FMitF: Track I: Safe, Efficient Persistent Memory Systems, 2220410

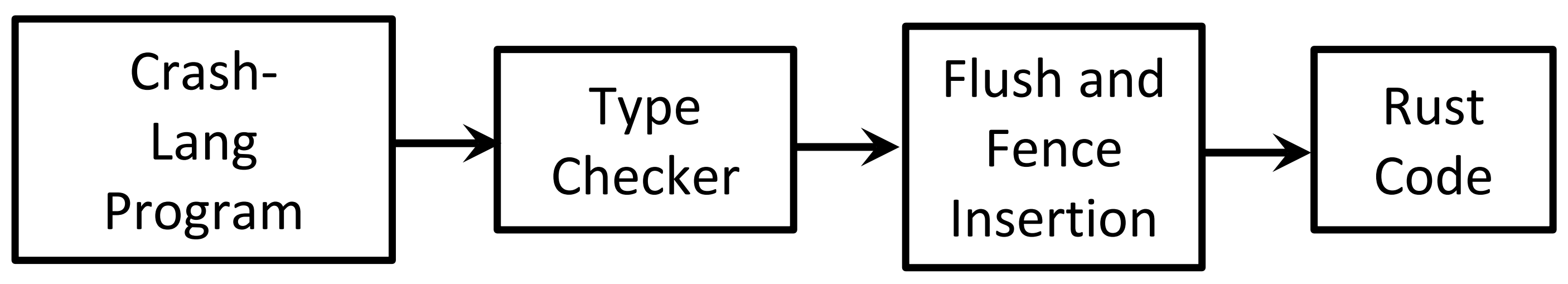
PI(s): Brian Demsky and Anton Burtsev
Institution(s): University of California, Irvine and University of Utah

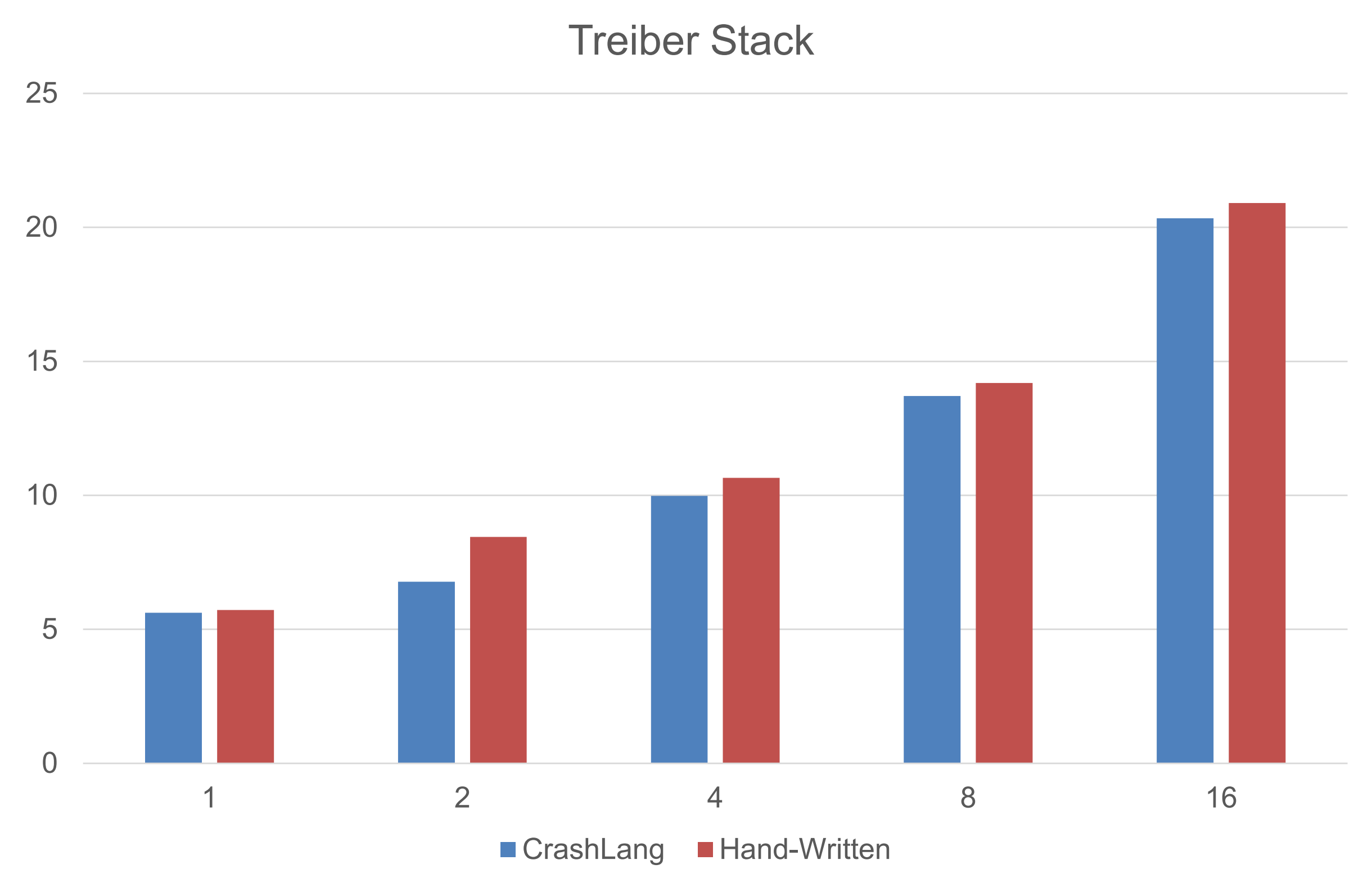
Challenge:
- Research is motivated by ensuring crash consistency of software systems that use emerging persistent memory and CXL shared memory technologies.
- Critical gap to be addressed is a lack of techniques to verify crash consistency of persistent memory software
- Transformative because the project will allow developers to have significantly more assurance that their software is crash consistent.

Objective:
Build tools to support the development of persistent memory software that is crash consistent by construction. Evaluate these tools on a range of high-performance systems to ensure that they are flexible enough to allow the construction of real-world systems.

Key Insight:
Operations on many crash consistent data structures have the following structure:
(1) Logically or physically private writes that do not change the logical state of the data structure.
(2) A commit store that logically changes the state of the data structure.
(3) A number of helping actions that change the physical state of the data structure, but not its logical state.

Use type systems to verify that code follows this structure.
Linear types + logical guards to check for private data.
Then prove that all well-typed programs are crash consistent.
Use type information to insert flush and fence operations.





```
private struct QNode {
    var elem : T;
    once next : QNode;
}
public struct Queue {
    spec head : QNode;
    help<next> tail : QNode;
    // Helping action CASes this value to tail.next.
    // Since next is a once field, the helping action
    // needs to be done when it is assigned.
}
public def enqueue :
    W (q : Queue, e : T) : void {
    let n = new Node;
    n.elem = consume e;
    tryEnqueue(q, consume n);
}
private def tryEnqueue :
    W (q : Queue, n : pristine QNode) : void {
    let oldTail = q.tail;
    if (try(oldTail.next = n))
        help q;
    } else {a
        tryEnqueue(q, consume n);
    }
}
```
Code for Treiber Stack

Formal reasoning about Rust code: Rust + Verus

Practical, low-burden reasoning about low-level systems code

- Linear types (Rust) - restricted aliasing model, no explicit annotations about heap
- Fast, automated SMT-based verification (Verus) - efficient encoding into Z3 (most proofs finish under a second)

Possibility to reason about unsafe pointer operations, avoid trust in unsafe Rust extensions and standard library.

Possibility to reason about concurrent code.

Sidestep complexity of verification, concentrate on models of the system and the hardware.

Information flow control to track unpersistent state

- Leverage Rust procedural macros to annotate code and track unpersisted state
- Use verification to reconstruct corner cases and develop persistence proof

Broader Impacts and Outreach:
. Persistent memory and CXL shared memory technologies have the potential to enable higher performance systems. Crash consistency is a major challenge in their development. The project has the potential to eliminate crash consistency bugs.
. The PIs have introduced topics related to this work into their graduate courses.

Broader Impacts:
Persistent memory and CXL shared memory has the potential to transform how programs store and process data. A major threat to this potential is crash consistency. The proposed research will develop new tools to assure safety even in the presence of crashes