector. ARBALEST leverages a helpful feature in LLVM that device directives can be executed on the host for debugging an OpenMP program. The OpenMP runtime can execute kernels by a group of dedicated CPU threads while simulating data movements using memory routines such as `malloc` and `memcpy`. Utilizing this feature, ARBALEST monitors the execution of an OpenMP program as if it were CPU-parallelized. ARBALEST maintains a *per-Variable State Machine* (*VSM*) to check the validity of each *mapped variable* (host variable mapped to the GPU and accessed in a kernel). When a read operation from either device tries to access a mapped variable while the variable does not have a valid value, ARBALEST reports this suspicious memory access as a data inconsistency.

However, we found a number of issues after testing ARBALEST. First, the original implementation of ARBALEST is based on LLVM 9 and OpenMP 5.0. Since LLVM 9 is the first release that supports device directives, the corresponding implementation of OpenMP is not mature, which may affect the accuracy of ARBALEST. Additionally, ARBALEST does not generate enough debugging information for detected data inconsistencies, making the bug report hard to understand. All these issues impede applying ARBALEST to real-world OpenMP programs.

In this paper, we introduce the redesign of ARBALEST, which is implemented on a more recent release of LLVM and addresses the aforementioned issues. The new tool, ARBALEST-VEC, also executes all kernels on the host, leverages the OpenMP Tool interface (OMPT) to capture the invocation of OpenMP constructs [5], and dynamically checks the validity of each mapped variable at byte granularity. During the dynamic analysis phase, ARBALEST-VEC applies three major optimizations to improve the quality of data inconsistency detection, including:

1) *accuracy*: more accurate data movement modeling,
2) *usability*: more comprehensive bug report, and
3) *performance*: dedicated shadow memory and SIMD-accelerated dynamic analysis.

To model the data movements between the host and the "virtual" GPU more accurately, ARBALEST-VEC adds a new OpenMP event into OMPT. OpenMP applies a series of complex rules for data movements. For example, the runtime applies a reference-count-based mechanism to determine redundant data movements, and programmers can overwrite

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00030

this default behavior using certain device directives. Built-in OMPT events pass either user-specified data movements in the source code or low-level memory operations on the GPU to the tool, none of which directly reveal the data movements carried out at runtime. Compared to these built-in events used by ARBALEST, the new event we introduced into OMPT can reduce the workload involved in correctly modeling the data movements. Furthermore, ARBALEST-VEC fully utilizes the improved debug information for OpenMP offloading [6]. This enhanced debug information, introduced in recent LLVM releases, records the corresponding OpenMP construct and mapped variable for each data movement. ARBALEST-VEC retrieves this information at runtime to generate clearer and more detailed bug reports for identified data inconsistencies, thereby enhancing the programmer's ability to understand and address those bugs quickly.

Apart from the optimizations for accuracy and usability, ARBALEST-VEC also tries to improve the performance of data inconsistency detection. ARBALEST-VEC applies SIMD instructions to accelerate the data inconsistency detection. A memory access may read/write a number of consecutive memory locations and render the identical state for these locations. Using SIMD instructions, ARBALEST-VEC efficiently processes the states of these contiguous memory locations in parallel, speeding up the identification of data inconsistencies. In addition, ARBALEST-VEC also applies a new strategy for shadow memory. Unlike ARBALEST, which embeds each variable's state into Archer's shadow cell, ARBALEST-VEC allocates separate shadow memory to store the VSM. Although this strategy may slightly increase the memory overhead, it mitigates the interference between Archer's race detection and ARBALEST-VEC's data inconsistency detection. The evaluation results demonstrate that ARBALEST-VEC offers lower time overhead than ARBALEST and exhibits more predictable memory usage. In the SPEC-ACCEL benchmark suite, ARBALEST-VEC shows a memory overhead ranging from 3.47× to 4.01×, with an average of 3.50×. While ARBALEST also has a memory overhead of 3.47×, ARBALEST-VEC maintains a comparable level of memory overhead. Notably, ARBALEST may incur significant memory overhead on certain OpenMP benchmarks (e.g., 552.pep), a problem that does not occur with ARBALEST-VEC.

We have released ARBALEST-VEC on GitHub. The source code of ARBALEST-VEC can be accessed through the following link: https://github.com/lechenyu/Arbalest-Vec.

The rest of this paper is organized as follows: Section II provides an overview of device directives and describes the probable root causes of data inconsistencies. Section III sheds light on some shortcomings in ARBALEST's design. Section IV introduces the architecture of ARBALEST-VEC, illustrating how we redesign different modules of dynamic data inconsistency detection to address ARBALEST's shortcomings. In Section V, we present and compare the evaluation results of ARBALEST-VEC and ARBALEST in terms of time and memory overhead. Finally, in Section VI, we discuss related prior research and explore potential future work in data inconsistency detection."

II. BACKGROUND

A. Device Directives

With device directives, programmers can describe a GPU kernel's behavior in a holistic manner, rather than explicitly specifying the computation within each thread. As we illustrated in lines 11-16 of Figure 1, programmers can use the combination of device directives to annotate a for-loop to be parallelized on the GPU, thereby delegating the error-pone work partitioning and loop bounds checking to the underlying runtime. This simplification not only makes the code more accessible but also reduces the likelihood of bugs associated with manual control of parallel execution details.

For each host variable accessed in a kernel, OpenMP infers the necessary data movements using *map-types*. Programmers may explicitly specify a variable's map-type using a `map` clause; otherwise, the OpenMP runtime can automatically infer the map-type using predefined rules. The common map-types are `alloc`, `to`, `from`, `tofrom`, `release`, and `delete`. The semantics of these map-types are:

- `alloc`, allocating an uninitialized GPU memory section for the host variable.
- `to`, transferring the host variable to the GPU,
- `from`, transferring the variable's value back to the host,
- `tofrom`, a combination of `to` and `from`,
- `release`, deallocating the GPU memory section for the host variable, and
- `delete`, similar to `release`, but ignoring the reference-count-based mechanism (to be introduced in following paragraphs)

Such data movements specified by `map` clauses are referred to as *data mappings* in the OpenMP specification.

For map-types `alloc` and `to`, they take effect before executing the kernel, while for map-types `from`, `release`, and `delete`, they take effect after the kernel terminates. OpenMP applies a reference-count-based mechanism to avoid redundant data movements. For a mapped variable that appears in multiple `map` clauses, its map-type is determined when encountering the first `map` clause; any subsequent `map` clauses are ignored by the runtime.

In Figure 1, both arrays `a` and `b` have two associated `map` clauses. The runtime tackles the `map` clauses in line 12 and 13 while ignoring the other two. Therefore, the map-types of arrays `a` and `b` are `from` and `alloc`, respectively. This runtime behavior of map-type deduction can be overwritten by adding an `always` modifier. All `map` clauses marked as "always" take effect regardless of the reference count maintained by the runtime.

For a mapped variable without an explicitly declared map-type, the OpenMP runtime can deduct its map type according to the variable type. The OpenMP runtime assigns a map-type of `tofrom` if the mapped variable is a static array/object. Otherwise, the runtime tackles it as a scalar variable (e.g., int/float/char/bool). The variable's value is transferred to the GPU in an implementation-specific manner for subsequent memory accesses in the kernels, but no data mapping is

settled for this mapped variable. A common misunderstanding regarding map-type deduction involves the rules for dynamic arrays. Due to the difficulty of inferring the array size for dynamic arrays, the OpenMP runtime tackles a dynamic array as a pointer, namely a scalar variable. Therefore, if the map-type is not explicitly specified, the OpenMP runtime transfers the host pointer instead of the entire array to the GPU.

B. *Data Inconsistency and the* `map` *Clause*

Data inconsistencies indicate that, at least on one device, a variable has an invalid value, and the value has been accessed by the program. According to the study of data inconsistencies in [3], both explicitly specified and implicitly deducted map-types can be the root causes of data inconsistencies. Programmers may use a map-type of `alloc` instead of `to` for a variable to be read on the GPU, as the bug presented in line 13 of Figure 1. With the map-type `alloc`, the runtime does not transfer the variable to the GPU before executing the kernel, rendering the variable uninitialized on the GPU. Similarly, using `alloc` instead of `from` may result in the loss of computation results produced by the GPU, potentially causing the host to use stale data for the remaining computations.

Implicit map-type deduction can help avoid data inconsistencies if the mapped variable is a static array. The predefined rules assign a map-type of `tofrom` for static arrays. As the most conservative map-type, it instructs the OpenMP runtime to always synchronize the values of array elements on different devices before and after the kernel execution. However, the predefined rules may not be appropriate for dynamic arrays. Since the dynamic array is tackled as a pointer, the runtime just transfers the value of the pointer to the GPU. This unexpected behavior can cause the kernel to access an invalid address, potentially leading to a buffer overflow during execution.

Apart from map-types, the incorrect array section in a `map` clause can also result in data inconsistencies. In line 13 of Figure 1, the program declares that the first half of array `b` should be mapped, while the whole array is accessed on the GPU (see lines 17 and 18). This programming error causes the remaining section of array `b` to have no valid copies on the GPU, and the reads on the GPU may lead to a buffer overflow. Sometimes, the buffer overflow can go unnoticed if the accessed memory location has already been allocated to other mapped variables. This scenario is quite common when multiple variables are mapped in a single kernel, making it difficult to detect and debug manually.

III. Issues in the Original ARBALEST

In this section, we introduce the details of some shortcomings we found in ARBALEST's original design. To the best of our knowledge, ARBALEST is the first dynamic analysis tool designed for data inconsistencies in OpenMP programs. ARBALEST executes an OpenMP GPU application as CPU-parallelized, launching the kernel on the host and simulating data mapping using host memory routines. As an extension to Archer, ARBALEST reuses Archer's shadow memory implementation, reserving four bits from each shadow cell

```
1  #define SIZE 1000000
2
3  int main() {
4    int *a = new int[SIZE];
5    int *b = new int[SIZE];
6
7    #pragma omp parallel for
8    for (int i = 0; i < SIZE; i++)
9      b[i] = i;
10
11   #pragma omp target data          \
12     map(from: a[0:SIZE])           \
13     map(alloc: b[0:SIZE / 2])
14   #pragma omp target parallel for  \
15     map(from: a[0:SIZE])           \
16     map(to:   b[0:SIZE / 2])
17   for (int i = 0; i < SIZE; i++) {
18     a[i] = b[i];
19   }
20 }
```

Fig. 1: OpenMP program exhibiting data inconsistencies in line 13. To fix this program, the highlighted `map` clause should be modified to `map(to: b[0:SIZE])`.

to record the state of mapped variables. However, since ARBALEST was developed before LLVM fully incorporated GPU support for OpenMP offloading, ARBALEST's design has several shortcomings.

A. *Inaccurate Modeling of Data Movement.*

ARBALEST registers callbacks through the OpenMP Tool interface (OMPT) to capture the invocation of OpenMP constructs. According to the OpenMP specification, there are two OMPT events related to data movements, `target_map` and `target_data_op`. Unfortunately, neither of them can accurately model the carried-out data movements. The OpenMP runtime issues `target_map` events for user-defined `map` clauses and implicitly deducted mapped variables. The occurrence of `target_map` event is not tied to the reference count, so the OpenMP runtime might notify dynamic analysis tools about an ignored `map` clause, e.g., the one in line 15 of Figure 1. To filter such `map` clauses, dynamic analysis tools need to track each mapped variable's reference count by themselves, which may incur additional overhead to program execution. The other event, `target_data_op`, is designed for memory operations involving the GPU. Tool developers may believe that tracking `target_data_op` events with flags `transfer_to` and `transfer_from` may be an alternative solution for data movements. However, OpenMP also offers device memory routines that enable programmers to transfer data between the host and the GPU explicitly. As a result, a `target_data_op` event may not indicate the presence of a `map` clause.

Figure 2 shows the OMPT events to be issued at runtime when executing the code snippet in Figure 1. We use different colors to denote OMPT events associated with arrays `a` and `b`. It is obvious that `target_map` events are quite noisy. Dynamic analysis tools must filter out those no-op `target_map` events (colored in purple) to accurately track the data movements.

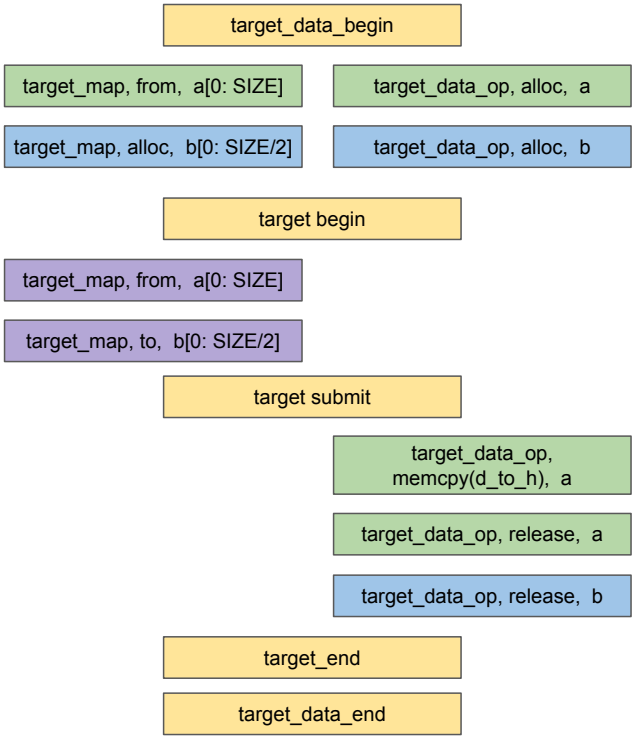

Fig. 2: OMPT events issued at runtime for Figure 1

Furthermore, using `target_data_op` events to track data movements is also challenging. A single `map` clause may incur multiple `target_data_op` events. Dynamic analysis tools have to correlate them correctly to analyze the effect of the `map` clause.

Another issue related to OMPT is that it lacks events for global variables. OpenMP programs may use a `declare target` construct to map global variables to the GPU. These constructs take effect when the OpenMP runtime loads the binary onto the GPU. According to the OpenMP specification, the `declare target` construct does not issue any OMPT event. As a result, dynamic analysis tools may not be able to monitor data movements for global variables.

B. Insufficient Debug Information

ARBALEST generates its bug report using a customized template from the LLVM sanitizer framework. The report includes the call stack and the memory location where the data inconsistency was detected. After scrutinizing the bug report, we found the provided debug information is insufficient. Programmers may determine the location of the data inconsistency by examining the call stack's top stack frame, but there might be multiple `map` clauses in the same line. Without additional debug information, figuring out the exact `map` clause that leads to data inconsistency is intricate. In open-source OpenMP benchmark suites like DRACC and SPEC-ACCEL, most kernels contain more than three `map` clauses, and programmers often condense them into a single line for a compact format. Therefore, we

```
==================================================
WARNING: ThreadSanitizer: data inconsistency (buffer overflow)
(pid=9482) on the target
  Read of size 4 at 0x7f936b0a4490 by thread T12:
    #0 .omp_outlined._debug__.1 error.cpp:21:12
    #1 .omp_outlined. error.cpp:17:3
    #2 __kmp_invoke_microtask <null>
    #3 .omp_outlined._debug__ error.cpp:17:3

  Location is heap block of size 2000000 at 0x7f936aebc000
allocated by main thread:
    #0 malloc tsan_interceptors_posix.cpp:667:5
    #1 DeviceTy::getTargetPointer(void*, void*, long, void*, bool, bool,
bool, bool, bool, bool, bool, AsyncInfoTy&, void*) <null>
(libomptarget.so.15)
    #2 main error.cpp:14:3
    #3 __libc_start_main libc-start.c:310

SUMMARY: ThreadSanitizer: data inconsistency (buffer overflow)
error.cpp:21:12 in .omp_outlined._debug__.1
==================================================
```

Fig. 3: Bug report of detected data inconsistency in Figure 1

infer that locating incorrect `map` clauses may commonly pose a challenge when applying ARBALEST to real-world OpenMP programs.

Figure 3 exhibits the bug report for one of the data inconsistencies in Figure 1. Although the bug report pinpoints the location of data inconsistency, little information is related to the incorrect `map` clause. The green "memory location" section only shows the memory access involved in the data inconsistency and the call stack when the variable is allocated. Through the limited debug information, programmers can only locate the `target data` construct in line 11. Additional effort is still required to determine the actual root cause of this data inconsistency, which is the highlighted `map` clause in line 13.

Apart from locating the `map` clauses, the debug information for the memory location is also insufficient. The bug report template leverages a symbolizer to translate memory locations into variable names. However, the symbolizer works quite unstable. When dealing with a dynamic array, it often fails to retrieve the corresponding name, which diminishes the effectiveness of the bug report.

C. Coarse-grind Dynamic Analysis

To reduce memory overhead during program execution, ARBALEST opts to reserve some bits from Archer's shadow cell instead of allocating separate shadow memory. This design choice brings about a drawback: a coarse-grind dynamic analysis. Since Archer allocates a fixed-size shadow cell for every eight-byte application memory, ARBALEST can only check the validity of mapped variables at eight-byte granularity. For memory accesses to any location within the eight-byte application memory, ARBALEST treats the effect as impacting the entire memory section. Although double-precision floating points and eight-byte integers are widely used in HPC areas, we also found that many OpenMP programs focus on four-byte variables. Consequently, coarse-grind variable state and

```
typedef enum ompt_device_mem_flag_t {
  ompt_device_mem_flag_to = 0x01,
  ompt_device_mem_flag_from = 0x02,
  ompt_device_mem_flag_alloc = 0x04,
  ompt_device_mem_flag_release = 0x08,
  ompt_device_mem_flag_associate = 0x10,
  ompt_device_mem_flag_disassociate = 0x20
} ompt_device_mem_flag_t;

typedef void (*ompt_callback_device_mem_t) (
  ompt_data_t *target_task_data,
  ompt_data_t *target_data,
  unsigned int device_mem_flag,
  void *host_base_addr,
  void *host_addr,
  int host_device_num,
  void *target_addr,
  int target_device_num,
  size_t bytes,
  const void *codeptr_ra,
  const char *var_name
);
```

Fig. 4: The declaration of `device_mem` callback

dynamic analysis might produce false positives and negatives when testing certain OpenMP programs.

IV. ARBALEST-VEC

This section introduces the new tool, ARBALEST-VEC. As a refined implementation of ARBALEST, we redesigned some modules when implementing the new tool on top of LLVM 15. This section focuses on these redesigned modules and explains how these enhancements address the limitations described in Section III.

A. Standalone OMPT Implementation

According to the LLVM repository's commit history, the OMPT support for device directives is unavailable in LLVM 15. Therefore, we wrote a standalone OMPT implementation for device directives. In addition, since existing OMPT events cannot accurately model the effect of `map` clauses, we designed a new OMPT event, `device_mem`, to group all corresponding memory operations related to the same `map` clause into a single event.

Figure 4 shows the function signature of the callback, which is attached to `device_mem` event. Since all conducted memory operations are recorded using a bitmap `device_mem_flag`, tools no longer need to group these operations by themselves, thereby mitigating the probability of incorrectly modeling data movements. Furthermore, `device_mem` events are associated with those effective `map` clauses. The runtime guarantees that whenever a `map` clause takes effect, a `device_mem` event is issued to notify the conducted semantics to all connected tools. Tools no longer need to track the reference count of each mapped variable, which helps reduce runtime overhead. Apart from ARBALEST-VEC, `device_mem` event has also been utilized in other tools [7].

We noticed that the latest LLVM 19 already supports these offloading-related OMPT events. After examining the implementation, we found that there still exists missing OMPT events. For example, `target`, `target_map` and `target_data_op` events associated with the `declare_target` construct. When OMPT support for device directives becomes fully functional in the future, we will switch to the built-in OMPT implementation in LLVM and stop maintaining our own implementation.

B. More Detailed Bug Report

To help programmers understand the root cause of detected data inconsistencies, we updated the bug report template and added more debug information related to the memory location. Compared to previous LLVM releases, LLVM 15 provides a more convenient way to retrieve the debug information of a **map** clause. For each `map` clause in an OpenMP program, LLVM embeds the clause's source code location and the involved mapped variable into a constant global object, `map_var_info`. Along with mapped variables, the corresponding `map_var_info` objects are also passed to the runtime when the OpenMP program invokes a device directive. We leverage the `map_var_info` object to help retrieve more accurate debug information. In ARBALEST-VEC, this object is stored along with the host and GPU addresses in an optimized interval-tree. When generating the bug report, ARBALEST-VEC can query the interval tree to translate memory locations into corresponding debug information.

Apart from using `map_var_info` objects, ARBALEST-VEC also use other techniques, such as compiler-time instrumentation, to retrieve more accurate debug information for global variables and dynamic arrays. Figure 5 shows the revised bug report ARBALEST-VEC generated. There is a new section (colored in orange) revealing that the data inconsistency is encountered on array `b`, with more detailed information about the invalid memory access. Compared to the original bug report in Figure 3, the additional section can help programmers understand the root cause of identified data inconsistencies efficiently.

C. Dedicated Shadow Memory and Vectorized Dynamic Analysis

As extensions to Archer, ARBALEST and ARBALEST-VEC can detect data races and data inconsistencies simultaneously. Both of them use shadow memory to track the status of mapped variables. Unlike ARBALEST, which reserves space within Archer's shadow memory, ARBALEST-VEC allocates a dedicated shadow memory to record the variable state. For every byte of application memory, ARBALEST-VEC allocates a one-byte shadow cell to store the variable state, known as the per-variable state machine (VSM). Additionally, four 32-byte shadow cells, maintained by the race detection routine inherited from Archer, are allocated for every eight bytes of application memory. Consequently, the overall memory overhead of ARBALEST-VEC is five times the memory usage of native execution.

Since ARBALEST-VEC has dedicated shadow memory for data inconsistency detection, it fully vectorizes such dynamic analysis using SIMD instructions. Figure 6 shows the routine to check VSMs for a section of application memory. With SIMD

```
==================================================
WARNING: ThreadSanitizer: data inconsistency (buffer overflow)
(pid=9482) on the target
  Read of size 4 at 0x7f936b0a4490 by thread T12:
    #0 .omp_outlined._debug__.1 error.cpp:21:12
    #1 .omp_outlined. error.cpp:17:3
    #2 __kmp_invoke_microtask <null>
    #3 .omp_outlined._debug__ error.cpp:17:3

  Location is heap block of size 2000000 at 0x7f936aebc000
allocated by main thread:
    #0 malloc tsan_interceptors_posix.cpp:667:5
    #1 DeviceTy::getTargetPointer(void*, void*, long, void*, bool, bool,
bool, bool, bool, bool, bool, AsyncInfoTy&, void*) <null>
(libomptarget.so.15)
    #2 main error.cpp:14:3
    #3 __libc_start_main libc-start.c:310

Variable/array involved in data inconsistency: b, max mapped
elements (4-byte element): 500000, element to be accessed:
500004

SUMMARY: ThreadSanitizer: data inconsistency (buffer overflow)
error.cpp:21:12 in .omp_outlined._debug__.1
==================================================
```

Fig. 5: ARBALEST-VEC's bug report for one of the data inconsistencies detected in Figure 1

```
1  ALWAYS_INLINE USED RawVsm* CheckVsmUtil(uptr addr,
       uptr size, u64 vmask) {
2    int range_mask = kVsmCellBitMap >> (kVsmCell -
       size);
3    uptr cell_start = RoundDown(addr, kVsmCell);
4    RawVsm *vp = MemToVsm(cell_start);
5    uptr offset =  addr - cell_start;
6    range_mask <<= offset;
7    m64 origin = _m_from_int64(LoadVsm8(vp));
8    m64 mask = _m_from_int64(vmask);
9    m64 result = _mm_cmpeq_pi8(_mm_and_si64(origin,
       mask), mask);
10   int rewrite_mask = _mm_movemask_pi8(result);
11   int error_bytes = range_mask ^ (range_mask &
       rewrite_mask);
12   // xor is non-zero => at least one bit is not
       the same as the range_mask
13   if (UNLIKELY(error_bytes)) {
14     uptr error_start_offset = __builtin_ffs(
         error_bytes) - 1;
15     return vp + error_start_offset;
16   } else {
17     return nullptr;
18   }
19 }
```

Fig. 6: The vectorized dynamic analysis for consecutive application memory

instructions, the dynamic analysis for data inconsistencies is converted to a group of bit operations upon the dedicated shadow memory for VSM, which improves the performance while achieving fine-grind byte-level dynamic analysis.

V. EVALUATION

In this section, we discuss the evaluations we conducted to gauge the effectiveness and efficiency of ARBALEST-VEC.

TABLE I: Execution time on SPEC-ACCEL in seconds

Benchmark	Original	ARBALEST	ARBALEST-VEC
503.postencil	1.74	39	37
504.polbm	9.80	90	80
514.pomriq	0.68	76	46
552.pep	30	82	66
554.pcg	23	412	389

TABLE II: Memory usage on SPEC-ACCEL in bytes

Benchmark	Original	ARBALEST	ARBALEST-VEC
503.postencil	267,858	823,448	954,518
504.polbm	843,399	2,536,552	2,955,568
514.pomriq	16,075	73,656	63,360
552.pep	1,841,735	9,218,620	7,382,508
554.pcg	2,500,654	8,684,624	8,682,766

A. Experiment Setup

We performed the evaluation on a single-node AMD server featuring dual 12-core 3.8 GHz Ryzen 9 3900X processors and 128 GB of memory, running Ubuntu 18.04.3 LTS. To mitigate the difference in infrastructure, we also ported the original implementation of ARBALEST to LLVM 15. Both tools are developed on the same checkout branch of LLVM 15; therefore, we can conduct a fair comparison between them.

For benchmarks, the evaluations utilized a group of open-source OpenMP benchmarks from SPEC-ACCEL [8], and all benchmarks are compiled using LLVM 15 with -O3. Additionally, we selected a suitable input size from the dataset to ensure that the benchmarks run long enough to minimize any bias introduced by the underlying operating system.

B. Comparison with ARBALEST on SPEC-ACCEL

Table I shows the execution time on five SPEC-ACCEL benchmarks. We found that ARBALEST-VEC achieved better performance on all benchmarks than ARBALEST. The dedicated shadow memory helped ARBALEST-VEC to mitigate the unnecessary interaction between data race detection and data inconsistencies, which may cause more slowdown due to the memory access ordering. In addition, SIMD instructions reduce the number of needed instructions to check a consecutive memory.

Apart from the execution time, we also compared the memory usage of the two tools. The detailed memory usage on each benchmark is listed in Table II. We found that ARBALEST-VEC offers lower time overhead than ARBALEST and exhibits more predictable memory usage. In the SPEC-ACCEL benchmark suite, ARBALEST-VEC shows a memory overhead ranging from 3.47× to 4.01×, with an average of 3.50×. While ARBALEST also has a memory overhead of 3.47×, ARBALEST-VEC maintains a comparable level of memory overhead. Notably, ARBALEST may incur significant memory overhead on certain OpenMP benchmarks (e.g., 552.pep), a problem that does not occur with ARBALEST-VEC.

VI. Related work

Dynamic debugging tools that target memory usage problems often use shadow memory to store historical access to certain memory addresses. AddressSanitizer [9] focuses on detecting out-of-bound access and use-after-free memory errors. The authors propose a new compact shadow memory encoding, as well as a customized memory allocator for the encoding. This efficient shadow mapping enables the tool to detect memory errors faster than previous similar works. The original implementation of ThreadSanitizer [10] uses 128 bits for each shadow cell. The most recent version reduces the size to 64 bits. During instrumentation, ThreadSanitizer inserts check for each read and write in the program. During runtime, the happens-before relation between two accesses is calculated by doing a bit-wise operation of the two shadow memory cells. MemorySanitizer [11] uses static compiler instrumentation to detect the use of uninitialized memory (UUM). It employs a 1-to-1 shadow mapping, as one-bit shadow memory corresponds to one-bit memory in the program. Compared with AddressSanitizer and ThreadSanitizer, the authors need to handle all possible instructions in the LLVM IR for MemorySanitizer.

The sanitizer tool family provides the foundation for developers to build upon, many of which are for OpenMP programs. Archer [4] annotates happens-before relationships in OMPT callbacks and uses ThreadSanitizer to do race detection. A similar approach can be found in TSAN-SPD3 [12]. However, the authors only reuse the instrumentation and shadow memory parts from ThreadSanitizer. The race detection algorithm is implemented upon a tree structure. In addition, researchers also introduced OpenMP sanitizer in 2019 [13], which detects mapping misuse through static analysis.

VII. Conclusion

In this paper, we introduce ARBALEST-VEC, a new tool designed to detect data inconsistencies in OpenMP offloading applications. ARBALEST-VEC addresses several shortcomings of its predecessor, ARBALEST, including improvements in accuracy, usability, and performance. Evaluation results show that ARBALEST-VEC incurs lower time overhead and offers more predictable memory usage compared to ARBALEST.

References

[1] L. Yu, J. Protze, O. Hernandez, and V. Sarkar, "A study of memory anomalies in openmp applications," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*. Springer, 2020, pp. 328–342.

[2] A. Schmitz, J. Protze, L. Yu, S. Schwitanski, and M. S. Müller, "Dataraceonaccelerator–a micro-benchmark suite for evaluating correctness tools targeting accelerators," in *European Conference on Parallel Processing*. Springer, 2019, pp. 245–257.

[3] L. Yu, J. Protze, O. Hernandez, and V. Sarkar, "Arbalest: dynamic detection of data mapping issues in heterogeneous openmp applications," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 464–474.

[4] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "ARCHER: effectively spotting data races in large OpenMP applications," in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 53–62.

[5] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185.

[6] J. Doerfert, J. Huber, and M. Cornelius, "Advancing openmp offload debugging capabilities in llvm," in *50th International Conference on Parallel Processing Workshop*, 2021, pp. 1–8.

[7] F. Jin, A. Tao, L. Yu, and V. Sarkar, "Visualizing correctness issues in openmp programs," in *Advancing OpenMP for Future Accelerators*. Cham: Springer Nature Switzerland, 2024, pp. 161–175.

[8] "SPEC ACCEL® Benchmark Suite 1.2," https://www.spec.org/accel/, 2017.

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[10] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*. ACM, 2009, pp. 62–71.

[11] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[12] L. Yu, F. Jin, J. Protze, and V. Sarkar, "Leveraging the dynamic program structure tree to detect data races in openmp programs," in *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2022, pp. 54–62.

[13] P. Barua, J. Shirako, W. Tsang, J. Paudel, W. Chen, and V. Sarkar, "OMPSan: Static Verification of OpenMP's Data Mapping constructs," in *International Workshop on OpenMP*. Springer, 2019.