A Language-based Serverless Function Accelerator

Emily Herbert
emilyherbert@cs.umass.edu
University of Massachusetts Amherst

Arjun Guha
a.guha@northeastern.edu
Northeastern University

Abstract

Building scalable and fault-tolerant cloud services typically requires expertise in systems and networking. In *serverless computing*, the cloud provider manages the OS, load balancing, failure recovery, and other low-level details, freeing the programmer to focus on application code. In return, the program (known as a *serverless function*) must be written in a supported programming language, and adhere to some simple rules.

This paper presents *Decontainerization*, which is an approach that transparently lowers the latency and resource utilization of a large class of serverless functions by using language-based sandboxing. In Decontainerization, we 1) use a source-to-source compiler to instrument serverless functions to produce interprocedural, execution trace trees; 2) compile trace trees to a safe subset of Rust; and 3) processes requests without dispatching to a virtual machine or container. This approach seamlessly transitions between language-based and OS-based sandboxing by leveraging the fact that serverless functions must tolerate re-execution for fault tolerance.

We implement Decontainerization in a new serverless platform that we call Breakout. Our experiments show that Decontainerization can significantly decrease the latency and resource utilization of serverless functions, e.g., decreasing response latency by of I/O bound functions by 3.28x. We also show that tracing has negligible cost, and that switching between two sandboxing modes is seamless.

1 Introduction

Cloud computing allows programmers to instantly rent computing resources and thus makes it possible to build web services that are fault-tolerant and can scale to meet rapidly changing demand. However, cloud computing also requires significant programmer expertise to use effectively. *Serverless computing* is a recent approach to cloud computing that makes it significantly easier for programmers to use the cloud. In serverless computing, the programmer writes a *serverless function*, which is typically a small web server, and uploads it to the cloud.[1] The cloud makes the serverless function available at a generated URL, and fully manages (virtualized) operating system, load-balancing, and auto-scaling for the programmer. In particular, the cloud transparently starts and stops concurrent instances of the serverless function as demand rises and falls. Moreover, the cloud terminates all instances if they are idle for an extended period of time. Due to this design, the programmer only pays a serverless function when it is actively processing a request. This is in contrast to traditional cloud computing, where virtual machines are billable even when they are idle.

A serverless function must satisfy several properties to work correctly. (1) It must be idempotent, which allows the cloud to safely re-send requests when it detects a possible failure [28]; (2) It must respond to every request in a few minutes, which allows the cloud to schedule the execution of serverless functions while minimizing the need for idle hardware [7]; and (3) It must use external storage for all persistent state, which allows the cloud to evict a running serverless function at any time. There is ongoing work to relax these requirements, so that a broader class of programs can leverage serverless computing (§6). However, many web applications and mobile app backends lend themselves to the serverless computing model, and thus serverless computing has grown rapidly in the last few years [13]. Serverless computing is now available from all major cloud providers, and there are several open source serverless computing platforms that can be deployed in private clouds [6, 7, 15, 25, 35].

Serverless computing uses fairly traditional cloud computing mechanisms under the hood. For example, most platforms use Node/V8 to run serverless functions written in JavaScript, and use virtual machines or containers to isolate (untrusted) serverless functions from the (trusted) cloud platform. Unfortunately, these mechanisms have some unavoidable costs. For example, request and response must be copied in and out of the isolated function, and it takes time a non-negligible amount of time to create a new container or virtual machine [7, 42].

In this paper, we present *Decontainerization*, a new approach to running serverless functions that uses language-based techniques to significantly improve performance and lower resource utilization. Decontainerization works as follows. (1) We instrument the source code of a serverless function to dynamically generate an *inter-procedural execution*

Conference'17, July 2017, Washington, DC, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Note that "serverless function" is a bit of a misnomer, since it is a complete program that may consist of several functions.

trace tree with callbacks. This is related to tracing JIT compilers, but the "hot path" in a serverless function includes asynchronous events, thus we have to develop this capability. (2) After receiving a few requests and growing the trace tree, we compile it to a safe fragment of Rust. (We insert runtime checks for certain safety properties that Rust does not provide.) (3) When compilation completes, we link the compiled trace tree into the serverless platform itself, and handle subsequent requests directly in the platform.

Decontainerization does not always succeed. Certain programming language features, such as `eval`, are not amenable to tracing. More significantly, dynamically generated execution traces may not cover all possible control-flow paths. When we receive a request that drives the compiled trace to an unseen path, we simply abort the fast-path (Rust), and retry execution in the slow-path (Node/V8 running in a container). This naive approach works for serverless functions, because they are already designed to be idempotent for fault tolerance.

We implement Decontainerization in a serverless platform that we call Breakout. Breakout supports serverless functions written in JavaScript, is built primarily in Rust, and runs as a collection of services hosted on Kubernetes [32]. Kubernetes is the basis of several other serverless platforms [6, 7, 15, 18, 20, 25, 31], we use it to run serverless functions and other components on a cluster of machines. We evaluate Breakout with a suite of six typical serverless functions .

Contributions In summary, our contributions are:

1. We show that it is possible to transparently accelerate serverless functions using language-based techniques, by exploiting the fact that serverless functions are idempotent and have transient in-memory state.
2. We present a source-to-source compiler and runtime system that instruments JavaScript code and dynamically generates an inter-procedural execution trace tree. A unique feature of our approach to tracing is that it includes asynchronous callbacks. In addition, our approach to source-level tracing uses a runtime system that grows the trace using zipper-like operations [26].
3. We present a compiler that translates trace trees to a safe subset of Rust, which minimizes the amount of new code that the serverless platform has to trust. This, combined with tracing, composes Decontainerization.
4. We implement Decontainerization in Breakout and evaluate it on six canonical serverless functions. We show that it can decrease response latency of serverless functions by 3.28x, can reduce CPU utilization by a factor of 0.20x (geometric mean), and may help alleviate the cold start problem.

The rest of this paper is organized as follows. §2 introduces serverless computing and the design of Breakout, §3 presents how we generate traces from JavaScript, §4 presents the trace-to-Rust compiler, §5 evaluates Breakout, §6 discusses related work, and §7 concludes.

Artifact Breakout, its benchmarking suite, and a Haskell model of our tracing semantics (§3), are publicly available and open-source. (Link removed for double blind review.)

2 Overview

In this section we introduce the serverless programming model using the Breakout API. We then discuss the design of traditional, container-based serverless platforms, and the extensions that Breakout makes.

2.1 Programming with Breakout

Figure 1a shows an example of a serverless function, written with Breakout, that authenticates users. The code is written in JavaScript and uses the Breakout API. The global `main` function is the entrypoint, and it receives a web request carrying a username and password (`req`). The function then fetches a dictionary of known users and their passwords from cloud storage (`resp`), validates the received username and password, and then responds with `'ok'` or `'error'`. This serverless function resembles a simple web server, but does choose a listening port, or decode the request. The serverless platform manages these low-level details.

The function illustrates an important detail: JavaScript does not support blocking I/O. Therefore, all I/O operations take a callback function and return immediately. For example, the `b.get` function takes two arguments: a URL to get, and a callback function that eventually receives the response. Therefore, the `main` function is also asynchronous. To return a response, the serverless function calls `b.respond` within a callback. All JavaScript-based serverless programming platforms have similar APIs that either use callbacks or promises.

2.2 Serverless Platform Design

Although Breakout has several unique features to support Decontainerization, its design and implementation is similar to other serverless platforms, and we first describe this basic design. Breakout has three primary kinds of components.

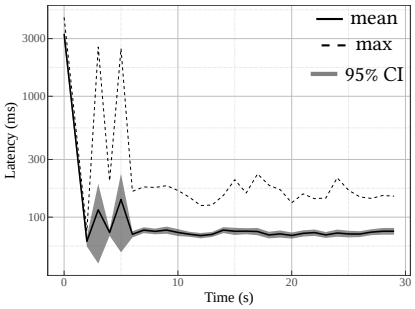
The *Controller* has an HTTP API for deploying and managing serverless functions, and assigns URLs to newly deployed functions. Thus the serverless function programmer interacts directly with the Controller to deploy their code.

The *Function Runner* has a small web server written in JavaScript. When created, the Function Runner receives the name of a serverless function f as an argument. The runner immediately downloads the source code of f, dynamically loads it, and then marks itself as ready to receive requests. A Function Runner runs in a container, which allows us to limit the CPU and memory utilized by the untrusted function f, also limit its disk and network access.

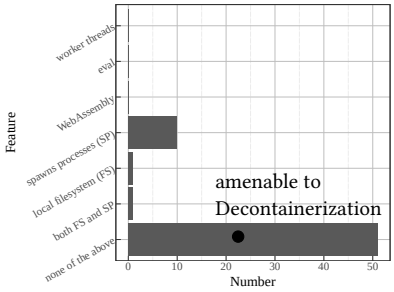
```
let b = require('breakout');

function main(req) {
  function F(resp) {
    let u = req.body.username;
    let p = req.body.password;
    if (resp[u] === p) {
      b.respond('ok');
    } else {
      b.respond('error');
    }
  }
  b.get('passwords.json', F);
}
```

(a) A simple serverless function.



(b) Sending requests to a Cloud Function.



(c) Breakdown of the features found in example serverless functions.

Figure 1. 1a is a serverless function to authenticate users against a remote list of passwords. 1b plots response latency over time, and illustrates cold starts vs warm starts. 1c classifies a repository of serverless functions, identifying functions that are and are not amenable to Decontainerization.

The *Dispatcher* is the most sophisticated component. Its primary role is to receive web requests for a serverless function and dispatch the request to an available Function Runner. However, when a Function Runner is not available, the Dispatcher is tasked with creating a new Function Runner (up to a configurable maximum), and with queuing pending requests until a runner is available. Conversely, when demand for a function falls, the Dispatcher destroys Function Runners to reclaim resources. Finally, the Dispatcher is also responsible for destroying Function Runners that exceed their time limit.

The aforementioned components run as a distributed system. Each component runs in a container and we use Kubernetes, which is a widely-used container orchestration system, to facilitate deploying components to multiple machines, replicating components for fault tolerance and scalability, and running untrusted components in isolation.

2.3 Requirements Imposed on Serverless Functions

Breakout, and other serverless platforms, impose several requirements on serverless functions, which arise due to the design of serverless platforms.

First, the programmer must assume that *in-memory state is transient*, since the platform may shutdown the function at any time to reclaim resources, or because a failure occurs. Serverless platforms also have *transient local disks*, thus all persistent state must be stored on local disks. In some serverless platforms, the local disk is read-only, and others do not allow any file I/O (e.g., [1, 12, 17]). Breakout adopts the latter approach for simplicity.

Second, the program must be able to tolerate *significant variation in request processing time*, because processing a request on new Function Runner (*cold start*) is significantly slower than processing a request on an available Function Runner (*warm start*). Figure 1b shows the latency observed while sending series of requests to a function hosted on Google Cloud Platform. The effects of cold start can be observed through an initial 5 seconds after the first request. Unfortunately, cold starts are unavoidable, since it is not cost-effective for the platform to always have idle Function Runners for every function.

Finally, the program must respond to every request before a *hard timeout* elapses (a few minutes on current platforms). This allows the serverless platform to better allocate resources. If a programmer needs to perform a lengthier computation, they need to break it up into smaller functions, or use other abstractions [7].

2.4 The Design of Breakout

Breakout modifies the components of a serverless platform to support Decontainerization as follows.

We modify the Function Runner to optionally instrument the serverless function to build an inter-procedural execution trace tree with callbacks (§3). It has a source-to-source compiler that instruments JavaScript code to build its own traces, and a small runtime system that facilitates tracing. For each serverless function f, we ensure that at most one Function Runner is building a trace tree, whereas all other function runners for f are executing the function normally.

We modify the Controller so that it can receive traces and dynamically compile them to a safe fragment of Rust, and insert additional dynamic checks for properties that the Rust type system does not ensure (§4). When trace compilation compiles, the Controller updates the modified Dispatcher, described below.

We modify the Dispatcher to either forward requests to function runners (as before), or process requests itself, using compiled traces. It runs each function f in one of three modes. 1) When first deployed, there is no trace available, and f is in *tracing mode*: it configures the first Function Runner for f to build a trace tree, whereas any other Function Runners run f normally. 2) After receiving a (configurable)

number of requests for f, the Dispatcher extracts its trace and sends it to the Controller to compile, shuts down the tracing function runner, and switches to *ordinary mode*. In this mode, the Dispatcher behaves like an ordinary serverless function Dispatcher for the function f. 3) After the Controller successfully compiles f, it updates the Dispatcher to link to the generated code, at which point the Dispatcher runs f in *decontainerized mode*: requests are processed on the Dispatcher itself. Switching to this mode does not immediately shutdown existing Function Runners. Instead, we allow the Dispatcher to terminate them after a period of inactivity, which is how they are normally terminated.

However, it is possible for the trace to be incomplete. When a trace receives a request that it cannot handle, it produces a special exception (☠) and the Dispatcher resends the request to a Function Runner. When this occurs, we can either switch back to *tracing mode* and hope to build a better trace, or give up on tracing and enter *ordinary mode*. Since tracing is not complete in general, it is possible for a function to "bounce" between tracing and decontainerized mode. To avoid this behavior, we limit the number of times this can occur, before switching to ordinary mode.

When Does Decontainerization Work? For a function to enter and remain in *decontainerized mode*, it must satisfy two criteria. First, we restrict the running time and memory utilization of each function to a few seconds and a few MB per request. (The exact limits are configurable, but higher limits degrade performance.) However, these limits are reasonable. A study of serverless workloads on Azure found that 50% of all serverless functions process events in less than one second (on average), and consume less than 170 MB of memory [42]. This is to be expected, because serverless functions often respond to events triggered by end-users of interactive systems.

In addition, we do not support serverless functions that use the local filesystem, spawn processes, use **`eval`**, use worker threads, or embed WebAssembly. Although some existing serverless platforms impose some of these restrictions (e.g., no processes, threads, or file system [12, 17]), there are more permissive platforms that do allow these behaviors. We wanted to understand the extent to which serverless functions written for these platforms actually use these features. We looked at a repository of 63 example JavaScript serverless functions, written for a variety of permissive serverless platforms [41]. Figure 1c shows the high-level breakdown of the functions and Appendix E contains a more detailed breakdown of the serverless examples repository. We found that 51 of the 63 functions would be amenable to Decontainerization.

3 From JavaScript to Dynamic Trace Trees

This section presents how the Function Runner turns a serverless function into a dynamically generated trace tree. The goal of tracing is to produce a trace tree such that on any input, the trace either 1) exhibits the same behavior as the original JavaScript program, or 2) aborts with an error that indicates unknown behavior.

Operators

op	::=	`+` | `-` | `*` | ⋯	

Expressions

e	::=	c	Constant
	|	x	Variable
	|	e_1 *op* e_2	Binary operation

Binding Forms

b	::=	e	Expression
	|	**`function`**`(`$x_1 \cdots x_n$`)` *blk*	Abstraction
	|	f`(`$e_1 \cdots e_n$`)`	Application

Block

blk	::=	`{` $s_1 \cdots s_n$ `}`

Statements

s	::=	**`let`** x `=` b`;`	Binding
	|	*blk*	Block
	|	**`if`** `(`e`)` s_1 **`else`** s_2	Conditional
	|	**`while`** `(`e`)` s	Loop
	|	x `=` b`;`	Assignment
	|	ℓ`:` s	Label
	|	**`break`** ℓ`;`	Break
	|	**`return`** e`;`	Return

Figure 2. The fragment of JavaScript that we use to present tracing. Breakout supports many other JavaScript features.

We present our approach using a fragment of JavaScript (Figure 2), which features first-class functions, assignable variables, conditionals, while loops, and structured jumps. We require all function definitions and applications to be named (similar to A Normal Form [19]). For features that are not in this fragment, Breakout uses two approaches. 1) The implementation natively supports a variety of features including objects (with prototype inheritance), arrays, and all JavaScript operators. These features do not affect the control-flow of a program, thus trace generation is routine. 2) Breakout supports many more features by translating them into equivalent features, e.g., all loops become `while` loops and all branches become `if`s.

Breakout does not support getters and setters, new meta-programming features such as object proxies, and `eval`. Since `eval` allows dynamically loading new code, it is at odds with our approach. If a program were to use `eval`, Breakout would abort tracing and fall back to using containers. However, we believe we could support the other features with more engineering effort. become `if`s. Breakout does not support getters and setters or new meta-programming features, although we believe support could be added with more engineering effort.

3.1 The Language of Traces

Breakout's generates trace trees as programs in a *trace language* (Figure 3). Many features of the trace language

Set of traced event-handlers

$T ::= n \rightharpoonup h$

Events

$ev ::=$ `'listen'` | `'get'` | `'post'` | $\cdots$

Event Handler

$h ::=$ **`handler`**`(`**`envId`**`:`x`,` **`argId`**`:`x`,` **`body`**`:`t`)`

l-values

$tlv ::= x$		Variable
	$*t\,.\,x$	Variable in environment
Addresses		
a	$::= t\,.\,x$	Address in environment
	$\mid \&x$	Address of variable

Blocks

$tblk ::= \{\ t_1 \cdots t_n \}$

Trace trees

$t ::= c$		Constant
	$\mid x$	Variable
	$\mid t_1\ op\ t_2$	Binary operation
	$\mid tblk$	Block
	$\mid$ **`if`** `(`t_1`)` t_2 **`else`** t_3	Conditionals
	$\mid$ **`while`** `(`t_1`)` $tblk$	Loops
	$\mid$ **`let`** x `=` t`;`	Variable declaration
	$\mid tlv$ `=` t`;`	Assignment
	$\mid \ell : t$	Labeled trace
	$\mid$ ☠	Unknown behavior
	$\mid$ **`break`** ℓ t`;`	Break with value
	$\mid$ **`event`**`(`ev`,` t_{arg}`,` t_{env}`,` n`)`	Event handler
	$\mid$ **`respond`**`(`t`)`	Response
	$\mid$ **`env`**`(`$x_1 : a_1$`,` $\cdots$`,` $x_n : a_n$`)`	Environment object
	$\mid *t\,.\,x$	Value in environment

Figure 3. The language of traces, most of which corresponds to JavaScript without functions. The boxed portions do not have JavaScript counterparts.

correspond directly to JavaScript. However, the trace language lacks user-defined functions, as they get eliminated by tracing. The language also includes several kinds of expressions that do not correspond to JavaScript—boxed in Figure 3–which we describe below. In this paper, we write JavaScript in `blue`, traces in `red`, and Rust in `orange`.

Unknown behavior Since the generated trace may not cover all possible code-paths in the serverless function, the language includes an expression that indicates unknown behavior (☠). Evaluating this expression aborts the language-based sandbox and restarts execution in a container.

Unified statements and expressions The trace language unifies expressions and statements. In addition, it unifies JavaScript's **`break`** and **`return`** statements into a single expression that breaks to a label and returns a value (**`break`** ℓ t). These choices make interprocedural tracing significantly easier, and because Rust has a similar design, they do not code generation harder.

Environment representation When several JavaScript functions close over a shared variable, their closures hold aliases to the same memory location. Although the trace language does not have first-class functions, it must correctly preserve this form of aliasing. Therefore, the language includes explicit environment objects (**`env`**), expressions to read a value from an environment ($*t\,.\,x$), read an address from an environment ($t\,.\,x$), and get the address of a variable ($\&x$).

Events handlers Event handler tracing is a unique feature of Breakout, which is driven by the fact that in typical serverless functions, all "hot paths" include callbacks. Without this feature, Breakout would only support trivial serverless functions that do not use external services.

The result of tracing is a set of numbered event handlers (**`handler`**). Each handler contains 1) the trace tree of the event handler body (**`body`**), which runs in response to the event, 2) the name of a variable that refers to the event itself (**`argId`**), and 3) the name of a variable that refers to an environment object (**`envId`**). In addition, handlers have a fourth field, which is the *value* of the environment (**`env`**). This value is only available at runtime, and thus does not appear in the trace language syntax. The environment allows us to support event handlers that close over variables in their environment, which are common in JavaScript.

For every request, the Dispatcher runs a small state machine that executes traced event handlers. We treat the main body of the program an event handler 0, and start execution by running it with empty **`envId`** and **`argId`**. The other event handlers run when the program issues an event using the **`event`** expression, which requires several arguments:

1. An event type (ev), which determines the kind of operation to perform, e.g., to send a web request or start a timer.
2. An event argument (t_{arg}), which is a trace that determines, for example, the URL to request or the duration of the timer.
3. The number (n) of the event handler whose traced body will be executed when the event completes.
4. The environment (t_{env}), which is a trace that refers to the environment object of the event handler.

During trace tree execution, when Breakout evaluates an **`event`** expression, it 1) stores the value of t_{env} and the handler n, 2) fires the event ev (implemented in Rust), and 3) when the event completes, it invokes the trace tree body of the handler n with t_{env} and the result from the event. The body may contain additional **`event`** expressions that repeat this process in the state machine. An end condition is reached when trace tree execution encounters an **`respond`** expression, which sends a response back to the original incoming request.

3.2 Trace Contexts

The Breakout runtime builds a trace program incrementally, which involves efficiently merging the trace of the current execution into an existing trace tree. To make this possible, Breakout uses an explicit representation of *trace contexts* (Figure 4). Similar to a continuation, a trace context (κ) is a representation of a trace with a "hole". (We write $\cdot$ for the

$\kappa ::= \cdot$	Empty context
$\mid$ SEQ($[t_1 \cdots t_{i-1}]$, $[t_{i+1} \cdots t_n]$, κ)	In a block, with $[t_1 \cdots t_{i-1}]$ already executed.
$\mid$ IFTRUE(t_1, t_2, κ)	In the true branch of an *if*, with condition t_1 and false branch t_2.
$\mid$ IFFALSE(t_1, t_2, κ)	In the false branch of an *if*, with condition t_1 and true branch t_2.
$\mid$ WHILE(t, κ)	In the body of a loop, with condition t.
$\mid$ LABEL(ℓ, κ)	In the body of a labeled trace, with label ℓ.
$\mid$ NAMED(x, κ)	In the body of a named variable x.

Figure 4. A trace context identifies a position within a trace in which the current statement is executing.

empty trace context.) For example, consider the following trace context:

$$\text{IFTRUE}(x{>}0, \text{☠}, \text{WHILE}(y{<}0, \cdot))$$

Above, IFTRUE indicates that current trace is immediately inside the true-branch of an `if`, which is within a `while` loop.

Each layer of the trace context carries enough information to completely reconstruct the trace. Thus IFTRUE carries the trace of the condition ($x{>}0$) and the else-branch (☠), and WHILE carries the trace of the loop guard ($y{<}0$). In this example, the ☠ indicates that execution has not yet entered the else-branch. Notice that the trace context represents the expressions around the hole "inside out". This representation makes trace context manipulation simpler and more efficient for the tracing runtime system.

Finally, we note that a trace context is *not* an evaluation context. For example, we can write the evaluation context of an *if* (`if` (t_1) t_2 `else` t_3) as `if` $(\cdot)$ t_2 `else` t_3. Informally, the evaluation context "remembers" both branches while evaluating the condition. In contrast, the IFTRUE trace context frame is equivalent to the evaluation context `if` $(t_1)\cdot$`else` t_3, which is not for ordinary evaluation.

3.3 Instrumenting JavaScript to Generate Traces

The Function Runner uses a source-to-source compiler to instrument a serverless function to build its own trace. The compiler is syntax-directed and relies on a small runtime system (Figure 6). This section focuses on tracing JavaScript programs that do not use event handlers (presented in §3.4).

The tracing runtime The internal state of the runtime system consists of three variables: 1) the trace of the currently executing statement (c), 2) its trace context (κ), and 3) a stack of traces that represent function arguments (α). A key invariant during tracing is that plugging c into κ produces a trace for the entire program. Therefore, when tracing begins, we initialize c to the unknown statement (☠), κ to the empty trace context ($\cdot$), and α to an empty stack [].

The runtime system has several functions that update its internal state and construct trace expressions. In these functions, we write $[\![t]\!]$ to denote the traced runtime representation of the expression t (i.e., to quote t). For example, $[\![x]\!]$ evaluates to a trace representation of the identifier x, whereas x evaluates to its value. Most functions in the runtime system receive quoted arguments. In our implementation, $[\![t]\!]$ is a JSON data structure. The runtime system (Figure 6) has four kinds of functions, described below.

First, several functions update the current trace (c), but leave the trace context unchanged. We use these functions to build representations of JavaScript statements that do not affect the control-flow of the program, such as declaring a variable (let) or assigning to a variable (set). If we think of the trace expression as a tree, these functions create leaf nodes in the expression tree.

Several functions push a new frame onto the trace context. The compiler inserts calls to these functions to record the control-flow of the program. Each function in this category has two cases. 1) If c is ☠, it creates a new context frame and leaves the current expression as ☠. If we think of the trace expression as a tree, this case occurs when we enter a node in the trace tree for the first time. 2) If c is not ☠, it uses the sub-expressions of c to create the context frame and update c itself. For example, if c is a *if* expression, ifTrue stores the condition and false-part in the trace context, and sets c to the true-part. Conversely, ifFalse sets c to the false-part. Thinking of the trace expression as a tree, this case occurs when we descend into a branch of a node that we have visited before, while preserving other branches in the trace context.

The function pop pops the top of the trace context, and uses it to update c to a new expression, which uses the previous value of c as a sub-expression. Thinking of the trace expression as a tree, we call pop to ascend from a node to its parent. We use the function popTo to trace `break` expressions, which transfer control out of a labeled block. This function calls pop repeatedly until it reaches a block with the desired label.

Finally, the functions pushArg and popArg push and pop traced expressions onto the stack of arguments (α), and the compiler uses them to instrument functions and applications.

Note that the current trace and its context effectively form a "zipper" [26] for a trace of the entire program, and the functions defined above are closely related to canonical zipper operation. However, the operations that create trace context frames are unconventional because they either move the focus of the zipper into an existing child node, or create a new child and then focus on it. Although we are using a zipper, the runtime system is stateful: the functions update c, κ, and

$$\rho : x \to [\![t]\!] \qquad \mathcal{L}[\![x]\!]\rho \triangleq \rho(x) \qquad \mathcal{E}[\![c]\!]\rho \triangleq [\![c]\!] \qquad \mathcal{E}[\![x]\!]\rho \triangleq \rho(x)$$

$$\mathcal{E}[\![e_1\ op\ e_2]\!]\rho \triangleq e_1'\ [\![op]\!]\ e_2' \quad \text{where } e_1' \triangleq \mathcal{E}[\![e_1]\!]\rho \quad e_2' \triangleq \mathcal{E}[\![e_2]\!]\rho$$

$$\mathcal{S}[\![\texttt{let } x = e;]\!]\rho \triangleq (\text{let}([\![x]\!], \mathcal{E}[\![e]\!]\rho); \texttt{let } x = e;, \rho[x \mapsto [\![x]\!]])$$

$$\mathcal{S}[\![\texttt{let } f = \texttt{function}(x_1 \cdots x_n)\, blk;]\!]\rho \triangleq (\text{let}([\![f]\!], [\![\rho]\!]); \texttt{let } f = \texttt{function}(x_1 \cdots x_n)\, blk'';, \rho[f \mapsto [\![f]\!]])$$
$$\text{where } s_1 \triangleq \text{let}([\![x_1]\!], \text{popArg}()) \cdots s_n \triangleq \text{let}([\![x_n]\!], \text{popArg}()) \quad \{s_1' \cdots s_m'\} \triangleq blk \quad (y_1 \cdots y_q) \triangleq \text{dom}(\rho)$$
$$\rho' \triangleq \rho \left[\begin{array}{l} x_1 \mapsto [\![x_1]\!] \cdots x_n \mapsto [\![x_n]\!], \\ y_1 \mapsto [\![env.y_1]\!] \cdots y_q \mapsto [\![env.y_q]\!] \end{array} \right]$$
$$(blk', \rho'') \triangleq \mathcal{S}[\![\{\text{let}([\![env]\!], \text{popArg}()); s_1 \cdots s_n; s_1' \cdots s_m'\}]\!]\rho' \quad blk'' \triangleq \{\text{label}(ret); blk'; \text{pop}()\}$$

$$\mathcal{S}[\![\texttt{let } r = f(e_1 \cdots e_n);]\!]\rho \triangleq (s_n \cdots s_1; \text{pushArg}([\![f]\!]); \text{named}([\![r]\!]); \texttt{let } r = f(e_1 \cdots e_n); \text{pop}();, \rho')$$
$$\text{where } s_1 \triangleq \text{pushArg}(\mathcal{E}[\![e_1]\!]\rho) \cdots s_n \triangleq \text{pushArg}(\mathcal{E}[\![e_n]\!]\rho) \quad \rho' \triangleq \rho[r \mapsto [\![r]\!]]$$

$$\mathcal{S}[\![lval = e;]\!]\rho \triangleq (\text{set}(\mathcal{L}[\![lval]\!]\rho, \mathcal{E}[\![e]\!]\rho); lval = e;, \rho])$$

$$\mathcal{S}[\![\{s_1 \cdots s_n\}]\!]\rho \triangleq (\{\text{enterSeq}(n); s_1'; \text{seqNext}(); s_2'; \cdots; s_n'; \text{pop}()\}, \rho) \text{ where } (s_1', \rho_1) \triangleq \mathcal{S}[\![s_1]\!]\rho \cdots (s_n', \rho_n) \triangleq \mathcal{S}[\![s_n]\!]\rho_{n-1}$$

$$\mathcal{S}[\![\texttt{if } (e)\ s_1 \texttt{ else } s_2]\!]\rho \triangleq (\texttt{if } (e)\ \{\text{ifTrue}(\mathcal{E}[\![e]\!]\rho); s_1'\} \texttt{ else } \{\text{ifFalse}(\mathcal{E}[\![e]\!]\rho); s_2'\}; \text{pop}(), \rho)$$
$$\text{where } (s_1', \rho_1) \triangleq \mathcal{S}[\![s_1]\!]\rho \quad (s_2', \rho_2) \triangleq \mathcal{S}[\![s_2]\!]\rho$$

$$\mathcal{S}[\![\texttt{while } (e)\ s]\!]\rho \triangleq (\text{while}(\mathcal{E}[\![e]\!]\rho); \texttt{while } (e)\ s'; \text{pop}(), \rho) \quad \text{where } (s', \rho') \triangleq \mathcal{S}[\![s]\!]\rho$$

$$\mathcal{S}[\![\ell : s]\!]\rho \triangleq (\text{label}(\ell); \ell : s', \rho) \quad \text{where } (s', \rho') = \mathcal{S}[\![s]\!]\rho$$

$$\mathcal{S}[\![\texttt{break } \ell;]\!]\rho \triangleq (\text{break}(\ell, \texttt{undefined}); \text{popTo}(\ell); \texttt{break } \ell;, \rho)$$

$$\mathcal{S}[\![\texttt{return } e;]\!]\rho \triangleq (\text{break}(ret, \mathcal{E}[\![e]\!]\rho); \text{popTo}(ret); \texttt{return } e;, \rho)$$

$$\mathcal{S}[\![[\![t]\!]]\!]\rho \triangleq ([\![t]\!], \rho)$$

Figure 5. The trace compiler.

Operations that create leaves in the trace tree

$\text{let}(x, t) \triangleq c = \texttt{let } x = t;$ $\quad \text{set}(t_1, t_2) \triangleq c = t_1 = t_2;$ $\quad \text{break}(\ell, t) \triangleq c = \texttt{break } \ell\ t;$

Operations that may create interior nodes in the trace tree

$\text{enterSeq}(n) \triangleq c = \text{☠}_1;\ \kappa = \text{SEQ}([], [\text{☠}_2 \cdots \text{☠}_n], \kappa)$	if $c = \text{☠}$
$\text{enterSeq}(n) \triangleq c = t_1;\ \kappa = \text{SEQ}([], [t_2 \cdots t_n], \kappa)$	if $c = \{t_1 \cdots t_n\}$
$\text{seqNext}() \triangleq c = t_{i+1};\ \kappa = \text{SEQ}([t_1 \cdots t_{i-1}, c], [t_{i+2} \cdots t_n], \kappa)$	if $\kappa = \text{SEQ}([t_1 \cdots t_{i-1}], [t_{i+1} \cdots t_n], \kappa)$
$\text{ifTrue}(t) \triangleq c = \text{☠};\ \kappa = \text{IFTRUE}(t, \text{☠}, \kappa)$	if $c = \text{☠}$
$\text{ifTrue}(t_1) \triangleq c = t_2;\ \kappa = \text{IFTRUE}(t_1, t_3, \kappa)$	if $c = \texttt{if } (t_1)\ t_2 \texttt{ else } t_3$
$\text{ifFalse}(t) \triangleq c = \text{☠};\ \kappa = \text{IFFALSE}(t, \text{☠}, \kappa)$	if $c = \text{☠}$
$\text{ifFalse}(t_1) \triangleq c = t_3;\ \kappa = \text{IFFALSE}(t_1, t_2, \kappa)$	if $c = \texttt{if } (t_1)\ t_2 \texttt{ else } t_3$
$\text{while}(t) \triangleq c = \text{☠};\ \kappa = \text{WHILE}(t, \kappa)$	if $c = \text{☠}$
$\text{while}(t_1) \triangleq c = t_2;\ \kappa = \text{WHILE}(t_1, \kappa)$	if $c = \texttt{while } (t_1)\ t_2$
$\text{label}(\ell) \triangleq c = \text{☠};\ \kappa = \text{LABEL}(\ell, \kappa)$	if $c = \text{☠}$
$\text{label}(\ell) \triangleq c = t;\ \kappa = \text{LABEL}(\ell, \kappa)$	if $c = \ell : t$
$\text{named}(x) \triangleq c = \text{☠};\ \kappa = \text{NAMED}(x, \kappa)$	if $c = \text{☠}$
$\text{named}(x) \triangleq c = t;\ \kappa = \text{NAMED}(x, \kappa)$	if $c = \texttt{let } x = t$

Operations that move from a node to its parent in the trace tree

$\text{pop}() \triangleq c = \texttt{if } (t_1)\ c \texttt{ else } t_2;\ \kappa = \kappa'$	if $\kappa = \text{IFTRUE}(t_1, t_2, \kappa')$
$\text{pop}() \triangleq c = \texttt{if } (t_1)\ t_2 \texttt{ else } c;\ \kappa\kappa'$	if $\kappa = \text{IFFALSE}(t_1, t_2, \kappa')$
$\text{pop}() \triangleq c = \texttt{while } (t)\ c;\ \kappa = \kappa'$	if $\kappa = \text{WHILE}(t, \kappa')$
$\text{pop}() \triangleq c = \{t_1 \cdots t_{i-1}; c; t_{i+1} \cdots t_n\};\ \kappa = \kappa'$	if $\kappa = \text{SEQ}([t_1 \cdots t_{i-1}], [t_{i+1} \cdots t_n], \kappa')$
$\text{pop}() \triangleq c = \ell :\ c;\ \kappa = \kappa'$	if $\kappa = \text{LABEL}(\ell, \kappa')$
$\text{pop}() \triangleq c = \texttt{let } x = c;\ \kappa = \kappa'$	if $\kappa = \text{NAMED}(x, \kappa')$
$\text{popTo}(\ell) \triangleq c = \ell :\ c;\ \kappa = \kappa'$	if $\kappa = \text{LABEL}(\ell, \kappa')$
$\text{popTo}(\ell) \triangleq \text{pop}();\ \text{popTo}(\ell);$	if $\kappa \neq \text{LABEL}(\ell, \kappa')$

Operations that manipulate the stack of argument traces

$\text{pushArg}(t) \triangleq \alpha = (t :: \alpha)$ $\quad \text{popArg}() \triangleq \alpha = \alpha';\ \texttt{return } t;\text{if } \alpha = (t :: \alpha)$

Figure 6. The functions provided by the tracing runtime system. We initialize $c = \text{☠}$, $\kappa = \cdot$, and $\alpha = []$.

α. Instead, the zipper-based approach is a clean abstraction for building the trace tree incrementally.

The tracing compiler The compiler (Figure 5) is syntax-directed compiler, defined by functions to compile statements ($\mathcal{S}$), expressions ($\mathcal{E}$), and l-values ($\mathcal{L}$). The compiler inserts calls to the runtime system, so the program builds a trace as a side-effect, and is otherwise unchanged. Compiling function declarations and applications requires the most work. The trace of a function application effectively inlines the trace of the function body. The compiler takes care to ensure that the traces correctly captures the semantics of JavaScript closures. The compiler must ensure that the trace of the function body can refer to variables that were in scope in the original JavaScript program, but are not in scope at the application site. For this to work, the compiler represents the trace of a function f as its environment (ρ), function

applications pass the environment on the trace argument stack, and we bind free variables in the function body to expressions that access fields of the environment.

Example: tracing a conditional Figure 7 shows an example of how the tracing compiler and runtime system operate. The program in Figure 7b uses branching to calculate $y = |x|$. In the figure, the code inserted by the compiler is shaded gray, and the original program is unshaded.

Figure 7a shows a first run of the program with $x<0$. The initial value of the current trace (c) is unknown (☠) and the initial trace context is empty ($\cdot$). The program enters the true-branch, and calls ifTrue in the runtime system, which 1) pushes an IfTrue frame onto the trace context with $x<0$ for the condition and ☠ for the false-branch (since it has not been executed); and 2) sets the c to ☠, since the body of the true-branch has not yet been executed. Next, inside the true-branch, the JavaScript code assigns y, and the inserted call to set updates c to a quoted representation of the assignment. Finally, after the *if* statement, the call to pop, pops the IfTrue frame off the trace context, plugging in the trace of the true-branch. In this final configuration, the trace context is empty, and the current trace represents the entire known program (with ☠ in the false-branch).

Figure 7c shows a second run of the program with $x \geq 0$. This run resumes tracing where the first run ended, thus we preserve the value of the current trace. Within the false-branch, we call ifFalse, which pushes an IfFalse frame onto the trace context. Moreover, since the current trace is already an *if*, ifFalse preserves the trace of the true branch that we calculated on the first run. After the call to ifFalse, the program assigns to y and records the assignment in c, similar to the first run. Therefore, when the program finally calls pop, c contains a complete trace of the false branch, and the IfFalse frame contains a complete trace of the true branch from the prior run. Therefore, the final value of c is a complete trace without any ☠s.

Example: tracing a function application In Appendix A, we present an example of tracing a function application, where the body uses a non-local variable.

3.4 Tracing Event Handlers

Tracing event handlers is a key feature of Breakout and involves two challenges: 1) managing trace trees for multiple active event handlers, and 2) supporting nested event handlers that capture non-local variables.

For example, the Breakout API has a function `get` which sends an HTTP request to a `url` and calls the `callback` function with the response. (Figure 9a shows the implementation with error handling elided.) To actually issue the request, `get` uses a function from a popular Node library called `request.get` (line 7). To manage tracing, `get` relies on three runtime system functions (Figure 8). 1) We call newHandler immediately before registering an event handler in JavaScript. This helper function reflects the newly created event handler by 1) creating a new **`handler`**, and 2) setting the current trace (c) to an **`event`**. Note that the body of the **`handler`** is initialized to ☠. However, as long as the event triggers a response, the ☠ will be replaced with the trace of the event handler. 2) We call loadHandler immediately after receiving an event. This function prepares the runtime to trace the callback by 1) pushing the traces of its environment and argument onto the argument stack, and 2) setting the current trace (c) to the trace in the handler (**`body`**). 3) Finally, we call saveHandler after the callback returns to store the current trace back into the handler. Therefore, if the callback executes multiple times, the trace in the handler gets restored, and thus can grow incrementally.

4 Compiling Traces to Rust

Finally, we present how the Controller compiles traces to Rust, the latter half of Decontainerization. Compiling traces to Rust has two major steps: 1) We impose CPU and memory limits on the program, and 2) We address the mismatch between the types of values in traces (which is dynamically typed) and Rust (which is statically typed). (Figure 13 in appendix B shows the subset of Rust targeted by the trace-to-Rust compiler.)

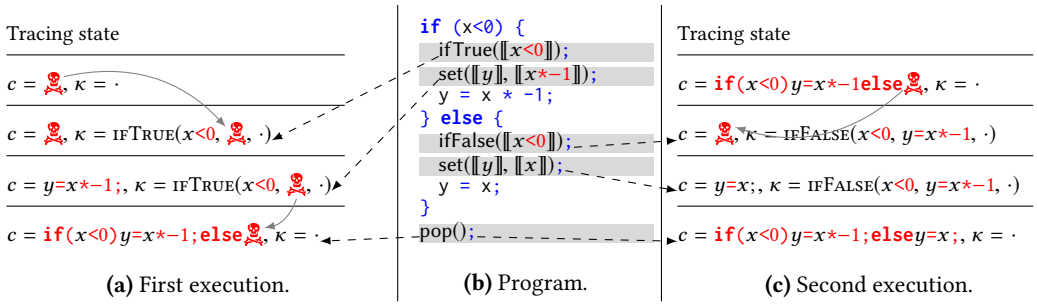
4.1 Static Types and Arena Allocation

Compiling the dynamically typed trace program to statically-typed Rust presents three separate issues.

Dynamic type In JavaScript, we can write expressions which produce a type error in Rust, such as `1 + true` (which evaluates to 2), so we inject JavaScript values into a *dynamic type* [2], defined as a Rust enumeration. Figure 14 (in appendix C) shows the Rust code for a simplified fragment of the dynamic type that we employ, which includes atomic values, as well as containers, such as objects. The dynamic type implements methods for all possible operations for all cases in its enumeration, and these methods may fail at runtime if there is a genuine type error.

Aliased, mutable pointers The Rust type system guarantees that all mutable pointers are unique, or *own* the values that they point to. Therefore, it is impossible for two mutable variables to point to the same value in memory. However, neither JavaScript nor the trace language have such restrictions. For code that truly requires multiple mutable references to the same object, the Rust standard library defines a container type (`RefCell`) that dynamically checks Rust's ownership rules, but prevents the value from crossing threads. Since JavaScript and trace programs are single-threaded, we use `RefCell` in our dynamic type to support aliasing (Figure 14, line 6 in appendix C).

Lifetimes and arena allocation Variables in Rust have a statically-determined lifetime. However, variables in traces

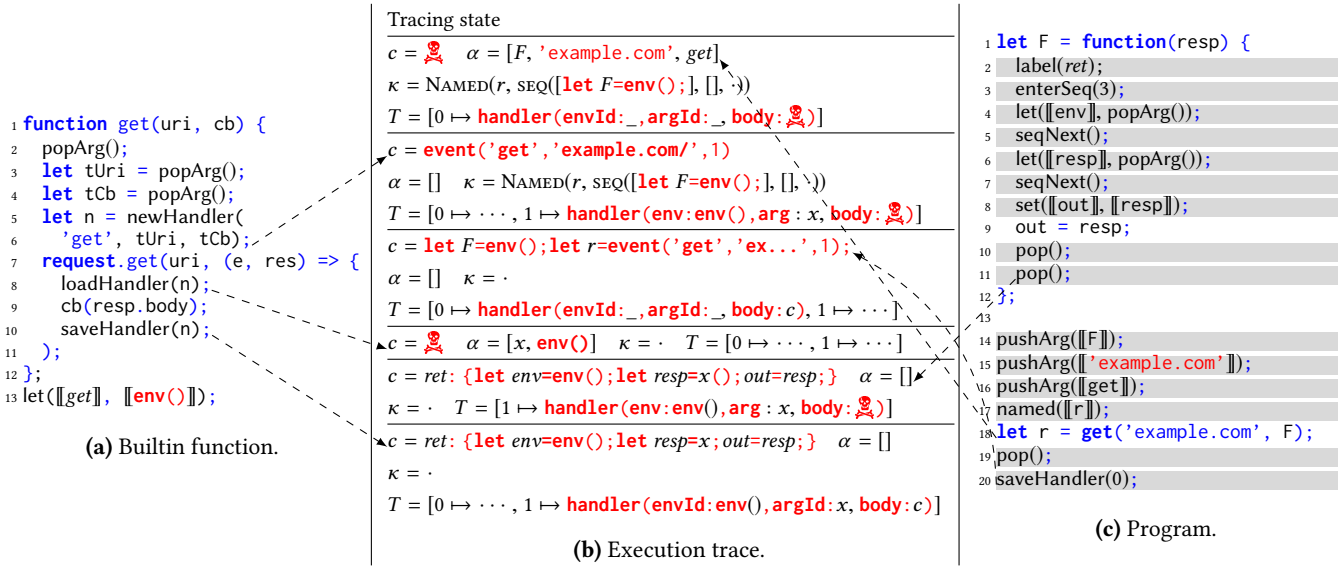


(a) First execution. **(b)** Program. **(c)** Second execution.

Figure 7. An example of incremental trace tree construction. The shaded lines are compiler-inserted, and the unshaded lines are in the original program. Figure 7a shows a run with input $x < 0$, and Figure 7a shows a second run with input $x \geq 0$.

newHandler(ev, t_{arg}, t_{env}) ≜ c=**event**(ev, t_{arg}, t_{env}, n); T=T[n ↦ **handler**(**envId**:env,**argId**:x, **body**:☠)];**return** n; where n, x are fresh if c= ☠
newHandler(ev, t_{arg}, t_{env}) ≜ **return** n; if c=**event**(ev, t_{arg}, t_{env}, n) $T(n)$ = **handler**(**envId**:env,**argId**:x, **body**:☠)
loadHandler(n) ≜ pushArg(h.**env**);pushArg(h.**argId**);c=h.**body**; $h = T(n)$
saveHandler(n) ≜ T=T[n ↦ $T(n)$ with **body** = c];

Figure 8. Runtime system for event handlers.



(a) Builtin function. **(b)** Execution trace. **(c)** Program.

Figure 9. Event handler example.

may be captured in environment objects, and thus have a lifetime that is not statically known. Rust supports a variety of workaround (e.g, reference counting and dynamic borrow checking). However, programs that use these features may leak memory, which is not safe for BREAKOUT. Instead, BREAKOUT uses an *arena* to store the values of a running trace program. Arena allocation simplifies lifetimes, since the lifetime of all values is the lifetime of the arena itself. However, the only way to free a value in an arena is to free all values in the arena at once. Fortunately, the serverless execution model gives us a natural point to allocate and clear the arena: we clear the arena immediately after each response, which is safe in a serverless context, because functions must tolerate transient memory.

4.2 Bounding Memory and Execution Time

Serverless computing relies on bounding the CPU and memory utilization of serverless functions. The arena allocator makes it easy to impose a memory bound: all values have the same lifetime as the allocator, and we impose a maximum limit on the size of the arena. Imposing a CPU utilization bound is more subtle, since BREAKOUT can run several trace programs in the same process, thus we cannot accurately account for the CPU utilization for an individual request. Instead, the trace-to-Rust compiler uses an instruction counter,

which it increments at the top of every loop and at the end of every invocation of the state machine, and we bound the number of Rust statements executed.

5 Evaluation

Our primary goal is to determine if Breakout can reduce the latency and resource usage of typical serverless functions.

5.1 Benchmarking

Benchmark Summary We develop six benchmarks:

1. *authorize* is equivalent to the running example in the paper (Figure 1a). It receives as input a username and password, fetches the password database (represented as a JSON object), and validates the input.
2. *upload* uploads a file to cloud storage. It receives the file in the body of a POST request and issues a POST request to upload it.
3. *status* updates build status information on GitHub. i.e., it can add a ✓ or ✗ next to a commit, with a link to a CI tool. The function takes care of mapping a simple input to the JSON format that the GitHub API requires.
4. *banking* simulates a banking application, with support for deposits and withdrawals (received over POST requests). It uses the Google Cloud Datastore API with transactional updates.
5. *autocomplete* implements autocomplete. Given a word as input, it returns a number of completions.
6. *maze* is more computationally expensive than the others. It finds the shortest path between two points in a maze on each request.

Experimental Setup We have tested Breakout on a three-node Kubernetes cluster. However, variation in network performance makes results hard to reproduce. Therefore, we run the following experiments on a single-node Kubernetes configuration, running on Dual-socket AMD EPYC 7282 with 128 GB RAM. We allocate 1 CPU core and 1 GB RAM to each Function Runner. These limits are configurable, and higher than the default resource limits for popular cloud-hosted serverless platforms.

Several benchmarks rely on external services (e.g., GitHub and Google Cloud Datastore). We tested them, but our experiments use mocks of these external services. Without mocks, our experiments would again suffer from variation in network performance. Additionally, our experiments issue thousands of requests per second and would trigger API rate-limits on these services.

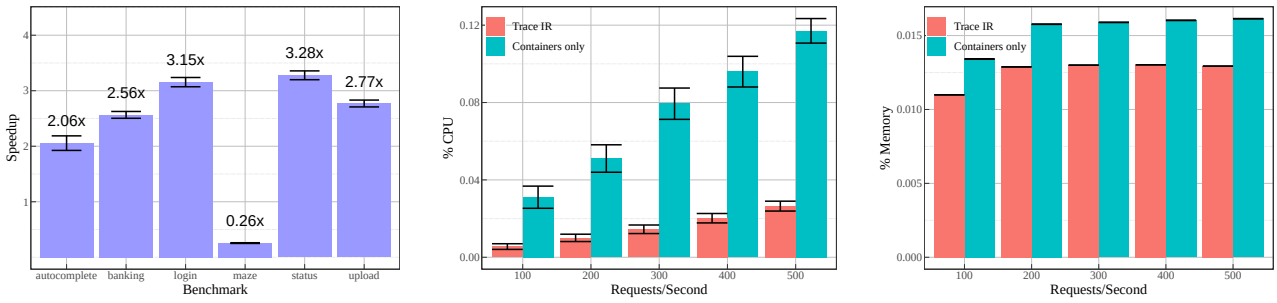
Steady-State Performance For our first experiment, we measure steady-state performance with and without Decontainerization. We send requests using ten concurrent streams, where each stream immediately issues another request the moment it receives a response. We measure end-to-end response latency and report the speedup.

We run each benchmark for 120 seconds, and start measurements after 75 seconds, which gives Breakout time to perform Decontainerization and start processing requests in Rust. The request arrival rate is high enough that Function Runners are never idle, and thus never shut down. Figure 10a shows the mean speedup for each benchmark with Breakout. In five of the six benchmarks, Breakout with Decontainerization is significantly faster, with speedups ranging from 2.06x to 3.28x. The outlier is the *maze* benchmark, which is discussed below.

Cold-to-Warm Performance Our second set of experiments examine the behavior of Breakout under cold starts. As in the previous section, we run each benchmark with and without Decontainerization, issuing events using ten concurrent event streams. We run each experiment for two minutes, starting with no running containers. Figure 11 plots the mean and maximum event processing latency over time.

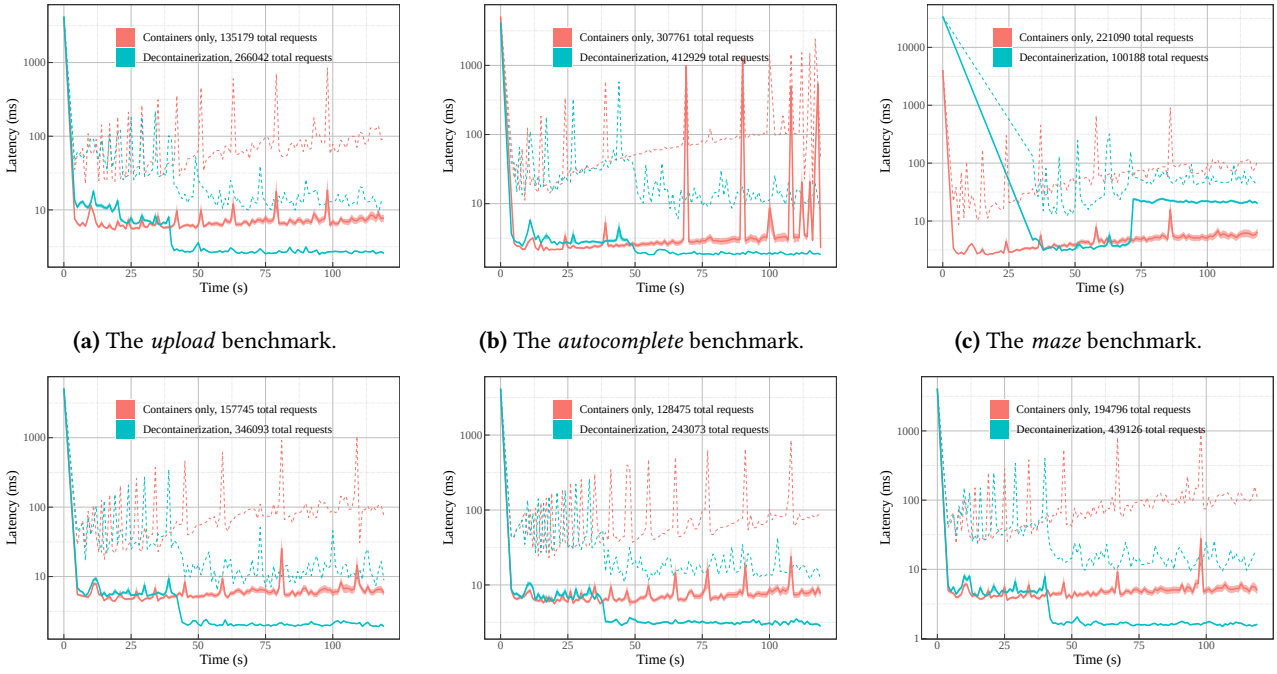
Let us examine *upload* in detail (Figure 11a). **Cold starts:** At $t = 0$, Breakout with Decontainerization and containers-only both exhibit cold starts (very high latency) as the containers warm up. *Note that the latency (y-axis) is on a log scale.* **Warm starts:** Since there are ten concurrent event streams, both cases start up the maximum number of containers (six), where one of the containers runs tracing for Decontainerization. Once they are all started, mean latency for both invokers dips to around 10 ms. However, tracing does incur some overhead, and we can see that the mean latency for Breakout with Decontainerization is slightly higher. **Breakout starts:** However, in the Breakout with Decontainerization case, within 40 seconds, Decontainerization is performed and requests start processing in Rust. Thus the mean latency *dips again* to 2.7 ms after 40 seconds. Over the duration of the 120 second benchmark, the Decontainerization case is able to process 1.96x *more* requests than with containers-only. Five out of six benchmarks exhibit this "double dip" behavior: first for warm starts, and then again once Breakout with Decontainerization starts its language-based sandbox. **Variability:** The plot also shows the event processing time has higher variability with containers. This occurs because there are ten concurrent connections and only six containers (one for each core) thus some events have to be queued. Breakout with Decontainerization runs in a single process, with one physical thread for each core. However, the Rust runtime system (Tokio) supports non-blocking I/O and is able to multiplex several running trace programs on a single physical thread, thus can process more events concurrently.

The outlier of the benchmarks is the *maze* benchmark, which runs at 0.26x the speed with Decontainerization. We believe that the reason for the slowdown is that *maze* uses a JavaScript array as a queue. JavaScript JITs support multiple array representations and optimize for this kind of behavior and the implementation of dequeueing (the `.shift` method)



(a) Speedup of Breakout. **(b)** CPU utilization of *authorize*. **(c)** Memory utilization of *authorize*.

Figure 10. CPU and memory utilization. The error bars show the 95% confidence interval.



(a) The *upload* benchmark. **(b)** The *autocomplete* benchmark. **(c)** The *maze* benchmark.

(d) The *authorize* benchmark. **(e)** The *banking* benchmark. **(f)** The *status* benchmark.

Figure 11. Cold-to-warm performance with and without Decontainerization. Each experiment runs for two minutes and begins with no containers loaded. Each graph summarizes the latency of events issued at a point in time, with $t = 0$ is the start of the experiment. The solid lines show the mean event latency, with the 95% confidence interval depicted by the shaded region around the mean. The dotted lines show the maximum latency.

in our Rust runtime system is an $O(n)$ operation. To test this theory, we ran an additional experiment with a simple function that performs `.shift` n times. We found that the performance of Breakout using Decontainerization degrades as n increases, and does so more significantly than the JavaScript JIT optimized array. The results are provided in Appendix D.

However, at the same time, the maximum latency (dashed salmon line) is significantly lower with Decontainerization than without! Because *maze* does not perform any asynchronous I/O, we cannot attribute this behavior to nonblocking I/O. It is hard to pinpoint the root cause of this behavior. One possibility is the difference is memory management: within the container, the program runs in a JavaScript VM that incurs brief GC pauses, whereas Breakout uses arena allocation, and clears the arena immediately after each response. However, this is a conjecture, and there are several

differences between Breakout with Decontainerization and with container-only execution.

Resource Utilization Our third experiment examines CPU and memory utilization. We use the *authorize* benchmark and vary the number of requests per second. The maximum number of requests per second that we issue is 500, because a higher request rate exceeds the rate at which containers can service requests. We examine resource utilization after the cold start period. As shown in Figure 10b, Breakout with Decontainerization has a lower CPU utilization than with containers-only by a factor of 0.20x (geometric mean). Figure 10c shows that Decontainerization lowers memory utilization by a factor of 0.81x (geometric mean).

An Alternative to Cold Starts Decontainerization does not eliminate cold start latency, since it needs the function to run in a container to build the trace program. However, traced programs present a new opportunity: since they are more lightweight than containers, the invoker can keep them resident significantly longer. For example, running *authorize* in 100 containers consumes 1.6 GB of physical memory. In contrast, an executable that contains 100 copies of the trace produced by *authorize* is 10 MB. In Breakout, the arena allocator frees memory after a response, thus the only memory consumed by a function that is loaded and idle, is the memory needed for its code, and for its entry in a dispatch table, which maps a URL to a function pointer.

6 Related Work

Serverless performance SAND [3] uses process isolation to improve the performance of serverless functions that are composed together; X-Containers [43] develops a new container architecture to speed up arbitrary microservices; MPSC [5] brings serverless computing to the edge; Costless [14] helps programmers explore the tradeoff between performance and cost; and GrandSLAm [30] improves microservice throughput by dynamic batching. Breakout differs from these solutions because it uses speculative acceleration techniques to bypass the container when possible. As long the code can be traced, Breakout is complementary.

Breakout exploits the fact that many serverless platforms require functions to be idempotent and tolerate transient in-memory state [28, 36]. In contrast, Ambrosia [24] provides a higher-level abstraction and relieves programmers from thinking about these low-level properties.

Boucher et al. [11] present a serverless platform that requires programmers to use Rust. Rust has a steep learning curve and—more fundamentally—does not guarantee resource isolation, deadlock freedom, memory leak freedom, and other critical properties [39]. In contrast, Breakout allows programmers to write JavaScript, generates Rust code, and uses dynamic checks when needed to ensure safety (§4).

Tracing and JITs Breakout compiles dynamically generated execution trace trees, which is an idea with a long history. Bulldog [16] generates execution traces statically for a VLIW processor. TraceMonkey [23] is an *intra*procedural tracing JIT for JavaScript. Spur [8] is an *inter*procedural tracing JIT for Microsoft CIL. RPython [10] enables meta-tracing, turning an annotated interpreter into a tracing JIT, whereas Truffle [44] partially evaluates an interpreter to build a JIT.

Tracing in Breakout differs from prior work in three key ways. 1) Since the target language is a high-level language (Rust), the language of traces is high-level itself. 2) Breakout is designed for serverless execution, and naively restarts the serverless function in a container when it goes off trace, whereas prior work has to seamlessly switch between JIT-generated code and the interpreter. 3) The traces that Breakout produces are include asynchronous event handlers.

Operating systems There are a handful of research operating systems that employ language-based sandboxing to isolate untrusted code. Singularity [27] processes are written in managed languages and disallow dynamically loading code. SPIN [9] and VINO [40] allow programs to dynamically extend the kernel. Traces in Breakout are analogous to an extension of the Dispatcher, written in a safe language. However, programmers do not have to write traces themselves, but they are generated from JavaScript instrumentation. Moreover, Breakout switches between language-based and container-based sandboxing as needed.

Domain-specific accelerators Weld [37] produces a common IR from data analytics applications that mix several libraries and languages. Numba [33] accelerates Python and NumPy code by JITing methods. Unlike Breakout, these systems do not employ tracing. TorchScript [38] uses tracing for PyTorch, but places several restrictions on the form of Python code in a model. These accelerators, including Breakout, exploit domain-specific properties to achieve speedups. However, the domain-specific properties of serverless computing are very different from data analytics, scientific computation, and deep learning, which makes uniquely suited for serverless computing.

Serverless as HPC Several projects use serverless computing for "on-demand HPC" [4, 21, 22, 29, 34]. Breakout is unlikely to help in these use-cases, because they rely on native binaries. However, for short-running, I/O intensive applications, our evaluation shows that Breakout can improve performance significantly.

7 Conclusion

We present Decontainerization, which is a serverless function accelerator technique that dynamically traces serverless functions written in JavaScript and compiles these traces to a safe fragment of Rust. We implement Decontainerization in Breakout, which processes requests using compiled traces

instead of dispatching to a container. Even though it does not support all functions, Breakout detects acceleration failures at runtime, and fallbacks to containers. Our approach relies on the fact that serverless functions must be idempotent for fault-tolerance, so a restart due to acceleration failure is no different any other restart.

References

[1] [n.d.]. Cloud Functions Execution Environment. https://cloud.google.com/functions/docs/concepts/exec?hl=da. Accessed: 2020-11-19.

[2] Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. 1995. Dynamic typing in polymorphic languages. *Journal of Functional Programming* 5, 1 (1995), 111–130.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *USENIX Annual Technical Conference (ATC)*.

[4] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *ACM Symposium on Cloud Computing (SOCC)*.

[5] Austin Aske and Xinghui Zhao. 2018. Supporting Multi-Provider Serverless Computing on the Edge. In *International Conference on Parallel Processing (ICPP)*.

[6] Azure Functions on Kubernetes with KEDA 2019. Azure Functions on Kubernetes with KEDA. https://docs.microsoft.com/en-us/azure/azure-functions/functions-kubernetes-keda. Accessed Nov 20 2020.

[7] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*.

[8] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: A Trace-based JIT Compiler for CIL. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[9] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[10] Carl Friedrich Bolz and Laurence Tratt. 2015. The impact of meta-tracing on VM design and implementation. *The Science of Computer Programming* (Feb. 2015), 408–421.

[11] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the "Micro" back in microservices. In *USENIX Annual Technical Conference (ATC)*.

[12] CloudFlare 2020. CloudFlare Workers Security Model. https://developers.cloudflare.com/workers/learning/security-model. Accessed Nov 19 2020.

[13] Sarah Conway. 2017. Cloud Native Technologies Are Scaling Production Applications. https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/. Accessed Nov 19 2020.

[14] Tarek Elgamal. 2018. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*.

[15] Alex Ellis. 2018. OpenFaaS. https://www.openfaas.com. Accessed Oct 12 2019.

[16] John R. Ellis. 1985. *Bulldog: A Compiler for VLIW Architectures*. Ph.D. Dissertation. New Haven, CT, USA.

[17] Fastly 2020. Fastly ComputeEdge. https://docs.fastly.com/products/compute-at-edge. Accessed Nov 19 2020.

[18] Fission 2020. Fission Github Repository. https://github.com/fission/fission. Accessed Nov 19 2020.

[19] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[20] Fn Project 2020. Fn Project Website. https://fnproject.io/. Accessed Nov 19 2020.

[21] Sadjad Fouladi, , Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers.

[22] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *USENIX Symposium on Networked System Design and Implementation (NSDI)*.

[23] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[24] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proceedings of the VLDB Endowment* 13, 5 (Jan. 2020), 588–âĂŞ601.

[25] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[26] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549âĂŞ554.

[27] Galen Hunt, Mark Aiken, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Jim Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. In *European Conference on Computer Systems (EuroSys)*.

[28] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, OOPSLA (2019).

[29] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Symposium on Cloud Computing*.

[30] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks. In *European Conference on Computer Systems (EuroSys)*.

[31] Kubeless 2020. Kubeless Website. https://kubeless.io/. Accessed Nov 19 2020.

[32] Kubernetes 2020. Kubernetes Github Repository. https://github.com/kubernetes/kubernetes. Accessed Nov 19 2020.

[33] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *LLVM Compiler Infrastructure in HPC (LLVM)*.

[34] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *International Conference on Cloud Computing (CLOUD)*.

[35] Nimbella 2020. Nimbella CLI Github Repository. https://github.com/nimbella/nimbella-cli. Accessed Nov 19 2020.

[36] Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, and Ana Milanova. 2020. Formalizing Event-Driven Behavior of Serverless Applications. In *European Symposium on Cloud Computing (ESOCC)*.

[37] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Parimarjan Negi, Rahul Palamuttam, Anil Shanbhag, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. In *International Conference on Very Large Data Bases (VLDB)*.

[38] PyTorch Contributors. 2018. TorchScript. https://pytorch.org/docs/master/jit.html. Accessed Nov 2 2019.

[39] Rust 2019. Behavior Not Considered Unsafe. https://doc.rust-lang.org/reference/behavior-not-considered-unsafe.html. Accessed Nov 3 2019.

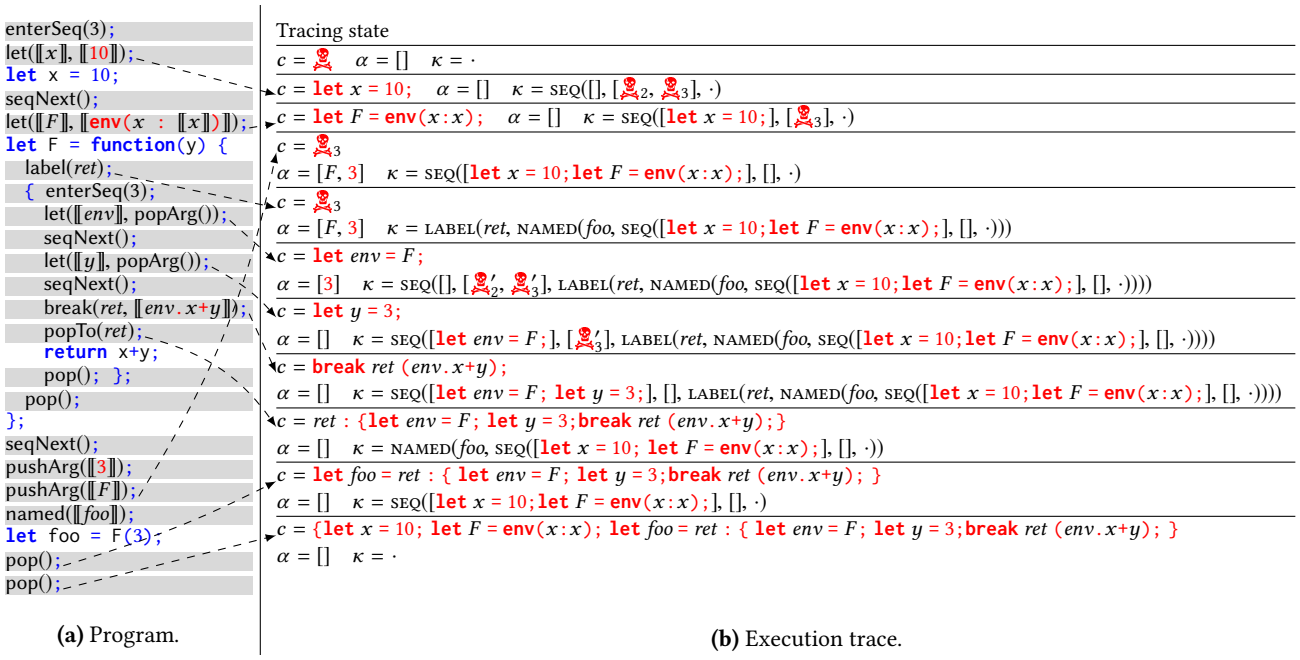
[40] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[41] serverless 2020. Repository of Serverless Function Examples. https://github.com/serverless/examples/commit/9eef07b09ee67c33e99c89c73b830a45b7da6ddb. Accessed Sep 30 2020.

[42] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. https://arxiv.org/abs/2003.03423.

[43] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Van Robbert Renesse, and Hakin Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[44] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wös, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.



(a) Program. **(b)** Execution trace.

Figure 12. An example of tracing a function application (shaded lines are compiler-inserted).

Operators

$op ::= + \mid - \mid * \mid \cdots$

Block

$blk ::= \{\ e_1 \cdots e_n \}$

Expressions

$e ::= ()$	Unit
$\mid c$	Constant
$\mid x$	Variable
$\mid e_1\ op\ e_2$	Binary operation
$\mid e?$	Try
$\mid e_1\texttt{.method(}e_2\texttt{,}\ e_3\texttt{,}\ \cdots\texttt{)}$	Method Call
$\mid \texttt{if (}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3$	Conditionals
$\mid \texttt{while (}e_1\texttt{)}\ blk$	Loops
$\mid \texttt{let}\ x = e\texttt{;}$	Variable declaration
$\mid blk$	Block
$\mid \texttt{'}\ell\texttt{: loop \{}$	Labeled block
$\mid\ \ e_1\texttt{;}\ e_2\texttt{;}\ \cdots\ \texttt{break '}\ell\ e_n\texttt{;}\texttt{\}}$	
$\mid \texttt{break '}\ell\ e\texttt{;}$	Break

Figure 13. The subset of Rust targeted by the trace-to-Rust compiler.

A Function Application Tracing Example

Figure 12 shows an example of tracing a function application, where the function F calculates the sum of its argument (x) and a variable that is free in its body (y). At the top of the program, the current trace is ☠, and at the end, the trace in c represents the entire program with F inlined. The figure shows the state of the tracing runtime at several key points. 1) The trace variable `F` is bound to an trace environment that is equivalent to the environment of the JavaScript function named `F`. 2) The program pushes and pops trace expressions from the argument stack (α). 3) The runtime system uses popTo before the **`return`**, which pops multiple frames off the context.

B Subset of Rust Targeted by the Trace Compiler

Figure 13 shows the subset of Rust targeted by the trace-to-Rust compiler. The subset features unified expressions and statements. In particular, the target subset excludes user-defined functions.

```
#[derive(Copy, Clone)]
pub enum Dyn<'a> {
  Int(i32),
  Bool(bool),
  Undefined,
  Object(&'a RefCell<Vec<'a, (&'a str, Dyn<'a>)>>),
}

impl<'a> Dyn<'a> {
  pub fn add(&self, other: &Dyn<a'>) -> Dyn<'a> {
    match (self, other) {
      (Dyn::Int(x), Dyn::Int(y)) => Dyn::Int(x + y),
      ...
    }
  }
}
```

Figure 14. A fragment of the dynamic type that Breakout uses to represent trace values.

C Dynamic Type for Trace Programs

Figure 14 shows a fragment of the dynamic type that Breakout employs to represent trace values. The function `add` is included in the fragment to demonstrate how to perform operations on two `&Dyn<a'>` values.

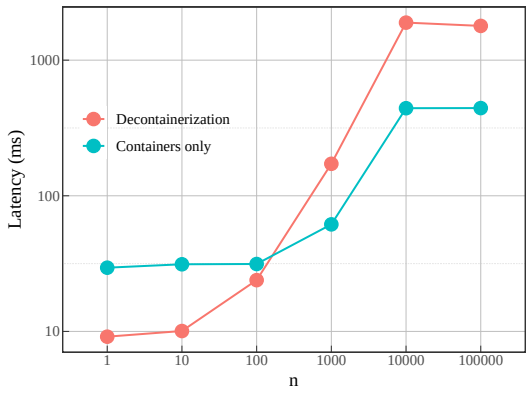


Figure 15. Warm-start performance of a function that performs `.shift` *n* number of times, comparing Breakout with Decontainerization to containers-only.

D The `.shift` Experiment

To get a better understanding of the core difference between a JavaScript JIT's optimized array and Breakout's prototype array implementation, we tested the warm-start performance of a simple function that performs `.shift` *n* number of times. Figure 15 shows the results, demonstrating that the performance of Breakout using Decontainerization degrades as *n* increases, and does so more significantly than the JavaScript JIT optimized array.

E Serverless Examples Repository

To understand the extent to which serverless functions contain features not amenable to Decontainerization, we statically evaluated a repository of 63 example functions written in JavaScript [41]. Table 1 contains a breakdown of the functions by name. A "✓" indicates that a function contains a particular feature and an "x" indicates that it does not.

	spawns processes	local filesystem	`eval`	WebAssembly	worker threads
aws-ffmpeg-layer	✓	✓	x	x	x
aws-node-alexa-skill	x	x	x	x	x
aws-node-auth0-cognito-custom-authorizers-api	x	x	x	x	x
aws-node-auth0-custom-authorizers-api	✓	x	x	x	x
aws-node-dynamic-image-resizer	✓	x	x	x	x
aws-node-dynamodb-backup	x	x	x	x	x
aws-node-env-variables	x	x	x	x	x
aws-node-env-variables-encrypted-in-a-file	x	x	x	x	x
aws-node-fetch-file-and-store-in-s3	x	x	x	x	x
aws-node-fullstack	✓	x	x	x	x
aws-node-function-compiled-with-babel	x	x	x	x	x
aws-node-github-check	x	x	x	x	x
aws-node-github-webhook-listener	x	x	x	x	x
aws-node-graphql-and-rds	✓	x	x	x	x
aws-node-graphql-api-with-dynamodb	x	x	x	x	x
aws-node-heroku-postgres	x	x	x	x	x
aws-node-iot-event	x	x	x	x	x
aws-node-mongodb-atlas	x	x	x	x	x
aws-node-oauth-dropbox-api	✓	x	x	x	x
aws-node-puppeteer	✓	x	x	x	x
aws-node-recursive-function	x	x	x	x	x
aws-node-rekognition-analysis-s3-image	x	x	x	x	x
aws-node-rest-api	x	x	x	x	x
aws-node-rest-api-mongodb	x	x	x	x	x
aws-node-rest-api-typescript	x	x	x	x	x
aws-node-rest-api-typescript-simple	x	x	x	x	x
aws-node-rest-api-with-dynamodb	x	x	x	x	x
aws-node-rest-api-with-dynamodb-and-offline	x	x	x	x	x
aws-node-s3-file-replicator	x	x	x	x	x
aws-node-scheduled-cron	x	x	x	x	x
aws-node-scheduled-weather	x	x	x	x	x
aws-node-serve-dynamic-html-via-http-endpoint	x	x	x	x	x
aws-node-serverless-gong	x	x	x	x	x
aws-node-ses-receive-email-body	x	x	x	x	x
aws-node-ses-receive-email-header	x	x	x	x	x
aws-node-shared-gateway	x	x	x	x	x
aws-node-signed-uploads	✓	x	x	x	x
aws-node-simple-http-endpoint	x	x	x	x	x
aws-node-simple-transcribe-s3	x	x	x	x	x
aws-node-single-page-app-via-cloudfront	x	✓	x	x	x
aws-node-stripe-integration	x	x	x	x	x
aws-node-telegram-echo-bot	x	x	x	x	x
aws-node-text-analysis-via-sns-post-processing	x	x	x	x	x
aws-node-twilio-send-text-message	x	x	x	x	x
aws-node-twitter-joke-bot	x	x	x	x	x
aws-node-typescript-apollo-lambda	x	x	x	x	x
aws-node-typescript-kinesis	✓	x	x	x	x
aws-node-typescript-nest	x	x	x	x	x
aws-node-typescript-rest-api-with-dynamodb	x	x	x	x	x
aws-node-typescript-sqs-standard	✓	x	x	x	x
aws-node-upload-to-s3-and-postprocess	✓	x	x	x	x
aws-node-vue-nuxt-ssr	x	x	x	x	x
aws-node-websockets-authorizers	x	x	x	x	x
azure-node-line-bot	x	x	x	x	x
azure-node-simple-http-endpoint	x	x	x	x	x
azure-node-telegram-bot	x	x	x	x	x
google-node-simple-http-endpoint	x	x	x	x	x
google-node-typescript-http-endpoint	x	x	x	x	x
openwhisk-node-and-docker-chaining-functions	x	x	x	x	x
openwhisk-node-chaining-functions	x	x	x	x	x
openwhisk-node-scheduled-cron	x	x	x	x	x
openwhisk-node-simple	x	x	x	x	x
openwhisk-node-simple-http-endpoint	x	x	x	x	x

Table 1. Breakdown of the features found in the serverless examples repository [41]. A "✓" indicates that a function contains a particular feature and an "x" indicates that it does not.