Mix Testing: Specifying and Testing ABI Compatibility of C/C++ Atomics Implementations

LUKE GEESON, University College London and Arm Ltd, United Kingdom
JAMES BROTHERSTON, University College London, United Kingdom
WILCO DIJKSTRA, Arm Ltd, United Kingdom
ALASTAIR F. DONALDSON, Imperial College London, United Kingdom
LEE SMITH, Arm Ltd, United Kingdom
TYLER SORENSEN, University of California at Santa Cruz, USA
JOHN WICKERSON, Imperial College London, United Kingdom

The correctness of complex software depends on the correctness of both the source code and the compilers that generate corresponding binary code. Compilers must do more than preserve the semantics of a single source file: they must ensure that generated binaries can be composed with other binaries to form a final executable. The compatibility of composition is ensured using an Application Binary Interface (ABI), which specifies details of calling conventions, exception handling, and so on. Unfortunately, there are no official ABIs for concurrent programs, so different atomics mappings, although correct in isolation, may induce bugs when composed. Indeed, today, mixing binaries generated by different compilers can lead to an erroneous resulting binary.

We present *mix testing*: a new technique designed to find compiler bugs when the instructions of a C/C++ test are separately compiled for multiple compatible architectures and then mixed together. We define a class of compiler bugs, coined *mixing bugs*, that arise when parts of a program are compiled separately using different mappings from C/C++ atomic operations to assembly sequences. To demonstrate the generality of mix testing, we have designed and implemented a tool, `atomic-mixer`, which we have used: (a) to reproduce one existing non-mixing bug that state-of-the-art concurrency testing tools are limited to being able to find (showing that `atomic-mixer` at least meets the capabilities of these tools), and (b) to find four previously-unknown mixing bugs in LLVM and GCC, and one prospective mixing bug in mappings proposed for the Java Virtual Machine. Lastly, we have worked with engineers at Arm to specify, for the first time, an atomics ABI for Armv8, and have used `atomic-mixer` to validate the LLVM and GCC compilers against it.

CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**.

Additional Key Words and Phrases: Compiler Testing, Concurrency, Interoperability

ACM Reference Format:
Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair F. Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson. 2024. Mix Testing: Specifying and Testing ABI Compatibility of C/C++ Atomics Implementations. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 287 (October 2024), 26 pages. https://doi.org/10.1145/3689727

Authors' Contact Information: Luke Geeson, University College London and Arm Ltd, London, United Kingdom, luke.geeson@cs.ucl.ac.uk; James Brotherston, University College London, London, United Kingdom, j.brotherston@ucl.ac.uk; Wilco Dijkstra, Arm Ltd, Cambridge, United Kingdom, wilco.dijkstra@arm.com; Alastair F. Donaldson, Imperial College London, London, United Kingdom, alastair.donaldson@imperial.ac.uk; Lee Smith, Arm Ltd, Cambridge, United Kingdom, lee.d.smith@acm.org; Tyler Sorensen, University of California at Santa Cruz, Santa Cruz, USA, tyler.sorensen@ucsc.edu; John Wickerson, Imperial College London, London, United Kingdom, j.wickerson@imperial.ac.uk.

ACM 2475-1421/2024/10-ART287
https://doi.org/10.1145/3689727

1 Introduction

Composing binaries is key to the correctness of today's software. Compilers must be semantics-preserving [29] in the sense that every behaviour of the compiled program must also be a behaviour of the source program under their respective semantic models. The question of *compositional* correctness arises when combining binaries produced by different compilers. Compositional correctness is ensured with an *application binary interface* (or ABI), which specifies compatible assembly code for compilers to implement. Processor designers publish many such ABIs [4, 39, 40], like the ABI for the Arm Architecture [4], but the ABI of *concurrent* programs is unexplored beyond early work [45]. We specify and test the concurrency ABI of today's compilers.

Multi-core processors are of course everywhere, and they implement *relaxed memory models* [3, 34, 36–38]. Memory models define the behaviours of concurrent programs, and in the case of ISO C/C++ [24] the behaviour a compiler should implement. These models have not prompted the development of official concurrency ABIs as far as we can tell, perhaps because there is a widely held belief that concurrent code should be compiled using one mapping (describing how atomic operations map to assembly). Supposedly, no guarantees are given if mappings are *mixed* together.

However, mixing mappings does occur in industry applications. Generally, mixing code is allowed for CPUs that can co-exist in the same shared-memory system, and mixing occurs in industry projects where portability is key. For instance, mixing MSVC and LLVM-generated code occurs on Windows on Arm [31] where MSVC's C/C++ STL accesses [26] (which use barriers), are mixed with LLVM's mappings (which use acquire/release instructions to access memory). Likewise, the developers of Mono, a tool for creating portable applications, insert barriers [44] when mixing LLVM's and GCC's mappings for the Arm architecture. Kernel developers [12] are cognizant of correctness issues associated with this mixing, and currently resolve them via online discussions.

Unfortunately, ABI-related concurrency bugs can arise when mixing. An ABI-related concurrency bug (herein a *mixing bug*) arises when the behaviour of a compiled (concurrent) program, as allowed by its architecture memory model, is not a behaviour of the source program under its source model. A mixing bug is a special kind of concurrency bug that arises when compiled (concurrent) programs are composed. The potential for mixing bugs has arisen as architectures have evolved, introducing new instructions, and with them new mappings from C/C++ to assembly. Without a concurrency ABI, there are no constraints on what compilers *must* do beyond those constraints imposed by the memory model, and mappings are chosen based on what is best for an architecture in isolation. Today's compilers have mixing bugs, as we now show.

Example 1.1. Consider the classic store-buffering test in Fig. 1(a), where each access has sequentially consistent [26] ordering. The outcome `{P0:t=0; P1:u=0}` is forbidden by the C/C++ memory model [24]. After compiling that whole program using `clang -O3 -march=armv7-a` (Fig. 1(b)), the compiled program does not exhibit `{P0:t=0; P1:u=0}` under either the unofficial Armv7-A model [35] or the newer Armv8 AArch32 model [32]. This is because the `store` operations map to assembly sequences that end with `DMB`s, which prevent reordering with the subsequent `load` (`LDR`) instruction. Compiling the whole program using `clang -O3 -march=armv8` (Fig. 1(c)) does not expose the outcome (under the Armv8 [32] model) either. With this mapping, the `store` no longer has a trailing fence; instead, the store-to-load reordering is enforced by mapping the atomic `load` to an acquire-load (`LDA`), which cannot be reordered with the store-release (`STL`).

However, the constituent operations of Fig. 1(a) may be compiled for different (compatible) architectures and mixed together. For example, suppose we separately compile the `store` operations using `-march=armv8` and the `load` operations using `-march=armv7-a`, and combine the resulting binaries into a final executable. This executable exhibits the unwanted outcome under the Armv8 model, because the Armv7-A mapping expects a barrier after the `store` that the Armv8 mapping

(a) The store-buffering litmus test in C-like code.

```
atomic_int *x = 0, *y = 0;
// Thread P0          || // Thread P1
store(x,1,sc);        || store(y,1,sc);
t = load(y,sc);       || u = load(x,sc);
// Outcome {P0:t=0; P1:u=0} must be forbidden.
```

(b) Compiling the whole program with `clang -march=armv7-a -O3` finds no bugs. The ● barriers preserve the store-to-load ordering.

```
// store(x,1,sc) ↴          || // store(y,1,sc) ↴
  MOV R1, #1                ||   MOV R1, #1
  DMB ISH                   ||   DMB ISH
  STR R1, [x]               ||   STR R1, [y]
● DMB ISH                   || ● DMB ISH
// t = load(y,sc) ↴         || // u = load(x,sc) ↴
  LDR R0, [y]               ||   LDR R0, [x]
  DMB ISH                   ||   DMB ISH
// Outcome {P0:t=0; P1:u=0} is forbidden.
```

(c) Compiling the whole program with `clang -march=armv8 -O3` finds no bugs. The store-releases (◓) and the load-acquires (◒) work together to preserve the store-to-load ordering.

```
// store(x,1,sc) ↴          || // store(y,1,sc) ↴
  MOV R1, #1                ||   MOV R1, #1
◓ STL R1, [x]               || ◓ STL R1, [y]
// t = load(y,sc) ↴         || // u = load(x,sc) ↴
◒ LDA R0, [y]               || ◒ LDA R0, [x]
// Outcome {P0:t=0; P1:u=0} is forbidden.
```

(d) Compiling the stores with `clang -march=armv8 -O3` and the loads with `clang -march=armv7-a -O3` reveals a mixing bug. The lone store-release (◓) is not sufficient to preserve the store-to-load ordering.

```
// store(x,1,sc) ↴          || // store(y,1,sc) ↴
  MOV R1, #1                ||   MOV R1, #1
◓ STL R1, [x]               || ◓ STL R1, [y]
// t = load(y,sc) ↴         || // u = load(x,sc) ↴
  LDR R0, [y]               ||   LDR R0, [x]
  DMB ISH                   ||   DMB ISH
// Outcome {P0:t=0; P1:u=0} is now allowed.
```

Fig. 1. Example of a mixing bug that cannot found by ordinary testing

does not provide, and the Armv8 mapping expects a load-acquire that the Armv7 mapping does not provide. This bug has been reported and confirmed [18]. The example can be fixed by adding a leading barrier in front of the `LDR` instruction for the the Armv7-A mapping.

Current techniques cannot find mixing bugs like the example above. Prior work [10, 23, 41, 52] that tests the compilation of concurrent programs operates on a *closed-world assumption* [42], finding bugs when *whole* programs are run through a compiler, using one atomics mapping.

We present the *mix testing* technique. Mix testing takes a C/C++ litmus test and a set of compiler profiles that cover different atomics mappings. Mix testing splits the litmus test into *instructions* that are compiled separately using each compiler profile, and each compiled instruction is then combined into one of *multiple* assembly litmus tests that represent combinations of concurrency implementations of the original C/C++ test. Concurrency-related compiler bugs are then detected (as detailed by Geeson and Smith [23]) by comparing the compiled program behaviour under its architecture model with the source program under the C/C++ model. We thus unearth valuable insights into the difficulty of testing the compilation of concurrent code, as a problem that cannot

simply be addressed by testing atomics mappings in isolation, but rather by strategically testing in the presence of exponentially many choices of mappings, both now and as architectures evolve.

We have put mix testing into practice by designing and implementing a new tool, `atomic-mixer`. We present an empirical evaluation showing that mix testing, via the `atomic-mixer` tool, improves on the state-of-the-art for the task of finding compiler bugs at the interface between multiple implementations of concurrency that are supposed to be compatible. Mix testing strictly generalises prior testing work with respect to a single compiler profile, finding bugs that are arguably more elusive since they exist on a larger test surface than that explored by prior work. We demonstrate the generality of mix testing, by using `atomic-mixer` to find a non mixing bug that prior concurrency testing work is limited to being able to find: this shows that `atomic-mixer` is at least as capable as these tools with respect to the kinds of bugs it can find. We then show that `atomic-mixer` can go further: we have used `atomic-mixer` to discover four previously unknown mixing bugs in LLVM and GCC, one of which has been fixed, and the others confirmed and triaged for fixing by compiler engineers. We found one of the four mixing bugs [13] in GCC's `_Atomic struct` implementation manually, since we rely on the `herd` simulator, which does not support `structs`. Lastly, we found a *prospective* mixing bug in mappings proposed for the Java Virtual Machine (JVM).

Significant work was required to reduce the complexity of mix testing, since the number of compiled tests expands exponentially in the size of the input programs and the number of compiler profiles under test. To bound complexity in practice, we developed an atomics ABI [22] for Armv8-A AArch64 with Arm's compiler teams. An atomics ABI is a specification of mappings between C/C++ atomic operations and assembly sequences along with a statement of their interoperability. We specify mappings from C/C++ atomics to AArch64 assembly sequences and special cases that must be implemented to prevent mixing bugs (and generally non-mixing bugs). As long as a compiler is ABI compatible, compiling tests that use any of the ABI's mappings will not induce mixing bugs. We use `atomic-mixer` to automatically validate ABI-compatibility of LLVM and GCC (modulo the bugs we found). As far as we know this is the industry's first open source specification of an atomics ABI with a tool to automatically check compatibility. Our contributions are as follows:

- We present the *mix testing* technique that mixes implementations of atomic operations, the `atomic-mixer` tool that implements this technique, and an artifact to reproduce our results.
- We define a special class of compiler bugs, coined *mixing bugs*, that arise when different parts of a program are compiled using different compiler mappings. We focus on concurrency-related mixing bugs, but emphasize that mix testing and mixing bugs are more general.
- We reproduce one existing non-mixing bug, find four previously unknown mixing bugs [13, 17–19] in LLVM and GCC, and one prospective mixing bug in proposed JVM mappings.
- We report on our experience working with engineers at Arm to publish the Armv8 Atomics ABI specification [22] which, to our knowledge, is the industry's first public atomics ABI.

Alongside a number of novel insights:

- The observation that mixing bugs can manifest when different parts of a program are compiled via different mappings (and that doing so *is* perfectly legal and commonplace).
- That it is not sufficient to test mappings in isolation. We identify a novel dimension for litmus testing and techniques to effectively "sample" the exponential search space of mix tests.
- Raising awareness of a blind-spot in current practice related to mixed compilation, and the role the ABI can play in specifying the interoperability of different atomics mappings.

The rest of this work is structured as follows. §2 covers the background, §3 covers the mix testing technique, §4 covers the `atomic-mixer` tool design, and challenges faced during implementation. We evaluate the efficacy of `atomic-mixer` in §5. In §6 we describe the Armv8 Atomics ABI. We end with related work in §7 and discuss conclusions in §8.

2 Background: Memory Models, Litmus Tests, and Compiler Testing

We explore mix testing in the literature using the bug report in Fig. 1.

$$\begin{array}{ll} \mathit{LitmusTest}_{\mathcal{L}} = \{\, \texttt{init} : \mathit{State},\ \texttt{prog} : \mathit{Prog}_{\mathcal{L}},\ \texttt{pred} : \mathit{Pred} \,\} & \mathit{Pred} = \mathit{State} \rightarrow \mathit{Bool} \\ \mathit{Prog}_{\mathcal{L}} = \mathit{Set}\,(\mathit{Thread}_{\mathcal{L}}) & \texttt{TID} = \{\, \texttt{P0}, \texttt{P1}, \ldots \,\} \\ \mathit{Thread}_{\mathcal{L}} = \{\, \texttt{instrs} : \mathit{Instrs}_{\mathcal{L}},\ \texttt{tid} : \texttt{TID} \,\} & \mathit{Instrs}_{\mathcal{L}} = \mathit{Set}\,(\mathit{Instr}_{\mathcal{L}}) \\ \mathit{Instr}_{\mathcal{L}} = \{\, \texttt{instr} : \mathcal{L}\text{-instr},\ \texttt{iid: IID} \} & \texttt{IID} = \{\, \texttt{P0\_0}, \texttt{P0\_1}, \ldots \,\} \end{array}$$

Fig. 2. We formalise litmus tests as labelled records

2.1 Litmus Tests, Executions, and Memory Models

Fig. 2 formalises the *litmus tests* in Fig. 1 as labelled records. Litmus tests consist of a fixed initial state (named `init`), a concurrent program written in language $\mathcal{L}$ (`prog`), and a predicate over the final state (`pred`). States are sets of assignments to shared data used in the concurrent program. A concurrent program consists of one or more threads of execution. Each thread consists of a list of instructions (`instrs`) and its *thread id* (`tid = P0, P1, ...`). Each instruction is referred to by a *instruction id* (`iid = P0_0, P0_1, ...`) as we will discuss the semantics of each instruction.

A litmus test checks whether there is an erroneous final state satisfying its predicate. The initial state in Fig. 1(a) assigns the value `0` to shared memory locations x and y. Each thread executes its instructions in parallel `P0||P1`, where each thread reads-from or writes-to (collectively *accesses*) shared memory locations.

Intuitively, threads *communicate* through shared memory, influencing the executions of other threads that read from memory. A sequence of accesses made by `P0||P1` defines an *execution* and the partial order on all accesses describes many possible executions. To complicate matters the instructions on each thread may be executed *out-of-order*, increasing the number of executions a litmus test may exhibit. Each execution finishes in one of several final states, known as *outcomes*. The predicate over these final states returns true if the specified final state(s) are reachable.

Memory consistency models filter out invalid executions. Some executions are forbidden according to language or architecture specifications and memory consistency models (herein models) apply predicates to executions to outlaw them. Models $\mathcal{M}_{\mathcal{L}}$ of many languages $\mathcal{L}$ exist including C/C++ RC11 [11], Armv8 AArch64 [3], RISC-V [34], IBM PowerPC [36], MIPS [37], and more. The set of executions allowed by a model $\mathcal{M}_{\mathcal{L}}$ characterises the *behaviour* of a litmus test $\mathcal{B}(\texttt{P0||P1}, \mathcal{M}_{\mathcal{L}})$.

Definition 2.1. Outcome. An outcome is a set of assignments to shared memory and thread-local data (e.g. `{P0:t=0; P1:u=0}`). Outcomes are the final states of executing a litmus test s from its initial state under a model $\mathcal{B}(s, \mathcal{M}_{\mathcal{L}})$. The set of outcomes is denoted $\mathit{Outcomes}(s, \mathcal{M}_{\mathcal{L}})$.

Run under C/C++ model
$\mathcal{B}(\texttt{P0||P1}, \mathcal{M}_{C/C++})$
⇓

```
{ P0:t=0; P1:u=1; }
{ P0:t=1; P1:u=0; }
{ P0:t=1; P1:u=1; }
```

Predicate not satisfied ✓

Example 2.2. Executing Fig. 1 (a) under the C/C++ RC11 model [11] is shown above. The outcome `{P0:t=0; P1:u=0}` is not present since RC11 forbids it. Fig. 1 captures the *store buffering* idiom. By checking the consistency of `{P0:t=0; P1:u=0}`, Fig. 1 tests whether the stores on each thread

can be reordered, or *buffered*, past the subsequent loads. In practice this can occur because of processor pipelines, caching, or other reasons. Prior work [23, 25] describes models and executions, but describing these concepts is not necessary to understand mix testing.

2.2 Compiler Testing and Concurrency-Related Compiler Bugs

We focus on compiler testing using concurrent C/C++ litmus tests. We input a syntactically valid C/C++ litmus test s to a compiler *comp* and observe its response. A compiler will either crash due to an internal error [51] or produce a binary that must be analysed for unexpected behaviour.

Given a litmus test s, A compiler bug arises if the behaviour of a compiled test $\mathcal{B}(comp(s))$ is not a behaviour of the source test $\mathcal{B}(s)$. This holds for all bugs, and so we focus on behaviours allowed by memory consistency models. Behaviours are characterised as program outcomes that arise due to the re-ordering of the observable effects of execution under some memory model $\mathcal{M}_{\mathcal{L}}$ that cannot be observed by running each thread in isolation (*ie* sequential execution). Further we are testing compilation from a source language, with associated memory model $\mathcal{M}_S$, to an assembly language, with associated memory model $\mathcal{M}_A$. Because these languages have different memory models, their allowed outcomes also differ. A *concurrency-related compiler bug* arises if there is an outcome of a compiled test allowed by its architecture memory model that is not an outcome of the source test under its source model:

Definition 2.3. Concurrency-related compiler bug. Let s be a well-defined *concurrent* source litmus test. Let $\mathcal{M}_S$ be the source model, and let $\mathcal{M}_A$ be the architecture model. Let $Outcomes(s, \mathcal{M}_S)$ be the set of allowed outcomes of s with respect to the model $\mathcal{M}_S$, and let $Outcomes(comp(s), \mathcal{M}_A)$ be the set of allowed outcomes of $c = comp(s)$ with respect to the model $\mathcal{M}_A$ after compilation with a compiler *comp*. Then *comp* exhibits a concurrency bug if $Outcomes(c, \mathcal{M}_A) \not\subseteq Outcomes(s, \mathcal{M}_S)$. Hereafter, we call concurrency-related compiler bugs *concurrency bugs*.

$$ConcurrencyBug(s, c) = Outcomes(c, \mathcal{M}_A) \not\subseteq Outcomes(s, \mathcal{M}_S)$$

3 Mix Testing: Automated Detection of Mixing Bugs

3.1 Definition

Fig. 3 details how the *mix testing* technique works. Given a C/C++ litmus test s, and a set P of compiler profiles under test, we produce a set C of compiled litmus tests. If any compiled litmus test exhibits a concurrency bug with respect to the source test then there is a mixing bug. Mix testing is defined as the process of (1) splitting up a source litmus test s into its *instructions*, (2) compiling each instruction separately using *compiler profiles*, (3) combining compiled instruction sequences

$$\begin{aligned}
&\textit{atomic-mixer}: LitmusTest_{src} \times Set(CompilerProfile) \\
&\quad \rightarrow Set(LitmusTest_{asm}) \\
&atomic\text{-}mixer(s, P) = combine(compile(split(s), P), s)
\end{aligned}$$

$$\begin{aligned}
&(1)\ split : LitmusTest_{src} \rightarrow Set(instrs_{src}) \\
&(2)\ compile : Set(instrs_{src}) \times Set(CompilerProfile) \\
&\quad \rightarrow Set(instrs_{asm}) \\
&(3)\ combine : Set(instrs_{asm}) \times LitmusTest_{src} \\
&\quad \rightarrow Set(LitmusTest_{asm})
\end{aligned}$$

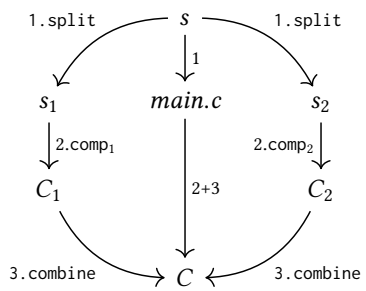

Fig. 3. Mix testing details. The splitting function chosen split a test into its instructions.

into *multiple* assembly litmus tests C, (4) checking whether any $c \in C$ exhibits a concurrency bug (Def. 2.3) with respect to s.

Mixing code generated with *different* compilers and architectures has the potential to find subtle bugs where code generated by different compiler profiles *should* be ABI-compatible, but turns out not to be. This approach has led to the discovery of a number of such bugs, as discussed in §5.2. We show how mix testing finds one such bug.

We split Fig. 1 into its constituent instructions using a *splitting function* (Def. 3.1). A splitting function takes a source litmus test and returns a set of its program instructions. Each instruction will be compiled separately and its instruction `iid` is used to recombine compiled sequences into one or more assembly litmus tests. Various splitting functions are explored in §3.3.

Definition 3.1. Splitting function. For a litmus test s:
$split(s) = \{\ instr \mid thread \in s.\texttt{prog},\ instr \in thread.\texttt{instrs}\ \}$

Each instruction is compiled separately using a *compiler profile*. A compiler profile is a description of a compiler, target architecture, and optimisation flags used to compile source code instructions. Compiler profiles are required to generate assembly sequences from C/C++ atomic operations. For example, the profile `"clang -march=armv7-a -O3"` compiles the `load` of y on `P0` Fig. 1(a) to the "`LDR;DMB`" sequence in Table. 1. We assume that the compilers under test use *fixed* (compile time) mappings between C/C++ atomics and assembly sequences. We explore runtime mappings in §5.3.

Definition 3.2. Compilation. For a set of source instructions *instrs* and a set P of profiles:
$compile(instrs, P) = \{\{\ \texttt{instr} = comp(i.\texttt{instr}), \texttt{iid} = i.\texttt{iid}\} \mid i \in instrs,\ comp \in P\ \}$

A compiler *implements* atomic operations. The compiler profile prompts the compiler to generate assembly sequences using mappings. Mappings are functions from atomic operations to instruction sequences. Since compilers emit different instructions based on the profile used, they typically implement *multiple* mappings. For instance LLVM implements the mappings in Table. 1 that lead to the tests in Fig. 1. We take a cross product of compiler profiles and source instructions to generate possible assembly sequences of each profile.

Table 1. Some of LLVM's sequentially consistent [26] mappings from C/C++ to Armv7-A and Armv8-A.

Atomic Operation	Compiler Profile	Assembly Sequence
`load(loc,sc)`	`clang -march=armv8 -O3`	`LDA R0, [loc]`
	`clang -march=armv7-a -O3`	`LDR R0, [loc]` `DMB ISH`
`store(loc,val,sc)`	`clang -march=armv8 -O3`	`MOV R1, #val` `STL R1, [loc]`

	`clang -march=armv7-a -O3`	`MOV R1, #val`
		`DMB ISH`
		`STR R1, [loc]`
		`DMB ISH`

We combine instruction sequences into compiled litmus tests using a *combining function* (Def. 3.3). Combining the compiled sequences produces exponentially many assembly litmus tests, all of which are valid combinations of the compiler profiles under test. We discuss how to prune this space of possible litmus tests, to prioritise litmus tests that are likely to be interesting, in §3.6.

Definition 3.3. Combining function. For a source litmus test s, and a set of compiled assembly sequences *asms* produced by splitting and compiling the instructions of s:

$$\begin{aligned} combine(asms, s) = \quad & let\ mk\ (asm) = (\texttt{init} = s.\texttt{init},\ \texttt{prog} = asm,\ \texttt{pred} = \texttt{s.pred})\ in \\ & map\ mk\ \{\ \{\ \texttt{instrs} = mk\_asm(asms, thread),\ \texttt{tid} = thread.\texttt{tid}\ \} \\ & \qquad \mid thread \in s.\texttt{prog}\ \} \\ mk\_asm(asms, thread) = \quad & \{\ asm \mid src \in thread.\texttt{instrs},\ asm \in asms, \\ & where\ src.\texttt{iid} = asm.\texttt{iid}\ \} \end{aligned}$$

Each source test compiles to multiple assembly tests. We check if any c in the set of compiled litmus tests C exhibits a bug with respect to the source litmus test s. We thus define a *mixing bug*:

Definition 3.4. Mixing bug: For a well-defined concurrent source program s and its set C of compiled litmus tests (given a splitting function, compiler profiles, and combining function):

$$MixingBug(s, C) = \exists c \in C, ConcurrencyBug(s, c) \qquad \text{(applies Def. 2.3)}$$

Example 3.5. Mix testing the test in Fig. 1(a) produces Fig. 1(d). Fig. 1(d) arises when the operations of Fig. 1(a) are compiled for Armv8-A and Armv7-A. Running Fig. 1(d) under the Armv8 model produces the outcomes below. The mixing bug occurs since the load of y on `P0` is compiled to the "`LDR;DMB`" sequence. Since the `LDR` instruction is missing a leading `DMB` barrier and it has no ordering semantics with respect to `STL`, it can reorder before the `STL` instruction on `P0`, leading to the outcome `{P0:R0=0; P1:R0=0}`.

$$\mathcal{B}(\texttt{P0}\|\texttt{P1}, \mathcal{M}_{Armv8})$$
$$\Downarrow$$

```
!!{ P0:R0=0; P1:R0=0; }!!
  { P0:R0=0; P1:R0=1; }
  { P0:R0=1; P1:R0=0; }
  { P0:R0=1; P1:R0=1; }
```

Predicate satisfied—bug ✗

3.2 Mix Test Notation

Since mix testing generates many litmus tests we introduce a *MixTest* notation in Fig. 4 to represent compiled tests that induce mixing bugs. A mix test is a labelled record consisting of a source litmus test (`test`) that is partitioned into its instructions and an *assignment* function from profiles to the instructions they compile (`assignment`). A thread is split into its instructions, then each instruction (represented by its `IID`) is compiled using a profile *comp* and sequenced together (`;`) to form a whole thread. The compiled test is represented using the mix test notation, for instance Fig. 1(d) is $(comp_1(\texttt{P0\_0}); comp_2(\texttt{P0\_1}))\|(comp_1(\texttt{P1\_0}); comp_2(\texttt{P1\_1}))$ and the record in Fig. 5.

$$\begin{aligned} MixTest\ = \{ & \texttt{test} : LitmusTest_{src}, \\ & \texttt{assignment} : CompilerProfile \longrightarrow Set(Instructions)\} \end{aligned}$$

Fig. 4. Mix test notation.

Fig. 1(d) test = $(comp_1(\texttt{P0\_0}); comp_2(\texttt{P0\_1}))\|(comp_1(\texttt{P1\_0}); comp_2(\texttt{P1\_1}))$.

Example:
{ test = Fig. 1(a),
assignment = {
$comp_1 \mapsto$ {P0_0, P1_0},
$comp_2 \mapsto$ {P0_1,P1_1}}}
where:
$comp_1$ = clang -march=armv8-a -O3
$comp_2$ = clang -march=armv7-a -O3

```
P0_0 = store(x,1,sc)
P0_1 = load(y)
P1_0 = store(y,1,sc)
P1_1 = load(x)
```

$comp_1$(P0_0) = "MOV;STL"
$comp_2$(P0_1) = "LDR;DMB"
$comp_1$(P1_0) = "MOV;STL"
$comp_2$(P1_1) = "LDR;DMB"

Fig. 5. Example of MixTest notation.

3.3 The Choice of Splitting Function

The splitting function determines the set of instructions I to be compiled separately. There is a trade-off here: the finer-grained the split, the more opportunities there are for problematic interactions between compiler mappings, but the larger the search space of possible sequences. The simplest function does not split the source test at all and just tests compilation using each profile $p \in P$, that is $|I| = 1$. This corresponds to (non-mix) testing as conducted by prior work [10, 23, 41, 52]. Mix testing strictly generalizes prior work, which compiles the whole program under one profile. The next function splits each source test s into its constituent K threads and compile those under different profiles, then $|I| = |K|$, and $|P|^{|K|}$ different choices. Intuitively bugs arise due to thread-local reordering [6, 41], so if each thread is compiled using one mapping and each mapping is *self*-consistent, then no bugs should arise. Since individual atomics mappings of each compiler have been rigorously tested, we expect most bugs found by splitting at the thread boundary are caught by non-mix testing. Therefore, we split litmus tests at the instruction level as shown in Fig. 6 ($|I| = 4$ in this case), so that I is bounded by the number of instructions of the input test. This function offers a good trade-off between the likelihood of finding bugs and the complexity of splitting the test into smaller fragments (given the simplicity of typical litmus tests).

3.4 Putting It All Together

Applying `atomic-mixer` to Fig. 1(a) is illustrated in Fig. 6, Fig. 7, and Fig. 8. `Atomic-mixer` produces the mix test in Fig. 4 that represents Fig. 1(d) using $(comp_1(\texttt{P0\_0}); comp_2(\texttt{P0\_1}))\|$ $(comp_1(\texttt{P1\_0}); comp_2(\texttt{P1\_1}))$.

The litmus tests generated are simple tests where each instruction is a load, store, barrier, loop, or conditional operations. We parse the program inside the C/C++ litmus test and replace each instruction (A,B) in the sequence A;B with $comp_1$(A);$comp_2$(B) (where $comp_{1/2}$ are profiles assigned by permutting all profiles under test). To handle conditionals or loops we check the condition for similar loads/stores, and then recurse into the loop body.

We use linkers to combine object files. We note that the linker will not apply link-time optimisation (LTO) unless the compilers used in the compilation and combination steps are the same. This is because LTO relies on GIMPLE and LLVM IR that is attached to object files, which is used to guide LTO. If the attached IRs of each file differ then the linker cannot optimise the assembly. When IRs differ the linker will instead emit branches to linked assembly code (instead of applying LTO to that code).

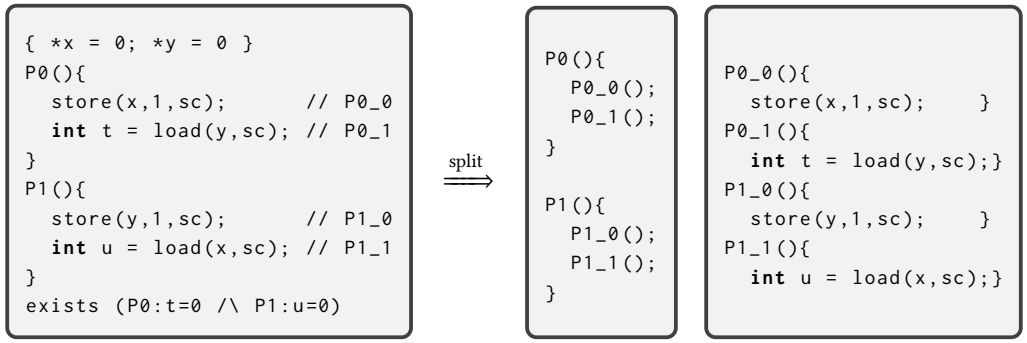

Fig. 6. Splitting Fig. 1(a) at the statement level produces multiple program instructions.

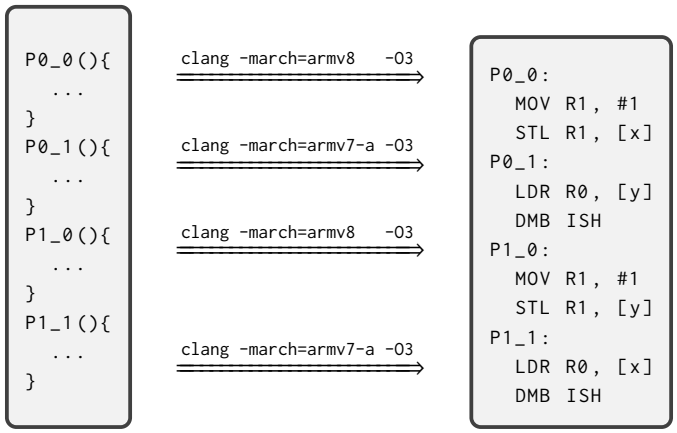

Fig. 7. Compilation using multiple profiles produces compiled instruction sequences.

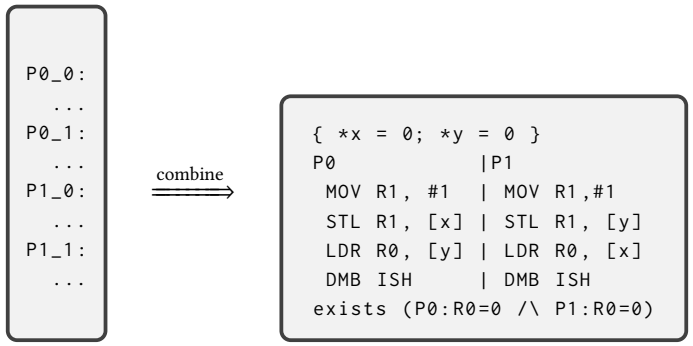

Fig. 8. Combining code and copying the initial state and predicate from Fig. 1 produces mixed tests.

```
{ *x = 0; *y = 0 }
P0                  | P1
  MOV X2, #1        |   MOV X2, #1
  BL P0_func_1      |   LDR X0, [%y]
  BL P0_func_2      |   DMB ISH
  B end             |   STR X2, [%x]
P0_func_1:          |
  LDR X0, [%x]      |   DMB ISH
  RET               |
P0_func_2:          |
  STR X2, [%y]      |
  RET               |
end:                |
exists (P0:X0 = 1 /\ P1:X0 = 1)
```

$$\mathcal{B}(\texttt{P0}\|\texttt{P1}, \mathcal{M}_{Armv8}) \Downarrow$$

```
{ P0:X0=0; P1:X0=0; }
{ P0:X0=0; P1:X0=1; }
{ P0:X0=1; P1:X0=0; }
!!{ P0:X0=1; P1:X0=1; }!!
```

Predicate satisfied—no constraints ✓

Fig. 9. (Left) AArch64 Load Buffering test where C/C++ relaxed loads are compiled to branch instructions on `P0`. (Right) outcomes under the AArch64 model [3]. The model allows the outcome `{P0:X0=1; P1:X0=1}`.

3.5 The Branching Problem

Splitting a test introduces function calls into each thread. A compiled litmus test has a corresponding branch instruction to the sequence that is separately compiled. For instance, GCC and LLVM generate *branch-with-link* and *return* instructions when targeting Armv8 AArch64. This can be problematic if processors implement the branch using a control-flow dependency that constrains the order of execution on each thread. Since we are looking for bugs that are exhibited by the re-ordering of observable events, we do not want to introduce such constraints.

Fortunately, each architecture we tested allows re-ordering across unconditional branches (Fig. 9). This means the effects of instructions after the branch can reorder before events of instructions prior. Intuitively, an unconditional branch is always taken, and so the micro-architecture is free to fill the pipeline with instructions on the branch taken as if the branch instruction itself was no-op. We empirically validated each compiler to check reordering (Fig. 9), and found both LLVM and GCC use call and return branches that allow reordering.

3.6 The Complexity of Mix Test Generation

The number of compiled litmus tests we generate is exponential in the number of compiler profiles and number of program instructions. Mix testing takes a set of S source litmus tests as input. Each $s \in S$ is split into a set of I program instructions using a splitting function (Def. 3.1). Each $i \in I$ is compiled separately using each compiler profile p in the set of P compiler profiles. Each source litmus test then yields a set C of different compiled litmus tests, where $|C| = |P|^{|I|}$. For example, mix testing Fig. 1 (a) yields $|P| = 2$, $|I| = 4$. That is $|C| = 16$ possible compiled tests. The number of $|C|$ tests for each $s \in S$ rapidly increases as the size of the input test increases. We therefore reduce S, P and I as much as possible whilst maximising the coverage of code generation. We do so by:

Curation of P: We omit compiler profiles that do not change the code generation of atomics relative to others. For instance `clang -O1`,`-O2`, and `-O3` use the same atomics mappings, but apply different optimisations. To maximise the chances of catching bugs we use `-O3`. We have worked with Arm's compiler experts to pick profiles that use different atomic mappings targeting Arm assembly. Depending on experts is a limitation (§5.5) we accept, but are open to automated techniques that find mappings as they arise.

Symmetry reduction on S: We do not generate source tests where the contents of each thread are simply swapped.

Bound $|I|$ by fixing the splitting function: The number of source instructions is determined by the splitting function. By splitting litmus tests at the instruction level we bound the number of generated tests, although the exponential complexity remains in theory. The number of compiled tests $|C|$ for each $s \in S$ is still exponential in I and P, and it is possible that duplicate tests exist in each set C. In the worst case when all compiler mappings in P are disjoint, that is a given C/C++ atomic operation compiles to a different instruction for each profile $p \in P$, the complexity is $|P|^{|I|}$ for each $s \in S$. In the best case when every compiler implements one set of mappings, we only need to test one compiler, and so $|P| = 1$ and the complexity is $1^{|I|}$ or 1 for each $s \in S$. The best case rarely happens in practice, since a given architecture has multiple possible atomics mappings, for each architecture sub-version, and hence multiple compiler profiles to test. For example, LLVM implements atomics differently for at least Armv8, Armv8.1, Armv8.2, Armv8.3, and Armv8.4. Further, new atomic instructions are announced with new architecture versions to improve performance of concurrent workloads. This means mix testing the compilation of concurrent programs is unfortunately a practical necessity at least until everyone agrees on an ABI that specifies common atomics mappings. Lastly, compilers are routinely revised and the code they generate often changes. As such our analysis is not exhaustive or even timeless, and we must periodically revise P and S as compilers are updated. It is not however surprising that the compiler profiles and tests suites must be updated.

3.7 The Scope of Our Testing

ISA-compatible assembly: We mix test compiled programs that are instruction-set architecture (ISA) *compatible*. This means their binary representation can be combined and executed without fault, as permitted by the envelope of the ISA. We do not yet require *atomics ABI-compatibility* in as far as we cannot find any official atomics ABIs outside of what we contribute in §6, but we do require that compiled programs are ABI-compatible in every other way (procedure call standards, exception handling, and so on). In general, mix-testing applies to code generated for CPUs of any architecture that can co-exist in the same shared-memory system, but for this work we limit our focus to a subset of recent Arm architectures.

Redundant mappings: It is desirable to omit repeated testing of instructions whose implementation remains the same. For instance, both LLVM and GCC implement a C/C++ `atomic_fence` operation as a `DMB` memory barrier when targeting the Armv8 architecture. In this case compiling using only one profile *should* suffice. Unfortunately, compilers do not implement common mappings. For instance, Arm's partners highlight [31] that MSVC emits two barriers for atomic read-modify-write operations, whereas LLVM emitted one at that time. Omitting test of mappings without an official ABI is risky and can lead to missed bugs. We therefore check redundant mappings and develop an Armv8 atomics ABI to define the envelope of compiler conformance going forward.

4 The `Atomic-mixer` Tool Implementation

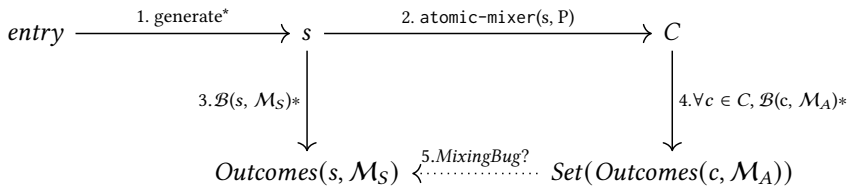

Fig. 10. Mix testing technique implementation using the new `atomic-mixer` tool and prior work* [1, 23].

4.1 Technique and Tool Implementation

We present the `atomic-mixer` tool and mix testing technique implementation. Fig. 10 shows how we implement the mix testing technique. We describe the mix testing process as follows:

(1) Generate a concurrent C/C++ litmus test s.
(2) Given a set P of compiler profiles, apply `atomic-mixer(s,P)` to get a set C of compiled tests.
(3) Collect $Outcomes(s, \mathcal{M}_S)$: the outcomes of simulating s under the source model $\mathcal{M}_S$ (Def. 2.1).
(4) For each $c \in C$ collect $Outcomes(c, \mathcal{M}_A)$: the outcomes under its architecture model $\mathcal{M}_A$.
(5) Check for mixing bugs (Def. 3.4).

We use the `Memalloy` [50] and `diy` [2] litmus test generators to produce tests S. We simulate source and compiled tests using the `herd` [1] simulator. We compare program outcomes using the `mcompare` tool [1]. The `atomic-mixer` tool itself extends the Téléchat toolchain [23], which handles the non-mix testing case by generating one compiled litmus test for each profile. The `atomic-mixer` tool increases coverage by enumerating atomics mappings of multiple profiles. Provided prior work is kept up to date, `atomic-mixer` is future-proof against the future evolution of programming languages, architectures, and their underlying memory models.

By extending Téléchat, `atomic-mixer` inherits a deterministic framework under authoritative models. Prior testing tools [41, 52] execute compiled programs on hardware to collect program outcomes (Def. 2.1). Hardware vendors are not however required to implement all behaviour permitted by the architecture specification. Consequently hardware-based testing may not exhibit all program behaviours, and bugs. Geeson and Smith [23] address this issue by parametrising testing under executable source and architecture models of Armv8 AArch64 [3] (official), RISC-V [34] (official), RC11 [11], Armv7 (unofficial) [35], Intel x86-64 [38], MIPS [37], IBM PowerPC [36], and more. By using models and simulation, `atomic-mixer` deterministically covers all possible outcomes of each test that terminates (up to bounds on loop unrolling).

4.2 Challenges Faced during Implementation

The key to efficient mix testing is knowing the number of mappings of a given atomic operation. There are many different compilers (for example, GCC and LLVM) and their code generation may change at any time to support new architectures, new optimisations, or modifications of existing implementations. A naïve approach is to test all compiler releases for each architecture. Despite the theoretical possibility that each release implements entirely different code generation, the reality is that few changes to atomics occur in practice. We therefore look for changes in code generation and only test each variant once. We describe simulation penalty and how we address it.

Simulation penalty: Simulating the behaviour of litmus tests under models is computationally complex. We must simulate each source program $s \in S$ to collect its behaviours. Then, for each such s we must simulate for every $c \in C$, where C is the set of compiled programs derived from s by `atomic-mixer`. Our goal is thus to reduce the number and size of the target test sets C for each s whilst increasing the coverage of code generated by compilers. We do so by hashing the generated assembly code of the litmus test.
We group C by hashes and check one representative of each group. It is possible that changing the compiler profile only changes one or two atomics mappings whilst other mappings remain unchanged. For example Armv8.3-A changes the mapping of acquire loads to use the `LDAPR` instruction instead of the existing Armv8 `LDAR` instruction. Since all other atomics remain unchanged many compiled programs will have the same static hash and behaviour under simulation. Only one program with a given hash needs to be simulated in a group. We compute hashes using the `mshowhashes` tool [1]. By doing so we only need to simulate one test from each group of tests with the same hash. Hashing drastically reduces the number of tests we must consider.

Handling programs that are larger than litmus tests: We limit our focus to small litmus tests of up to five threads, and up to 20 lines of code. The bottleneck of our flow is the `herd` simulator, which we use to compute the allowed behaviours of source and assembly tests under source and target models, respectively. As the number of threads grows, exhaustive simulation quickly becomes infeasible. It may be possible to use other tools, but `herd` is attractive for our current needs since it is easily extensible. That said, we don't expect much would be gained by moving to larger test cases, since we are focusing on testing mappings interoperability, and bugs that cannot be expressed using small litmus tests are rare (but certainly not impossible – we are aware of one bug [28]).

Mix testing is I/O bound: Even if we can discard duplicates using hashing, `atomic-mixer` must still *generate* them. `atomic-mixer` must generate all $c \in C$ as `mshowhashes` cannot compute the hashes until it has tests.

Adapting to changing architectures and language standards: This is a challenge for any concurrency testing tool. The C/C++ language and Arm Architecture specifications undergo numerous changes as implementors provide feedback and new requirements. Memory models, and testing tools that rely on models, must adapt in turn. We use `herd` to simulate the source C program and the compiled program. `herd` takes a cat [1] file describing the desired model, so changing language is a simple matter of changing this file. The models changed several times during this project, causing no issues.

5 Evaluation

We evaluated the mix testing technique and `atomic-mixer` tool by conducting a number of case studies using LLVM and GCC. We show that mix testing strictly generalises testing with respect to a single compiler profile by using `atomic-mixer` to find (non mixing) bugs that prior work is limited to being able to find (§5.1), and discover four previously unknown mixing bugs (§5.2) of which one was found manually (§5.4). We also found a mixing bug in mappings proposed for the JVM (§5.3). We cover the limitations of mix testing (in §5.5).

5.1 Reproducing an Existing (Non-mixing) Bug

Since mix testing with one profile corresponds to non-mix testing it follows that `atomic-mixer` should be able to reproduce existing bugs. We reproduce a (non-mixing) bug found by Geeson and Smith 2024 [23] using `atomic-mixer`.

Example 5.1. Consider the Message Passing test in Fig. 11 (left). Fig. 11 (middle) shows that the outcome {P1:r0=0; y=2} is forbidden by the RC11 model [11]. When compiled to target Armv8.2-A the compiled program (Fig. 17) exhibits the outcome under the Armv8 AArch64 [3] model. Fig. 11 (right) shows the outcomes of `atomic-mixer`-generated test (Fig. 17) allowed by the Armv8 AArch64 model [3]. These outcomes match those in the bug report [14]. This bug has since been fixed in LLVM by Arm's engineers.

```
{ *x = 0; *y = 0; }
P0() {
  store(x,1,rlx);
  fence(rel);
  store(y,1,rlx);
}
P1() {
  atomic_exchange(y,2,rel);
  fence(acq);
  int r0 = load(x,rlx);
}
exists (P1:r0=0 /\ y=2)
```

$\mathcal{B}(\mathsf{P0}\|\mathsf{P1}, \mathcal{M}_{C/C++})$ ⇓

```
{ P1:r0=0; y=1; }

{ P1:r0=1; y=1; }
{ P1:r0=1; y=2; }
```

⇓ Predicate not satisfied ✓

$\mathcal{B}(comp(\mathsf{P0})\|comp(\mathsf{P1}), \mathcal{M}_{Armv8})$ ⇓

```
  { P1:r0=0; y=1; }
!!{ P1:r0=0; y=2; }!!
  { P1:r0=1; y=1; }
  { P1:r0=1; y=2; }
```

⇓ Predicate satisfied—bug ✗

$comp$ = "clang -march=armv8.2-a -O3"

Fig. 11. `atomic-mixer` finds a non-mixing bug [14]. rlx = `relaxed`, rel = `release`, and acq = `acquire`.

5.2 Finding Bugs the State-of-the-Art Cannot

Mix testing can find bugs that current tools cannot, since they require mixing and are thus out of scope. We checked a compiler patch [30] and found and reported [17] a mixing bug that was missed by Geeson and Smith when they tested it back in January 2023. This example highlights the difficulty of testing the compilation of concurrency as a problem that cannot be addressed by testing atomics mappings in isolation, but rather by strategic testing in the presence of exponentially many choices of mappings. Mix testing takes the field forward both in terms of what is possible conceptually (mixing bugs) and what is possible in today's tools.

```
{ i128 *x, *y = 0; }

P0 () {
  store(x,1,sc);//P0_0
  i128 r0=load(y,sc);//P0_1
}
P1 () {
  store(y,1,sc);//P1_0
  i128 r0=load(x,sc);//P1_1
}

exists(P0:r0=0 /\ P1:r0 = 0)
```

$\mathcal{B}(\mathsf{P0}\|\mathsf{P1}, \mathcal{M}_{C/C++})$ ⇓

```
{ P0:r0=0; P1:r0=1; }
{ P0:r0=1; P1:r0=0; }
{ P0:r0=1; P1:r0=1; }
```

⇓ Predicate not satisfied ✓

$\mathcal{B}(comp_2(\mathsf{P0\_0}); comp_1(\mathsf{P0\_1})\| (comp_2(\mathsf{P1\_0}); comp_1(\mathsf{P1\_1})), \mathcal{M}_{Armv8})$ ⇓

```
!!{ P0:r0=0; P1:r0=0; }!!
  { P0:r0=0; P1:r0=1; }
  { P0:r0=1; P1:r0=0; }
  { P0:r0=1; P1:r0=1; }
```

⇓ Predicate satisfied—bug ✗

$comp_1$ = "clang -march=armv8.4-a -O3"
$comp_2$ = "clang -march=armv8-a -O3"

Fig. 12. Mixing bug [17]. The outcome {P0:r0=0; P1:r0=0} is forbidden by the C/C++ model [24], but the compiled program exhibits it under the AArch64 model [3]. sc = `seq_cst`, and i128 = `_Atomic __int128`.

Example 5.2. Consider the test in Fig. 12 (left). Fig. 12 (middle) shows that the outcome of the exists clause {P0:r0=0; P1:r0=0} is forbidden by the C/C++ model [24]. Fig. 12 (right) shows a mixing bug arises when mix testing the source test using profiles targeting Armv8-A and Armv8.4-A. Fig. 13 shows the mix test generated by `atomic-mixer`. In this case the load pair (LDP) of x on P1 has no leading barrier, and since LDP has no ordering semantics, its effects can be reordered before the store-release exclusive pair instruction (STLXP) on P1. The compiled program exhibits the outcome {P0:r0=0; P1:r0=0} under the AArch64 model [3].

```
{ *x = 0; *y = 0; }
P0                  |P1
 MOV X2, #1         | MOV X6, #1
 DMB ISH            |loop:LDAXP X1,X2,[%P1_y]
 STP X1,X2,[%P0_x]| STLXP W4,X5,X6,[%P1_y]
 DMB ISH            | CBNZ W4, loop
 LDP X4,X0,[%P0_y]| LDP X4, X0, [%P1_x]
 DMB ISH            | DMB ISH

exists (P0:X0 = 0 /\ P1:X0 = 0)
```

MixTest = {
    test=Fig. 12 (left),
    assignment = map}
where:
map={$comp_1$ ↦{P0_0,P0_1,P1_1},
     $comp_2$ ↦{P1_0} }
$comp_1$="clang -march=armv8.4-a -O3"
$comp_2$="clang -march=armv8-a -O3"

Fig. 13. Mix test that exposes mixing bug: $(comp_1(\texttt{P0\_0}); comp_1(\texttt{P0\_1}))\|(comp_2(\texttt{P1\_0}); comp_1(\texttt{P1\_1}))$.

Until now, only experts in both compilers and concurrency would be likely to find such a bug. The bug would not be caught by the state-of-the-art tools, since they do not conduct mix testing. The test in Fig. 12 is not unusual by concurrency standards, but the mixing bug is likely detectable only if the user has detailed knowledge of the atomics mappings in today's compilers and the concurrency experience needed to reproduce it. Indeed, neither concurrency architects nor engineers caught this bug. We worked closely with Arm's engineers to report the bug [17]. In the process, we developed mix testing and fostered a team of compiler engineers who handle queries regarding the compilation of atomics going forward[1].

We found three mixing bugs automatically [17–19], and one bug manually [13].

(1) 32-bit sequentially consistent load is missing a barrier: See Fig. 1, report [18], and §3.
(2) 64-bit sequentially consistent load is missing a barrier: See report [19]. An analogue of (1), but for 64-bit loads when compiling to target 32-bit systems.
(3) 128-bit sequentially consistent load is missing a barrier: See report [17], Fig. 12, and §5.2.
(4) `_Atomic` struct size and alignment differ between LLVM and GCC. See report [13], Fig. 16, and §5.4.

Each bug is triggered by a different tests and profiles. These tests were found using variants of store buffering tests with either sequentially consistent stores or read-modify-write operations. Picking the size of accesses from 32, 64, or 128-bits triggers different code paths in GCC and LLVM. Further, the Armv7 and Armv8 AArch64 back-ends are different targets in LLVM, and their code-generation is triggered by different compiler profiles. In other words we found three unique bugs. Generally, the test inputs and compiler profiles are not *orthogonal*. The choice of test and profile cannot be arbitrarily varied but must be chosen to find bugs. This is problematic, since the search space is exponential in these inputs (§3.6). Mix testing thus relies on good choices of profiles and tests to trigger the conditions for bugs.

It is reasonable to question whether mixing bugs only arise when mixing acquire-release and barrier-based implementations. We now explore a mixing bug that does not require barriers.

[1]For compiler and ABI inquiries please contact: arm.eabi@arm.com

5.3 Finding Mixing Bugs in Proposed Mappings

One of Arm's partners approached Arm's compiler teams with a proposal to the change the default mappings (Table. 2) of sequentially consistent [26] loads and stores when compiling for the release consistency processor consistency extension (RCPC). The RCPC extension introduces the LDAPR instruction whose effects can reorder before prior store-release (STLR) instructions that access different memory locations. The LDAPR instruction has the potential [47] to improve performance over the LDAR instruction (the current load implementation). Replacing LDAR with LDAPR alone however is unsound since it can reorder with prior stores (STLR). Instead, Arm's partners proposed to both strengthen the STLR with a trailing barrier (DMB ISH), and relax loads to use LDAPR, effectively preventing non-mixing bugs. We applied mix testing to show that this case would not be correct when mixing their proposed mappings in with code targeting Armv8-A.

Table 2. One of Arm's partners asked if relaxing SC loads, and strengthening SC stores would be sound.

Atomic Operation	Compiler Profile	Assembly Sequence
load(loc,sc)	clang -march=armv8 -O3 (current)	LDAR W0, [loc]
	clang -march=proposed -O3	LDAPR W0, [loc]
store(loc,val,sc)	clang -march=armv8 -O3 (current)	MOV W1, #val STLR W1, [loc]
	clang -march=proposed -O3	MOV W1, #val STLR W1, [loc] DMB ISH

Example 5.3. Consider the test in Fig. 14 (left). Fig. 14 (middle) shows that the outcome of the exists clause {P0:r0=0; P1:r0=0} is forbidden by the C/C++ model [24]. Fig. 14 (right) shows a mixing bug arises when mix testing the source test using the mappings in Table. 2. Fig. 15 shows the mix test we manually found (no compiler implements the proposed mappings without which atomic-mixer cannot work). In Fig. 15 the store-release (STLR) instruction on P0 has no trailing barrier, and the effects of executing the LDAPR can be reordered before the effects of the STLR. The compiled program exhibits the outcome {P0:r0=0; P1:r0=0} under the AArch64 model [3].

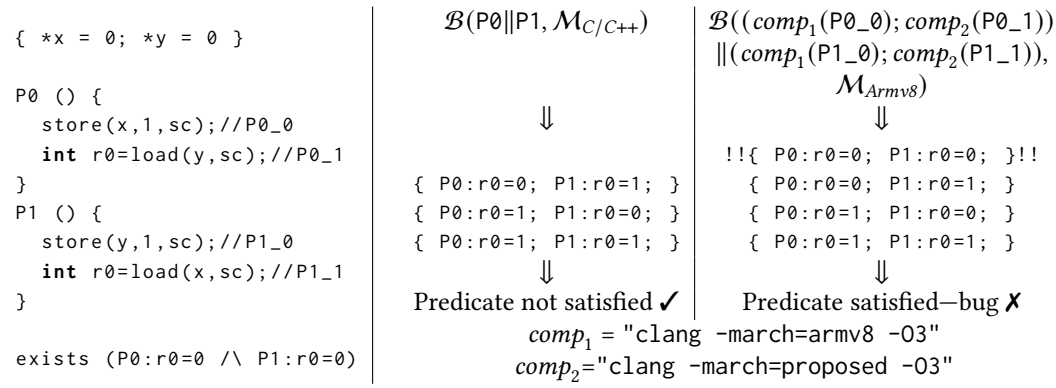

Fig. 14. A mixing bug arises if $comp_1$ and $comp_2$ mappings are mixed. The outcome {P0:r0=0; P1:r0=0} is forbidden by the C/C++ model [24], but the compiled program exhibits it under AArch64 [3]. sc = seq_cst.

```
{ *x = 0; *y = 0 }
 P0                | P1
  MOV W1, #1       |   MOV W1, #1
  STLR W1,[%P0_x] |   STLR W1,[%P0_y]
  LDAPR W0,[%P0_y]|   LDAPR W0,[%P0_x]

exists (P0:X0=0 /\ P1:X0=0)
```

MixTest = {
test = Fig. 14 (left),
assignment = map}
where:
map = { $comp_1 \mapsto$ {P0_0,P1_0},
$comp_2 \mapsto$ {P0_1,P1_1} }
$comp_1$ = above, $comp_2$ = above

Fig. 15. Mixing bug without barriers: $(comp_1(\texttt{P0\_0}); comp_2(\texttt{P0\_1}))\|(comp_1(\texttt{P1\_0}); comp_2(\texttt{P1\_1}))$.

There is nothing wrong with the proposed mappings, provided *all* stores are strengthened. In general we cannot know if a compilation unit will be mixed with other code for different (yet compatible) architectures. As long as multiple mappings exist, they may be mixed. The user can either guarantee the whole program is *always* compiled using the proposed mappings or otherwise every compiler implementation must change. This requires that every compiler that supports Armv8-A and above (including LLVM, GCC, and MSVC) strengthens their SC stores with a trailing barrier. Unfortunately such a wide reaching change is unlikely to be accepted in practice. This proposal constitutes an ABI break with respect to today's compilers.

It is possible to use the proposed mappings without mixing bugs. Arm's partner wanted to change the Java Virtual Machine (JVM) implementation to use the proposed mappings. When used in isolation these mappings are sound, since the JVM uses a JIT compiler that can dynamically generate code using the proposed mappings all at once. However there are three cases where mixing bugs can arise. Firstly, heap locations may be written to by the JVM's C++ code using SC atomics (the `STLR` instruction), but then later read by Java volatiles (using `LDAPR`). This can be fixed by inserting barriers after every C/C++ store in the JVM source. Secondly, a user's C++ code may share a memory buffer with Java code (for example, a `java.nio.ByteBuffer`), where the C/C++ code stores to the buffer and Java loads from it (using `VarHandle::getVolatile`). Again, the user must insert barriers after C/C++ stores. Thirdly, bugs may arise if the JVM interacts with C/C++ through foreign function interfaces (FFI) such as JNI (for instance using an API call to `SetIntField`). Assuming the JVM does not synchronize at FFI boundaries (see §3.5), barriers must added here too. Assuming these cases are handled, there are (probably) no mixing bugs.

5.4 Mixing Bugs in `_Atomic struct` Implementations

This bug [13] was discovered manually while developing the ABI in §6. Manual effort was required since `atomic-mixer` depends on `herd`, which doesn't support `structs`. This bug arises when two different compilers translate code for the same ISA. We discovered that GCC and LLVM have incompatible implementations of `_Atomic structs`. Both the size and alignment requirement calculated in Fig. 16 differs between compilers. The `sizeof` operator is used to determine the storage allocation and size of atomic instructions to be used. In this case GCC's engineers chose to use an inefficient locking call (`atomic_load`), whereas LLVM's engineers used a load acquire (`LDAR`) instruction. GCCs engineers chose to use the locking call, since there aren't any instructions to handle unaligned atomics or oddly-sized types, LLVMs engineers chose to use `LDAR` on the basis that every other access would share the same alignment values. Mixing code generated by both is problematic since LLVM may write struct padding bits to memory where GCC allocates entirely unrelated data—mixing LLVM and GCC code can invalidate data and hence program execution in unknown ways. The solution is to overalign and pad atomic types to the next supported atomic size.

```
typedef _Atomic struct { char a[5]; } X;
int size_x = sizeof(X); // GCC=5, LLVM=8
int align_x = __alignof(X); // GCC= 1, LLVM=8

X f(X *p) { return *p; }
// GCC=bl __atomic_load, LLVM=LDAR X2, [loc]

X g(X *p[2]) { return *p[1]; }
// GCC=__atomic_load, LLVM=LDR X1, [loc,#8]; LDAR X2, [X1]
```

Fig. 16. Mixing struct implementations. This issue also effects x86 code generation.

5.5 Limitations

There are three limitations that contribute to the complexity of mix testing. We rely on experts to provide the compiler profiles, we depend on model-based tooling, and require test generators that have good atomics coverage.

Fortunately, there are practical solutions to these issues. Typically, only a small number of compiler profiles introduce *new* atomics mappings, and so we only need to test those. Second, we assume herdtools [1] implements all instructions we test. We thus added new features to `herd` including an 128-bit signed integer type [15, 16] to handle Fig. 12. Lastly, the tests we used to find mixing bugs (§5.2) are generated by `Memalloy` [50] and `diy` [2], although Fig. 11 is not generated by today's tools [20]. In this case, we compare assembly program outcomes (§4.2) to find programs that induce bugs.

6 Industry Impact

In this section we cover our experience applying mix testing in industry. We worked closely with Arm's compiler teams to develop an *atomics application binary interface* (ABI) that specifies the mappings of source-level atomics into AArch64 assembly sequences. As far as we know this is the industry's first public specification of an atomics ABI with an accompanying tool (`atomic-mixer`) that can find bugs in non-compliant compilers. We summarize the specification as it is today, and refer the reader to the published document for updates [22].

6.1 An ABI Specification of Armv8 Atomics

The ABI is defined by a list of atomics mappings (1, below), accessed through compiler profiles that generate atomic instructions. Each mapping is correct if it satisfies a declarative statement of atomics ABI compatibility (2) when mix tested on a large sample of the test space (3).

6.1.1 Listing Atomics Mappings. We test atomics mappings produced by the compiler profiles in LLVM and GCC that use `-march=armv8+{lse|rcpc|rcpc3|lse128}/armv8.4-a`. Since architecture sub-versions such as `-march=armv8.1-a` imply some of these flags they are omitted.

Example 6.1. Table. 3 shows how a 32-bit integer exchange maps to either a compare-and-swap sequence when Armv8.0 is selected or a swap instruction (`SWP`) if the Armv8.1-a is selected.

6.1.2 Statement of ABI Compatibility. A compiler that implements the stated mappings is ABI-Compatible with respect to other compilers that implement the ABI.

In other words, given a set of compiler profiles, a splitting function (Def. 3.1), and C/C++ litmus test set S, the mappings are correct with respect to S if mix testing finds no mixing bugs (Def. 3.4). This definition comes with the constraint that this is not a correctness guarantee, but rather a statement backed up by bounded testing. Verifying the compilation of concurrent programs

Table 3. An exchange maps to a compare-and-swap loop or a SWPL instruction.

Atomic Operation	Compiler Profile	Assembly Sequence

`atomic_exchange(loc,val,release)`	Base (`-march=armv8`)	`MOV W2, #val`
		`lbl: LDXR W4,[loc]`
		`STLXR W3, W2,[loc]`
		`CBNZ W3,lbl`

	`+lse`	`MOV W2, #val` `SWPL W2, W4,[loc]`

under relaxed models is undecidable [7]. Instead we test programs with a fixed initial state, loop unroll factor, and no recursion. The ABI does not make any statement about the compatibility of compilers outside the test bounds specified, the provided mappings are not exhaustive, the document makes no statement about the compatibility of optimised programs, nor any statements concerning the performance of compiled programs under the provided mappings. Nevertheless, a rigorous statement of ABI compatibility backed by an effective regression testing method and tool, is a significant improvement.

6.1.3 Mix Testing 6.1.1 using 6.1.2. We generate a number of concurrency tests, checking ABI compatibility of compilers that implement the mappings in the document. At the time of writing the mixing bugs reported in GCC are fixed, but not in LLVM.

6.2 Using `Atomic-mixer` to Test ABI Compatibility

By following the steps in §4.1 we mix test LLVM and GCC given compiler profiles and tests as input. We generate tests that involve patterns of C/C++ atomic operations, memory order parameters, barriers, control-flow and straight line code up to 5 threads in size. These tests are not exhaustive but aim to test atomic operations introduced in C/C++11. The ABI specifies mappings for C11 atomic operations for 8, 16, 32, 64, and 128-bit width accesses for both signed and unsigned integer types. Each atomic operation maps to multiple assembly sequences. Table. 4 defines all the combinations of test, compiler, and architecture under test.

Table 4. We test combinations of C/C++ constructs × Acccess width/sign × Order × Arch.

C/C++ constructs:	`(atomic operations|non-atomic operations|barriers|control-flow|straight-line code)+`
Access width/sign:	`(u)int(8|16|32|64)_t`
Memory Order:	`(relaxed|acquire|release|acquire-release|seq-cst)+`
Target Architecture:	`(armv8|armv8+lse|armv8+rcpc|armv8+rcpc3|armv8+lse128 | armv8.4-a)+`

We generated thousands of C/C++ litmus tests using `diy` [1] and applied `atomic-mixer` to get millions of AArch64 assembly litmus tests. We used `mshowhashes` to remove redundant compiled tests (see §4.2) and `herd` [1] to search for mixing bugs. We parallelised [46] mix testing (with load balancing to reduce swap usage) on a 224 core ThunderX2 using 100GB runtime footprint and found no mixing bugs besides those we document in §5.2. We do not auto-generate tests for all mappings in the ABI, since the `diy` [1] generator does not support all read-modify-write operations, such as `fetch_add`. We manually constructed tests with unsupported operations and applied `atomic-mixer` to show there are no more mixing bugs in these cases.

6.3 Variation of Atomic Mappings in Practice

There are many implementations of a given atomic operation in practice. Considering only the Armv8-A AArch64 backends of LLVM and GCC, there are up to 5 different mappings for each primitive, but many primitives also have mappings for each size and signedness. In addition, individual mappings were changed in compiler patches, but the changes were not consistently applied. As a result, LLVM and GCC are not currently interoperable, but specifying the ABI is one step towards addressing this. Altogether, the ABI specification fills over seventeen A4 pages, even with as much duplication removed as possible. Beyond this, there are also mappings used by proprietary compilers such as MSVC, which we have not yet considered. We expect that other compiler implementations for RISC-V, Intel x86, and IBM PowerPC have the potential for ABI mixing bugs too.

6.4 Special Cases

We detail two special cases that compilers should handle. These bugs were found by prior work [23].

Read-modify-write should preserve read: Exchange can map to SWPL instructions (Table.3). However according to the Arm Architecture Reference Manual [5] *instructions where the destination register is* WZR *or* XZR*, are not regarded as doing a read for the purpose of a* DMB LD *barrier*. The bug in Fig. 11 arises since the effects of executing a SWPL may be reordered past the acquire fence (Fig. 17), we propose that compilers do not rewrite the destination register to be the zero register (WZR) in this case. This also applies to mappings using LD<OP> or CAS.

```
{ *x = 0; *y = 0 }
  P0                |  P1
   MOV W9,#1        |    MOV W9,#2
   STR W9,[%P0_x]   |    SWPL W9, WZR,[%P1_y]
   DMB ISH          |    DMB ISHLD
   STR W9,[%P0_y]   |    LDR W8,[%P1_x]

exists (P1:X8=0 /\ [y]=2)
```

$\mathcal{B}(\texttt{P0}\|\texttt{P1}, \mathcal{M}_{Armv8}) \Downarrow$

```
  { P1:X8=0; [y]=1; }
!!{ P1:X8=0; [y]=2; }!!
  { P1:X8=1; [y]=1; }
  { P1:X8=1; [y]=2; }
```

Predicate satisfied—bug ✗

Fig. 17. The compiled version of Fig. 11. The SWPL destination register is the WZR zero register.

Mutable const-qualified 128-bit data: Registers in AArch64 state hold 64-bit values. To load 128 bits atomically we must use a compare-and-swap loop (see Table.5) when Armv8.0 is selected. If const-qualified memory is marked read-only (and stored in, for example, .rodata) then executing the store-exclusive pair (STXP) instruction will crash the program. We propose that compliant implementations should mark const-qualified atomic locations as mutable. This also affects x86 code generation [43] of 64-bit access.

Table 5. Some mappings for an 128-bit atomic load, in this case a compare-and-swap loop or an LDP instruction.

Atomic Operation	Compiler Profile	Assembly Sequence
load(loc,relaxed)	Base (-march=armv8)	lbl: LDXP X9, X10, [loc] STXP W3, X9, X10, [loc] CBNZ W3,lbl
	+lse	LDP W2, W4, [loc]

6.5 Sub-ABIs, ABI-Islands, and the Baseline ABI

There is no one 'true' ABI, but rather a specification that serves most purposes. The ABI we provide represents a *baseline* specification for any implementation that aspires to be compatible across all versions of the Armv8 architecture. Ideally, mainstream implementations such as LLVM and GCC will adhere to this ABI in the future. This ABI does *not* prevent implementors from creating their own ABI, whether it is a subset of the baseline (a *sub-ABI*) or an altogether different set of mappings (a disjoint *ABI-island*). A sub-ABI could induce mixing bugs on unsupported architectures (like in §5.3) and it would be up to the user of that sub-ABI to ensure such a situation cannot arise. Likewise, implementors may rely on an entirely different set of mappings that are disjoint from the baseline specification. Such an ABI-island would require similar restrictions to ensure correct execution. All ABI variants are of course relative to a baseline existing in the first place.

We observed that the absence of an explicit baseline led to the definition of implicit sub-ABIs. As architecture extensions (*ie* fast new instructions) are introduced users quickly identify prospective mappings that offer performance improvements for their workloads. These sub-ABIs guide compiler development as they arise, but lacked ABI specification and testing until now. We provide a baseline ABI as guidance, firstly because mixing bugs have been introduced by accident (§5.2). Secondly, there have been numerous attempts to optimise special atomic sequences (see §6.4), motivating the need to collect these cases together. Thirdly, engineers have been asked whether the same set of prospective mappings is correct by multiple different partners, and writing down the known cases helps rule out incorrect mappings. Lastly, the collective knowledge of atomics ABIs exists as a series of online discussions and web pages [45], which are unfortunately outdated or have altogether disappeared (for instance when LLVM migrated from Phabricator to GitHub). We provide an ABI to help engineers and reduce the chances of mixing bugs arising in the future.

6.6 Future ABI Extensions

The published atomics ABI [22] contributes to an open ABI for the Arm Architecture. We hope that Arm's partners will submit requests to the ABI team[2] so that more compilers can be validated and their mappings added to the ABI (if they differ from the current ABI). Further, the ABI team may consider new mappings if future architectures introduce new atomic instructions.

7 Related Work

Neither compiler testing, memory models, nor interoperability is new. Previous work focuses on each topic in isolation whereas we combine them. We summarise the relevant work and show how mix testing improves upon them.

Compiler testing for sequential code: Testing the compilation of sequential code has found hundreds of bugs in the past [27, 53]. There are broadly two approaches: *differential testing* (see CSmith [53]) and *metamorphic testing* (see Orion [27]). In each case, techniques such as fuzzing and mutation are used to find bugs using many C/C++ features—a large test surface. Testing the compilation of concurrency samples a tiny test surface, namely the mappings from C/C++ atomics to assembly. Finding concurrency-related bugs is harder, since they require specific conditions to arise. As such prior concurrency testing work [10, 23, 41, 52] found between two and six bugs each owing to the challenges of generating good tests and observing all behaviour. Mix testing finds four concurrency-related bugs but strictly generalises concurrency testing with respect to a single compiler profile, finding bugs that prior techniques cannot. We note that sequential testing techniques target the entire surface of the C/C++ languages, while our work is focused on the small, yet critical, surface of C/C++

[2]Please submit a request on GitHub [4] or contact arm.eabi@arm.com

atomics. Mix testing could be applied more broadly to find (not necessarily concurrency-related) ABI issues between compilers, which we defer to future work. We do not do any fuzzing or mutation.

Compiler testing for concurrent code: There has been a wealth of work on testing since the advent of the C/C++ model by Boehm and Adve [9], Batty [8] and others. Ševčík led the way on a theory of sound optimisations [48] and verified compilation [49]. The `cmmtest` [41] and `validc` [10] tools are based on this theory and represent early efforts to find bugs. These tools compare the *executions* of source and compiled tests. The `C4` tool [52] simplifies testing by comparing *outcomes* of executions; this is amenable to testing as compiler engineers know it. Unfortunately, `cmmtest` and `C4` rely on hardware executions, which may not exhibit all architecturally allowed outcomes, if the hardware exhibits them at all. Further `validc` does not test compilation down to the assembly level and may miss bugs in target dependent optimisations. Geeson and Smith 2024 [23] address these issues by parametrising testing over source *and* architecture models. Téléchat is the simplest tool, comparing outcomes of source and compiled programs under their respective models.

Relaxed memory models: In the beginning, Lamport [26] coined the term *sequential consistency* (SC). A *relaxed* model is one that removes one or more constraints on the SC model. Fig. 1 uses SC accesses but is tested using relaxed models of C/C++ [11] and Armv8. Proposals of new C/C++ models provide soundness proofs and *compilation schemes* that suggest correct atomic mappings for compilers to implement (see for instance, Fig. 9 of Lahav et. al [25]). Unfortunately, such proofs are quickly outdated, since compilers implement *multiple* mappings that can change. Mix testing exposes this fact and the subsequent bugs that follow. Whilst we do not contribute any work on the models themselves, mix testing implies there may exist mixing cases in soundness proofs that have not been considered. Geeson and Smith [23] conduct large-scale differential testing of various C/C++ models, but we leave mixed implementation soundness proofs to further work.

Language interoperability: Interoperability is a long-standing concern of engineers deploying portable code. The state-of-the-art testing techniques [10, 41, 52] assume the *whole* program is compiled at once, using one set of atomics mappings. Unfortunately, this is an unrealistic view as production code bases are often compiled separately. Perconti and Ahmed [42] call this the *closed-world* assumption and contribute correctness theorems for verification purposes. Mix testing is a testing analogue of this idea that we applied to production compilers and found mixing bugs. Our work is closer to *combinatorial interaction testing* (CIT) [33] that samples the test space to reduce the explosion of possible test parameters. CIT differs in that it constructs tests by exploiting the *orthogonality* of test parameters whereas our choice of test input is coupled to each compiler profile under test. Since each atomic operation is implemented as a compiler intrinsic (whose code generation can change) that is accessed through compiler flags (that also change), we rely on expertise to pick these parameters. We document a number of practical (§3.7), theoretical (§3.6), and knowledge-based (§6.1) techniques to reduce the complexity of mix testing.

8 Conclusions and Further Work

We present the *mix testing* technique and the `atomic-mixer` tool for testing the compilation of concurrent programs that mix atomic implementations. We explore the mix testing idea (§3), technique and tool design, and problems we faced during implementation (§4) with a reproducible artifact (see the Data-Availability statement below). We define a special kind of concurrency bug: the mixing bug (Def. 3.4) and found four previously unknown mixing bugs in LLVM and GCC (§5.2). We explore the exponential complexity of mix testing (§3.6) and practical ways to reduce the test

generation (§3.7) and simulation penalty (§4.2). We found a bug missed by the state-of-the-art on the same inputs (§5.2). We expose the scale of testing the compilation of concurrency, as a problem that cannot be addressed by testing atomics mappings in isolation, but rather by testing in the presence of exponentially my many choices of mappings. This holds both now and in the future, as long as there are compatible atomics mappings in compilers.

We show how the complexity of mix testing can be reduced in practice through our industry experience, notably by publishing an atomics application binary interface (ABI - §6.1) for Armv8 AArch64 atomics implementations. As far as we know, this is the industry's first publicly documented [22] specification of an atomics ABI with an accompanying tool that finds bugs in non-compliant compilers. The ABI reduces the complexity of mix testing to a smaller (but still exponential) number of implementations to test and provides validation for Arm's partners who build against it. Whilst developing the ABI, we assisted with queries from Arm's partners (§5.2) regarding the correct mixing of atomics.

Data-Availability Statement

The software that supports this paper is available on Zenodo [21].

Acknowledgments

We thank Earl Barr, Richard Grisenthwaite, Al Grant, Kyrylo Tkachov, Tomas Matheson, Sam Ellis, Ties Stuij, Nick Gasson, Luc Maranget, Arm's Compiler Teams and Arm Architecture & Technology Group for their feedback and assistance. This work was supported by the Engineering and Physical Sciences Research Council [EP/V519625/1]. This work was also supported by EPSRC project [EP/R006865/1]. The views of the authors expressed in this paper are not endorsed by Arm or any other company mentioned.

References

[1] Jade Alglave and Luc Maranget. 2021. herdtools7. https://github.com/herd/herdtools7. Accessed: 2019-10-06.

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in Weak Memory Models (Extended Version). *Form. Methods Syst. Des.* 40, 2 (April 2012), 170–205. https://doi.org/10.1007/s10703-011-0135-z

[3] Arm-Limited. 2021. Armv8 `AArch64` Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat.

[4] Arm-Limited. 2024. https://github.com/ARM-software/abi-aa/issues. Accessed: 2024-14-02.

[5] Arm-Limited. 2024. *Arm Architecture Reference Manual.* Arm-Limited, Cambridge, UK. https://developer.arm.com/documentation/ddi0487/latest/

[6] Arvind Arvind and Jan-Willem Maessen. 2006. Memory Model = Instruction Reordering + Store Atomicity. *SIGARCH Comput. Archit. News* 34, 2 (may 2006), 29–40. https://doi.org/10.1145/1150019.1136489

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the Verification Problem for Weak Memory Models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/1706299.1706303

[8] Mark John Batty. 2014. *The C11 and C++11 Concurrency Model.* Ph. D. Dissertation. University of Cambridge.

[9] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[10] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating Optimizations of Concurrent C/C++ Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2854038.2854051

[11] Simon Colin. 2022. RC11 Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/rc11.cat.

[12] Will Deacon. 2014. arm64: atomics: fix use of acquire + release for full barrier semantics. http://lists.infradead.org/pipermail/linux-arm-kernel/2014-February/229588.html.

[13] Wilco Dijkstra. 2024. Alignment of _Atomic structs incompatible between GCC and LLVM. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=115954.

[14] Luke Geeson. 2023. [AArch64]: Atomic Exchange Allows Reordering past Acquire Fence . https://github.com/llvm/llvm-project/issues/68428.

[15] Luke Geeson. 2023. Add (__)(u)int128(_t) parsing to CType. https://github.com/herd/herdtools7/pull/520.

[16] Luke Geeson. 2023. [lib/gen]: fixed __int128 parsing. https://github.com/herd/herdtools7/pull/553.

[17] Luke Geeson. 2024. [AArch64]: 128-bit Sequentially Consistent load allows reordering before prior store when armv8 and armv8.4 implementations are Mixed. https://github.com/llvm/llvm-project/issues/81978.

[18] Luke Geeson. 2024. [Armv7-a]: Sequentially Consistent Load Allows Reordering of Prior Store when Implementations are Mixed. https://github.com/llvm/llvm-project/issues/65541#issuecomment-1709229837.

[19] Luke Geeson. 2024. [Armv7/v8 Mixing Bug]: 64-bit Sequentially Consistent Load can be Reordered before Store of RMW when v7 and v8 Implementations are Mixed. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=111416.

[20] Luke Geeson. 2024. Weak Memory Demands Model-based Compiler Testing. https://doi.org/10.48550/arXiv.2401.09474 arXiv:2401.09474 [cs.PL]

[21] Luke Geeson, James Brotherston, James Dijkstra, Alastair F. Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson. 2024. Artifact for "Mix Testing: Specifying and Testing ABI Compatibility Of C/C++ Atomics Implementations". https://doi.org/10.5281/zenodo.13625822

[22] Luke Geeson and Wilco Dijkstra. 2024. C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture. https://github.com/ARM-software/abi-aa/releases/tag/2024Q3.

[23] Luke Geeson and Lee Smith. 2024. Compiler Testing with Relaxed Memory Models. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 334–348. https://doi.org/10.1109/CGO57630.2024.10444836

[24] Open ISO-C-Std. 2022. ISO/IEC 9899:201x. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2912.pdf.

[25] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11 *(PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[26] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. *SIGPLAN Not.* 49, 6 (June 2014), 216–226. https://doi.org/10.1145/2666356.2594334

[28] Sung-Hwan Lee. 2023. A miscompilation bug in LICMPass (concurrency). https://github.com/llvm/llvm-project/issues/64188.

[29] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[30] LLVM-Phabricator. 2023. [AArch64] Codegen for FEAT_LRCPC3. https://reviews.llvm.org/D141429#inline-1378324.

[31] LLVM-Phabricator. 2023. [WoA] Use fences for sequentially consistent stores/writes. https://reviews.llvm.org/D141748.

[32] Arm Ltd. 2022. `AArch32` Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch32.cat.

[33] Robert Mandl. 1985. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM* 28, 10 (oct 1985), 1054–1058. https://doi.org/10.1145/4372.4375

[34] Luc Maranget. 2022. RISC-V Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/riscv.cat.

[35] Luc Maranget and Jade Alglave. 2022. `ARM` Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/arm.cat.

[36] Luc Maranget and Jade Alglave. 2022. IBM PowerPC Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/ppc.cat.

[37] Luc Maranget and Jade Alglave. 2023. MIPS Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/mips.cat.

[38] Luc Maranget and Jade Alglave. 2023. x86-64 Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat.

[39] Microsoft. 2024. ELF x86-64-ABI psABI. https://gitlab.com/x86-psABIs/x86-64-ABI. Accessed: 2024-02-07.

[40] Microsoft. 2024. Overview of x64 ABI conventions. https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170. Accessed: 2024-02-07.

[41] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2491956.2491967

[42] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

[43] programmerjake. 2023. x86-32 -mno-x87 64-bit atomic load miscompilation. https://github.com/llvm/llvm-project/issues/64969.

[44] Mono Project. 2024. The Mono Project, atomic source code. https://github.com/mono/mono/blob/44e6226c31d8ffcae58f81350d71a728edecfe22/mono/utils/atomic.h#L209. Accessed: 2024-02-29.
[45] Peter Sewell and Jaroslav Ševčík. 2014. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.
[46] Ole Tange. 2023. *GNU Parallel 20230222*. https://doi.org/10.5281/zenodo.7668338 GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them..
[47] Kyrylo Tkachov. 2024. Enabling the LDAPR instructions for C/C++ compilers. https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/enabling-rcpc-in-gcc-and-llvm.
[48] Jaroslav Ševčík. 2011. Safe Optimisations for Shared-memory Concurrent Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534
[49] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. https://doi.org/10.1145/2487241.2487248
[50] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models *(POPL 2017)*. ACM, New York, NY, USA, 190–204. https://doi.org/10.1145/3009837.3009838
[51] Wikipedia. 2024. Internal Compiler Errors. https://en.wikipedia.org/wiki/Compilation_error#Internal_Compiler_Errors.
[52] Matt Windsor, Alastair F. Donaldson, and John Wickerson. 2021. C4: The C Compiler Concurrency Checker. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. ACM, New York, NY, USA, 670–673. https://doi.org/10.1145/3460319.3469079
[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Received 2024-04-06; accepted 2024-08-18