Acknowledgement

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

Multi-level Debugging for Multi-stage, Parallelizing Compilers

Richard Xia, Tayfun Elmas, Shoaib Kamil, Armando Fox, and Koushik Sen
Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley
Email: {rxia, elmas, skamil, fox, ksen}@eecs.berkeley.edu

***Abstract*—A multi-stage compilation framework transforms portions of programs written in a productivity-level language into an efficiency-level language, such as C, with explicit hardware-specific optimizations. It is challenging for compiler programmers to debug errors in the compilation because they must perform complicated end-to-end reasoning, relating the programs across the multiple stages of compilation. To simplify this debugging effort, we present *multi-level debugging*, a novel combination of error-checking algorithms in a multi-stage compilation environment. Our method particularly aims to model and check sequential and parallel notions of nondeterminism and related bugs introduced by the compilation. Using our method, the programmer can systematically eliminate potential sources of the bug in the compilation process and focus only on the real source. We demonstrated on two real multi-stage compilers the effectiveness of multi-stage debugging in simplifying the diagnosis of manually-injected bugs as well as in an actual bug encountered during compiler development.**

I. Introduction

Domain-expert programmers using productivity-level languages (PLLs) desire scalable application performance, but usually rely on experts in efficiency-level languages (ELLs) and explicit optimizations such as parallel programming to achieve it. Recently multi-stage compilation frameworks (e.g., [7], [8]) have been proposed to maximize the reuse of efficiency programmers' work. Such frameworks transform domain-specific portions of programs written in a PLL (without explicit parallelism) into an ELL, such as C, C++, or Cuda, with explicit hardware-specific optimizations. At each stage of compilation either the program is optimized within the same language (e.g., loop unrolling, cache blocking), the program is translated to another language (e.g., from Python to C++), or some sequential statements (e.g., `for` loops) are replaced by their parallel versions, where the parallelism is exposed via frameworks such as OpenMP, OpenCL, or Cilk Plus. Moreover, during these stages, the compiler may introduce into the program different notions of nondeterminism: either by sequential constructs such as `for` loops with nondeterministically ordered iterations or by parallel constructs that run by nondeterministically scheduled threads. This sophisticated process may introduce various nondeterminism and synchronization related bugs into the optimized program.

If an execution of the program optimized by a multi-stage compilation produces an unexpected outcome, e.g., violates a test assertion or crashes, then debugging the error is a significant challenge for the compiler writer. For instance, the multi-stage compilation produces a sequence of intermediate

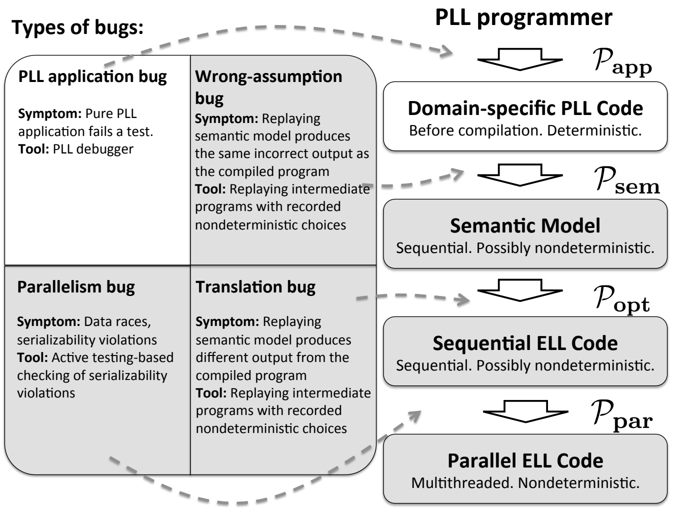
Types of bugs:
PLL programmer
PLL application bug
Symptom: Pure PLL application fails a test.
Tool: PLL debugger
Wrong-assumption bug
Symptom: Replaying semantic model produces the same incorrect output as the compiled program
Tool: Replaying intermediate programs with recorded nondeterministic choices
Parallelism bug
Symptom: Data races, serializability violations
Tool: Active testing-based checking of serializability violations
Translation bug
Symptom: Replaying semantic model produces different output from the compiled program
Tool: Replaying intermediate programs with recorded nondeterministic choices
$\mathcal{P}_{\textbf{app}}$
Domain-specific PLL Code
Before compilation. Deterministic.
$\mathcal{P}_{\textbf{sem}}$
Semantic Model
Sequential. Possibly nondeterministic.
$\mathcal{P}_{\textbf{opt}}$
Sequential ELL Code
Sequential. Possibly nondeterministic.
$\mathcal{P}_{\textbf{par}}$
Parallel ELL Code
Multithreaded. Nondeterministic.

Fig. 1. Overview of bugs that may occur in a multi-stage compilation and related compilation stages. Our work focuses on gray-colored bugs and stages.

programs with different levels of abstraction and in different languages. Debugging the problem across all these levels of abstraction becomes complicated because of excessive amount of code generated by the compiler or lack of enough semantic information tracked along with the program to relate the programs across the stages of the compilation.

In this work, we propose *multi-level debugging*, a novel methodology that simplifies the overwhelming process of debugging a multi-stage compiler. Our key observation is that, *instead of performing an end-to-end reasoning about the input and final output of the compilation, one can directly identify the specific type of the bug and its originating stage by incorporating additional runtime analyses over the intermediate versions of the program*. For this, we classify the bugs that may occur in a multi-stage compilation environment (shown in Figure 1) and combine a collection of runtime analysis techniques to detect them, each technique running at a different stage of the compilation and checking for a particular bug type.

Sequential and parallel notions of *nondeterminism* introduced by multi-stage compilation exacerbates the reasoning about the compiled program and imposes unique debugging challenges to the compiler writer. Therefore, our contributions primarily focus on understanding and checking the effects of such nondeterminism.

First, we identify and tackle three types of bugs:

1) *Nondeterminism-related bugs:*

a) *Wrong-assumption bugs* are introduced when the PLL programmer makes an incorrect assumption about the (sequential) nondeterministic semantics of the optimized computation. In this case, the PLL programmer expects a specific deterministic execution of a construct, e.g., a `for` loop implementing a left-to-right reduction of a list, but the compiler treats the construct as a nondeterministic one, e.g., to a nondeterministic-`for` loop producing unanticipated reduction orders.

b) *Parallelism bugs* are introduced when replacing sequential statements with their parallel versions in the ELL code. They are caused by missing or incorrect synchronization on shared variables, and often manifest themselves as parallelism-related errors such as data races and serializability violations.

2) *Translation bugs* are introduced when translating the PLL code to ELL code (e.g., from Python to C++). Such a bug can be due to a mistranslation of a single operator (e.g., using $-$ instead of $+$) or a high-level feature in the PLL (e.g., improper translation of a list comprehension in Python to a `while` loop in C++). In addition, we consider incorrect local optimizations in the ELL code as translation bugs.

We do not address the case of bugs introduced in the PLL application itself, since these bugs can be treated using the debugging tools already available in the PLL, such as the Python debugger. The bugs given above, however, cannot be found or detected using PLL debuggers. ELL debuggers can help to understand such bugs in the ELL code, but such debuggers do not provide any support to correlate the root cause of those bugs to the compilation stages.

Second, we present a formalism for modeling the various intermediate representations of a program in a multi-stage compilation (Section II). In order to reason about different notions of nondeterminism, we model sequential nondeterminism with two constructs, `nd_for` and `nd_if`, and parallel nondeterminism with `par_for` loop. In multi-level debugging, we expose these constructs to the multi-stage compiler and track an explicit relationship between them across stages (e.g., which `nd_for` loops are parallelized to which `par_for` loops) in order to enable interaction of our error-checking techniques running at different stages. Using our formalism, we clearly describe how each of the previously mentioned bugs can be identified with respect to these transformations.

Third, we combine, for the first time in a multi-stage compilation environment, a collection of runtime analysis techniques (Section III). These techniques add more thorough checking over the compilation process in order to help the compiler writer to distinguish nondeterminism-related bugs from other kinds of bugs. *Techniques for parallelism-error detection:* We exploit active testing [29] and serializability-violation detection algorithms [4] originally developed for single-stage debugging of programs with structured parallelism. Our algorithms make use of relationship between the nondeterministic sequential constructs in the intermediate version of the program and their parallelized forms in the final program in order to separate checking of parallelism errors from other kinds of errors. *Techniques for replaying programs with nondeterminism*: We propose an algorithm for checking translation and wrong-assumption bugs. The algorithm replays an intermediate program with nondeterministic sequential constructs using the resolution of this nondeterminism from a more optimized version of the program—i.e., the former program mimics the nondeterministic choices made in an execution of the latter program. This allows the compiler writer to correlate the executions of programs generated at different stages of the compilation. By comparing the result of the replayed execution with the original execution, the programmer can distinguish between (i) errors due to unanticipated introduction of nondeterminism by the earlier stages of the compilation and (ii) errors due to incorrect translation of other, deterministic constructs in the later stages.

Finally, we evaluated the usability of our multi-level debugging method by implementing our method in two different multi-stage compilers in a Selective Embedded Just-In-Time Specialization (SEJITS) framework [7] (Section IV) and detecting manually-injected bugs comprising the types described above. We first confirmed that these compilers are bug-free by applying them on a number of test programs and enabling our tools to check for errors during the testing. Then, we conducted two case studies in which we manually injected typical examples to nondeterminism-related and translation bugs, in order to evaluate the effectiveness of our method for identifying the correct type and cause of the inserted bug. For each bug, our tools successfully detected violations due to the bug and reported sufficient information to quickly find the actual cause of the bug. We present the case studies in Sections V and VI and discuss the results in Section VII.

II. Background: Multi-stage Compilation

In this section, we present a formalism for modeling the various intermediate representations of a program in a multi-stage compilation, common to embedded DSL frameworks. Then, in the next section, we explain on this model the techniques we use in our multi-level debugging method.

We denote a program under multi-stage compilation by $\mathcal{P}$ and distinguish the versions of a program at different stages of the compilation using subscripts, e.g., $\mathcal{P}_{\mathbf{opt}}$. We model each program as a function of type $T_{in} \rightarrow T_{out}$, i.e., it takes a single argument of type T_{in} and returns a single value of type T_{out}. We assume that the program is closed except for this input argument. Let $\mathcal{P}(v)$ denote the set of all possible values that can be produced by an execution of $\mathcal{P}$ given input value v. A program $\mathcal{P}$ is said to be *deterministic* if for each input v, every execution of $\mathcal{P}$ produces the same output value, i.e., $\mathcal{P}(v)$ is singleton.

A compiler translates a program $\mathcal{P}_{\mathbf{0}}$ in an iterative process: it generates different versions of the program $\mathcal{P}_{\mathbf{1}}, ..., \mathcal{P}_{\mathbf{n}}$ and finally executes the most optimized program $\mathcal{P}_{\mathbf{n}}$. We denote each compilation stage by $\mathcal{P}_{\mathbf{i}} \dashrightarrow \mathcal{P}_{\mathbf{i+1}}$, where $\mathcal{P}_{\mathbf{i+1}}$ is obtained from $\mathcal{P}_{\mathbf{i}}$ by applying a domain-specific transformation or a performance optimization.

```
1 class SimpleKernel(StencilKernel):
2   def kernel(self, in_grid, out_grid):
3     for p in out_grid.interior_points():
4       for q in in_grid.neighbors(p, 1):
5         out_grid[p] = out_grid[p] + in_grid[q]
```

Fig. 2. A simple stencil kernel in Python.

```
1 class SimpleKernel(StencilKernel):
2   def kernel(self, in_grid, out_grid):
3     nd_for x in out_grid.interior_x_indices():
4       nd_for y in out_grid.interior_y_indices():
5         nd_for nx in [x-1, x, x+1]:
6           nd_for ny in [y-1, y, y+1]:
7             out_grid[x,y] = out_grid[x,y] + in_grid[nx, ny]
```

Fig. 3. A simplified semantic model for `SimpleKernel`.

```
1  void kernel_unroll_4(PyObject *in_grid, PyObject *out_grid) {
2    .....
3    int z;
4    #pragma omp parallel for
5    for (int x = 1; (x <= 8); x = (x + 1)) {
6      for (int y = 1; (y <= (8 - 3)); y = (y + (1 * 4))) {
7        z = _to_array_index(x, y); // an inner point
8        // combine neighbor values
9        _my_out_grid[z] = _my_out_grid[z] -
10                     _my_in_grid[_to_array_index(x, y-1)];
11       .....
12        z = _to_array_index(x, y+1); // next inner point
13       _my_out_grid[z] = ..... // combine neighbor values
14       .....
15        z = _to_array_index(x, y+2); // next inner point
16       _my_out_grid[z] = ..... // combine neighbor values
17       .....
18       z = _to_array_index(x, y+3); // next inner point
19       _my_out_grid[z] = ..... // combine neighbor values
20       .....
21 } } }
```

Fig. 4. C++ code after unrolling and parallelizing loops in the semantic model. Shaded lines are used in Section V to explain the bugs in the compiler.

To simplify our presentation and comply with the following sections, in which we demonstrate the use of our method in SEJITS, we consider a representative, three-stage compilation in SEJITS: $\mathcal{P}_{\mathbf{app}} \dashrightarrow \mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}} \dashrightarrow \mathcal{P}_{\mathbf{par}}$

Each stage of the compilation exhibits a different common transformation pattern we observed in our experiments with real SEJITS compilers. However, a compiler may contain more stages than given here (by applying each pattern multiple times). We next present the programs obtained during this representative compilation (marked in Figure 1). To be concrete, we also overview an example compilation of a stencil kernel.

A. Application-level program

$\mathcal{P}_{\mathbf{app}}$ represents the application-level program written by a user of SEJITS. $\mathcal{P}_{\mathbf{app}}$ uses a restricted, domain-specific subset of the language that describes the high-level computation without expressing how this computation is carried out in detail. We assume that $\mathcal{P}_{\mathbf{app}}$ is sequential and deterministic; that is, it does not contain nondeterministic or parallel constructs.

Example. Figure 2 shows an example to $\mathcal{P}_{\mathbf{app}}$, Python code containing a stencil kernel called `SimpleKernel`. `SimpleKernel` takes a two-dimensional input grid `in_grid`. For each point `p` in the grid, it applies a function (in our case *sum*) of that point and its neighborhood (over which `q` ranges), and writes the result of that function to the corresponding point in an output grid (`out_grid`). Notice that the interior points and the neighborhood of a point (and their traversals) are specified in a high-level, abstract way. ■

B. Semantic model

In program $\mathcal{P}_{\mathbf{sem}}$, the high-level computation in $\mathcal{P}_{\mathbf{app}}$ is mapped to more detailed constructs for carrying out the computation. The stage $\mathcal{P}_{\mathbf{app}} \dashrightarrow \mathcal{P}_{\mathbf{sem}}$ performs a global, domain-specific transformation in the structure of the program.

Example. Figure 3 shows a simplified Python-like code for the semantic model of our stencil kernel in Figure 2 (for `nd_for` loops see below). In essence, this model is an intermediate representation that maps the `kernel` function to another function describing in more detail how the stencil computation defined at a high level in $\mathcal{P}_{\mathbf{app}}$ will be carried out. In particular, the semantic model expands the loop over abstract interior points (at line 3 of Figure 2) to two loops over the `x` and `y` dimensions of the input grid (lines 3-4 of Figure 3). Moreover, the loop over neighbors of an interior point (at line 4 of Figure 2) is expanded to two loops (over `nx` and `ny`) traversing the specific indices in the neighborhood of an interior point. ■

In the semantic model $\mathcal{P}_{\mathbf{sem}}$ some computations are described using nondeterministic constructs in order to enable further optimizations. This nondeterminism is resolved by the latter, more refined versions of the program in a way that yields the best performance. For example, nondeterminism allows opportunity for parallelization, because each nondeterministic task can be executed by a separate thread. To specify the nondeterminism, our methodology exposes to the SEJITS compiler two nondeterministic constructs:

- `nd_for`: A nondeterministic-for (`nd_for`) loop allows its iterations to run in a different order than the default, deterministic order dictated by regular `for` loops.
- `nd_if`: A nondeterministic conditional `nd_if` does not evaluate a boolean expression but choses which branch to execute nondeterministically, useful for modeling boolean expressions which depend on state shared between multiple threads.

For all these statements, the nondeterminism is resolved independently at each runtime invocation of the statement. We assume that, under a mechanism that resolves all the nondeterministic choices in the execution, the semantic model $\mathcal{P}_{\mathbf{sem}}$ is executable. Section III-B presents such a mechanism.

As explained in Section III-A, program $\mathcal{P}_{\mathbf{sem}}$ with constructs `nd_for` and `nd_if` becomes a sequential (specification) artifact for checking the correctness of the succeeding, parallelized versions of the program. Such sequential programs with explicit nondeterminism allows our algorithms to check independently (i) the correctness of the modifications on the sequential aspects of the program by former stages of the compilation and (ii) the correctness of the parallelism introduced at later stages of the compilation.

Example. In Figure 3 the semantic model marks all the loops (at lines 3-6) as `nd_for` loops. Thus, $\mathcal{P}_{\mathbf{sem}}$ leaves the order in which internal points and their neighbor points are traversed nondeterministic. This nondeterminism allows later stages to perform optimizations on these loops, e.g., unrolling or parallelizing, that alter the default ordering of iterations and would otherwise be invalid for a deterministic `for` loop. ■

C. Optimized program

While stage $\mathcal{P}_{\mathbf{app}} \dashrightarrow \mathcal{P}_{\mathbf{sem}}$ performs domain-specific, global transformations and optimizations, stage $\mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}}$ performs domain-independent, local optimizations, similar to compiler optimizations or JIT compilations. This includes rewriting program $\mathcal{P}_{\mathbf{sem}}$ in an efficiency language and further optimizations within the statements, e.g., rewriting numerical expressions on arrays to vector instructions.

Example. For our stencil example, the stage $\mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}}$ translates the semantic model in Python (Figure 3) to sequential C++ code shown in Figure 4 *without the OpenMP pragma at line 4*. Program $\mathcal{P}_{\mathbf{opt}}$ resolves the nondeterminism in $\mathcal{P}_{\mathbf{sem}}$, i.e., the order of `nd_for` iterations, as follows. The compiler unrolls (i) every consecutive four iterations of the loop at line 4 traversing the *y* dimension of the grid and (ii) the loops at lines 5-6 traversing the neighborhood of the current point. This optimization is allowed because the `nd_for` loops specify that any ordering is valid. The fixed order in which these loops are unrolled and the remaining, deterministic part of the loop over `y` at line 6 resolve the nondeterminism in the `nd_for` loops at lines 4-6 of the semantic model. ■

D. Parallelized program

$\mathcal{P}_{\mathbf{par}}$ represents the most optimized program generated by the compiler. It adds to $\mathcal{P}_{\mathbf{opt}}$ parallelism by translating some sequential statements to parallel constructs to be executed by multiple threads. We assume that $\mathcal{P}_{\mathbf{par}}$ introduces parallelism in a structured way as follows: A `for` loop (regular or `nd_for`) is transformed to a parallel-for (`par_for`) statement. At runtime, some or all iterations of the loop may be executed concurrently by multiple threads.

Running some statements in parallel introduces a second source of nondeterminism, since threads are interleaved under a nondeterministic scheduler. In fact, parallelizing an `nd_for` loop in $\mathcal{P}_{\mathbf{opt}}$ to a `par_for` loop is a way to resolve the nondeterminism in programs; that is the runtime scheduling of the threads determines the order in which the iterations of the `nd_for` loop are executed. In Section III, we use this runtime order to replay `nd_for` loops sequentially.

Example. The final program $\mathcal{P}_{\mathbf{par}}$ for our stencil example is obtained by parallelizing the loop over `x` at line 5 in Figure 4. For this, the compiler appends the pragma `omp parallel for` right before the loop. (In Section III, we will use `par_for` to represent such parallel loops.) In the parallel version, the iterations of the loop are run by multiple threads concurrently. In this case, the runtime schedule of threads executing loop iterations gives the same effect of nondeterministically reordering the iterations in the corresponding `nd_for` loop (over dimension `x` at line 3) in the semantic model. ■

In summary, as a result of the SEJITS compilation, the three-line body of `kernel` code in Figure 2 becomes a long, complicated C++ function (consisting of 30 LOC) in Figure 4, and now includes parallelism. Moreover, while not modeled here, the compiler can generate multiple variants of C++ code as a result of applying different sets of optimizations for tuning for the most efficient variant. Debugging a problem under this multi-stage compilation is a challenge, and our multi-level debugging method aims to tackle this challenge.

III. Multi-level Debugging of Compilers

Having introduced different versions of the program during the multi-stage compilation, the correctness condition of the compilation is as follows:

C: *Assume that $\mathcal{P}_{\mathbf{app}}$ is correct, i.e., it passes all tests. Then, for each input value I, $\mathcal{P}_{\mathbf{par}}(I) = \mathcal{P}_{\mathbf{app}}(I)$.*

Recall that, $\mathcal{P}_{\mathbf{app}}$ is assumed to be deterministic. Thus, the condition **C** implies that the parallel program is also deterministic: for each input I, it always produces the same output independent of the scheduling of threads. This condition ensures that the compiled program ($\mathcal{P}_{\mathbf{par}}$) cannot exhibit a behavior (i.e., an input-output relation) that is not allowed by the original program ($\mathcal{P}_{\mathbf{app}}$). Since we assume that $\mathcal{P}_{\mathbf{app}}$ passes all tests, $\mathcal{P}_{\mathbf{par}}$ failing a test indicates that $\mathcal{P}_{\mathbf{par}}$ contains an extra, unintended behavior introduced by one of the compilation stages. Identifying the real cause of this unintended behavior by considering all these stages together is a challenge. In this paper, we reduce the testing of condition **C** by applying a set of checking algorithms at different stages of the compilation. Section IV explains how these algorithms are applied in practice within the SEJITS framework.

Each algorithm in our method considers a particular stage, $\mathcal{P}_{\mathbf{i}} \dashrightarrow \mathcal{P}_{\mathbf{i+1}}$. First, we treat $\mathcal{P}_{\mathbf{i}}$ as a specification, i.e., assume that $\mathcal{P}_{\mathbf{i}}$ is correct. Then, we adapt **C** to that particular stage:

C': *Assume that $\mathcal{P}_{\mathbf{i}}$ is correct, i.e., it passes all tests. Then, for each input value I, $\mathcal{P}_{\mathbf{i+1}}(I) = \mathcal{P}_{\mathbf{i}}(I)$.*

To test the condition **C'**, the algorithm performs one or more specific checks over programs $\mathcal{P}_{\mathbf{i}}$ and $\mathcal{P}_{\mathbf{i+1}}$. If the compiled program fails a test and the algorithm (running at stage $\mathcal{P}_{\mathbf{i}} \dashrightarrow \mathcal{P}_{\mathbf{i+1}}$) detects a violation of a condition it checks, then the compiler writer can directly focus on the code that transforms $\mathcal{P}_{\mathbf{i}}$ to $\mathcal{P}_{\mathbf{i+1}}$ to debug the problem. The algorithm also provides diagnostic information so that the cause of the bug can be localized within the suspicious stage.

A. Checking for parallelism errors

Our first tool runs at the final translation stage $\mathcal{P}_{\mathbf{opt}} \dashrightarrow \mathcal{P}_{\mathbf{par}}$ and helps to debug parallelism bugs. In previous work [4] we showed that under the parallelization scheme in which a `nd_for` loop is transformed to a `par_for` loop, condition **C'** can be tested by checking serializability violations [24] within bodies of these parallel statements. During this, we use $\mathcal{P}_{\mathbf{opt}}$ as a *nondeterministic sequential (NDSeq) specification* of $\mathcal{P}_{\mathbf{par}}$. That is, we assume that, for a fixed input I, output $\mathcal{P}_{\mathbf{opt}}(I)$ is correct and deterministic (independent of how the sequential nondeterminism in $\mathcal{P}_{\mathbf{opt}}$ is resolved). Thus, the interleavings of threads in $\mathcal{P}_{\mathbf{par}}$ should not introduce any extra output for input I, which is showed in [4] to hold when all the parallel statements are serializable.

To check for the serializability violations, our tool combines active testing-based atomicity checking [25] and our notion of improved serializability checking in [4]. For this, we define a *transaction* as each execution of a loop body in a `par_for` statement. During an execution of $\mathcal{P}_{\mathbf{par}}$ transactions run by different threads may interleave with each other.

Given an execution E of $\mathcal{P}_{\mathbf{par}}$, our tool works on E in two phases. In the first phase, it finds data races in E. A data race occurs when two different threads access a common variable and at least one of these accesses is a write and there is no synchronization between the two accesses. Our tool uses these data races for two purposes: First, they allow us to identify the variables shared across different threads (i.e., transactions). Let $ShrVars$ be the set of such shared variables. Second, we identify the code locations at which a variable from $ShrVar$ is accessed as potential buggy points. We automatically add to these points perturbations in the thread schedule. This makes a parallelism bug more likely to occur.

In the second phase, we apply the algorithm in [4] to check if execution E is serializable. Execution E is said to be *serializable* if there exists a *single-threaded* execution E' of $\mathcal{P}_{\mathbf{par}}$ (i) with the same input as E and (ii) consisting of the same operations as E (with the same side effects, but possibly in a different order of execution). If so, both executions E and E' produce the same output for the given input.

To show that E is serializable, our algorithm tracks the accesses to variables in $ShrVars$ by different transactions. It records all the accesses in a graph structure where each vertex is a separate transaction. We add an edge to this graph from a transaction $t1$ to $t2$ (by different threads) whenever there is a data race between two accesses by $t1$ and $t2$ and the access by $t1$ is performed earlier than $t2$ in the execution. A cycle in this graph signals a parallelization problem causing a serializability violation involving the transactions in the cycle. The programmer can analyze these transactions and their accesses involved in the cycle to diagnose the problem. In general, a parallelization problem is caused by (1) incorrectly sharing a variable which is supposed to be local, or (2) insufficient synchronization of accesses to shared variables. The parallelization errors presented in Section V and Section VI are examples to cases (1) and (2), respectively.

Absence of cycles (i.e., serializability violations) in E implies that E is serializable. For real and complex programs, the standard serializability checking algorithm [24] may report a large number of false alarms, i.e., the checks by the algorithm fails, even though the transactions are serializable and thus there is no parallelism error. Our algorithm in [4] improves the precision of the standard algorithm by relaxing condition (ii) above. For this, we use nondeterministic `nd_if` statements in the program $\mathcal{P}_{\mathbf{opt}}$ to rule out benign data races between threads that does not affect the output of the computation, and thus, can be safely ignored by the analysis. This allow us to rule out cycles in the transaction graph that do not correspond to real serializability violations and report fewer false alarms.

B. Checking for translation and wrong-assumption bugs

Suppose that an execution E of $\mathcal{P}_{\mathbf{par}}$ fails a test, but the algorithm described above for parallelism errors gives no warning for E. Thus, the corresponding single-threaded execution of $\mathcal{P}_{\mathbf{opt}}$ also fails the test, and the bug should be introduced before the last (parallelizing) compilation stage: either $\mathcal{P}_{\mathbf{app}} \dashrightarrow \mathcal{P}_{\mathbf{sem}}$ or $\mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}}$.

To identify the stage introducing the bug, our second algorithm replays execution E of $\mathcal{P}_{\mathbf{par}}$ by program $\mathcal{P}_{\mathbf{sem}}$. Let E' be the replayed execution of $\mathcal{P}_{\mathbf{sem}}$. Note that, all nondeterministic constructs in $\mathcal{P}_{\mathbf{sem}}$ (`nd_if` and `nd_for`) are either replaced with equivalent deterministic operations in $\mathcal{P}_{\mathbf{opt}}$ or translated to parallel constructs in $\mathcal{P}_{\mathbf{par}}$. While generating the replayed execution E' of $\mathcal{P}_{\mathbf{sem}}$, we use the resolution of the nondeterminism in execution E. For this, we require that for each nondeterministic (`nd_for` and `nd_if`) statement in $\mathcal{P}_{\mathbf{sem}}$, there exists a corresponding loop or conditional statement in $\mathcal{P}_{\mathbf{par}}$, so that we can execute the former statements in E' by replaying the same order of iterations and branches in the latter. Since the stage $\mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}}$ makes only local transformations and the stage $\mathcal{P}_{\mathbf{opt}} \dashrightarrow \mathcal{P}_{\mathbf{par}}$ makes well-formed parallelizations of loops and straight code, this correspondence is straightforward.

- When an `nd_for` statement is invoked in E', we use the order of iterations in the corresponding instance of the loop in execution E. For example, if the loop is parallelized in $\mathcal{P}_{\mathbf{par}}$, then the order of transactions in E gives the order in which the iterations are executed during the replay. This order is obtained by our parallelization checker by computing a topological sorting of the (directed) transaction graph.
- When an `nd_if` statement is invoked in E', we execute the same branch in the corresponding instance of the statement in E.

Next, we make a case split by comparing the outputs produced by E and E':

- If the output values are equal, then we conclude that the translation $\mathcal{P}_{\mathbf{app}} \dashrightarrow \mathcal{P}_{\mathbf{sem}}$ is problematic, because $\mathcal{P}_{\mathbf{sem}}$ can produce the same (incorrect) output produced by $\mathcal{P}_{\mathbf{opt}}$ in E. Then, the compiler writer needs to search for the cause of this output by focusing on the domain-specific transformations, especially how the nondeterminism is introduced, when generating $\mathcal{P}_{\mathbf{sem}}$.
- If the output values are different (and the output of $\mathcal{P}_{\mathbf{sem}}$ is correct), then we conclude that the translation $\mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}}$ is problematic, because even though the executions resolve the nondeterminism in the same way, they compute different results. Then, the compiler writer can rule out problems due to nondeterminism and focus on the translation of the deterministic statements.

IV. Implementation in SEJITS

We have implemented our multi-level debugging method as a plug-in[1] to the Asp SEJITS framework [7]. The SEJITS approach enables building small, domain-specific compilers embedded in high-level languages; the Python framework that implements this approach is BSD-licensed open source and is called Asp[2]. Users write their programs in these embedded DSLs and, invisibly, the compilers translate user-supplied code using phased compilation as described in Section II, compile the resulting code, and execute it, returning values to the Python interpreter. The framework provides mechanisms to simplify development, including a unified tree transformation framework and infrastructure to define typed intermediate forms, as well as common code transformations used in compilers (e.g. loop unrolling and blocking) to be used in optimization passes. A number of embedded DSLs across a variety of domains, targeting multiple backends, have been developed using the framework, which is under active development.

Our checking tools are enabled/disabled by setting an environment flag `MULTI_LEVEL_DEBUG` before running the compiler on the input program. Once enabled, each tool is triggered at the relevant stage of the compilation and instruments the intermediate program using an API provided by Asp. During the execution of the compiled program, the instrumentation records information about various events (e.g., memory accesses, function calls) required by the checking algorithm. While the parallelism checker runs online (i.e., checks errors while the program is still executing) the replay mechanism runs offline (i.e., checks errors after the compiled program terminates). The errors detected are reported to the user at the end of the execution and relevant detailed error descriptions are written to files to be used during the debugging. We next give tool-specific details.

A. *Implementing the parallelism checker*

We implemented our parallelism checker as an extra transformation stage within the SEJITS compilers after the parallelized program ($\mathcal{P}_{\mathbf{par}}$) is generated. In the SEJITS compilers in our study, the parallelism is introduced by using OpenMP pragmas `omp parallel` and `omp parallel for` before statements to be parallelized. Thus, our tool instruments the statements annotated by these pragmas to check for parallelism errors. For other parallelism paradigms (such as Pthreads or Cilk Plus), similar annotations would be introduced during parallelization. The instrumentation consists of (1) marking the beginning and end of each parallel statement, (2) marking the beginning and end of each transaction, a statement executed by a separate thread, and (3) reads and writes from every possibly-shared variable. The granularity of instrumentation for (3) can be adjusted as needed: while the tool can track every individual memory-access instruction, it can also be configured to treat data structures (e.g., a list of doubles) as shared variables and track coarse-grained operations on these structures (e.g., add/lookup/remove operations of the list) as variable accesses. We follow the former approach in the stencil compiler and the latter in the TinyCU compiler described later.

[1]The version of the framework containing the multi-level debugging tools is available at https://github.com/richardxia/asp-multilevel-debug.

[2]Asp is SEJITS for Python, available at http://www.sejits.com.

During the execution of the compiled program, our tool analyzes each invocation of a parallel statement separately, since these invocations are separated from each other using implicit barriers at the end of the statement. As discussed in Section III-A, our tool works in two phases. For every execution of a parallel statement, our tool first records the variables on which at least one data race is detected. In this way, our tool identifies the variables to be tracked for the second phase of the checking (for serializability violations). In the second phase, the tool collects the accesses performed by transactions and constructs the transaction graph. When the currently-running parallel statement ends, the tool checks for cycles in the transaction graph and writes any cycles to a report file in a human-readable form. The report lists the shared variables and the source code locations accessing these variables. For each serializability violation, the report contains the exact thread schedule and the statements involved in the violation. If there is no parallelism error is detected, the tool produces a trace file containing the accesses to certain variables in the parallel loops, e.g., loop-index variables. This trace is used to compute the (total) order of iterations in parallel loops, which is used by the replay mechanism to guide the execution of `nd_for` loops as discussed below.

B. *Implementing the replay mechanism*

The replay mechanism checks for translation errors by resolving nondeterminism in the semantic model ($\mathcal{P}_{\mathbf{sem}}$) using execution traces of the compiled program ($\mathcal{P}_{\mathbf{par}}$). In the case that no parallelization optimizations are performed, the $\mathcal{P}_{\mathbf{opt}}$ is equivalent to $\mathcal{P}_{\mathbf{par}}$. The replay mechanism executes $\mathcal{P}_{\mathbf{sem}}$ using a recorded execution trace of the compiled program containing information about how the nondeterministic statements in $\mathcal{P}_{\mathbf{sem}}$ are resolved, for example, loop-iteration ordering. Our parallelism checker, which instruments $\mathcal{P}_{\mathbf{par}}$, generates this execution trace as described in Section IV-A. Note that we assume that $\mathcal{P}_{\mathbf{sem}}$ is executable once all the nondeterministic choices in the execution is given.

In order to replay $\mathcal{P}_{\mathbf{sem}}$ with information from $\mathcal{P}_{\mathbf{par}}$, we first require the compiler writer to annotate certain statements to explicitly correlate them to traced statements. The compiler writer annotates nondeterministic statements within the $\mathcal{P}_{\mathbf{sem}}$ (e.g., `nd_for`), indicating to our replay tool that a reordering is valid and that it should determine the reordering from the trace. In addition, assignment statements may optionally be annotated in order to indicate to the tool that it should check whether values match between traced and the replayed executions, allowing for finer granularity of debugging output.

For annotated loops, the tool replaces the default Python iterators with an iterator that returns loop iterations in the same order as the traced execution. Annotated assignment statements are replaced with statements that both perform the assignment and compare the replayed value with the traced value. While replaying, if any intermediate assignment values

```
1  class SimpleTest(unittest.TestCase):
2    def test1(self):
3      kernel = TestKernel()
4      in_grid = StencilGrid([10,10])
5      out_grid = StencilGrid([10,10])
6      in_grid[2,3] = 1
7      in_grid[4,3] = 2
8      in_grid[3,2] = 3
9      in_grid[3,4] = 4
10     # COMPILE AND RUN KERNEL
11     kernel.kernel(in_grid, out_grid)
12     # CHECK SOME CONDITION ON OUTPUT
13     self.assertEqual(out_grid[3,3], 10)
```

Fig. 5. A simple stencil kernel and a unit test for the kernel. When called at line 11, the code from Fig 2 is compiled to parallel C++ code and run.

do not match traced values, then the tool will report an error and indicate which intermediate values mismatch. In addition, if the traced loop iterations are inconsistent with the iterations available in the replayed execution (e.g. number of iterations do not match), the tool will also report an error.

If the replayed execution differs from the traced execution in any way or if the final result differs, then the tool reports a *translation bug* and provides information about the mismatch. Otherwise, if the replayed execution exactly matches traced execution and returns the same value as the $\mathcal{P}_{\mathbf{par}}$, the tool reports the error as a *wrong-assumption bug* and warns the programmer about a possible problem about the nondeterminism in the semantic model.

V. Case Study: Debugging of a Stencil Compiler

Figure 5 shows a class named `StencilTest` to unit test the stencil kernel shown in Figure 2. To test this kernel, we fix an input grid (lines 6-9), invoke the kernel on this input grid (line 11), and check using an assertion whether the kernel has computed the correct output (line 13). Note that, when the kernel is called at line 11, the stencil compiler framework first translates the kernel to a semantic model in Python (shown in Figure 3) and then translates the kernel to highly-efficient, parallelized C++ code (shown in Figure 4). Then, the resulting C++ function is called, and the return value of the function is mapped back to a Python-level object before the kernel returns.

Now suppose that the assertion at line 13 fails when the kernel is compiled by SEJITS. Given the complexity of the multi-stage translation from Python to C++ code outlined in Section II, it is a tedious job to debug the problem. We next explain how we apply our multi-level debugging tools to simplify this process, in particular to identify the type of a bug and focus on the relevant stage of the stencil compiler. We present the debugging process, in the *bottom-up* fashion similarly to Section III, where the programmer starts with focusing on errors introduced at the latter stages of the compilation. Ruling out errors in this way allows us to use information and guarantees from the algorithms in the latter stages while checking errors at the former stages.

A. Parallelization bug

Consider the C++ fragment in Figure 4 after parallelizing a loop in the C++ code. In this code, the two-dimensional (input and output) grids in the Python code are mapped to a one-dimensional C++ array. At each iteration of the inner loop, the variables `x` and `y` are used to compute indices into the input array (to read from) and an index `z` into the output array (to write to). Line 4 is added to the sequential C++ code to add parallelism to the kernel. The OpenMP directive `#pragma omp parallel for` indicates that the `for` loop after the pragma may be executed by multiple threads, each running a separate iteration of the loop.

Adding the directive at line 4 results in an *incorrect* parallelization as described next. Before adding the parallelization directive, the variable `z` is intended to be a local variable: every loop iteration computes a different value for `z` and accesses a distinct location in the output grid. However, adding the directive *after* the definition of `z` makes it shared by the OpenMP threads executing the loop iterations. This sharing of `z` does not cause a problem *if* the OpenMP threads executing iterations are not interleaved with each other. Otherwise, two different threads may access the same location in the output grid, causing one of the threads to overwrite the value written by the other and produce an incorrect grid at the end.

Unfortunately, for two reasons, it is very likely that the compiler writer deploys her code without even noticing that such a bug exists. First, such erroneous interleavings are very sensitive to the timings of threads and occur once in thousands of executions. Thus, one may conclude that the parallelization is correct after testing many times without observing an error. Second, since the only change between the sequential and parallel variants of the C++ code is a single-line `#pragma` directive, the effect of parallelizing the loop on `z` is not obvious from the static code (i.e., there is no sign of this sharing). Thus, during code review one could easily miss this detail. Therefore, such nondeterministic bugs due to thread interleavings are insidious, and tools for detecting them are essential.

For our running example, our parallelism checker (explained in Section III-A) detects data races on the points of output grid and reports that the iterations of the loop are not serializable. That is, in between a thread $t1$ reading from and updating a point in the output grid, another thread $t2$ accesses (reads and/or writes) the same point. This indicates that points in the output grid are *unexpectedly* shared among threads. Since we were not expecting this sharing, the accesses to the grid are not properly synchronized, resulting in the parallelism errors detected by the tool. Further inspection of our tool's report reveals why points of the output grid have become shared: Our tool also detects data races on (shared) variable `z`, which was intended to be local, but is now shared by threads.

We correct the compiler by moving the definition of variable `z` inside the parallelized loop body (right after line 5). This change makes the variable `z` thread-local, and after that our tool does not report any parallelism-related errors.

B. Translation bug

The next type of bug is introduced when translating the Python semantic model to (sequential) C++ code. Now suppose that our tool for checking parallelization errors introduced above is enabled and the tool does not give any warnings. In this case, we can safely rule out parallelization-related problems and search for the cause of the bug in the former,

```
1 class SimpleKernel(StencilKernel):
2   def kernel(self, in_grid, out_grid):
3     for x in out_grid.interior_points():
4       for y in in_grid.neighbors(x, 0):
5         out_grid[x] = (2 * out_grid[x]) + in_grid[y]

6 class SimpleTest(unittest.TestCase):
7   def test1(self):
8     kernel = TestKernel()
9     in_grid = StencilGrid([10,10])
10    in_grid.neighbor_definition=[[(-1,0),(1,0),(0,-1),(0,1)]]
11    out_grid = StencilGrid([10,10])
12    in_grid[2,3] = 1
13    in_grid[4,3] = 2
14    in_grid[3,2] = 3
15    in_grid[3,4] = 4
16    # COMPILE AND RUN KERNEL
17    kernel.kernel(in_grid, out_grid)
18    # CHECK SOME CONDITION ON OUTPUT
19    self.assertEqual(out_grid[3,3], 26)
```

Fig. 6. Another kernel using the same stencil compiler in SEJITS. In this case, the function at line 5 is not associative, thus the nondeterminism introduced by the transformations causes the compiled program to fail the test.

sequential versions of the program. To improve this search, we use the replay mechanism explained in Section III-B to debug the problem. The mechanism helps to distinguish translation bugs from wrong-assumption bugs by checking whether a bug is caused by the nondeterminism introduced by the compiler or by the incorrect translation of statements.

Our replay tool tracks the execution of the parallel C++ version of the program and collects information how the nondeterminism in the semantic model is resolved in the parallel program. For each parallel loop (at line 5 of Figure 4), the tool computes the total order of threads running the loop, giving the order of iterations for the corresponding loop marked nondeterministic in the semantic model. Then, it replays the semantic model (in Python) in Figure 3, but resolving the nondeterminism in the `nd_for` loops in the same way as the parallel execution. That is, iterations of a `nd_for` loop are executed by following a serial ordering that is conflict-equivalent to the parallel execution. For this, we compute the order of `nd_for` iterations by using the order of accesses to the loop-index variables in the parallel execution and mapping this information back to the semantic model. Finally, the tool compares the output (`out_grid`) produced by the semantic model with that of the parallel C++ program. If the outputs match, then this means the body of Python loop is translated to C++ correctly. Otherwise, a mismatch indicates that *even after following the same order of iterations in both executions*, the C++ code produces a different output. Therefore, we understand that the bug is not related to the nondeterminism introduced by the compiler but the translation of each loop iteration. In our example, the shaded line 8 of Figure 4 contains the bug: incorrectly replacing $+$ with $-$.

C. Wrong-assumption bug

Wrong-assumption bugs occur because a user of the compiler makes incorrect assumptions about the nondeterministic semantics introduced by the compiler. In this case, the original Python code runs with by-default-deterministic statements and passes a test case, but the compiled code fails the test, as it contains nondeterministic computations.

To demonstrate such bugs, we modify the kernel code in Figure 2 to the one in Figure 6. At line 5, we replace the (associative) + operation with a non-associative one. Since the new function is non-associative, we also specify at line 10 our own neighbor definition, i.e., an explicit order to traverse the neighborhood of each point. We expect the loop enumerating the neighbors of the current point to follow this order; visiting the neighbors in a different order will result in a different result and make the test fail.

However, the C++ code in Figure 4 does not follow this assumption by unrolling the loop over the neighbors in a different order. (Since we change the kernel function, the C++ at lines 7-20 may differ.) Recall that the semantic model in Figure 3 expands the abstract loop in Figure 2 for traversing the neighborhood of an interior point to two nondeterministic-`for` (`nd_for`) loops. This allows the stage $\mathcal{P}_{\mathbf{sem}} \dashrightarrow \mathcal{P}_{\mathbf{opt}}$ to unroll these loops in the C++ version in Figure 4, but following a different order of iterations than the one we expected (line 10 Figure 6). As a result, the C++ code produces an incorrect output and violates the assertion.

Our replaying mechanism described above helps us to identify the cause of the bug as nondeterminism introduced by the semantic model. In particular, the execution of $\mathcal{P}_{\mathbf{par}}$ and the replayed execution of the $\mathcal{P}_{\mathbf{sem}}$ (guided by the ordering of iterations from $\mathcal{P}_{\mathbf{par}}$) produce the same output, indicating that the translation from Python model to C++ is correct. This result, together with the absence of parallelism errors, implies that the problem is at an earlier stage: $\mathcal{P}_{\mathbf{app}} \dashrightarrow \mathcal{P}_{\mathbf{sem}}$.

Next, we compare the replayed execution of $\mathcal{P}_{\mathbf{sem}}$ with an expected execution of the pure Python program. Our tools helps in this phase by showing the recorded order of the loops and the intermediate results of the loops used during the replay. Then, we easily notice that the loop over the neighbors in $\mathcal{P}_{\mathbf{sem}}$ execution is different from the order specified at line 10 of Figure 6. To fix the bug, we must either rewrite the program without relying on a specific ordering of the neighborhood iteration or we must extend the semantics of the compiler to allow the PLL programmer to give an explicit order to traverse the neighborhood.

VI. Case Study: Debugging of a TinyCU Compiler

We now consider a compiler based on the Copperhead [6] framework, which maps a data-parallel subset of Python to parallel platforms including CUDA and OpenMP. In Copperhead, the programmer writes Python code using predefined procedures such as `map`, `reduce`, and `scatter`. The program is then dynamically compiled to an underlying parallel platform using the platform's own efficient data-parallel constructs. We implemented a subset of the Copperhead operations as a SEJITS compiler called TinyCU.

A. TinyCU compiler

TinyCU implements the `map`, `sum`, and `reduce` functions on simple lists and emits parallel OpenMP code. TinyCU transforms an input program ($\mathcal{P}_{\mathbf{app}}$) into a semantic model ($\mathcal{P}_{\mathbf{sem}}$) consisting of the previous three functions, arithmetic operations, and function calls. Fig 7 shows an example

TinyCU program. During the parallelization stage, TinyCU transforms calls to `map` into data-parallel loops annotated with a pragma `omp parallel for`, as shown in Fig 8. Similarly, for `reduce` TinyCU emits a parallel tree reduction which operates on each pair of elements in parallel and recursively reduces the results into a single value. This requires the reduction function to be *associative*.

```
class ReduceKernel(TinyCU):
  def doubler(self, x):
    return 2 * x
  def reducer(self, x, y):
    return x + 2 * y

  def run(self, in):
    return reduce(self.reducer, map(self.doubler, in))

class ReduceTest(unittest.TestCase):
  def test1(self):
    kernel = ReduceKernel()
    output = kernel.run([1, 2, 3, 4])
    self.assertEqual(output, 54)
```

Fig. 7. A simplified TinyCU kernel and a unit test for the kernel.

B. Parallelization bug

The TinyCU kernel parallelizes loops in C++ using `omp parallel for` pragmas. Consider the following fragment of the generated program implementing a *sum* reduction over a vector of double values.

```
.....
double accum = 0.0; # accum is shared by threads
 #pragma omp parallel for reduction(+:accum)
for (int sum_i = 0; sum_i <= tmp0->size(); sum_i++) {
    accum = (accum + (*tmp0)[sum_i]);
}
.....
```

The code accumulates the values of in the vector in a *shared* variable `accum`. Using the attribute `reduction(+:accum)` ensures that each update of `accum` in the loop is atomic. That is, the OpenMP runtime will emit code to ensure that each iteration of the `for` loop will run without harmfully interfering with concurrently running iterations. Thus, we do not have to use any synchronization operations to protect `accum`.

Suppose that we forget to use `reduction(+:accum)` and think that OpenMP will impose sufficient synchronization to ensure the correctness of the reduction. We are wrong in this assumption because without this pragma attribute, OpenMP does not add any synchronization to the loop body. As a result, threads may be interleaved with each other before and after accessing `accum`. This breaks our intention that each iteration of the loop updates `accum` atomically.

When executing the loop without `reduction(+:accum)`, our tool reported serializability violations when reading and writing `accum`. By examining the trace of a violation, we were able to see that threads are accessing `accum` arbitrarily without any synchronization. Adding `reduction(+:accum)` back solved the problem, and our tool reported no parallelism-related errors, indicating that the interleavings of threads cause no unintended nondeterminism in the parallelized program.

C. Translation bug

In implementing TinyCU, we discovered that we had actually introduced a bug in the compilation of the `map` function. Fig 8 shows the compiled code containing the error. The shaded portion of line 6 shows the error, in which the upper array bound is one greater than the correct bound. This bug was introduced due to a misunderstanding of the loop bounds of the `For` constructor, incorrectly assuming that the upper bound was exclusive rather than inclusive. When executing the replay tool, it reports that the traced execution has too many iterations, indicating a translation error.

```
1  std::vector<double>* run() {
2    ......
3    std::vector<double>* tmp0_inner =
4        new std::vector<double>(*tmp0);
5    #pragma omp parallel for
6    for (int i = 0;  i <= tmp0->size() ; i = (i + 1)) {
7      (*tmp0_inner)[i] = doubler((*tmp0)[i]);
8    }
9    ......
10 }
```

Fig. 8. Parallelized C++ code representing the `map` operation with a bug shaded. The loop bound check is incorrect and ignores the last array element.

D. Wrong-assumption bug

To demonstrate a wrong-assumption bug, we used the `reduce` operator. In TinyCU, `reduce` requires that the reduction function be commutative and associative, allowing the compiler to execute reductions in any order it chooses. The underlying implementation of `reduce` in the optimized TinyCU code is a tree-reduction, which allows multiple elements of the reduction to be run in parallel. However, the default Python `reduce` function performs a left-to-right reduction. In Fig 7, the programmer has defined a reduction function which is not associative, but the programmer has assumed that the reduction function operates on elements in left-to-right order. When executing the replay tool, the replayed execution iterates over the reduction in the same tree-reduction order and obtains the same intermediate assignment values but still fails the test case. The tool reports a wrong-assumption bug because the replayed execution matches the optimized execution.

VII. Discussion

We now highlight common results from our case studies. When implementing a compiler, bugs are likely to fall into one of the categories: parallelization, translation, and wrong-assumption bugs. Thus, the effect of such a bug on the behavior of the program can be detected by focusing on the behavior of intermediate programs, rather than a complicated, end-to-end reasoning about the input and final output programs of the entire compilation. Identifying the right algorithms and properties to check on the intermediate programs is crucial to simplify this end-to-end reasoning without resorting to equivalence checking between each stage, which may be fragile given that the compilers use specialization, rendering their results correct only for the given input. Furthermore, because of the multi-stage nature of the compilers we study, applying existing debugging algorithms without accounting for the staged compilation will not assist in isolating the buggy code as well as our approach since the algorithms will only identify the bug in the final output of the compiler.

All of the bugs we study manifest themselves at the very end of the execution as incorrect output. Thus, all stages of the compilation are equally likely to be buggy and it is therefore difficult to identify the buggy stage. Our method helps the user in this difficult task by applying checking algorithms to intermediate programs, eliminating each as the source of the bug in turn. Moreover, our method makes the parallelism-related bugs more visible to the programmer (by perturbing the thread schedule to force bugs to manifest). For example, while the buggy thread schedules in our stencil and TinyCU examples occur in only a fraction of test runs, the instrumentation causes the error to happen in every execution. This avoids incorrectly declaring the tests successful and deploying the compilers with bugs in them.

Completeness Our method is not complete: Our tools may not detect a violation even though the compiled program fails a test. Such cases indicate the need for additional techniques to focus on other properties of the program, e.g., memory-leak detection [10], to integrate into the multi-level debugging.

VIII. Related work

DSEL Compilers Domain-specific embedded languages [18] attempt to override the limitations of traditional DSLs by embedding them in a high-level language. Like Asp, the Delite framework [8] enables writing embedded DSL compilers in a host language, in their case Scala. Delite also uses phased compilation, with a unified intermediate form across DSLs; its structure is amenable to the debugging methodologies in this work. For compilers that use language macro systems such as those in Lisp and Racket [26], instrumentation as used here may be inserted in the transformation macros.

Parallelism correctness We use the NDSeq approach [4] to specify and check parallelism correctness, which uses as specification a sequential version of the program with explicit nondeterminism generated by former stages of the compilation. Several other parallel correctness criteria have been studied for parallel programs. These criteria include data-race freedom [22], [30], atomicity [15], and linearizability [17]. Among all, NDSeq is the most appropriate approach for multi-level debugging, because it provides a *complete* separation of concerns about the parallelism and other sequential aspects of the program. NDSeq specification differs from determinism specification and checking [5], [3], [27] in that NDSeq not only allows one to specify that the final state is independent of the thread schedule, but also allows one to specify that the final state that depends on thread schedule is equivalent to the state arising due to nondeterministic choices in the NDSeq.

Fixing bugs In this work we do not focus on fixing the bugs, which can be incorporated to the compilation environment to identify fixes to certain problems. For example, there exist lock allocation-based approaches [14], [21], [20] to infer efficient locking schemes for parallel code regions and refactoring techniques [12], [28] that help the programmer in writing code with efficient-but-tricky locking techniques.

Fault diagnosis For complicated bug scenarios, the identifying the real cause of the bug, even after categorizing the bug and relating its causes to particular compilation stage, may still be a challenge. In such cases, more advanced techniques on software fault localization [1], [11], [13], bug reproduction [2], [19], and code inspection [16] can be integrated into our method to identify the root cause of the bug—i.e., the particular components or code regions in the compiler—by exploiting the recorded trace of the failing execution.

Replay The method of tracing and replaying applications has been used in other techniques for debugging nondeterministic code [23], [9]. Our technique is novel in that we trace an execution of an optimized program to replay on less-optimized versions of the same program to locate errors in the compilation process.

Acknowledgments

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *ASE*, 2009.
[2] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP*, 2008.
[3] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.
[4] J. Burnim, T. Elmas, G. Necula, and K. Sen. NDSeq: Runtime checking for nondeterministic sequential specifications of parallel correctness. In *PLDI*, 2011.
[5] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.
[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *PPoPP*, 2011.
[7] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective, just-in-time specialization. In *PMEA*, 2009.
[8] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*, 2011.
[9] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98*.
[10] J. Clause and A. Orso. Leakpoint: pinpointing the causes of memory leaks. In *ICSE*, 2010.
[11] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP*, 2005.
[12] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in java. In *OOPSLA*, 2009.
[13] M. B. Dwyer, R. Purandare, and S. Person. Runtime verification in context: can optimizing error detection improve fault diagnosis? In *RV*, 2010.
[14] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, 2007.
[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.
[16] J. H. Hayes, I. R. Chemannoor, and E. A. Holbrook. Improved code defect detection with fault links. *Softw. Test., Verif. Reliab.*, 2011.
[17] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 1990.
[18] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 1996.
[19] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *ICSE*, 2012.
[20] P. Liu and C. Zhang. Axis: automatically fixing atomicity violations through solving control constraints. In *ICSE*, 2012.
[21] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[22] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Prog. Lang. Syst.*, 1992.
[23] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *WODA 2005*.
[24] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.
[25] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.
[26] I. PLT Scheme. The racket language, 2012.
[27] C. Sadowski, S. Freund, and C. Flanagan. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *ESOP*, 2009.
[28] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring java programs for flexible locking. In *ICSE*, 2011.
[29] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
[30] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.