2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00055

ARBALEST: Dynamic Detection of Data Mapping Issues in Heterogeneous OpenMP Applications

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

Joachim Protze
IT Center
RWTH Aachen University
Aachen, Germany
protze@itc.rwth-aachen.de

Oscar Hernandez
Computer Science Research
Oak Ridge National Laboratory
Oak Ridge, USA
oscar@ornl.gov

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

***Abstract*—From OpenMP 4.0 onwards, programmers can offload code regions to accelerators by using the *target offloading* feature. However, incorrect usage of target offloading constructs may incur data mapping issues. A data mapping issue occurs when the host fails to observe updates on the accelerator or vice versa. It may further lead to multiple memory issues such as use of uninitialized memory, use of stale data, and data race. To the best of our knowledge, currently there is no prior work on dynamic detection of data mapping issues in heterogeneous OpenMP applications.**

In this paper, we identify possible root causes of data mapping issues in OpenMP's standard memory model and the unified memory model. We find that data mapping issues primarily result from incorrect settings of `map` and `nowait` clauses in target offloading constructs. Further, the novel unified memory model introduced in OpenMP 5.0 cannot avoid the occurrence of data mapping issues. To mitigate the difficulty of detecting data mapping issues, we propose ARBALEST, an on-the-fly data mapping issue detector for OpenMP applications. For each variable mapped to the accelerator, ARBALEST's detection algorithm leverages a state machine to track the last write's visibility. ARBALEST requires constant storage space for each memory location and takes amortized constant time per memory access. To demonstrate ARBALEST's effectiveness, an experimental comparison with four other dynamic analysis tools (Valgrind, Archer, AddressSanitizer, MemorySanitizer) has been carried out on a number of open-source benchmark suites. The evaluation results show that ARBALEST delivers demonstrably better precision than the other four tools, and its execution time overhead is comparable to that of state-of-the-art dynamic analysis tools.

***Index Terms*—Dynamic Analysis, Concurrency Bug Detection, Data Mapping Issue, OpenMP, Accelerator**

I. INTRODUCTION

In recent years, accelerators, especially NVIDIA GPUs, are becoming widely available in high-performance computing (HPC) platforms. According to the latest Top500 list, there are 139 supercomputers using NVIDIA GPUs, which includes six out of the top ten machines. Considering the massive compute power of accelerators, programmers are turning to heterogeneous programming when developing parallel applications [1]–[3]. Many parallel programming models, such as OpenMP [4] and Kokkos [5], have added support for accelerators to follow the trend of heterogeneous programming.

OpenMP has been used extensively for multithreaded parallel programming in the past decades. Starting from the 4.0 version, OpenMP introduced a new feature, *target offloading*, which enables migrating computation to accelerators during the program execution. Target offloading is applicable to a broad spectrum of accelerators. It provides a set of generic constructs (*device directives*) to help programmers declare compute kernels (code regions to be executed on the accelerator) and data movement. Furthermore, target offloading delegates many burdensome programming tasks to the underlying runtime, for instance, memory management on the accelerator. Compared to other accelerator programming models such as CUDA [6] and ROCm [7], target offloading requires less engineering effort to achieve similar performance [8], [9].

In this paper, we focus on the data movement and memory management aspects of target offloading. Considering distinct architectures adopted by accelerator manufacturers, OpenMP utilizes a hardware-agnostic abstraction, *data mapping*, to represent data movement between the host and an available accelerator. A data mapping is specified by a `map` clause with two key parameters (see Figure 1):

- *mapped variable*, the variable or array section involved in the data mapping;
- *map-type*, the effect of data mapping, e.g., transferring the variable to/from the accelerator

For each mapped variable, there is an associated storage location on the host and accelerator. To comply with the OpenMP specification, we refer to these two kinds of storage as *original variable* (OV) and *corresponding variable* (CV), respectively. The OpenMP runtime automatically manages the lifecycle of CV and makes its physical address transparent to programmers. When compiling a compute kernel, the compiler transforms mapped variable accesses to operate on the CV. In certain OpenMP implementations, OV and CV may reside in the same physical memory (e.g., NVIDIA GPUs' Unified Memory). Otherwise, the two storage locations are independent and may have inconsistent values.

The OpenMP runtime takes care of all low-level operations related to data mapping (e.g., allocating/deallocating the corresponding copy, physical address lookup), and keeps the mapping transparent to programmers. Thus data mapping significantly reduces the engineering effort for experienced OpenMP programmers. On the other hand, correctly understanding OpenMP's data mapping rules remains challenging for

```
1  #define N 5000
2
3  int a[N], b[N*N], c[N];
4
5  init(a, b, c);
6
7  #pragma omp target
8    map(to:a[0:N])
9    map(alloc:b[0:N*N]) // mapping type should be "to"
10   map(tofrom:c[0:N])
11 {
12   #pragma omp teams distribute
13   #pragma omp parallel for
14     for(int i=0; i<N; i++)
15       for(int j=0; j<N; j++)
16         c[i]+=b[j+i*N]*a[j]; // data mapping issue
17 }
```

Fig. 1: Data mapping issue in DRACC_OMP_022. The `target` construct declares a compute kernel from line 11 to line 17. The associated `map` clause in line 9 indicates that array b's CV is allocated but not initialized, which causes a use of uninitialized memory (UUM) in line 16.

programmers. Multiple types of bugs may arise from incorrect data mappings such as use of uninitialized memory (UUM), use of stale data (USD), and data race. They may further affect the program execution, resulting in an erroneous output or even program crashes. According to feedback gathered in recent OpenMP hackathons, such incorrect usage of data mappings is a source of programming complexity. Programmers would like to be aware of these issues even if they turn out not to be bugs at runtime.

UUM and USD may stem from incorrect usage of `map` clause or other target offloading constructs, including a) missing a necessary data movement (e.g., the absence of a `map` clause or `update` construct), b) incorrect host variable or array section, and c) incorrect map-type. Moreover, failing to order data mappings and memory accesses may result in data races. In this paper, we refer to such errors resulting from incorrect data mappings as *data mapping issues*. Figure 1 illustrates a real-world example from DRACC. DRACC is a benchmark suite designed to evaluate the effectiveness of program analysis tools [10]. It reveals a number of concurrency bugs related to heterogeneous programming.

In contrast to well-studied concurrency bugs, such as data race [11]–[15], deadlock [16], [17], and barrier divergence [18], data mapping issues in heterogeneous computing are poorly understood. Data mapping issues can be manifested as different observable anomalies, and no existing tool can guarantee soundness and completeness when detecting data mapping issues. We have evaluated four dynamic analysis tools on benchmarks from DRACC: Valgrind[19], Archer [15], AddressSanitizer (ASan) [20], and MemorySanitizer (MSan) [21]. The result shows that none of them can accurately identify all known data mapping issues. We also observed that these tools could only tackle a subset of data mapping issues due to limitations in their detection algorithms (for details of this evaluation, please refer to Section VI).

Based on a comprehensive study of data mapping issues, we observe that the root cause of data mapping issues lies in the inconsistency between OV and CV. As a result, we can define a data mapping issue as follows:

Definition 1 (Data Mapping Issue)**.** *For a read on the host that can receive a value from a write to the same variable on the accelerator (or vice versa), a data mapping issue occurs if the read does not observe the write; namely, the read fails to return the value placed by the write.*

Based on Definition 1, an OpenMP application should always generate a consistent result regardless of the accelerator's features (e.g., unified memory). This definition describes the behavior of data mapping issues through the visibility of write operations. Since an operation's visibility can be examined at runtime, we propose ARBALEST, a dynamic data mapping issue detector for OpenMP applications. Inspired by cache coherence protocols, ARBALEST uses a state machine to track the state of each variable involved in a data mapping. A state transition is triggered when the application issues a memory access. ARBALEST reports a data mapping issue when a state machine transitions to the 'illegal' state. ARBALEST requires $\mathcal{O}(1)$ storage space for each variable, and takes $\mathcal{O}(1)$ time to complete a state transition.

ARBALEST is built upon Archer [15] to fully reuse the LLVM sanitizer infrastructure [22] and OMPT interface [23]. ARBALEST leverages a convenient feature of target offloading that the host can be treated as a 'virtual' accelerator. By setting the host as the destination of offloading, compute kernels will be executed by another group of threads, and memory transfers will be simulated by dynamic memory allocation (`malloc`/`free`) and memory block copy (`memcpy`). With these features, ARBALEST utilizes dynamic analysis techniques designed for CPU applications (e.g. shadow memory [24]) to implement the detection algorithm, which reduces the complexity of implementation.

In summary, this paper makes the following contributions:

- We present a clear definition of data mapping issues based on our study of the OpenMP specification and open-source benchmarks.
- We reason about the relation between data mapping issues and the underlying accelerator memory model. Our reasoning demonstrates that unified memory cannot prevent the occurrence of data mapping issues.
- We have implemented ARBALEST, an on-the-fly data mapping issue detector for OpenMP applications. ARBALEST can pinpoint all data mapping issues that the application encounters.
- We have conducted evaluations with benchmarks from DRACC [10] and SPEC-ACCEL [25] to compare the precision and performance of ARBALEST relative to four other tools. The result shows that ARBALEST reported all known data mapping issues in these benchmarks, while the average time overhead to the program execution is acceptable.

- ARBALEST is publicly available at https://github.com/lechenyu/Arbalest.git.

The remainder of this paper is organized as follows: In Section II, we introduce target offloading's execution model and all associated constructs. In Section III, we present case studies for known data mapping issues and explain the refined definition. Section IV illustrates ARBALEST's detection algorithm, followed by the implementation of ARBALEST in Section V. We evaluate the correctness and performance of ARBALEST in Section VI. We summarize some related work in Section VII, and finally, in Section VIII, we briefly conclude with some possible directions for future work.

II. OpenMP Target Offloading

As a major change to OpenMP's execution model, target offloading was added to the specification in OpenMP 4.0. The community continued the efforts to add more functionalities and clarifications in subsequent releases. In this section, we introduce the details of target offloading in accordance with the latest OpenMP 5.1.

A. Execution Model

OpenMP applies a generic execution model to support accelerators from different manufacturers (e.g., NVIDIA, AMD, ARM). An accelerator is abstracted as a compute resource with independent processing units and memory space. The execution model is host-centric such that an OpenMP application consists of a *host program* and a number of *target regions*. OpenMP uses the term target region to refer to a compute kernel in the program. The host program begins execution on the host (usually CPU), and may offload computation and data to available accelerators. After submitting a target region to an accelerator, the host program either proceeds or waits for the termination of the target region.

A target region may be executed on a broad spectrum of accelerators. A well-formed OpenMP application will show consistent behavior when offloading the same target region to different accelerators. In addition, a target region can also be offloaded to an available CPU. By setting a CPU as the target device, programmers can examine the interaction between the host program and target regions using common debuggers without GPU support (e.g., GDB, LLDB). This configuration eases the debugging for OpenMP applications.

B. Device Directives

The corresponding constructs for target offloading are *device directives*, including:

- the `target` construct annotating a target region,
- the `target data` construct declaring data mappings for the enclosed code region,
- the `target enter data` and `target exit data` constructs for an unstructured version of `target data` construct, and
- the `target update` construct indicating a synchronization between OV and CV.

The comprehensive behaviors of device directives are described in section 2.14 of the OpenMP specification [4].

For `target`, `target data`, `target enter data`, and `target exit data` constructs, multiple `map` clauses may be present to specify data mappings. Proper memory transfer will be carried out at the beginning and end of the associated code region, according to the map-types of specified data mappings.

Table I lists the semantics of predefined map-types in terms of memory transfer. To mitigate unnecessary memory transfer, OpenMP applies a reference counting algorithm along with data mapping. The underlying runtime maintains a counter for each mapped variable to indicate whether the variable's CV has been created. No memory transfer will be carried out if the CV already exists on the accelerator.

For `target update` construct, `to` and `from` clauses specify the direction of memory transfer. The reference counting is not applied to this construct.

If a `nowait` clause is present in a device directive, the host thread can continue execution after launching the construct; otherwise the host thread will block until the construct terminates. We refer to target regions with `nowait` clauses as *asynchronous compute kernels*, and without `nowait` clauses as *synchronous compute kernels*.

III. Data Mapping Issue

According to the specification, an accelerator's architecture details are transparent to OpenMP. Thus the data mapping semantics in an OpenMP application should be consistent regardless of the underlying accelerator. However, the runtime behavior of data mapping issues may vary among distinct accelerator memory models, e.g., architectures with/without *unified memory* [6]. In this section, we first introduce a few case studies to analyze probable runtime behaviors of data mapping issues in a separate memory model. Furthermore, we take unified memory into account and discuss its effect on data mapping issues.

A. Separate Memory Model

OpenMP can be implemented on hardware exposing a separate memory model. Host and each accelerator have isolated memory space, and explicit memory transfers are indispensable to maintain the consistency between a variable's OV and CV. According to Definition 1 and the semantics of device directives, the probable runtime behaviors of data mapping issues are UUM, USD, and data race.

Figure 1 presents a data mapping issue resulting in a UUM. To illustrate the other two types of data mapping issues, we show a buggy OpenMP application in Figure 2. There are two data mapping issues residing in lines 5 and 16. The read in line 5 results in a USD as it fails to observe the write in line 3. The root cause is the incorrect map-type of variable `a` in the first target region. To fix this data mapping issue, a's map-type should be set to `tofrom`. For line 16, the result of the read to a is nondeterministic. Due to the `nowait` clause, the target region from lines 9 to 12 may happen in parallel with the host program, and hence the memory transfer at the end of

TABLE I: Map-types in OpenMP. The corresponding semantics is described in pseudocode. `OV` and `CV` refer to a mapped variable's storage location on the host and accelerator. `ref_count` returns the associated counter of `CV`. `exist` checks whether `CV` has been created on the accelerator, namely `ref_count(CV)==0`. `new` allocates a memory block for `CV` in accelerator memory, and `delete` frees the allocated memory block. `memcpy(dst, src)` copies data between `OV` and `CV`.

Map-Type	Effect on Entry to the Associated Region
to, tofrom	$if\ (!exist(CV))\ \{$ $new\ CV;$ $memcpy(CV, OV);$ $ref\_count(CV) = 1;$ $\}\ else\ \{$ $ref\_count(CV) \mathrel{+}= 1; \}$
from, alloc	$if\ (!exist(CV))\ \{$ $new\ CV;$ $ref\_count(CV) = 1;$ $\}\ else\ \{$ $ref\_count(CV) \mathrel{+}= 1; \}$
Map-Type	**Effect on Exit from the Associated Region**
from, tofrom	$ref\_count(CV) \mathrel{-}= 1;$ $if\ (ref\_count(CV) == 0)\ \{$ $memcpy(OV, CV);$ $delete\ CV; \}$
to, alloc, release	$ref\_count(CV) \mathrel{-}= 1;$ $if\ (ref\_count(CV) == 0)\ \{$ $delete\ CV; \}$
delete	$ref\_count(CV) = 0;$ $delete\ CV;$

```
1  int a = 1;
2  #pragma omp target map(to: a)
3    a += 1;
4  // read stale data on the host
5  printf("a = %d\n", a);
6
7  #pragma omp target data map(tofrom: a)
8  {
9    #pragma omp target nowait
10   {
11     a = 3;
12   }
13   a += 1;
14 }
15 // nondeterministic result of a
16 printf("a = %d\n", a);
```

Fig. 2: Possible Data Mapping Issues in OpenMP Applications

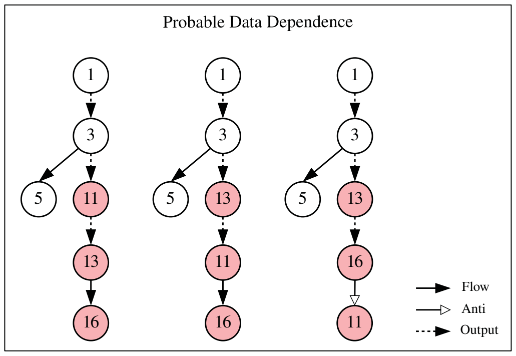


Fig. 3: Dynamic Data Dependence Graph for Figure 2

the target data region (line 14) may conflict with the memory access in lines 13 and 16. The read in line 16 may return the value placed by the write in line 11 or 13, depending on the interleaving of the host program and the target region. Figure 3 shows the corresponding data dependence graphs for probable interleavings. To fix the data mapping issue in line 16, the application should maintain a correct happens-before relation for the host and accelerator, e.g., adding a synchronization operation before the write to a in line 13.

B. Unified Memory Model

To simplify accelerator programming, recent accelerators adopt unified memory, a virtual shared memory space among the host and accelerators. Each memory location in unified memory is *transparently accessible* to all processing units, and programmers do not need to insert explicit memory transfers into the application. Leveraging unified memory to implement target offloading is optional for an OpenMP runtime. An OpenMP applications can use a `requires` directive to specify the necessity of unified memory for the application's correctness.

With unified memory, programmers might assume that modifying a variable on the host is directly visible on the accelerator, which prevents the occurrence of data mapping issues. For example, Pascal and later NVIDIA GPUs apply an on-demand page migration policy for unified memory. A GPU page fault is raised when the GPU accesses an absent page, and the GPU driver will automatically transport the page from host to GPU. On-demand page migration helps resolve data mapping issues in a data-race-free OpenMP application.

Without a cache coherence protocol, data mapping issues might still occur. OpenMP introduces flush operations to synchronize threads' temporary view (cache) with the memory space, removing the need for cache coherence. Therefore, data mapping issues may arise when updates on the host are not flushed into memory before subsequent reads on the GPU. Considering the implicit cross-device flush operations before and after each target region, such buggy cases can only happen if updates on the host and reads on the GPU occur concurrently without proper synchronization.

C. Repairing Data Mapping Issues

With an integrated static/dynamic analysis module, an OpenMP implementation can repair a subset of data mapping issues. Some pioneering work has been conducted in this direction [26], albeit the prototype has not been deployed for production use. When identifying data mapping issues resulting in USDs, the OpenMP runtime can carry out memory transfers between OV and CV to make their values consistent. For data mapping issues leading to data races, the compiler can insert additional *depend* clauses to the associated target regions or provide compiler diagnostic messages to help programmers identify the detected bugs' root causes.

IV. Dynamic Data Mapping Issue Detection

In this section, we present our approach for data mapping issue detection and describe its implementation in ARBALEST.

A. Variable State Machine

Definition 1 indicates that the root cause of data mapping issues is the inconsistency between OV and CV. Some OpenMP implementations might allocate a shared memory location for OV and CV [27], but programmers should not rely on a specific implementation when examining an OpenMP application's correctness.

Assume OV and CV are independent, and all operations except memory transfer can only access the storage location on the current device. A write to CV makes OV's value *invalid*. Without a synchronization, subsequent reads to OV always retrieve the invalid value, namely a data mapping issue. Similarly, a write to OV followed by reads to CV also results in data mapping issues. Therefore, the validity of OV and CV can be utilized to determine the occurrence of data mapping issues.

We propose a *variable state machine* (VSM) to track the status of a mapped variable. For a mapped array section, each element is tracked independently by VSM. As depicted in Figure 4, VSM consists of four distinct states:

- $invalid$, neither of the two storage locations has a valid value,
- $host$, only OV has the valid value,
- $target$, only CV has the valid value, and
- $consistent$, the two storage locations are consistent and valid.

The following operations can cause transitions which are depicted as edges in VSM:

- $read_{host}$/$read_{target}$, which gets the value from OV/CV;
- $write_{host}$/$write_{target}$, which sets a new value into OV/CV;
- $update_{host}$/$update_{target}$, which synchronizes OV and CV using the value in CV/OV; and
- $allocate$/$release$, which allocates/deallocates CV on the accelerator.

The associated label on an edge indicates the operations triggering the transition. For conciseness, the subscript is omitted if both two kinds of read, write, or update operations trigger the same transition in a certain state.

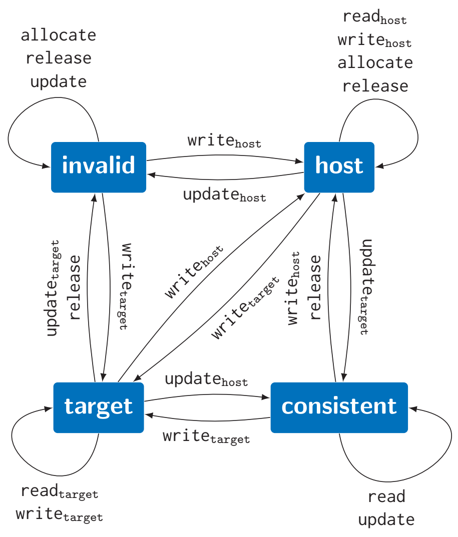


Fig. 4: Per Variable State Machine

B. States and Transitions in VSM

Invalid: no storage location has a valid value. VSM uses $invalid$ as the initial state since each variable is uninitialized at the beginning. A data mapping issue is reported when a read operation to OV/CV happens. In $invalid$, a write on the host/accelerator makes the state jump to $host$/$target$, while other operations do not change the state.

Host: OV has a valid value, and CV is either unallocated or invalid. A data mapping issue is reported when a $read_{target}$ operation happens. In $host$, A $write_{target}$ operation makes CV hold the latest value, so that the variable's state jumps to $target$. A memory transfer from CV to OV ($update_{host}$) changes the state to $invalid$ since OV has been overwritten by the invalid value in CV. In addition, the variable's state enters $consistent$ after a memory transfer from OV to CV ($update_{target}$) synchronizes the two storage locations.

Target: similar to $host$ except that OV has an invalid value. A data mapping issue is reported when a $read_{host}$ operation happens. For transitions, the key difference from $host$ is that there are two operations switching the variable's state from $target$ to $invalid$ ($update_{target}$ and $release$).

Consistent: both the OV and CV are valid. Thus no data mapping issue will occur in $consistent$. A write operation on the host/target changes the state to $host$/$target$ as the two storage locations are no longer consistent.

In total there are three situations resulting in data mapping issues: a) a $read_{host}$/$read_{target}$ operation in $invalid$, b) a $read_{target}$ operation in $host$, and c) a $read_{host}$ operation in $target$. In VSM, there are no corresponding transitions for these situations. VSM reports data mapping issues when it fails to find out the next state for the current operation.

C. The Complexity of VSM Based Detection Algorithm

For higher accuracy, ARBALEST applies VSM at 8-byte granularity when examining memory accesses. For every 8-

byte of a mapped variable/array sections, ARBALEST allocates a fixed-size *shadow memory* [24] to record those states in Figure 4. The shadow memory is a 2-tuple that marks the 8-byte memory section's validity in OV and CV. Upon allocating the variable on the host, all associated tuples are initialized as $invalid$ ($[Host: 0, Accel: 0]$) because the variable's CV has not been created and its OV has not been explicitly initialized. The shadow memory is attached to OV and bijectively mapped to OV's address. ARBALEST uses an interval tree to maintain the relationship between OV and CV. When a memory access to CV happens, a $\mathcal{O}(\log_2(m))$-time lookup in the interval tree is conducted to retrieve the associated shadow memory, where m is the number of mapped variables/array sections. Considering m is small in most OpenMP applications, and ARBALEST can cache the latest lookup, the overhead of lookup will be amortized over the total number of memory accesses.

For all operations except read, ARBALEST performs an $\mathcal{O}(1)$-time update to the corresponding state. When encountering a read operation, ARBALEST conducts an $\mathcal{O}(1)$-time comparison to detect data mapping issues. In addition, each access to CV needs an $\mathcal{O}(\log_2(m))$-time lookup to locate OV and retrieve the state. In summary, ARBALEST requires linear space with the size of mapped variables and amortized $\mathcal{O}(1)$ time per operation. Moreover, ARBALEST implements VSM in a lock-free manner. The $\mathcal{O}(1)$-time update and comparison are conducted by the atomic compare-and-swap instruction, which allows a fully concurrent analysis during program execution.

Applying VSM at byte-level granularity is requisite for soundness. For a mapped variable, OpenMP does not enforce any constraints on the granularity of operations, and hence every byte may be manipulated independently. Assuming a compute kernel only modifies a few bytes in a CV, and fails to transfer the update back to the corresponding OV. A coarse-grained check may lead to false alarms if subsequent read operations only access those intact bytes in OV. Since most operations in scientific applications are performed in double-precision arithmetic, ARBALEST keeps track of read and write operations at 8-byte granularity to assure soundness.

The state transition diagram shown in Figure 4 assumes that the OpenMP application only utilizes a single accelerator. By extending states in VSM, the algorithm can support multiple accelerators. For an OpenMP application using n accelerators, the variable state is an $(n+1)$-tuple in which n elements mark the validity of the n associated storage locations on accelerators. The space overhead increases to $\mathcal{O}(n+1)$, and accordingly, time complexity also becomes $\mathcal{O}(n+1)$ to maintain the state.

D. Extension for Buffer Overflow

An OpenMP application may encounter buffer overflow after setting an erroneous array section in the `map` clause. For example, the host program only maps the first half of an array to the accelerator, while the compute kernel loops through the whole array. Since the OpenMP runtime manages CV's physical address, the buffer overflow becomes an undefined behavior. It may retrieve a valid value from an adjacent memory location which happens to be another variable's CV. In such a situation, the buffer overflow does not result in a data mapping issue, but the application's final output is not trustworthy.

ARBALEST extends the definition of data mapping issues to capture buffer overflows in a CV. When a compute kernel accesses a mapped array section, the operation must perform on the array's CV; otherwise, ARBALEST will report an error. ARBALEST leverages the interval tree to find out buffer overflows. Using LLVM IR, ARBALEST records CV's base address. For each memory access to CV, ARBALEST compares the accessed physical address with CV's base address. If the two addresses belong to different intervals in the interval tree, ARBALEST reports it as a data-mapping-related buffer overflow.

E. General Data Mapping Issue Detection Method for OpenMP Applications

The VSM based detection algorithm precisely reports data mapping issues in the observed execution trace. These data mapping issues are real bugs in the application since they further lead to UUM or USD. However, VSM only examines a single schedule of compute kernels. For OpenMP applications using asynchronous compute kernels, there exist multiple possible schedules. VSM may lead to false negatives if the hidden data mapping issues do not manifest in the observed schedule.

To tackle asynchronous compute kernels, we introduce Theorem 1, a sufficient and necessary condition for data-mapping-issue-freedom.

Theorem 1 (Data-mapping-issue-freedom Theorem)**.** *For an OpenMP application containing asynchronous compute kernels, the application is free of data mapping issues if*

1) *the OpenMP application is free of data races, and*
2) *no data mapping issue is detected by VSM when executing all asynchronous compute kernels in a synchronous manner (the host thread suspends until the termination of the asynchronous compute kernel).*

Due to the page limit, we omit the formal proof of Theorem 1. We present informal proof here to reveal the relationship between data races and data mapping issues.

Informal Proof of Theorem 1.

Hypothesis 1, data-race-freedom, indicates that all $read$, $write$, and $update$ operations to the same variable are correctly ordered. Hypothesis 2 indicates that in a specific schedule, there is no data mapping issue. Let α be the observed execution trace in which asynchronous compute kernels execute in a synchronous manner, and β be the trace of another probable schedule. We assume the schedule of compute kernels does not affect the program path at runtime. So that α and β perform the same set of operations. We prove Theorem 1 by contradiction.

Suppose the two hypotheses hold, and there is a data mapping issue in β triggered by a $read$ operation r. According to the definition of data mapping issue, we have that α returns the value of a write operation w in α, and a different write operation $\hat{w}$ in β. It indicates in β

- the happens-before order between w and $\hat{w}$ is violated, or

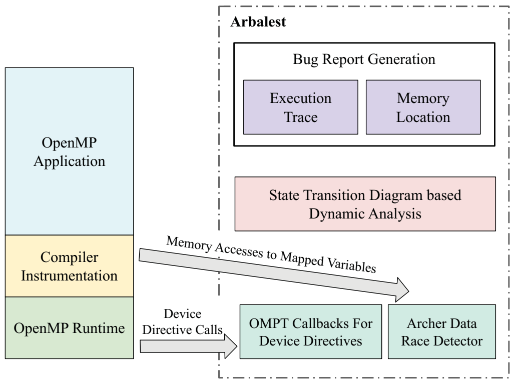

Arbalest
Bug Report Generation
Execution Trace
Memory Location
OpenMP Application
State Transition Diagram based Dynamic Analysis
Compiler Instrumentation
Memory Accesses to Mapped Variables
OpenMP Runtime
Device Directive Calls
OMPT Callbacks For Device Directives
Archer Data Race Detector

Fig. 5: ARBALEST Architecture

- an $update$ operation, which is performed between w and r in α, is reordered before w or after r.

Both cases indicate data races in β, which contradicts Hypothesis 1 that the application is free of data races. Thus the assumption cannot be satisfied, and Theorem 1 is proved. □

V. ARBALEST

ARBALEST is implemented as an extension to Archer to reuse the instrumentation pass, OMPT callbacks, and shadow memory implementation. As shown in Figure 5, ARBALEST consists of three modules: a) runtime data collection, b) dynamic analysis, and c) bug report generation.

When analyzing an OpenMP application, ARBALEST uses the 'x86_64-pc-linux-gnu' target architecture in LLVM to simulate the offloading of compute kernels and memory blocks. Compute kernels are executed by a group of CPU threads, and memory transfer is simulated by dynamic memory allocation (malloc/free) and memory block copy (memcpy).

A. Runtime Data Collection

During the program execution, ARBALEST monitors the invocation of OpenMP constructs as well as memory accesses to mapped variables. It leverages the OMPT interface to capture the invocation events of OpenMP constructs. Because the OMPT implementation in LLVM 9.0 is incomplete for device directives, ARBALEST uses an alternative OMPT implementation developed by researchers at Rice University[1]. For memory accesses, ARBALEST utilizes Archer's instrumentation pass. The injected callbacks provide the details of memory accesses (e.g., accessed memory location) to the dynamic analysis.

When implementing ARBALEST, we found that OMPT does not provide correct mapping information for global variables. We reported this omission to the OMPT committee. We proposed that the OpenMP runtime should provide event callbacks for those implicit data mappings, which are carried out during the initialization of the runtime or the device. We also reported some shortcomings in OMPT with respect to data mapping and synchronization. For example, the callback for the `target` construct does not distinguish synchronous and asynchronous target regions, which limits the possible dynamic analysis for asynchronous compute kernels.

[1]https://github.com/jmellorcrummey/llvm-openmp-5/

TABLE II: Shadow State in ARBALEST

Field	Size
IsOVValid	1 bit
IsCVValid	1 bit
IsOVInitialized	1 bit
IsCVInitialized	1 bit
TID (Thread Id)	12 bits
Scalar Clock	42 bits
IsWrite	1 bit
Access Size (1, 2, 4 or 8)	2 bits
Address Offset (0..7)	3 bits

B. Shadow State

ARBALEST reuses Archer's shadow memory implementation to store the variable state. For every aligned 8-byte word of application memory, Archer maps it into four *shadow states* using direct address mapping. These four shadow states store vector clocks for data race detection. ARBALEST reserves the first four bits of a shadow state for data mapping issue detection. The corresponding encoding is presented in Table II. ARBALEST uses the first two bits to represent the states in VSM. The following two bits are used for the bug report. When reporting data mapping issues, ARBALEST also delineates observed anomalies in the bug report. UUMs and USDs can not be distinguished by VSM, so that ARBALEST uses two additional bits to record the initialization of OV and CV, respectively.

C. Bug Report

ARBALEST reuses the bug report template in Archer. When a data mapping issue is detected, ARBALEST fills in corresponding debugging information into the template to generate a comprehensive bug report. An example of ARBALEST's bug report is present in Figure 7. The bug report illustrates the observed anomaly, as well as the stack trace and memory locations involved in the data mapping issue. Such information can help programmers gain a better understanding of the detected bug.

VI. EVALUATION

We validate the effectiveness of ARBALEST with two sets of experiments. First, we compare ARBALEST's precision with other analysis tools using the DRACC benchmark suite [10]. Second, we compared ARBALEST's performance relative to four state-of-the-art dynamic analysis tools on SPEC-ACCEL 1.2 [25].

A. Compared Analysis Tools

To the best of our knowledge, currently there are no other dynamic analysis tools designed for data mapping issues. The closest work to ARBALEST is dynamic data race/memory error detectors. These tools may capture a data mapping issue when it leads to a data race/memory error. Similar to ARBALEST, both data race detectors and memory error detectors consist of an instrumentation module and a shadow-memory-based detection algorithm. The instrumentation module determines the memory accesses to be checked, and the shadow memory stores the status of memory locations at runtime.

To evaluate the precision and performance, we compared ARBALEST with a state-of-the-art data race detector and three memory error detectors: Valgrind [19], Archer [15], AddressSanitizer (ASan) [20], and MemorySanitizer (MSan) [21]. Valgrind is a dynamic instrumentation framework with a group of debugging and analysis tools. Valgrind's *memcheck* tool is a widely-used memory anomaly detector for C/C++ applications. Archer is a data race detector for OpenMP applications. It is available in LLVM's OpenMP sub-project as an extension to the ThreadSanitizer (TSan) race detector. ASan and MSan are another two tools from LLVM. ASan is capable of detecting multiple kinds of memory errors (e.g., buffer overflow, use of freed memory) while MSan is a dynamic detector for UUMs. ARBALEST and the three tools from LLVM utilize the same tool infrastructure (LLVM sanitizer infrastructure [22]) so that the difference in implementation has less effect on the evaluation results.

In our evaluation, we used Valgrind 3.15, Archer from LLVM 9.0, and ASan and MSan from LLVM 11.0. We selected tools from different LLVM releases because ASan and MSan do not work correctly before LLVM 11.0. They either crush or report false alarms on a majority of OpenMP benchmarks. We found that the program crash results from a segment fault in the OpenMP runtime, and the false alarms are related to the OpenMP constructs. For ASan and MSan from LLVM 11.0, we did not observe similar issues in the evaluation. Currently we are consulting the LLVM OpenMP team to locate the root cause of these issues.

B. Experimental Setup

Our experimental machine is a compute node of the NEC cluster. The machine has two 24-core Intel Platinum 8160 processors, 384 GB of memory, and two NVIDIA Volta GPUs, running CentOS 7 (Linux 3.10.0). For precision comparison, we conducted experiments on all 56 OpenMP benchmarks from DRACC 1.0[2]. For performance comparison, we measured the time and space overhead for ARBALEST and the other four tools, using five benchmarks from SPEC ACCEL 1.2. The five benchmarks were compiled with Clang 9.0/11.0 at the -O3 optimization level, and executed by 24 threads bound to one socket. We skipped the other ten OpenMP benchmarks in SPEC ACCEL 1.2 because Clang 9.0 cannot compile them successfully.

C. Precision Evaluation on DRACC

Table III shows the precision comparison on 16 buggy DRACC benchmarks. Each benchmark has a known data mapping issue which will result in a memory error at runtime. Column 1 records the benchmark ID (e.g., 22 refers to benchmark DRACC_OMP_022), column 2 describes the

[2]https://github.com/RWTH-HPC/DRACC

TABLE III: Effectiveness Comparison on DRACC Benchmarks

Benchmark ID	Effect	Effectiveness				
		Arbalest	Valgrind	Archer	ASan	MSan
22, 24, 49, 50, 51	UUM	✓	-	-	-	✓
23, 25, 28, 29, 30, 31	BO	✓	✓	-	✓	-
26, 27, 32, 33, 34	USD	✓	-	-	-	-
Overall		16/16	6/16	0/16	6/16	5/16

resulting memory error, and columns 3 - 7 describe the results of ARBALEST and other dynamic analysis tools. In column 2, UUM, BO, and USD stand for use of uninitialized memory, buffer overflow, and use of stale data, respectively. In columns 3 - 7, '✓' denotes the tool correctly reports the data mapping issue, while '$-$' denotes the data mapping issue is not detected. Moreover. Table III skips the results on the other 40 DRACC benchmarks because none of the five tools report a false positive when the benchmark is free of data mapping issues.

ARBALEST outperformed the other four tools on precision. ARBALEST precisely reported data mapping issues in all 16 benchmarks. Valgrind only reported 6 data mapping issues, and Archer did not report any data mapping issues. ASan and MSan detected 6 and 5 data mapping issues out of the 16 known bugs. Apart from ARBALEST, each tool can only tackle a subset of data mapping issues. Besides, only ARBALEST detected the data mapping issues in the third row. For DRACC_OMP_026, DRACC_OMP_027, DRACC_OMP_032, and DRACC_OMP_033, the data mapping issues finally result in USDs in the program execution. Since none of the other four tools take USDs into account, they failed to capture the manifested data mapping issues. For DRACC_OMP_034, the data mapping issue leads to a UUM in a compute kernel. MSan and Valgrind missed this bug, even if they are designed for UUMs. It turned out that MSan and Valgrind did not precisely model the semantics of all OpenMP constructs due to the lack of OMPT.

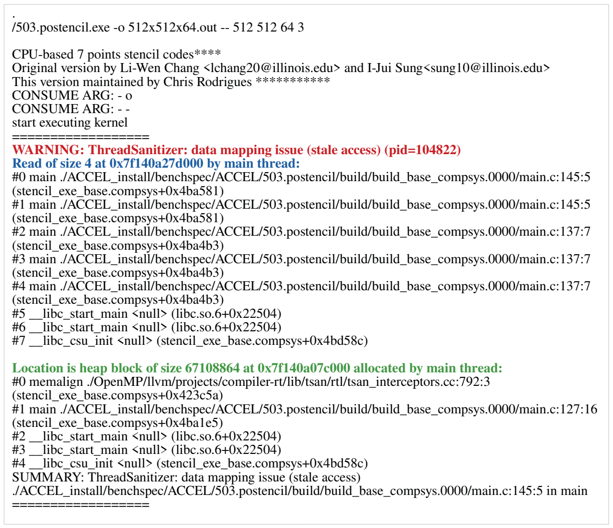
D. Experiment on 503.postencil

We applied a real-world data mapping issue to measure ARBALEST's effectiveness further. 503.postencil is a performance benchmark from the SPEC-ACCEL benchmark suite. According to the changelog [28], there exists a data mapping issue in the 1.2 version. The buggy code snippet is shown in Figure 6. The function `cpu_stencil` is a compute kernel. After launching it in line 137, the host program swaps the pointers of the two variables which have been mapped to the accelerator. The swap operation makes the two variables' CVs inconsistent with their OVs. In odd iterations, the calculation results on the accelerator will not be copied back to the correct OVs, which leads to a data mapping issue.

Figure 7 presents ARBALEST's result on 503.postencil. The stack trace indicates that there is a data mapping issue in line 139, which is the output function. The bug report demonstrates that ARBALEST successfully detects the hidden data mapping issue.

```
124 float *h_A0, *h_Anext;
125 int size = nx * ny * nz;
126 size_t alignment = SPEC_ALIGNMENT_SIZE;
127 h_A0 = memalign(alignment, sizeof(float)*size);
128 h_Anext = memalign(alignment, sizeof(float)*size);
129
130 generate_data(h_A0,nx,ny,nz,0);
131 generate_data(h_Anext,nx,ny,nz,0);
132
133 #pragma omp target data map(to: h_A0[0:size])
        map(tofrom: h_Anext[0:size])
134 {
135   printf("start executing kernel\n");
136   for(int t=0; t<iteration; t++){
137     cpu_stencil(c0,c1,h_A0,h_Anext,nx,ny,nz);
138     float *temp = h_A0;
139     h_A0 = h_Anext;
140     h_Anext = temp;
141   }
142 }
143
144 if (param->outFile) {
145   outputData(param->outFile,h_Anext,nx,ny,nz);
146 }
```

Fig. 6: Buggy Code Section in 503.postencil



```
.
/503.postencil.exe -o 512x512x64.out -- 512 512 64 3

CPU-based 7 points stencil codes****
Original version by Li-Wen Chang <lchang20@illinois.edu> and I-Jui Sung<sung10@illinois.edu>
This version maintained by Chris Rodrigues ***********
CONSUME ARG: - o
CONSUME ARG: - -
start executing kernel
==================
WARNING: ThreadSanitizer: data mapping issue (stale access) (pid=104822)
Read of size 4 at 0x7f140a27d000 by main thread:
#0 main ./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:145:5
(stencil_exe_base.compsys+0x4ba581)
#1 main ./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:145:5
(stencil_exe_base.compsys+0x4ba581)
#2 main ./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:137:7
(stencil_exe_base.compsys+0x4ba4b3)
#3 main ./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:137:7
(stencil_exe_base.compsys+0x4ba4b3)
#4 main ./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:137:7
(stencil_exe_base.compsys+0x4ba4b3)
#5 __libc_start_main <null> (libc.so.6+0x22504)
#6 __libc_start_main <null> (libc.so.6+0x22504)
#7 __libc_csu_init <null> (stencil_exe_base.compsys+0x4bd58c)

Location is heap block of size 67108864 at 0x7f140a07c000 allocated by main thread:
#0 memalign ./OpenMP/llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:792:3
(stencil_exe_base.compsys+0x423c5a)
#1 main ./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:127:16
(stencil_exe_base.compsys+0x4ba1e5)
#2 __libc_start_main <null> (libc.so.6+0x22504)
#3 __libc_start_main <null> (libc.so.6+0x22504)
#4 __libc_csu_init <null> (stencil_exe_base.compsys+0x4bd58c)
SUMMARY: ThreadSanitizer: data mapping issue (stale access)
./ACCEL_install/benchspec/ACCEL/503.postencil/build/build_base_compsys.0000/main.c:145:5 in main
==================
```

Fig. 7: ARBALEST's Output on 503.postencil

E. Time Overhead

Since DRACC benchmarks are not designed for performance evaluation, we carried out the performance comparison on SPEC-ACCEL benchmarks. Figure 8 shows the execution time of each tool. "Native-GPU" and "Native-CPU" denote the original execution time on the NVIDIA GPU and CPU.

On all benchmarks, ARBALEST incurred similar overheads with Archer. Since data race detection is much more expensive, ARBALEST's execution time is dominated by Archer's race detection routine. ARBALEST outperformed Valgrind on three benchmarks (504.polbm, 514.pomriq, and 554.pcg) and achieved similar execution time on the other two benchmarks. Valgrind uses dynamic binary instrumentation to monitor the

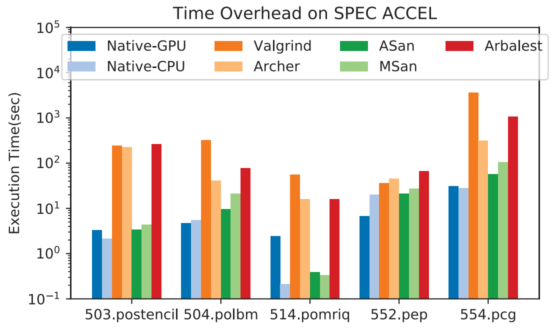


Fig. 8: Time Overhead on SPEC ACCEL

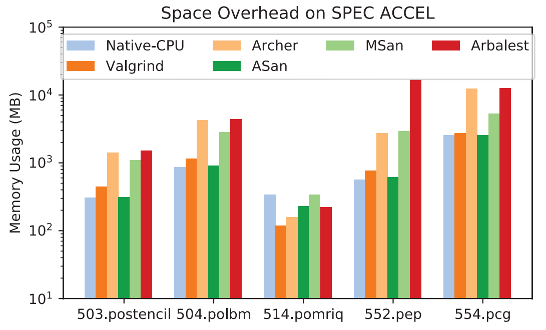


Fig. 9: Space Overhead on SPEC ACCEL

program execution. Therefore the overhead is much higher than the LLVM-based compile-time instrumentation. In addition, ARBALEST's performance is close to ASan and MSan on two benchmarks.

In total ARBALEST incurred 3.32× - 120.2×s slowdown to the native execution on the CPU. Considering the precision of ARBALEST, the time overhead is acceptable. Currently we make ARBALEST a pure dynamic analysis tool. In the future, we will apply some static analysis techniques to improve ARBALEST's performance further.

F. Space Overhead

Figure 9 lists the memory usage on five SPEC-ACCEL benchmarks. Since all tools except Valgrind utilize the same shadow memory implementation in the LLVM sanitizer framework, the incurred space overheads are quite close. ARBALEST encodes its own state into Archer's shadow state when detecting data mapping issues, so that ARBALEST should not incur additional space overhead. On all benchmarks except 552.pep, ARBALEST's memory usage is close to Archer, which matches our expectations. On 552.pep, ARBALEST incurred significant space overhead to the program execution. We estimate that ARBALEST's operations on the shadow state have a negative effect on the shadow state eviction policy. We are working on this benchmark to locate the root cause of the unexpected space overhead.

G. Comparison with Static Data Mapping Issue Detection

OMPSan [29] is a static data mapping issue detector for OpenMP applications. It assumes that the *serial elision* property is held in OpenMP[3]. By comparing an OpenMP application's dataflow with the serial elision version, OMPSan reports all inconsistent def-use relations as data mapping issues. As a static analysis tool, OMPSan reasons about the dataflow without actually running the OpenMP application, so that the overhead is relatively low.

Because OMPSan is still in the development phase, we did not find an available release. We have contacted OMPSan's authors to get the evaluation results. The authors confirmed that OMPSan pinpointed all 16 known data mapping issues in the DRACC benchmark suite. However, OMPSan missed the data mapping issue in 503.postencil because of the complex dataflow. OMPSan's effectiveness relies heavily on the alias analysis pass. Due to the lack of runtime information, OMPSan's alias analysis may generate inaccurate results, which further leads to false positives and false negatives.

VII. Related Work

In this section, we relate our work to the state-of-art studies in three areas, including data mapping issue detection, cache coherence, and programming models similar to OpenMP.

A. Data Mapping Issue Detection and Cache Coherence

Unlike ARBALEST and OMPSan, prior work has proposed other schemes to tackle data mapping issues, for example, data mapping issue avoidance. X10CUDA [30] and OpenARC [31] are two compiler frameworks applying automatic memory management to avoid data mapping issues. The underlying runtime tracks the coherence status of shared data using a state transition diagram, which is similar to ARBALEST. Necessary memory transfers are inserted into the application to avoid data mapping issues. However, X10CUDA and OpenARC track the coherence status at a coarse granularity for better performance (a single state for the whole array section), and neither of these two works takes asynchronous compute kernels into consideration.

OpenMP's data mapping can be viewed as a form of software cache coherence. Inspired by cache coherence protocols, ARBALEST applies a state machine to track each mapped variable's status. Cache coherence has been well studied in the past decades [32]. Corresponding techniques are wildly used in dynamic analysis tools to detect and repair memory vulnerabilities. Given the popularity of heterogeneous architectures and parallel applications, recent research also focuses on flexible coherence interfaces for cache coherence protocols [33]. These coherence interfaces are applicable for a wide range of devices while the incurred overhead is low. ARBALEST may also leverage an available coherence interface to embed the state machine into the cache line state, making data mapping issue detection much more efficient.

[3]With certain OpenMP constructs/clauses, the serial elision property does not hold, e.g., using firstprivate variables in OpenMP tasks.

B. Data Consistency Issue in MPI applications.

OpenMP data mapping issues can be understood as data consistency issues that also arise in other parallel programming models. In [34], Hoefler et al. formalized the semantics of one-sided communication operations in MPI 3.0. Considering MPI defines two distinct memory models, the *unified memory model* and the *separate memory model*, in the specification, the authors reasoned about each memory model's effect on one-sided communication operations respectively. The unified memory model relies on hardware-managed data consistency, making MPI applications free of data consistency issues. On the other hand, the separate memory model enforces that the consistency of associated memory regions, which may reside on different nodes, is correctly managed by programmers. Therefore incorrect usage of MPI constructs in the separate memory model may result in data consistency issues.

Currently, ARBALEST cannot tackle MPI constructs, but the VSM based detection algorithm is still applicable to MPI applications. The algorithm can be integrated into a dynamic MPI anomaly detector (e.g., MUST [16]) to pinpoint data consistency issues in MPI applications.

VIII. Conclusion and Future Work

In this paper, we comprehensively analyze data mapping issues in OpenMP applications and propose ARBALEST, an on-the-fly data mapping issue detector. ARBALEST can precisely report data mapping issues in an OpenMP application while incurring constant time and space overhead to the program execution. For a given input, ARBALEST can find out data mapping issues in all possibles schedules. The evaluation on a set of open-source benchmarks shows that ARBALEST correctly handles all OpenMP constructs and incurs acceptable time and space overhead.

Furthermore, we identified several OMPT's shortcomings related to target offloading in the OpenMP specification. We worked together with the OMPT committee in refining OMPT's interface to expose all synchronization semantics necessary for our analysis.

For future research, we plan to combine some static analysis techniques with ARBALEST to improve the efficiency of dynamic data mapping issue detection. We also plan to extend ARBALEST further to support other accelerator programming models, such as OpenACC and Kokkos.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE). Part of this work was possible under funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA). This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United

States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] G. Balduzzi, A. Chatterjee *et al.*, "Accelerating DCA++ (dynamical cluster approximation) scientific application on the summit supercomputer," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 433–444.

[2] I. Nisa, J. Li *et al.*, "Load-balanced sparse mttkrp on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 123–133.

[3] H. Huang and E. Chow, "Accelerating quantum chemistry with vectorized and batched integrals," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 529–542.

[4] OpenMP Architecture Review Board, "OpenMP Application Programming Interface 5.1," https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, 2020, accessed: December 5, 2020.

[5] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[6] NVIDIA, "CUDA C Programming Guide," https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2020, accessed: December 5, 2020.

[7] AMD, "ROCm API References," https://rocm-documentation.readthedocs.io/en/latest/ROCm_API_References/ROCm-API-References.html, 2020, accessed: December 5, 2020.

[8] S. Memeti, L. Li *et al.*, "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM, 2017, pp. 1–6.

[9] N. Hemsoth, "Parallel Programming Approaches for Accelerated Systems Compared," https://www.nextplatform.com/2017/04/24/parallel-programming-approaches-accelerated-systems-compared/, 2017.

[10] A. Schmitz, J. Protze *et al.*, "DataRaceOnAccelerator–a micro-benchmark suite for evaluating correctness tools targeting accelerators," in *European Conference on Parallel Processing*. Springer, 2019, pp. 245–257.

[11] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 121–133.

[12] Y. Gu and J. Mellor-Crummey, "Dynamic data race detection for OpenMP programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 61.

[13] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 24–33.

[14] 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.

[15] S. Atzeni, G. Gopalakrishnan *et al.*, "ARCHER: effectively spotting data races in large OpenMP applications," in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 53–62.

[16] T. Hilbrich, J. Protze *et al.*, "MPI runtime error detection with MUST: advances in deadlock detection," *Scientific Programming*, vol. 21, no. 3-4, pp. 109–121, 2013.

[17] C. Voss, T. Cogumbreiro, and V. Sarkar, "Transitive joins: a sound and efficient online deadlock-avoidance policy." in *PPoPP*, 2019, pp. 378–390.

[18] A. Betts, N. Chong *et al.*, "GPUverify: a verifier for GPU kernels," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 113–132.

[19] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[20] K. Serebryany, D. Bruening *et al.*, "AddressSanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[21] 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.

[22] Google, "LLVM Sanitizer Infrastructure," https://github.com/google/sanitizers/wiki, 2015.

[23] J. Protze, J. Hahnfeld *et al.*, "OpenMP tools interface: Synchronization information for data race detection," in *International Workshop on OpenMP*. Springer, 2017, pp. 249–265.

[24] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*, 2007, pp. 65–74.

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

[26] P. Barua, J. Zhao, and V. Sarkar, "OmpMemOpt: Optimized Memory Movement for Heterogeneous Computing," in *European Conference on Parallel Processing*. Springer, 2020, pp. 200–216.

[27] L. Li and B. Chapman, "Compiler assisted hybrid implicit and explicit GPU memory management under unified address space," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–16.

[28] "SPEC ACCEL® Changes," https://www.spec.org/accel/Docs/changes-in-v1.3.html#bmark503.postencil, 2019.

[29] P. Barua, J. Shirako *et al.*, "OMPSan: Static Verification of OpenMP's Data Mapping constructs," in *International Workshop on OpenMP*. Springer, 2019.

[30] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 33–42.

[31] S. Lee, D. Li, and J. S. Vetter, "Interactive program debugging and optimization for directive-based, efficient gpu computing," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 481–490.

[32] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[33] J. Alsop, M. Sinclair, and S. Adve, "Spandex: a flexible interface for efficient heterogeneous coherence," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 261–274.

[34] T. Hoefler, J. Dinan *et al.*, "Remote memory access programming in MPI-3," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, p. 9, 2015.