Compiling Distributed System Models with PGo

Finn Hackett  
University of British Columbia  
Canada

Shayan Hosseini  
University of British Columbia  
Canada

Renato Costa  
University of British Columbia  
Canada

Matthew Do  
University of British Columbia  
Canada

Ivan Beschastnikh  
University of British Columbia  
Canada

ABSTRACT

Distributed systems are difficult to design and implement correctly. In response, both research and industry are exploring applications of formal methods to distributed systems. A key challenge in this domain is the missing link between the formal design of a system and its implementation. Today, practitioners bridge this link through manual effort.

We present a language called Modular PlusCal (MPCal) that extends PlusCal by cleanly separating the model of a system from a model of its environment. We then present a compiler tool-chain called PGo that automatically translates MPCal models to TLA+ for model checking, and that also compiles MPCal models to runnable Go code. PGo provides system designers with a new ability to model and check their designs, and then re-use their modeling efforts to mechanically extract runnable implementations of their designs.

Our evaluation shows that the PGo approach works for complex models: we model check, compile, and evaluate the performance of MPCal systems based on Raft, CRDTs, and primary-backup. Compared to previous work, PGo requires less time to develop a checked model and derive a fully working implementation. With PGo we created a formally checked Raft model and its corresponding implementation in under 1 person-month, which is 3× less time than Ivy. Our evaluation shows that a PGo-based Raft KV store with three nodes has 41% higher throughput than a Raft KV store based on Ivy, the highest performing verified Raft-based KV store from related work. A PGo-based CRDT set has a latency within 2× of a CRDT set implementation from SoundCloud called Roshi.

CCS CONCEPTS

- Software and its engineering → Formal software verification; Compilers; System modeling languages; - Computing methodologies → Distributed computing methodologies.

KEYWORDS

Formal methods, Distributed systems, Compilers, PlusCal, TLA+

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS ’23, March 25–29, 2023, Vancouver, BC, Canada  © 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.  ACM ISBN 978-1-4503-9916-6/23/03… $15.00  https://doi.org/10.1145/3575693.3575695

1 INTRODUCTION

Distributed systems remain challenging to design and to build. As a result, systems in production often contain bugs that degrade performance [26], cause service outage [28, 77] and even data loss [27]. Many tools have been developed to aid developers in building more correct distributed systems. Some tools help developers during design to verify that their logic is correct, before they write code [47, 79]. Others, such as tracing [41, 61], runtime checking [53, 54], and debugging frameworks [4, 5, 29, 48, 74, 82, 86], help explain and check runtime behavior. However, there remains a gap between models of system design and runnable code. In particular, at present, there is no easy-to-use way to translate a verified distributed system design into an implementation.

Systems such as Verdi [79] and IronFleet [37] have codified this translation into formal frameworks that help users prove the translation with theorem provers such as Coq and Dafny. This is a major success, but these frameworks require substantial effort and are too complex to be used by non-researchers.

One implication of the gap between models and implementations is that formal methods developed for models of distributed systems are limited in their impact. Another implication is that the systems community has focused on applying model checking to distributed system implementations [49, 58, 83]. Although this line of work has yielded impressive results, it is fundamentally handicapped due to the state explosion inherent to implementations.

Our goal is to translate verified distributed system models into implementations that exhibit some refinement of the modeled behavior. For this, we designed a source language called Modular PlusCal (MPCal) that is a strict superset of PlusCal [47]. To our knowledge, MPCal is the first PlusCal extension that makes it possible to derive runnable code from PlusCal models. This automation avoids the practical overhead and potential for human error in first modeling a system and then separately writing an implementation. Additionally, building a DSL on top of PlusCal with support for multithreading and non-determinism as primitives allows us to implement these more intelligently.

To translate a formal MPCal model into Go, we designed and implemented a new compiler called PGo (Figure 1). The key to PGo’s translation is MPCal’s separation between the description
We show that PGo makes it easier to build verified distributed systems. We present the design of PGo, a compiler from MPCal to runnable Go code. PGo introduces three new abstractions: (1) archetypes contain only the system definition, (2) mapping macros encapsulate environment behavior, and (3) resources describe a system’s environment dependencies. Due to the environment details being omitted during compilation, PGo also provides a Go library, PGo-distys, which offers runtime replacements for a variety of common MPCal environment abstractions.

In summary, this paper makes the following contributions:

- We introduce MPCal, a language to bridge a system model with its implementation. Building on PlusCal, which has little support for abstraction or information hiding, MPCal explicitly distinguishes the system from its environment with three new abstractions: (1) archetypes, (2) mapping macros, and (3) resources (§3).
- We present the design of PGo, a compiler from MPCal to runnable code. We show how to map PlusCal’s support for non-determinism and TLA⁺’s set-theory-first untyped semantics to the practical Go-based implementations generated by PGo (§4).
- We show that PGo makes it easier to build verified distributed systems, providing at least 3× reduction in development time for Raft as compared to Ivy [24]. PGo-based systems also have good performance. For example, on a YCSB benchmark, a PGo-based Raft KV store with three nodes has 41% higher throughput and 42% lower latency than a Raft KV store based on Ivy. Compared to etcd, which is a production grade Raft-based KV, our system has the same latency and 21% of etcd’s throughput (§7).

2 ASSUMPTIONS AND BACKGROUND

We now review the assumptions of our work, as well as the background on model checking, TLA⁺, and PlusCal.

2.1 Trusted Computing Base Assumptions

Model correctness. A developer writes a model of their system in MPCal. The overall model-focused process is shown on the left of Figure 1. The developer trusts that the MPCal→PlusCal compilation provided by PGo is correct, and that the PlusCal→TLA⁺ translation provided by the TLC toolbox is correct. Next, the user trusts the model checker (TLC) or the TLA⁺ proof system (TLAPS). TLC further requires the developer to judiciously constrain the model’s state space (which is usually unbounded), while TLAPS helps with the writing of machine-checked proofs. Note that when combining separately-checked components (discussed in §4.3), the combination is unverified, and the developer has to also trust that the components are compatible and combined correctly.

Implementation correctness. Once the model is complete, the developer uses PGo to compile the model into Go. This implementation-focused process is shown on the right side of Figure 1. The developer must trust that the MPCal→Go compilation with PGo is correct, i.e., the PGo-generated Go code conforms to the TLA⁺ spec that was checked with TLC.

As well, the developer must trust any hand-written glue Go code that they write to bootstrap the system, and all PGo’s distys libraries that are hand-written in Go. The developer must also trust the Go runtime and the underlying systems software stack when deploying their system. When considering liveness properties in an implementation context, the developer must additionally consider factors such as the Go runtime scheduler and network delivery guarantees. For example, schedulers are usually not fair in the TLA⁺ sense, so care must be taken regarding liveness assumptions in MPCal. When using relaxed resources (§4.2.1), developers must additionally trust that the performance optimizations do not introduce behaviors that are not allowed by the MPCal specification.

2.2 Background

We now review the ideas that our work builds on and use a simple lock service as a running example. In this system, there is a central lock server that manages a lock. There are several clients that request to acquire/release the lock.

Model checking determines if a model of a system satisfies a certain specification. A model checker exhaustively explores the system’s state space to determine if the specification is satisfied in every possible model execution. If an execution is found to violate a property, the model checker outputs a counterexample trace that explains how the system reached an invalid state.

Correctness specifications are written as system properties. It is common to divide these into two categories: safety and liveness properties [2, 44]. Safety asserts that bad states are not reachable from the initial state; liveness properties express that the system must eventually do something good. For example, the mutual exclusion safety property for the lock service is that no two clients can hold the lock at the same time. In this property, ClientSet is the set of all clients in the system, and stateij is i-th client’s state:

\[ \exists i, j \in \text{ClientSet} : i \neq j \land \text{state}_{ij} = \text{HasLock} \land \text{state}_{ji} = \text{HasLock} \]

TLA⁺ [46] is a declarative language for modeling systems based on TLA [45], a variant of Pnueli’s Temporal Logic [70], which uses first-order logic and set theory. A TLA⁺ model can be checked against safety and liveness properties with the TLC model checker.

A TLA⁺ model consists of several predicates that define the system’s initial state as well as the transition relation that describes
variables network = \{id \in NodeSet |-> <<>>\};

\* fair keyword specifies assumption of fair scheduling
fair process (Server = 1)

fair process (q = <<>>)

variables msg, q = <<>>;

{ 
serverLoop:
while (TRUE) {
serverReceive:
\* wait cond statement blocks the process until
\* the cond becomes true
await Len(network[self]) > 0;
msg := Head(network[self]);

network[self] := Append(network[self], GrantMsg);
serverRespond:
\* if q is empty
\* if q is not empty (/= is the not equals operator)
\* q := Append(q, msg.from);
}\ 
else {
q := Tail(q);
if (q /= <<>>) {
\* network[Head(q)] := Append(network[Head(q)], GrantMsg);
}\ 
};
}

Listing 1: Lock server specification in PlusCal.

How the system can make progress. The transitions define atomic steps in the system and a sequence of them form a system behavior. PlusCal [47] is an algorithm description language that can be compiled to TLA+ . Using PlusCal, a user specifies a system in a procedural style: different processes in a system have their behavior defined by statements, and interact using familiar control flow constructs such as if statements and while loops. Listing 1 shows a PlusCal model of a lock server, our running example. Users can compile PlusCal specs into TLA+ using a PlusCal translator. This allows them to use the TLC model checker on PlusCal specs.

A PlusCal spec consists of one or more processes. Each process runs sequentially and processes can run concurrently. The user can use synchronization mechanisms (such as await) and global variables. PlusCal requires the user to structure the processes around labels (such as serverReceive and serverRespond in Listing 1). A block of statements within a label is an atomic step in the model, similar to the “atomic block” concept from concurrent programming languages like Fortress [1] and Chapel [10]. The label notation is particular to PlusCal, combining atomicity with the C-like ability to name control flow targets that can be reached via goto. We also use the term critical sections to refer to labels. In compilation to TLA+, each PlusCal label block is translated to a TLA+ transition. Labels present the designer with a trade-off: more labels allow for more concurrency (as interleavings between labels in different processes), which may be more realistic. But, this realism comes at the cost of exponential growth of the state space.

3 MODULAR PLUSCAL

In this section we describe MPCal, a language that we designed to make implementing tools like PGo possible for the PlusCal language family. MPCal adds to PlusCal a separation between the modeled system and its environment, as well as mechanisms to bind system and environment definitions together.

Why not compile PlusCal? PlusCal mixes details of the system with the environment. This makes it impossible for PGo to detect the parts that should compile to executable code, and the parts that model environment semantics and should be ignored. For example, Listing 1 mixes the semantics of a lock server with a model of a buffered network connection.

MPCal explicitly distinguishes the system and its environment (Figure 2). At the same time, MPCal preserves PlusCal’s flexibility in capturing arbitrary environment semantics. MPCal extends PlusCal with three complementary implementation hiding primitives: (1) archetype definitions, (2) archetype instantiations, and (3) mapping macros.

Archetype definitions (left of Figure 2) are a template for processes. Unlike PlusCal processes, archetypes must be separable from the surrounding specification for compilation by PGo, so they prevent access to the specification’s global state by default. An archetype defines the system model and owns a set of local variables, which only one process (instantiated from the template) can access. An archetype may take parameters called resources, which describe the system’s environment dependencies. An archetype definition provides all the information PGo needs to generate an executable system, including dependency injection points for interfacing with its environment, such as communication mechanisms, storage, liveness detectors, etc. Archetype are discussed in §3.1.

The right side of Figure 2 refers to the system’s environment. The environment is made available to archetypes via resources that act like state variables and have a narrow read/write API. However, resources are more flexible: while they can be used to access global state variables as is idiomatic in PlusCal, their read/write operations can be re-defined to arbitrary PlusCal code via mapping macros. Mapping macros provide a way to inject different types of environment semantics into archetype definitions. This allows PGo to translate an MPCal specification into a set of model-checkable PlusCal processes (e.g., Listing 1). Mapping macros also define the interface between an archetype’s PGo-generated implementation and its runtime environment. We discuss mapping macros in §3.2 and how to combine them with archetype definitions in §3.3.

1 Our use of the term “macro” is a historic detail due to PlusCal, which uses them extensively: mapping macros are better understood as a dependency injection mechanism, which we explain in terms of plain TLA+ (§3.3).
archetype AServer(ref network[_])
variables msg, q = <<>>;
!
serverLoop:
while (TRUE) {
serverReceive:
msg := network[self];
serverRespond:
if (msg.type = LockMsg) {
  if (q /= <<>>) {
    network[msg.from] := GrantMsg;
  }
  q := Append(q, msg.from);
}
else if (msg.type = UnlockMsg) {
  q := Tail(q);
  if (q /= <<>>) {
    network[Head(q)] := GrantMsg;
  }
};
};

mapping macro ReliableFIFOLink {
read {
  await Len($variable) > 0;
  with (readMsg = Head($variable)) {
    $variable := Tail($variable);
    yield readMsg;
  }
};
write {
  yield Append($variable, $value);
}
}

Listing 2: MPCal specification corresponding to Listing 1.

3.1 The Archetype Process Abstraction

To generate code from a specification, it must be clear which parts of an MPCal specification PGo should compile. Archetype definitions form self-contained descriptions of parts of a distributed system. Their parameterization allows them to be meaningful both during verification, and when compiled into Go code.

The archetype definition starting on line 1 of Listing 2 describes a slice of the same process as in Listing 1, with all of the model checking concerns abstracted away. Instances of network, which is defined as the parameter ref network[_], replace the network semantics mixed into Listing 1 with a handle to externally-defined environment semantics. Accessing the value of network[addr] is a network receive from addr, including all necessary buffer manipulation. Similarly, assigning to network[addr] is a network send\(^2\).

3.2 Mapping macros

Mapping macros provide verification-specific, abstract details of a specification’s environment, acting as oracles of the environment’s true set of possible behaviors. These can be arbitrary PlusCal code, and their compilation is best explained as conversion to TLA\(^+\) operators. Mapping macros inject code into the read/write operations on an MPCal resource, so the two operations they expose are read and write and these take two implicit parameters each. The read takes as parameters $variable$, the underlying state variable to operate on, and $yield(_)$, a continuation into which to pass a computed value. The write operation takes parameters $variable$, again the underlying state variable to operate on, and $value$, the value being written by the caller (an archetype), which the write may transform prior to writing it to the underlying state variable using the $yield$ statement.

For example, Listing 3 gives a description in TLA\(^+\) of the mapping macro on line 23 of Listing 2, a definition of reliable FIFO network semantics. Some liberties taken for the sake of presentation aside, these two translated operators can be called from a compiled MPCal archetype during reads and writes to a resource, injecting custom behavior in each case. The read operation restricts evaluation to a situation where the underlying $variable$, assumed to be a TLA\(^+\) sequence, is non-empty. If that condition is satisfied, it uses TLA\(^+\) sequence manipulation primitives to pop the first element from the underlying state variable and pass the resulting value to the parameter $yield$. Intentionally named the same as the MPCal keyword here to highlight the equivalence, calling the $yield$ parameter should invoke the rest of the enclosing critical section which depends on the yielded value, passing control back to the TLA\(^+\) code that originally invoked the read operation. The write operation has no output, and treats $yield$ differently: it takes a $value$ to write, and uses the TLA\(^+\) Append operation to push the provided value onto the end of the underlying state variable, which is still expected to be a TLA\(^+\) sequence. The $yield$ statement in this case translates to just a state variable assignment.

For context, Listing 4 illustrates how the label defined on line 6 of Listing 2 might be translated into TLA\(^+\). The label is translated into a TLA\(^+\) operator, and the access to network[self] is wrapped in a call to the previously-defined ReliableFIFOLink_read. The assignment to msg is passed to the read operation as a continuation, to which the read operation will pass its computed value.

Mapping macros allow environment details to be provided on archetype instantiation, with minimal restriction on how they are specified or what they might do. Once an MPCal archetype instantiation is expanded into a PlusCal process definition, the result is functionally equivalent to hand-written PlusCal (i.e., the MPCal in Listing 2 behaves equivalently to the PlusCal in Listing 1).
3.3 Instantiating an Archetype Process

An archetype instantiation refers to an existing archetype definition, parameterizing it with the verification-specific information that it lacks. The result can be compiled into a PlusCal process that can be presented to TLC for model checking.

For example, the instantiation in line 39 of Listing 2 supplies the network global variable, defined on line 36, as the $\text{A}Server$ archetype’s ref network[ ] parameter. That is, in the resulting process, all references to network will refer to the global variable of the same name. The additional clause mapping network[ ] via ReliableFIFOLink indicates that the mapping macro reliableFIFOLink should be applied to those same references to network, rewriting the archetype code to insert the same network semantics as in Listing 1. The [ ] syntax indicates that we want to map indexed accesses to network. This means that indexing into network should provide access to any one of a collection of mailboxes, whose individual semantics are piecewise defined by ReliableFIFOLink.

3.4 Compiling MPCal with PGo

PGo (Figure 1) is a source-to-source compiler with two distinct targets: it compiles MPCal code to PlusCal, and also compiles the same MPCal code to Go. PGo includes three key features, which combine to integrate MPCal with both Go and PlusCal: (1) compilation from MPCal to both Go and PlusCal; (2) a Go-language representation of arbitrary TLA* data, supporting the vast majority of constructs usable with the TLC model checker; and (3) a hand-written Go framework, PGo-distsys, which provides a main application loop required by MPCal algorithms, as well as state management, operation scheduling, and multiple modes of environment interaction.

4 GENERATING GO CODE FROM MPCAL

In order to produce a runnable implementation from MPCal, PGo generates Go code that implements a combination of TLA*, PlusCal, and special MPCal semantics. We show how we deal with the unique challenges presented by the required concurrency semantics, the need for executing non-deterministic code, and TLA*’s set-theoretic data and expressions.

PGo and its libraries are part of the trusted computing base of the output system. It is important to minimize this trusted computing base. PGo’s Go code generation is therefore minimalistic by design. PGo generates largely unoptimized code and leaves many static configurations like component linking until runtime rather than pre-compiling them. This is not a fundamental design decision.

In exchange for simplicity as a compiler, the execution of an MPCal archetype relies on a separate distsys Go library with which the PGo-generated code interfaces. A significant part of the complexity lies in MPCal’s critical section semantics (§4.2), which are provided by distsys as part of a complete application loop. As a result, PGo’s generated code is a collection of individual implementations of all the labels in an MPCal model, alongside some metadata, upon which the distsys main loop will provide the necessary critical section and inter-label control flow semantics. What PGo ultimately avoids is protocol bugs, rather than lower-level bugs in I/O implementation code, so we consider it secondary that compilation, distsys, and configuration be verified.

4.1 MPCal Statements and TLA* Values

Listing 5 lists the compiled Go output for the serverReceive label in Listing 2. This is an almost direct translation of the original MPCal code.

Resources are managed at runtime by distsys, and can be accessed via the critical section’s single parameter, iface. Lines 6–9 acquire handles to the resource-local variable msg and the archetype parameter ref network[ ]. These handles can then be manipulated using methods provided by iface. At line 13, iface.Read corresponds to the indexed read from the network resource at line 7 of MPCal Listing 2. At line 17, iface.Write corresponds to the write to the local variable msg on that same line of the MPCal. At line 21, iface.Goto corresponds to the implicit jump to the immediately following critical section on line 8 of Listing 2, which is made explicit during compilation.

TLA* values. As with statements, TLA* values and their operations are almost entirely implemented using library code. Expressions are mapped to Go function calls, with, for example, a simple TLA* expression such as $2 + 3$ compiling to tla.PlusSymbol(tla.MakeNumber(2), tla.MakeNumber(3)).

Insofar as is supported by other tooling (such as TLC), TLA* values include the following data types: booleans, 32-bit integers, strings, sets, heterogeneous sequences, key-value mappings (“functions” and “records”).

TLA* only allows immutable data and state transitions between immutable system snapshots. We implemented atomic values such as booleans, integers, and strings with wrapped Go native types. For compound values, naively using common mutability-optimized data structures in an immutable context will lead to copying whenever a value needs updating, which can cause performance bugs. Instead, PGo represents compound TLA* values using Hash-Array Mapped Tries [3, 40], which supports O(1) updates via structural sharing, ensuring more predictable performance for compiled TLA* output.

1 All finitely representable data of built-in types is supported.
4.2 MPCal Concurrency Semantics

A PGo-generated system must refine MPCal’s model-level concurrency semantics, which are identical to PlusCal. For this refinement, PGo’s generated code must support more coarse-grained representations of environment actions. In contrast, previous approaches [37, 69, 79] are single-threaded and based on fine-grained instantaneous environment interactions. PGo needs to provide environment interaction mechanisms that are expressive and flexible. These mechanisms need to be capable of representing any environment interaction, which means PGo’s generated implementation must be able to interact with more abstract, coarser-grained environment definitions. Flexibility is also required to allow developers to use specialized high performance implementation techniques if needed.

The execution of a single MPCal process is a sequence of critical sections. An execution of several processes interleaves the critical sections across processes in an arbitrary way to form a total order. Within this total order, each critical section executes atomically, requiring that only system state before or after execution of a critical section is observed.

These semantics entail the following properties: (1) the successful execution of any MPCal critical section must be serializable relative to other critical sections; (2) if a critical section cannot complete, a correct runtime implementation must ensure no partial execution is observed and roll back any changes.

4.2.1 Coordinating Resource Implementations. For resources that are local to an MPCal process, there is no need to deal with concurrent accesses. It is therefore reasonable to store a cached copy of local state before the current critical section and roll back to it if needed. In general, however, MPCal resources provide access to arbitrary non-local data, either on different threads or different machines, and can follow any semantics (message passing, arbitrary shared state, etc). Maintaining concurrency semantics with shared resources requires coordination.

To coordinate an MPCal process’s interaction with its resources, distsys provides a main loop which manages the execution of a single MPCal process. While following the process’s local sequence of critical sections, the main loop uses a collection of abstract operations to implement atomicity and non-determinism. PGo explicitly supports both optimistic and pessimistic concurrency, as well as the deferred evaluation of side-effects. This need for flexibility prevents more straightforward techniques like systematic locking of shared resources.

Listing 6 lists the interface that Go-based distsys resources must satisfy. ReadValue and WriteValue represent the input and output interactions with the critical section’s environment. We discussed these alongside Listing 5, so we focus on the remaining three methods.

Listing 6: Go archetype resource interface definition.

type ArchetypeResource interface {
  Abort() error
  Commit() error
  PreCommit() error
  ReadValue() *(tla.Value, error)
  WriteValue(value tla.Value) error
}

4.2.2 Practical Example: Replicating a Lock Server. The MPCal model in Listing 2 describes a single lock server. We build on this example and build a multi-coordinator lock service that uses shared state among lock servers. For this, we define q as a shared resource and instantiate multiple lock servers. The result is Listing 7, which builds on a mostly-unchanged version of Listing 2.

Listing 7: Changes to Listing 2 to make the lock server distributed.

archetype AServer(ref network[, ref q])

variables msg;

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network</td>
<td>[id</td>
</tr>
<tr>
<td>q</td>
<td>&lt;&lt;&lt;</td>
</tr>
</tbody>
</table>

fair process (Server |in {1, 2, 3}) ==

instance AServer(ref network[, ref q])

mapping network[,] via ReliableFIFOLink;
Why is consensus among resources needed in this example? Consider the label `serverRespond` in Listing 2, which now operates on a shared queue \( q \). This critical section modifies the shared queue and sends a message through the network. Due to MPcal semantics, these two operations must occur atomically.

Now consider `serverReceive`, which performs a single network operation. Here, a resource implementing the full resource consensus API might have unnecessary overhead: even network, which conceptually represents a reliable network send/receive, would be required by consensus to confirm that the message payload can be safely received during pre-commit. This would introduce extra latency. MPcal helps to reclaim the performance loss by weakening resource semantics with extra domain knowledge. For example, the `serverReceive` critical section contains exactly one resource operation that might fail, and a relaxed resource implementation can be used in this case. This implementation can stub the pre-commit check as trivially succeeding, and instead try to robustly send messages, knowing that any failures do not require coordination.

### 4.3 Using Verified Code from MPcal

Many resource implementations that we provide are hand-written in Go. But, such resources pose a risk as they add to a system’s trusted computing base (TCB). To decrease the TCB and because resources may encapsulate complex distributed protocols (e.g., distributed mailboxes, Raft), it is important to allow resource implementations to be verified as well.

In its current state, PGo is not able to combine different MPcal specifications in a sound way: verification applies to one specification at a time. But, it is possible to link MPcal specifications manually. We explore this option as a proof of concept by using distsys to link together generated code from independent MPcal models.

We separately specified and model checked the Raft protocol [68] and an abstract distributed key-value store. Then we compiled each of these models and connected them in Go to derive a complete Raft-based key-value store, which we call PGo-RaftKV-Mod. In this case, we modeled each system’s interface with the other as a pair of input-output channels, and, for each system, we specified an abstraction of the other system’s allowable behaviors for model checking purposes.

This technique makes specifications smaller and easier to reason about. It also reduces the state space of each component model, which makes it possible to scale MPcal models beyond the conventionally viable limits of model-checked systems. In exchange, it is up to the user to ensure that the contract between the two systems remains valid. We consider this problem a target for further tooling and leave the topic for future investigation.

In §7, we compare the runtime performance of this modular key-value store with several others, including a monolithic MPcal Raft key-value store specification.

### 4.4 Fault Tolerance

Here we describe how we can build fault tolerant distributed systems using PGo. Fault tolerance is dependent on the system model and failure model. We currently support an asynchronous computing model in which nodes might fail with crash failure semantics [9] and/or network partitions might happen. We leave other models to future work. For practical usage, see the prototypes we evaluate in §7. We used these fault tolerance techniques to build the implementations discussed there.

**Modeling failure behavior.** Network faults in PGo can be expressed using a `mapping macro` with weak guarantees. For example, Listing 8 describes a faulty network link. When we send a message through this link, it might or might not deliver the message. We express this non-deterministic behavior using the `either` statement. Listing 9 describes an idiom for crash failures. Placed at the end of a critical section that might fail, this block ensures that either nothing happens, or the process jumps to an unreachable failure state. During model checking, `ExploreFail` may be set to `TRUE` to enable non-deterministic failure exploration.

At runtime, since failure would occur as a consequence of the environment, `ExploreFail` is set to `FALSE`. We also have to make sure that a failed process will not receive any network messages; we do that by adding a toggle to network links. This node failure strategy leaves the developer with full control of where they do or do not want to model failure.

**Handling failures with failure detectors.** Producing a model of failure handling from which PGo can generate a reasonable implementation can be subtle. For example, an `either` statement can be used to explore failures with model checking, but this expresses what could happen. This type of pattern would allow an implementation to spontaneously handle a failure, regardless of whether any failure was detected.
The desired behavior in an implementation is that we try the network send first and in case of timeout we execute failure handling code. However, MPCal has no notion of time. Our solution is to use \textit{failure detectors} to abstract time and prune unwanted executions. Listing \ref{listing:timeout} shows how this idiom is applied. Given an appropriate implementation of \texttt{fd}, the failure handling branch can only be taken when the \texttt{fd} resource reports the remote process as having failed. If \texttt{fd(id)} yields \texttt{FALSE}, then that branch will be rolled back and the other one can be attempted. This design allows the inclusion of practical failure checks at runtime and allows verification to be parameterized by failure detectors. Balancing concerns of model checking complexity and correctness, failure detection can be defined as anything from a random boolean, to a theoretically perfect process failure detector \cite{hackett2019mpcal}.

5 \textsc{Generating PlusCal From MPCal}

A major portion of PGo’s PlusCal generation has already been described in §3. Due to lack of space we only briefly touch on it here. PlusCal output has a one-to-one correspondence to input MPCal, except where MPCal-only directives are expanded. Two key transformations are PGo’s basic-block decomposition of MPCal, which is also used when generating Go, and PGo’s rewrite-based implementation of variable reassignment in PlusCal.

\textbf{Basic-block decomposition}. This technique simplifies PGo’s syntactic transformation design. PGo operates on a transformed version of the input MPCal called the \textit{basic-block transformation}. Almost all the code within critical sections is preserved, except that implicit control flow is made explicit via synthesized jump statements. This removes the need for PGo to reason about the contextual relationships between critical sections.

\textbf{Reassignment in PlusCal}. In PlusCal, within the same critical section, the same state variable may be assigned at most once. Since MPCal’s abstractions can hide assignments, we removed this limitation from MPCal and PGo automatically applies the necessary rewrites to ensure that its output is valid PlusCal.

6 \textsc{Implementation}

PGo is an open source project \cite{pgokdistsys}. The PGo compiler is implemented in 6,170 SLOC of Scala 2.13, and PGo-distsys is implemented in Go 1.18 (Table 1). The \textbf{PGo compiler} is implemented using standard functional techniques applied to an immutable AST. Much of the compiler pipeline is composed of AST term-rewriting passes inspired by Viper’s model \cite{viper}. Our TLA'\footnote{The MPCal along with the compiled PlusCal, TLA', and compiled and glue Go code for these systems are in our GitHub repository \cite{pgokdistsys}.} parser is implemented using Scala’s parser combinator library. This parser supports all of TLA' version 1, alongside a pragmatic subset of TLA' 2.

\textbf{PGo-distsys} implements the runtime code needed by PGo-generated code. The core MPCal support defines the abstract implementations of the resource interface and the main loop algorithm from §4.2, as well as the necessary accessor and management methods. The TLA' data model implements everything to do with manipulating TLA' values. Most of PGo’s generated expression code are calls to functions implemented by this module. The rest of the code contains resource implementations, which compiled archetypes use for input/output.

7 \textsc{Evaluation}

In our evaluation we aim to answer three questions: (1) What is the development effort of constructing MPCal specifications? (2) How does the performance of PGo-based systems compare against other verified and manually written systems? (3) What is the performance overhead of using verified code from MPCal?

7.1 \textsc{Evaluated Systems and Methodology}

Table 2 lists the seven systems we have constructed using MPCal.\footnote{The MPCal along with the compiled PlusCal, TLA', and compiled and glue Go code for these systems are in our GitHub repository \cite{pgokdistsys}.} We built Raft-based systems by following a draft of the original TLA' spec \cite{hackett2021mpcal}. PGo-RaftKV is a monolithic Raft KV store specification. Part of its specification is available in Appendix A. Distributed-KV is an abstract key-value store with no consensus component. PGo-RaftKV-Mod is a modular composition of the pure Raft protocol specification and Distributed KV as described in §4.3. PGo-PBKVK is a primary-backup key-value store where the primary synchronously replicates KV requests to backup nodes. PGo-CRDT is an add-wins observed removed set (AWORSet) state-based CRDT \cite{crdt} that uses vector clocks for merging and conflict resolution.

We evaluate the performance of several systems from Table 2 in Sections 7.4 to 7.6. We ran our experiments on Azure with each system deployed across a set of Ubuntu 20.04 Standard_B8ms VMs, using default Azure Cloud routing. We made a best effort to fully re-initialize server state between measurements, and repeated each of our benchmarking scenarios 5 times for reliability. Each scenario consisted of tens of thousands of operations, and ran for 10 minutes on average. We also inspected network interface metrics to ensure that we were not saturating any network connections. We report medians of the trials, and use whiskers on bar graphs to show the 10th and 90th percentiles.

7.2 \textsc{Development Effort}

All MPCal specifications were written by the first two authors (two Computer Science graduate students in their 1st and 2nd year). Table 2 lists their effort in person days. The most complex system we have developed is Raft and a KV store based on Raft. Table 2 also lists the number of archetypes and SLOC in each MPCal spec, and SLOC for the Go code we hand-wrote to bootstrap the generated Go implementation of each system.
Table 2: Systems we developed using PGo. Our evaluation focuses on the bolded systems: (1) PGo-RaftKV-Mod, which is a modular composition of Raft protocol and Distributed KV (see §4.3), (2) monolithic PGo-RaftKV, (3) PGo-PBKv, and (4) PGo-CRDT.

<table>
<thead>
<tr>
<th>System</th>
<th>Effort (person days)</th>
<th>Properties model checked</th>
<th>Checked # states</th>
<th>Checking time (m)</th>
<th>Archetype Count</th>
<th>MPCal SLOC</th>
<th>Glue Go SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raft protocol</td>
<td>22</td>
<td>Five Raft properties [68]</td>
<td>$2.7 \times 10^7$</td>
<td>312</td>
<td>9</td>
<td>771</td>
<td>676</td>
</tr>
<tr>
<td>Distributed KV</td>
<td>3</td>
<td>Client interaction, consistency</td>
<td>$2.6 \times 10^7$</td>
<td>4</td>
<td>3</td>
<td>256</td>
<td>383</td>
</tr>
<tr>
<td>PGo-RaftKV-Mod</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1059</td>
</tr>
<tr>
<td>PGo-RaftKV</td>
<td>25</td>
<td>Client interaction plus Raft</td>
<td>$3.1 \times 10^8$</td>
<td>404</td>
<td>7</td>
<td>758</td>
<td>1099</td>
</tr>
<tr>
<td>Lock service</td>
<td>2</td>
<td>Mutual exclusion and liveness</td>
<td>$4.6 \times 10^7$</td>
<td>73</td>
<td>2</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>PGo-PBKv</td>
<td>10</td>
<td>Strong consistency</td>
<td>$4.5 \times 10^8$</td>
<td>235</td>
<td>4</td>
<td>420</td>
<td>270</td>
</tr>
<tr>
<td>PGo-CRDT</td>
<td>10</td>
<td>Convergence and termination</td>
<td>$5.8 \times 10^8$</td>
<td>3954</td>
<td>2</td>
<td>160</td>
<td>185</td>
</tr>
</tbody>
</table>

Table 2 shows that building PGo-RaftKV required less than one person-month of effort, while building the similar system in Ivy [24] needed 3 person-months, Verdi [79] took 12 person-months, and IronFleet [37] required 18 person-months. We find our results encouraging. However, we note that all of these numbers, including ours, are anecdotal and self-reported. They are also based on researchers who are not representative of the average software developer. Future work should explore user studies to evaluate the usability of tools in this space.

While building these systems we found ourselves reusing mapping macros across systems that have identical or similar assumptions about failures or the environment. We have also developed and re-used several implementations of common resources like the network and the file system.

7.3 Model Checking Performance

Table 2 lists the properties we specified and checked, the states that the TLC checker explored, and TLC checking time. We ran our experiments on a machine with 64 CPU cores and 128 GB of RAM. TLC is an exhaustive model checker: it will cover the entire reachable state space if it is given and then terminate. This provides stronger guarantees than heuristically sampling the state space. This also means that TLC is a bounded model checker: it must be given a finite state space to explore. We therefore needed to restrict each system’s state space. For example, for the Raft protocol and PGo-RaftKV our model checking configuration assumed 3 servers, one round of election, and at most two entries committed to the log with at most one node failure. More powerful machines would allow for looser bounds and more checked states in Table 2.

While effective at finding bugs, our use of model checking provides weaker guarantees than theorem provers, which are popular in related work. Theorem provers allow verification to consider infinite state spaces using symbolic techniques, and do not require bounding a model’s state space. In exchange, theorem provers require more guidance during verification than model checkers. This guidance, however, becomes additional information that must be updated alongside the specification if changes are made. In comparison, model checking adapts more easily to changing specifications as it requires less information to begin with. For example, when we refactored PGo-RaftKV to add extra threads of execution as a performance optimization, we were able to simply re-run TLC on the new version.

We did not compare the overhead of model checking MPCal relative to PlusCal because we could not find working PlusCal/TLA+ models for the systems we considered. However, we manually reviewed the PGo-generated PlusCal for all systems, and are confident that the checking overhead is minimal.

7.4 Performance of Raft-based KV Stores

PGo-RaftKV uses TCP and BadgerDB [57, 72], an embedded KV store for durable store. We also evaluate PGo-RaftKV-Mod, our proof of concept for modular verification, with the goal of measuring the overhead of linking separately verified MPCal specifications. We compare our Raft-based KV stores against several verified KV stores: a KV store verified in Verdi [79], called Vard [80]; a KV store verified in Dafny, called IronKV [37, 38]; and a KV store verified in Ivy, that we call Ivy-Raft [24]. All these KV stores are Raft-based, except IronKV, which is based on MultiPaxos. Each implementation is extracted as OCaml, C++, and C++ respectively. All implementations interact with the underlying platform using custom shim code. These shims communicate via plain TCP or UDP. IronKV supports SSL, but its original evaluation did not use this, so we leave SSL disabled. PGo-RaftKV and Vard implement disk-based durability, whereas IronKV and Ivy-Raft do not. To see if this had an impact, we re-ran a set of benchmarks with disk-based durability disabled, and noticed a significant change in throughput.

We present benchmark results for etcd v3.5.4 [22] as a baseline. etcd is a widely used Raft-based KV store implemented in Go. We attempted to additionally evaluate Coyote [18] and StateRight [63], as they appeared to have Raft and Paxos prototypes respectively. But, we found that these prototypes were incomparable with practical consensus implementations. Coyote’s Raft prototype was confirmed by the authors to be only intended as a model checking target. Similarly, StateRight authors confirmed that their Paxos prototype was single-degree: it could only agree on a single value during an execution.

We evaluate these KV stores with the YCSB benchmark [16] and measure throughput and latency. We consider five YCSB workloads: (A) 50/50 read/update Zipfian, (B) 95/5 read/update Zipfian, (C) read-only Zipfian, (D) 95/5 read/update latest (most recently inserted records are at the head of the Zipfian distribution), (E) 50/50 read/modify-write (causally linked read/write) Zipfian.

To avoid re-implementing the YCSB codebase and workload generators in a language compatible with the related KV stores’ original client libraries, we wrote custom Java clients to interact

---

6We omit YCSB workload E as our systems do not support scans.
We compared the relationship between latency and throughput for the systems we benchmark in Figure 4. This comparison uses clusters of size 3 running workload A, and plots the median throughput and client latency curve across every number of clients used when calculating maximum throughput for Figure 3. Note that Vard has been omitted from the plot for readability: its maximum throughput was 31 op/s and its minimum latency was 738 ms. This comparison shows that overall PGo-RaftKV has 42% lower median latency than Ivy-Raft’s, which allows greater concurrency than Raft’s core specification. While PGo-RaftKV leverages more multi-threading than related work, it is still based on the original Raft TLA specification, and does not deviate significantly from the core protocol specification. Second, PGo-RaftKV uses inherently less efficient immutable data structures in its compiled TLA. In exchange for asymptotically good performance, these data structures are known to have significant overheads compared to mutable variants. We leave addressing these issues and moving our performance closer to a production-grade tool, such as etcd, to future work.

We did not need to do this for etcd, because etcd has a dedicated Java client library. We worked closely with the authors of Ivy-Raft to build and debug their artifact. We were ultimately not able to extract working C++ code from their model using any version of the C++ tool, and our measurements of Ivy-Raft’s performance rely on already-extracted C++ files from one of the Ivy author’s archives.

Two of the systems we evaluate use a variant of the YCSB workloads, using values that are only a few bytes long: both Vard and Ivy-Raft’s implementations suffer from buffer overflow issues when handling larger messages, impacting their availability. We tested the other systems with this version of the YCSB workloads, and found that this did not make a significant difference to our measurements.

Figure 3 shows the performance of PGo-RaftKV and PGo-RaftKV-Mod, alongside related work KV stores, across YCSB workloads.

All systems used 3-node clusters. We repeated the benchmarks for each workload while varying the number of concurrent clients, until we reached peak possible throughput for each system and workload, and recorded that peak number. PGo-RaftKV had the highest throughput across all workloads. It outperformed Ivy-Raft (the closest performing system) in overall mean throughput by 41%. This shows that PGo’s architecture generates more flexible implementations than related work, allowing us to precisely optimize I/O behavior and produce an efficient multi-threaded implementation. Our optimizations include dividing the MPCal model into several communicating processes, each dedicated to performing a single task, allowing more concurrent processing than related work. The multi-threading transformation required editing and recompiling the model, which we also model checked to verify that it remained correct. Additionally, we tuned timeout values and the delay between attempts at log synchronization between nodes to maximize runtime performance. This was done on the Go side, as our model does not reason about physical time.

We also observe that PGo-RaftKV-Mod has lower performance than PGo-RaftKV, with a maximum throughput a little below that of Ivy-Raft. This suggests that separating a system into two MPCal models may incur some performance overhead. The difference could also be due to us spending more time working on tuning PGo-RaftKV’s implementation.

All systems, including PGo-RaftKV, substantially under-perform the etcd baseline (not shown): etcd achieved peak throughput between 5,866 and 10,504 op/s across all workloads. We believe PGo-RaftKV’s has lower throughput than etcd for two reasons. First, etcd’s architecture allows much more concurrency in processing clients’ requests compared to PGo-RaftKV. This is due to a design difference between etcd and all the other Raft-based KV stores we evaluated: etcd implements a thread extension of Raft [66], which allows greater concurrency than Raft’s core specification. While PGo-RaftKV leverages more multi-threading than related work, it is still based on the original Raft TLA specification, and does not deviate significantly from the core protocol specification. Second, PGo-RaftKV uses inherently less efficient immutable data structures in its compiled TLA. In exchange for asymptotically good performance, these data structures are known to have significant overheads compared to mutable variants. We leave addressing these issues and moving our performance closer to a production-grade tool, such as etcd, to future work.

We compare the relationship between latency and throughput for the systems we benchmark in Figure 4. This comparison uses clusters of size 3 running workload A, and plots the median throughput and client latency curve across every number of clients used when calculating maximum throughput for Figure 3. Note that Vard has been omitted from the plot for readability: its maximum throughput was 31 op/s and its minimum latency was 738 ms. This comparison shows that overall PGo-RaftKV has 42% lower median latency than the lowest-latency related work, Ivy-Raft, and similar latency to etcd. This lower latency is likely due to our ability to generate and tune multi-threaded implementations, which are able to internally buffer data and perform tasks concurrently where possible, rather than strictly following the model’s higher-level totally-ordered semantics. Note that introducing threading to verified systems, like those based IronFleet, can require a fundamental re-design.

---

7We thank the authors of IronKV and Ivy-Raft for their assistance in working with their artifacts and reverse-engineering their original client code. We make our client code, YCSB drivers, and the Ivy-Raft C++ code available [55].
of the correctness properties and a re-writing of proofs. By contrast, PGo’s lighter-weight verification options allow for more agile performance optimization of verified distributed systems.

Figure 5 considers the scalability of each system for varying cluster sizes, using workload A. As with Figure 3, for each cluster size and system we found the number of concurrent clients that resulted in peak throughput. For consensus-based systems, it is expected that peak throughput will decrease as cluster size increases, because more coordination work is needed. This effect is shown in Figure 5. Interestingly, at cluster size of 9, PGo-RaftKV and Ivy-Raft do not have measurably different peak throughputs. This could be due to a difference in the efficiency of PGo-RaftKV’s consensus implementation as compared to Ivy-Raft. We believe that PGo-RaftKV’s use of multiple threads per node confers less of an advantage at high cluster sizes, and the relative efficiency of Ivy-Raft’s C++ implementation might begin to have an effect.

Figure 6 shows PGo-RaftKV’s fault tolerance in action during an execution of YCSB workload A with cluster size 5, plotting throughput over time. The plot shows a leader failure at about 22s. And after a timeout, the clients timeout and look for a new leader. We later kill a follower at about 41s, which has a minimal effect.

7.5 Performance of Primary-Backup KV stores
PGo-PBKV is a distributed key-value store based on the primary-backup protocol. PGo-PBKV has a primary node that synchronously replicates data to one or more backup nodes. We evaluated PGo-PBKV and compared it against Redis [73], a widely-used key-value store written in C. Redis’ replication uses a primary-backup protocol, which we ran in synchronous replication mode to better match PGo-PBKV’s behavior. We also tried to evaluate Verdi’s primary-backup system, but we confirmed with the authors that they had never written runtime glue code for it because it was only used for proof purposes. We deployed both PGo-PBKV and Redis on three machines, one primary and two backups, and used the YCSB workload A to evaluate them. The peak throughput of PGo-PBKV is 340 op/s, while Redis handled over 50,000 op/s. The poor performance of PGo-PBKV is due to a lack of protocol optimizations and tuning. In particular, PGo-PBKV does not support batching for replicating incoming requests, which requires a more complex set of correctness properties and implementation semantics than what we had time to implement.

7.6 Performance of CRDT-based Systems
We evaluated a state-based CRDT set, PGo-CRDT, and compare it against an open source CRDT set from SoundCloud called Roshi [8].

To compare these systems, we measured how long it took for all nodes’ states to converge to the same value (convergence time). In our experiment, every node executes multiple rounds. In round \( r, \) node \( n \) adds the pair \( ⟨ r, i ⟩ \) to the set and then waits until its set has all pairs of form \( ⟨ r, i ⟩ \), for every node \( i \). For each round, we measured the time from when a node updates its local set until the above condition is satisfied. We repeated this process for a total of 100 rounds. Note that in both systems the updates are applied locally, and then each node broadcasts its state every 50ms.

Figure 7 shows that Roshi has up to 2x better performance than PGo-CRDT, although PGo-CRDT scales more consistently. While smaller than the difference between PGo-RaftKV and etcd, this difference is also likely due to Roshi having more person-hours dedicated to tuning and optimization, as well as potential inefficiencies in the data structures used by PGo’s compiled output.

8 RELATED WORK
Model-checked DSLs. PGo and MPCal are similar to domain-specific languages intended for developing model-checked distributed systems. P [17, 19] provides a verifiable state machine model similar to MPCal, but with a lower-level C-like language augmented with actor-like primitives. Mace [42, 43] offers a model based on nested state machines, operating as a DSL integrated with C++. Mace lacks MPCal’s abstraction capabilities. StateRight [63] is a model checking-oriented DSL in Rust that represents distributed systems expressed as state machines, making similar tradeoffs to Mace. It offers exhaustive model checking options and benefits from Rust’s strong low-level safety guarantees. Coyote [18] acts as an implementation model checker for unmodified C# code, with an optional actor-based DSL.

Automated theorem-proving. Verdi [79] and Adore [39] provide libraries for Coq [78] and offer implementation extraction. Verdi focuses on relaxing assumptions via refinement, and Adore reduces proof effort using a protocol abstraction. EventML [71] targets Nuprl instead of Coq and uses a logic based on causal order of events. PSync [21] supports semi-automated verification and assumes a round-based program structure. Disel [75] is a Coq DSL for writing and verifying imperative specifications using a Hoare-style logic designed to allow easy composition of verified components. Chapar [51] is another Coq DSL, specialized to the specification and verification of key-value stores and their clients. IronFleet [37]
provides tools that allow developers to prove that realistic implementations refine a high-level specification in Dafny [50]. Ivy [69], DuoAI [84], DistAI [85], SWISS [36], and H [60] decrease the effort to come up with inductive invariants for verification. Note that Ivy the verifier and Ivy-Raft the KV store [24] are distinct works by different authors. Sift [59] is a proof composition methodology that relies on automated refinement. Armada [20] provides a C-like specification language for verified concurrent programs.

PGo differs from work in this category in that it does not specify how verification must be done. Model checking of TLA+ can provide practically useful levels of confidence via state space exploration, without requiring formal proofs [25, 65], though proofs of safety properties for a TLA+ model are possible via TLAPS [14]. Prior work has observed that multiple techniques are necessary for practical verification results [7], including model checking. Another key difference (discussed in §4.3), PGo’s support for modular verification is unsound: linkage of verified components is unchecked.

Note that presentation differences may hide the common TCB between PGo and these projects. In practice, we found that all systems similar to PGo must trust: the verifier, the code generator, the OS, some configuration, and some scheduling and I/O code. Maude. Maude [15] supports specifying, verifying, and generating distributed system implementations [32]. Support is limited to state machines communicating via message passing. To define environment behavior, Maude provides a sockets abstraction; it is not clear how it can provide higher level abstractions as in MPCal.

Model-checking implementations. Previous work has applied the idea of state-space exploration directly to system implementations [30, 49, 58, 64, 76, 83], as opposed to their abstractions. This work is pragmatic and overcomes difficult specification issues [23]. But, this type of model checking is limited in scalability since system implementations contain more concurrency and a larger state space than system models.

Go systems tooling. Recent work has proposed tools to check and fix Go concurrency issues [55, 56], as well as verify Go code [11, 12, 81]. This work is complementary to our own, since it can further increase a user’s confidence in the Go output from PGo.

9 CONCLUSION
In this paper we bridge the gap between distributed system models and their implementations via compilation (in contrast to formal verification or program synthesis). We presented the design of the MPCal language and the PGo compiler tool-chain to compile MPCal models to TLA+ for model checking, and to running Go code. Our evaluation shows that PGo is capable of building complex distributed systems, such as a Raft-based key-value store. The resulting systems perform at least 40% better than verified systems from related work and also take at least 3x less time to construct.

In our future work we hope to decrease the TCB of the PGo tool-chain, by improving the modular verification workflow and exploring compiler-assisted runtime verification.

Ultimately, we believe that a compiler will encourage developers to specify their systems, since it will derive the majority of an implementation for free. A compiler will also help researchers to focus their efforts on more broadly applicable techniques.

DATA-AVAILABILITY STATEMENT
We make public a snapshot of our raw evaluation results and tooling [31], and our compiler [34]. See Appendix B for more details.

ACKNOWLEDGMENTS
We thank Brandon Zhang, Yi Fan (Bob) Yang, Adam Geller, Ruben Palrecha, Tannis Ye, Shizuko Akamoto, and Zack Granman for their contributions to PGo at different stages over the years. We would also like to thank the anonymous referees of ASPLOS’23 and OSDI’22 and our shepherd for invaluable suggestions. This research is supported in part by the NSERC Discovery Grants (RGPIN-2020-05203, RGPIN-2014-04870), by AWS through an Amazon Research Award, by an Azure Education Grant, and by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

A PGO-RAFTKV SPECIFICATION APPENDIX

Figure 8: Partial architecture of PGo-RaftKV. Arrows show the interaction between archetypes and mapping macros. The direction of each read/write arrow denotes the direction of data flow.

This appendix contains a selection of our PGo-RaftKV specification. PGo-RaftKV consists of servers and clients. Figure 8 shows a partial architecture of PGo-RaftKV, including some of its archetypes and mapping macros. A server has several archetypes that run concurrently. AServerHandler relays incoming messages to other components. The remaining components are named after the aspects of the Raft protocol to which they correspond.

The user-facing archetype AClient relays input requests from a channel (reqCh) to instances of AServer via ReliableFIFOLink. It passes relevant responses back to the user via the channel (respCh).

We include some of our MPCal definitions on the next page.
B ARTIFACT APPENDIX

B.1 Abstract
Our artifact has two components. We provide the PGo compiler itself, which can compile MPCal specifications, and we also provide a method for reproducing our performance results from §7 (except for Figure 6). The PGo compiler is available at https://github.com/DistCompiler/pgo [34], and can be used to compile MPCal specifications. Tools for reproducing our performance results are available at https://github.com/DistCompiler/pgo-artifact [31]. We describe how to set up both of these.

B.2 Artifact Check-list (Meta-information)

B.2.1 The PGo Compiler.
- Compilation: Scala 2.13, Go 1.18, and sbt 1.6.2 as build system.
- Experiments: our unit tests can be run, and should pass.
- How much disk space required (approximately)?: 200MB.
- How much time is needed to prepare workflow (approximately)?: 15 minutes.
- Publicly available?: yes.
- Code licenses (if publicly available)?: Apache-2.0
- Workflow framework used?: no.
- Archived (provide DOI)?: https://doi.org/10.5281/zenodo.7430244

B.2.2 Performance Results.
- Program: our fork of the YCSB benchmarking suite [16] with additional backends, included.
- Binary: our included binaries in the image/ folder are Linux-specific and tested on Ubuntu 20.04.
- Run-time environment: each of our benchmarking machines is assumed to run Ubuntu 20.04.
- Hardware: up to 15 machines networked together. We originally used Standard_B8ms VMs provisioned on Microsoft Azure with default network routing, but any fleet of VMs or bare metal machines with 8 CPUs with 32GB of RAM and a fully connected network topology should be appropriate.
- Run-time state: experiments assume uncontended network and CPU.
- Metrics: latency, throughput.
- Output: live output is console, which will be stored in the results/ folder. We provide a post-processor that can translate the folders of console output generated by our experiments into CSV format for processing via our Jupyter notebook. We provide the results we feature in the paper in the results_paper/ folder. While each individual benchmark execution will take 10 minutes or less, budget multiple days to rederive all data points we include in our paper.
- Experiments: we provide an automated runner for running our experiments, and a Jupyter notebook containing our data processing steps. We include a complete set of configuration files and automation scripts for our tools, and we provide instructions on how to customize this configuration to account for different situations.
- How much disk space required (approximately)?: 1-2GB for the main folder. Deployed VMs will consume extra disk space.
- How much time is needed to prepare workflow (approximately)?: 5 hours to set up Azure or Vagrant; budget multiple days if attempting bare metal deployment
- How much time is needed to complete experiments (approximately)?: budget multiple days to rederive all data points we include in the paper. Some systems are flaky, which may require monitoring and restarting the benchmarking process. Our runner will start over at the last valid result if interrupted.

B.3 How to Download and Run the PGo Compiler
To just use the PGo compiler, we provide these instructions. For more information, consult the project’s README.md. To reproduce our experimental results, see the next section. First, clone the git repository at branch asplos23, and enter the created folder:

```bash
$ git clone --branch asplos23
-- https://github.com/DistCompiler/pgo
$ cd pgo
```

To build PGo, you will need to install the sbt build tool 1.6.2 (https://www.scala-sbt.org/), and Go 1.18 or later (https://go.dev/).

Once the dependencies are installed, you can build and run PGo via the sbt command. All other dependencies will be downloaded automatically by the build process. Build PGo and run its sanity tests:

```bash
$ sbt test
```

The systems we have build in MPCal are available in systems/. Enter any subfolder, and the Makefile will contain verification and compilation commands relevant to that system. For example, run the model checker on one of the MPCal models:

```bash
$ make mc
```

Note that we provide pre-generated code from PGo for each system for ease of use. To regenerate these files, re-run PGo on the .tla files. This applies both to generated Go and TLA* code.

For more information, consult the repository’s README.md.

B.4 Description
The following is exclusively about the tools necessary to reproduce our evaluation, available at https://github.com/DistCompiler/pgo-artifact. A version of this appendix is reproduced in the README.md file, including additional detail where noted. Our benchmark runner is a pre-compiled collection of JAR files runnable on Linux. Its source code is provided in the azbench/ folder.

The benchmark runner machines, which will be controlled by the benchmark runner, require that a large number of dependencies be installed, as described in image/provision.sh. We document this process further below.

B.4.1 How to Access. Clone the repository as shown below, recursing over submodules. Some but not all dependencies are included as submodules.

```bash
$ git clone --recursive-submodules
← https://github.com/DistCompiler/pgo-artifact
```

B.4.2 Hardware Dependencies. Our original experiments were run on Microsoft Azure VMs. We provide an automated workflow for recreating our setup, as long as you have a locally logged-in Microsoft Azure account with $600 USD available.

Given that requiring Azure credits is not ideal, we also support two other modes of operation: creating local VMs via Vagrant, or provisioning machines by hand. Local VMs are not expected to produce meaningful results.
B.4.3 Software Dependencies. On the machine that will be used to collect experimental results, our experiment runner’s software dependencies are just a working installation of Java 11+. To run our data processing, we additionally require Jupyter with ipykernel 6.13.0 or compatible, as well as pandas 1.3.5, matplotlib 3.5.1, and numpy 1.21.5. For our Vagrant-based provisioning solution, Vagrant 2.2.16 or compatible is required to run the provided Vagrant files. For our Azure-based provisioning solution, Azure CLI 2.41.0 or compatible is required to log into the Azure account that you will use with our provider.

All other dependencies must be installed on remote machines that our benchmark runner controls. Our provisioning script image/provision.sh should be considered authoritative for versions and build steps. The script expects an Ubuntu 20.04 environment with the current directory set to a copy of the image/ directory at ~/image/. Note that the remote machine workloads are incompatible with Java 16+ due to a deprecated feature used by the Java YCSB implementation.

B.5 Installation

Our installation process has three variations, each of which has a tradeoff in terms of faithfulness to our original setup, ease of use, and financial investment. In all cases, the included benchmark runner ./azurebench will run all the experiments listed in experiments.json and deposit the results in results/. Consult the tool’s --help for information on tunable values. Note that --settling-delay 20 is necessary to run the Ivy-Raft benchmark, as that system takes some time to successfully elect a leader.

B.5.1 Manage Machines with Vagrant. This is the easiest solution to setup, as it launches all the required servers as VMs on the local machine with Vagrant. It is unlikely to produce useful results, but it is an easy way to see that the experiments can be run at all.

Complete setup instructions for this configuration are provided in the artifact’s README.md. Ensure the Vagrant VMs described in vagrant_fleet/Vagrantfile are running with the custom box we describe, and the correct static_server_map.json is present.

Once this is done, the following command will run some simple experiments on those VMs:

$ ./azurebench --settling-delay 20 .

B.5.2 Manage Machines with Azure. Given the funds, the most accurate method to reproduce our results is to run experiments on Microsoft Azure servers.

To do this, install Azure CLI \(^8\) and log in using the account and tenant to which you intend to charge experiments. Note down your tenant ID and your subscription ID.

Once this is done, launching the provisioning and experiment running process can be done with this command:

$ ./azurebench --settling-delay 20 --azure-subscription <subscription ID> --azure-tenant-id <tenant ID>

See our artifact’s README.md for additional notes on managing this process.

B.5.3 Manage Machines Manually. See our artifact’s README.md file for information on how to do this.

---

B.6 Experiment Workflow

Each experiment run by ./azurebench will be recorded in a sub-folder of results/. This includes logs containing the outputs from all the SSH sessions used. If a run was successful, results.txt will exist and contain the results as human-readable text. If a run was unsuccessful or interrupted, results.txt will not exist and the other files will indicate what happened.

Which experiments occur is controlled by the experiments.json file, which lists configuration values and shell commands to execute when running experiments. The serverCount key indicates how many servers each experiment requires, excluding one additional client machine. All script dependencies for experiments exist in the image/ folder, so a script invoked by the name foo can be inspected by reading image/foo.

On checkout, the initial contents of experiments.json is a copy of experiments_simple.json. This is a small workload designed to ensure all kinds of experiment can be performed. The true set of experiments from the paper, including our machine-dependent tuning values, is in experiments_full.json. Copying that over to experiments.json will cause all experiments from the paper to be run in full.

Note that for results describing peak throughput (Figures 3 and 5), our configuration lists the values at which we measured peak throughput on our machines. Results are known to vary even across different Azure VMs of the same type. To recreate meaningful results, we recommend splitting the experiments into two passes: varying only the number of client threads, then varying workload and cluster size. This initial set of experiments is in experiments_tuning.json. The number of client threads that causes the highest throughput should then be edited into key threadCount of the template experiments_tuned.json, which will gather data that depends on peak throughput.

B.7 Evaluation and Expected Results

To run our full set of experiments, run the following commands:

$ cp experiments_full.json experiments.json
$ ./azurebench --settling-delay 20 . # specify Azure IDs if needed

Once complete, graphs-python.ipynb can be used to parse data from results/ and recreate each of the performance graphs from this paper. Which cell corresponds to which figure is annotated in the comments.

Our existing data set is included under the name results_paper/. To test that the notebook is set up properly, you can copy that data over to results/ and see the same graphs from the paper regenerated.

We expect a recreation of our results to preserve the relationships between artifact performance numbers, but not the numbers themselves.

B.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

---

\(^8\) https://learn.microsoft.com/en-us/cli/azure/install-azure-cli


[80] James R. Wilcox, Doug Woes, Pavel Panchevka, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2022. verdi-raft commit at which Vard implementation was evaluated. https://github.com/verdi-raft/tree/ea99a2453c3b0bc31b49d36a3b456f2ad23babe1.


