Ignis: Scaling Distribution-Oblivious Systems with Light-Touch Distribution

Nikos Vasilakis
University of Pennsylvania, USA
nikos@vasilak.is

Ben Karel
University of Pennsylvania, USA
karel@seas.upenn.edu

Yash Palkhiwala
University of Pennsylvania, USA
yashp@seas.upenn.edu

John Sonchack
University of Pennsylvania, USA
jsonch@seas.upenn.edu

André DeHon
University of Pennsylvania, USA
andre@acm.org

Jonathan M. Smith
University of Pennsylvania, USA
jms@cis.upenn.edu

Abstract

Distributed systems offer notable benefits over their centralized counterparts. Reaping these benefits, however, requires burdensome developer effort to identify and rewrite bottlenecked components. *Light-touch distribution* is a new approach that converts a legacy system into a distributed one using automated transformations. Transformations operate at the boundaries of bottlenecked modules and are parametrizable by light distribution recipes that guide the intended semantics of the resulting distribution. Transformations and recipes operate at runtime, adapting to load by scaling out only saturated components. Our IGNIS prototype shows substantial speedups, attractive elasticity characteristics, and memory gains over full replication, achieved by small and backward-compatible code changes.

CCS Concepts • **Computer systems organization → Distributed architectures**; *Cloud computing*; • **Software and its engineering** → *Extra-functional properties*; *Software as a service orchestration system.*

Keywords Distribution, Profiling, Load detection, Transformations, Scale-out, Scalability, Parallelism, Decomposition

ACM Reference Format:
Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. 2019. Ignis: Scaling Distribution-Oblivious Systems with Light-Touch Distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314586

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314586

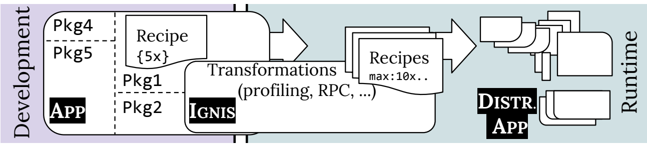

Fig. 1. Schematic of light-touch distribution. Module structure is used at runtime to automatically scale systems out, guided by recipes (*Cf.*§1).

1 Introduction

Distributed systems can speed up computations, mitigate resource-exhaustion attacks, improve fault-tolerance, and balance load during spikes. Yet, only a minority of developers, employed by the select few companies that deal with massive datasets, have the luxury of engineering software systems with distribution baked in from the start. The remaining majority starts by developing and deploying software in a centralized manner—that is, *until* there is a significant change of requirements, such as a load increase.

When this happens, developers try to identify affected parts of the system and manually re-write them to exploit distribution. The scope of such rewrites, and therefore the cost of manual effort, can vary considerably. Often, they only focus on a few parts of the system—for example, upgrading to a distributed storage layer. More rarely, companies rewrite entire systems (*e.g.*, Twitter's Ruby-to-Scala rewrite [55]), a process that is notoriously difficult under schedule constraints and competitive pressures [80, 93]. The manual effort is expensive, and can introduce new bugs, cascading changes, or regressions of previously fixed performance issues, especially since software today makes extensive use of third-party modules [85]. Could the process of identifying bottlenecks, generating a distributed version of the system, and scaling it out at runtime be *significantly* automated?

The core insight behind this work is that, instead of manually building scalability into the system, valuable human effort should only be spent on instructing the system *how* to scale. As long as developers have sprinkled the program with hints, it should *automatically detect* and *dynamically adapt* to load. We term this automation and associated control-plane hints *light-touch distribution* (Fig. 1).

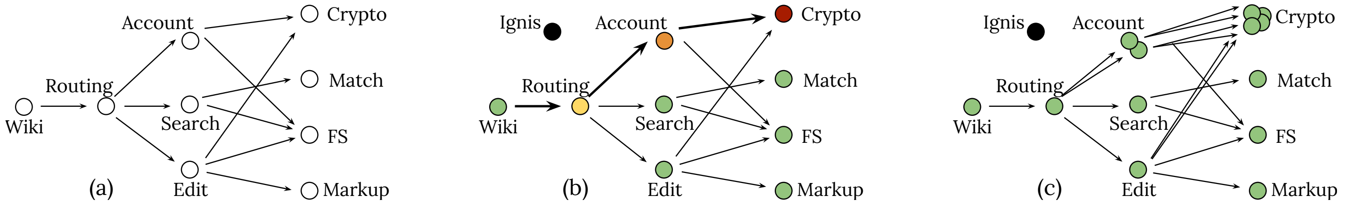

Fig. 2. Case study. Wiki module relationships (a) monitored continuously at runtime to identify bottlenecks (b) and automatically scale them out (c) (*Cf.*§3).

Light-touch distribution involves two components: (i) automated programmatic *transformations* that operate at module boundaries for detecting and adapting to load; and (ii) distribution *recipes*, lightweight annotations that guide the semantics of the resulting distributed application. Transformations automate most of the process, but depend on recipes for key semantic decisions that affect soundness. Both transformations and recipes operate at runtime, which offers significant benefits: applications can respond dynamically to increased load by scaling out, can selectively replicate saturated components instead of whole applications, and can avoid over-provisioning by scaling back when load subsides.

Light-touch distribution occupies a known middle-ground [10, 54, 86] between flexibility and automation (§9), and is enabled today by a confluence of trends in software development—namely, the increasingly pervasive use of (i) dynamic, interpreted languages, (ii) fine-grained modules with clear boundaries, and (iii) cooperatively concurrent, continuation-passing programming styles (CPS). Examples of such environments include JavaScript, Julia, and Lua; our Ignis prototype targets server-side JavaScript (§7).

We begin with an example of applying light-touch distribution (§2), and continue with an overview of Ignis (§3). Sections 4–6 highlight our key contributions:

- §4 introduces *load-detection transformations* that collect windowed statistics about load at each module boundary. Control-plane coordination with a global view of load and available resources helps decide when to initiate scale-out of a bottlenecked module.
- §5 presents a set of parametrizable *distribution transformations* that transparently scale modules out. These transformations can create module replicas, hook communication channels among them, schedule requests, and forward side-effects such as mutation and collection of memory.
- §6 outlines *distribution recipes*, lightweight annotations that guide the semantics of the resulting distribution. They offer significant flexibility by parametrizing transformations, including tuning state management, replication consistency, event propagation, and colocation preferences.

We then outline Ignis' implementation (§7), evaluate it using a combination of micro-benchmarks and real systems (§8), discusses related prior work (§9), and close with possible future directions (§10).

2 Background and Motivation

We use a wiki engine to illustrate difficulties in scaling out applications (§2.1), outline light-touch distribution (§2.2), apply Ignis to alleviate the aforementioned difficulties (§2.3), and outline the trends that enable this approach today (§2.4).

2.1 Case Study: A Wiki Engine

Fig. 2*a* shows the (simplified) module structure of a wiki engine [29]. Modules—development-time constructs usually glued together without a full understanding of their internals—are represented as vertices. The resulting dependencies, which in modern applications can be thousands [85], are depicted as edges connecting importing *parent* modules with imported *child* modules.

A sharp increase in sign-in attempts can saturate the `account` module. Logically unrelated parts of the system competing for the same resource (*e.g.*, CPU), such as document `edit`ing and `search`ing, will also be affected.

Developers use various techniques to understand such problems. For example, collected traces can be replayed against off-line versions of the system and statistical profiling can identify hot code-paths. These techniques, however, require some degree of *manual* effort: capturing traces, setting up testbeds, replaying traces, analyzing statistics, and debugging performance are all tedious and time-consuming tasks. Pervasiveness of third-party modules and heavy code reuse in modern applications compound the challenge, as the causes may lie deep in the dependency chain.

Detecting bottlenecks is not easy, but its effort is dwarfed by that of rewriting parts of an application to exploit distribution. Extensive code changes, orchestration of multiple jobs, service discovery, and scheduling over multiple replicas are all difficult and error-prone tasks, and must be repeated for every new bottleneck.

Light-touch distribution attempts to automate as much of this process as possible without requiring development in a new programming language or model.

2.2 Light-touch Distribution with Ignis

Ignis detects and scales out bottlenecked components by interposing on the application's module boundaries. It is introduced as a backward-compatible, drop-in replacement of the language's module system. It can be imported as an application-specific module (*e.g.*, `ignis` package) or can

```
1 let pbkdf2 = require("crypto").pbkdf2; //module
2 let slt = users[usr].salt; // usr, pswd via form
3 let h = users[usr].hash.toString();
4 pbkdf2(pswd, slt, 10000, 512, (e, d) => {
5  (h == d)? resp.send(200) : resp.send(401);
6 });
```

Fig. 3. Example bottleneck. pbkdf2 at the account-crypto boundary makes the crypto module a good candidate for scale-out (*Cf.*§2.3).

be bundled with a custom language runtime (*e.g.*, Ignis-powered Lua) replacing its system-wide module system. It starts by dynamically replacing the import function: instead of simply locating and loading a module, the function yields to Ignis, which applies a series of transformations to modules with the goal of interposing on their boundaries.

Transformations depend on several configuration details related to recipes, but can be coarsely grouped into three broad classes: (i) profiling and decision-making, (ii) spawning and distribution, and (iii) single-system retrofitting. Profiling transformations build a statistical model of module pressure (Fig. 2*b*) and rank candidate modules. Distribution transformations replicate bottlenecked modules, create communication channels among them, and balance load across all replicas (Fig. 2*c*). Single-system transformations selectively back-port the semantics of a single runtime. To guide the intended semantics (and associated trade-offs), developers annotate transformations with optional distribution recipes.

2.3 Ignis-powered Wiki Engine

To show how to apply Ignis on the performance problem outlined earlier (§2.1), Fig. 3 zooms into the authentication section of the account module: it imports the built-in crypto module (line 1) and invokes pbkdf2 (4) which, upon completion, calls a provided continuation function (5). Ignis augments require (1) to return a wrapper of pbkdf2. The wrapper monitors pbkdf2's calls at the account-crypto boundary. Upon load increase, it identifies pbkdf2 as a bottleneck and marks crypto as a candidate for scale-out.

To scale out, Ignis launches a few fresh replicas of the crypto module and starts spreading remote procedure calls (RPCs) among them. RPCs require serializing arguments, sending them to one of the remote replicas, and calling pbkdf2 there. Results are sent back to the account module, which passes them to the provided continuation.

Ignis also augments require to take a recipe as an additional, optional argument. A recipe σ at require("crypto", σ) (1) would constrain pbkdf2's scale-out. For example, a σ equal to {order:true} would have forced ordering semantics on calls and their results across all crypto replicas. Luckily, pbkdf2 is a pure function, which can be determined even in the complete absence of annotations. This exemplifies a case where light-touch distribution can obtain benefits even without any developer effort.

2.4 Simplifying Trends

Light-touch distribution is significantly simplified by the increasingly pervasive use of certain features today.

Packages and Modules Modules provide an implicit and fine-grained component architecture [26, 27, 35, 62] that applications can be partitioned across [18, 37, 63, 74, 91]. They encapsulate state behind small and tight interfaces, simplifying and minimizing transformations. Their boundaries clearly mark self-contained components that can be configured to execute remotely—including built-ins, such as crypto. Multi-thousand-module dependency graphs enable profiling and decomposition at a very high resolution, aiding bottleneck detection and memory consumption at scale.

Programming Styles Today's popular programming styles blur the line between local and remote execution. Cooperative concurrency grants scheduling control to the executing code, event-driven programming is naturally message-passing, and continuations enable (hidden) parallelism: independent continuations do not impose ordering constraints—two continuations λ_1 and λ_2 of sequenced calls $f_1(\ldots,\lambda_1)$; $f_2(\ldots,\lambda_2)$ can be interleaved in any order (otherwise, f_2 would have been included in λ_1). As such, components can be distributed across multiple nodes, even if there is no underlying single system image [3, 5, 9, 13, 49, 52, 94].

Dynamic Language Interpretation Dynamic languages have features—*e.g.*, name (re-)binding, value introspection, dynamic code evaluation, and access interposition—that enable runtime transformations [41, 42]. They conveniently unify module identification with interposition: a single function or function-like operator locates a module, interprets it, and applies transformations before exposing its interface in the caller context. As a result, monitoring and distribution can be performed at runtime and without forcing users into specific programming models [20, 60, 61, 68, 92].

3 System Overview

This section presents an overview of Ignis (§3.1) and outlines the structure of transformations (§3.2).

3.1 Transformations vs. Recipes

Ignis' responsibilities are divided between transformations and recipes, similar to the separation of mechanism and policy in the operating systems literature [50]. Transformations provide the mere mechanism for automating profiling and distribution, including creating remote references, copying structures, propagating events, *etc.* Semantics-related concerns are offloaded to recipes—*i.e.*, policies that encode developer knowledge about the behavior of modules.

Distribution Recipes Developers start by annotating selected imports with distribution recipes. Recipes are declarative runtime configuration objects, expressed using a domain-specific language embedded in the source language. They

```
p  ::=  s ∈ String  |  n ∈ Number  |  b ∈ Bool  |  ∅
v  ::=  p  |  (x,...) => {e}  |  {s:v,...}  |  [v,...]
e  ::=  x  |  v  |  (x = e) e  |  e(e)  |  e[e] = e
```

Listing 1. Module interface language, used in transformations. Modules return non-primitive values v, manipulatable via expressions e.

declare the intended semantics of the resulting distribution, tuning trade-offs that are fundamental in distributed systems [1, 30, 53]. For example, calls to a module may need to maintain ordering and changes to a module's state may need to be reflected across its replicas.

Profiling Once provided with a few recipes, Ignis starts monitoring the performance of the corresponding modules in order to detect opportunities for distribution. A key observation is the semantic isomorphism between calling a function and passing a message [48, 81]. This allows viewing a series of calls as a stream of messages. Module boundaries can be viewed as (virtual) queues of messages that await processing. Overwhelming a module causes its ingress queue to grow. At some point, the waiting time of newly-arrived messages becomes longer than the time to send the messages to a remote copy of the module, run the call there, and return the results back to their intended recipient.

Scaling Out Once this point is reached, Ignis attempts to scale out a module while selectively maintaining single-runtime semantics. Scaling out is achieved by spawning a module replica and replacing its local use with a thin client that disperses calls across all replicas. On each call, arguments are sent to a remote replica and results from the replica are returned to the thin client. The selection of which single-runtime semantics to maintain is tunable by distribution recipes, and implemented via additional transformations: converting local-memory pointers to meaningful distributed ones, forwarding side-effects such as memory allocation and collection, providing distributed versions of core built-in libraries, and enforcing ordering (when required).

3.2 Structure of Transformations

Transformations are used pervasively throughout Ignis, and are abstracted via a few parametrizable templates. Templates map different types of values (List. 1) to a generic handler for each type. Transformations have these handlers parametrized to achieve concrete goals such as monitoring, serialization, and scale-out. Simplified instances are described in the following two sections (§4–5).

Transformations can be applied to any value in the language, such as an object returned from a module or an exception about to be sent across the network. The general case of such a value is a directed acyclic graph (DAG). The types of its vertices can be coarsely grouped into primitives, functions, and objects. Objects map strings to other values, pointing to other vertices in the DAG. Transformations start by walking the DAG from the root vertex and processing component values based on their types. They do not mutate original values, but first copy them, apply transformations to the copies, and return copies to the caller. They only partially explore the object graph, as they do not peak through function closure environments. Fortunately, this aligns well with our goal of monitoring activity at module boundaries and ignoring module-internal activity.

As an example, consider transforming the `crypto` module (§2.3). Ignis traverses the object returned by `crypto` and replaces functions such as `pbkdf2` with wrappers whose specifics depend on the intended goal: profiling wrappers call the original function in between statistics collection, and RPC wrappers forward the call to a remote replica.

4 Decision-Making

The task of monitoring performance and detecting opportunities for scale-out is logically split into (i) a decentralized set of profiling agents that operate at module boundaries (§4.1), and (ii) a centralized coordinator that builds a holistic understanding of what—and when—to scale out (§4.2).

Profiling is accomplished by wrapping module interfaces with logic that generates a model of the current workload. Each module boundary collects its own statistics based on a combination of recipes and instructions from the coordinator. Profile generation can operate at a high resolution in time and space: (i) at every function call entering a module, and (ii) on thousands of modules across an application.

The coordinator oversees all profile-generation agents, collects periodic summaries from them, and ranks their needs. It is also responsible for creating a map of available resources and checking their status and health. While coordination operates at a lower resolution than profile generation, it allows monitoring some boundaries more closely than others.

The division of labor in deciding when to initiate scale-out is somewhat delicate. Agents running at the boundaries should not depend on the coordinator for online decision-making, as after a first scale-out they may be executing on different nodes. As such, they should have enough logic to make an online decision. To solve this, the coordinator pushes periodic guideline updates to the agents, which merge them with their local configurations (*e.g.*, module-specific recipes that override default/global recipes).

4.1 Profile Generation

Profiling transformations insert agents modeling queues at the module boundaries. Observing queue metrics such as arrival rate and wait time, agents build an understanding of the pressure applied at their boundary.

Transformations Profiling transformations focus on function (and function-like) values (Fig. 4), for which they generate and attach code that monitors calls and returns. Specifically, each vertex in the DAG returned from a module is recursively replaced with a wrapper:

```
ptf (e: DAG) : DAG := match e with
| Obj ((s, v) :: xs) -> Obj ((s, ptf v) :: ptf xs)
| Arr (v :: vs)      -> Arr ((ptf v) :: ptf xs)
| Fun f -> Fun (args) => {nq(); f(args); dq();}
| __                 -> toStats(e)
end
```

Fig. 4. Profiling transformation. Functions are wrapped with prologue (nq) and epilogue (dq) operations that record statistics (*Cf.*§4.1).

- function values are wrapped in functions that add a prologue/epilogue pair recording profiling data.
- mutable values have their getter and setter methods similarly wrapped with prologue/epilogue wrappers.
- values of all other types are left unmodified.

Ignis now mediates between parent and child modules. Wrappers record statistics about the their encapsulated functions as well as queue characteristics of outstanding calls.

In the cases of non-blocking (*i.e.*, asynchronous) interfaces, the prologue includes code for wrapping the continuation argument (*i.e.*, callback function) before passing it to the encapsulated callee; the continuation wrapper records metadata similar to the case of returns for blocking (*i.e.*, synchronous) interfaces. As function invocations may support reentrant concurrency (*e.g.*, `fs` module), marks are added (§5.2) to match prologues with their respective epilogues.

Statistics The wrapper epilogue has access to several raw metrics related to profiling (Tab. 1). These metrics need to be composed into a model that allows Ignis to decide whether to scale out a module.

A simple idea would be to detect when the number of concurrent requests ϱ exceed the number of replicas R. When this happens, assuming R does not exceed the number of CPUs P, Ignis could start a fresh new replica:

$$R_{new} = \begin{cases} R + 1, & \varrho > R \wedge R < P \\ R - 1, & \varrho < R \end{cases} \tag{1}$$

This approach omits a few important issues. First, we would like to model and account for the overheads of scaling out. These overheads involve context switching, round-trip times, and other systemic overheads δ, as well as one-off startup costs δ_0 when spawning a replica (§8). Second, as some operations (*e.g.*, slow I/O) are intrinsically concurrent, we would want to allow for at least some concurrency before paying the cost of scale-out. The available room for concurrency, however, is not visible at the level of individual boundaries; thus, agents can model a virtual queue by taking a windowed, weighted average of wait-times l_i:

Tab. 1. Example metrics. Module boundary agents see only (high-frequency) local metrics; coordinators receive weighted summaries but have end-to-end visibility across the system (*Cf.*§4.1).

Metric	Module Agent	Controller
Arrival Freq.	Call Freq.	Arrival Th/put
Queue Size	Queued Items	Avg. size
Processing Time	Downstream Latency	Lifetime
Call Type	Func/Method/Prop. Access	Call Ratios
Failure Ratio	Exceptions	Summary
CPU Number/Type	Locally available	Total
etc.	*etc.*	*etc.*

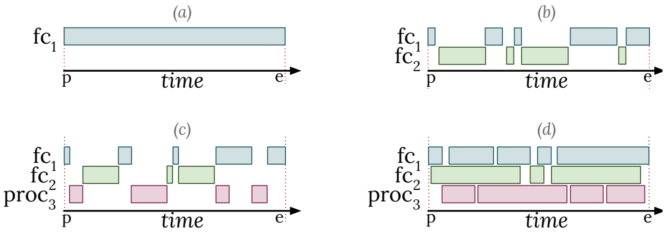

Fig. 5. Accounting accuracy. Only observing the prologue-to-epilogue timings $e - p$ for a function fc_1 does not allow distinguishing among (a) uninterrupted function call, (b) concurrent interleaving with another call, (c) descheduled in favor of a different process, (d) parallel execution (*Cf.*§4.1).

$$R_{new} = \begin{cases} R + 1, & \varrho \times \sum_{i=1}^{t} w_i l_i > \delta + \delta_0 \\ R - 1, & \varrho \times \sum_{i=1}^{t} w_i l_i < \delta + a * \delta_0 \end{cases} \tag{2}$$

Scale-in (*i.e.*, −1) might not seem as important, but constrained environments benefit from quick re-allocation. However, after scale-in, the system often ends up just scaling out the same module. To avoid such oscillation, a small reclamation delay a increases the system's confidence that the workload has moved away from a specific pattern.

Our prototype (§8) uses a combination of eq. (1) and eq. (2). Eq. (1) is used for short-running processes where there are not enough samples to feed the weighted average.

Challenges A few details on call styles, exception handling, and the module cache are worth noting.

In the case of blocking interfaces, no items will be arriving at the boundary before previous items finish executing. As such, there is no meaningful notion of queues (*i.e.*, queues will always have zero items). Instead, Ignis wrappers calculate windowed averages over *individual* past calls, without modeling concurrently pending ones. If a blocking interface is marked as a potential bottleneck, Ignis will suggest conversion to a concurrent, non-blocking version as an intermediate step before distribution; runtime transformations (§5) generate and link the new interface automatically.

In the case of non-blocking interfaces, functions will be called as soon as items arrive. Gathering accurate statistics for individual calls can be challenging, because OS-internal queuing and reordering is not visible to the boundary wrapper. Specifically, a prologue–epilogue time interval can mean any one of several scenarios (Fig. 5). Fortunately, as services are deployed for some time prior to saturation, Ignis has the benefit of collecting accurate long-term runtime statistics. The problem is further alleviated by Ignis's ability to distinguish the concurrent from the non-concurrent case, by checking the number of concurrently pending calls in the function wrappers.

In cases of runtime exceptions, the control flow bypasses the epilogue, skipping statistics collection. For these cases, Ignis wraps the encapsulated function call with an exception handler that calls the epilogue and rethrows the exception.

For consistency and performance purposes, module systems maintain a cache of loaded modules. When an existing module is imported again in a different part of the codebase, they return a cached reference to the original module. To precisely attribute load to the right bridge between modules, Ignis' profiling transformations return a *fresh* wrapper function upon each load. For example, if modules A and B import C, B may be placing 10× more load on C than A does. Such a 1:M function mapping does not affect module consistency and has negligible effects on performance (except in cases of meta-programming (§5.2)) but offers noticeable improvements to load attribution.

4.2 Application-wide Coordination

Ignis starts by setting up an application-wide coordinator—a logically centralized control hub that builds a registry of the available resources, interfaces with the agents at module boundaries, analyzes distribution recipes, and pushes guidelines to the boundary agents. Ignis daemons, cut-down versions of the coordinator, execute on other hosts that be used for scale-out, reporting on local resources, listening for replication requests, and transforming replica interfaces (§5).

Resource Registry On every node, coordinators poll the underlying environment for software and hardware information. Newly configured daemons report this information up towards the parent coordinator.

Information on the software environment focuses on the operating system, language runtime, and various built-in libraries. Among other reasons, this is important for modules compiled to execute natively as well as module dependencies that need to be installed globally. For example, imagine a module that needs root permissions but happens to not be available on a specific node. As Ignis cannot set up this module during runtime, it will not be able to replicate and schedule calls to that module on this particular node.

Hardware information includes memory configuration, CPU speed, and bus speed. Information about the characteristics of the network (*e.g.*, latency) is continuous, and extracted periodically from the performance of call traffic.

Boundary Registration After import (§2.2) and transformation (§4), the newly transformed boundary registers with the coordinator. The rewired `require` function notifies the coordinator with a message that includes identifiers of the parent and child modules, pointers to the original and transformed DAG handles, and a pointer to the local recipe.

The new boundary is added to a list of boundaries monitored by the coordinator. The list is ranked by need-to-replicate, recalculated with every new update the coordinator receives from a boundary. The calculation focuses on the fraction of the execution time taken by each module, and is updated at a frequency set by the coordinator.

If the update leads to changes in the list, boundary agents whose ranking changed are notified. This notification contains guidelines that allow modules to make online decisions, including the minimum and maximum number of replicas and the threshold values of the formulas (1) and (2).

Load Attribution Agents running at each boundary cannot "see through" modules in order to attribute load among a module and its dependencies correctly. For example, an agent at the outermost boundary A of a dependency tree A⟶B⟶C does not know how much of the latency comes from B.

To solve this problem, Ignis relies on the coordinator, which understands the structure of dependencies and can correctly calculate how much of the latency comes from each module. Such a calculation is more complicated than a simple subtraction, as a module may invoke interfaces from multiple modules at the same time. The technique of creating 1:M wrappers (end of §4.1) alleviates much of the problem.

5 Distributing Modules

Transforming a system into a distributed version amounts to scaling out individual bottlenecked modules (§5.1) while selectively maintaining the illusion of a single runtime (§5.2).

5.1 Scaling Out

To scale a module out, Ignis (i) creates a new process importing the module and Ignis-specific libraries, (ii) sets up a communication channel between the old and new process, and (iii) schedules calls across all the replicas.

Setup Ignis spawns a new operating system process on a node that fits certain criteria, such as light load, acceptable latency, and compatible versions of packages.

The new process binds to a fresh ($IP, port$) pair, used both as a communication handle and as a unique node identifier. Nodes communicate over TCP even for processes colocated on the same machine, as TCP is system-agnostic and hides the distinction between local and remote communication. The newly spawned node first loads a copy of Ignis to (i) set up the channels and, when needed, extract characteristics of the hardware, (ii) interface with the coordinator on the parent process (or higher in the module/process hierarchy), and (iii) further respond to increased load *within* that module.

Transformations Scale-out transformations focus on replacing a local module with a thin client that forwards calls to a set of remote modules (Fig. 6). To create a thin client of the same type as the original module, Ignis inspects the DAG

```
dtf (e: DAG) : DAG := match e with
| Obj ((s, v) :: vs) -> Obj ((s, dtf v) :: dtf vs)
| Arr (v :: vs)      -> Arr ((dtf v) :: dtf xs)
| Fun f              -> toRPC(e)
| __                 -> toInterposed(e)
end
```

Fig. 6. Distribution transformation. Ignis' toRPC function takes a function f_1 and returns a function f_2 that, when called called, sends the arguments to f_1 and calls it. Primitives are wrapped with interposition (*Cf.*§5.1).

returned by the import call in the new process. It recursively replaces each node in the DAG with a wrapper:

- primitive values are wrapped with an interposition mechanism that records and propagates changes.
- function values become RPC stubs that serialize arguments, send them via the channel, and collect results.
- mutable values have their getter and setter functions replaced with RPC stubs similar to functions.
- exceptions are re-thrown in the parent context after inspection from Ignis running on the parent module.

Ignis maintains a distributed map from module identifiers to (a set of) channel pointers. If a module is already loaded, Ignis retrieves the channel pointer and returns the previously-wrapped DAG; this is useful, for example, in cases where dependencies have a diamond shape (*i.e.*, two different modules import the same module).

Scheduling Unless instructed otherwise (§6), calls to replicas are scheduled in a round-robin fashion.

Blocking calls yield to the Ignis scheduler, which picks a replica, serializes given arguments, sends them through the channel to the chosen replica, and waits for a response. The child-side wrapper de-serializes arguments, calls the required method, and sends results back through the channel. For non-blocking calls, the parent module wrapper registers an event listener that invokes the provided continuation when results become available on the channel.

Challenges A few technicalities on asynchronous replica spawn and module resolution are worth noting.

Spawning a new process takes several tens to hundreds of milliseconds (§8), which should be off the critical path—especially at the point in the execution of the program when Ignis is in utmost need for performance. Thus, Ignis spawns each module replica in asynchronous, non-blocking mode while calls are served by the original module. When the replica completes initialization and is ready to handle calls, the Ignis coordinator on the child sends a message with all the information described in §4.2 to the parent process.

To be able to locate modules in the new environment, Ignis needs to patch the module resolution algorithm at the replica. For replicas running on the same physical host, locating a module is relatively easy: absolute modules are stored in well-known locations in the environment (*e.g.*, Python's sys.path), modules local to the project are stored within the application (*e.g.*, JavaScript's node_modules), and modules relative to the current location are prefix-resolved at runtime, by having Ignis prefix the module path appropriately.

Replicas running on different hosts require a more complex runtime resolution, which is avoided by prefetching modules. To prefetch modules, the dependency chain is analyzed upon startup by the coordinator (§4.2). Project-local modules are extracted at startup and pre-fetched upon daemon setup. While some disk space is wasted as the majority of these modules will not be used, latency on the critical path is avoided. Absolute modules are resolved similarly and re-introduced as project-local modules on the new host. Modules relative to the current location either work unmodified (first replica on a host) or are prefix-resolved by Ignis (subsequent replicas).

5.2 Maintaining (the Illusion of) a Single Runtime

This section describes several techniques related to transformations intended to maintain the original application behavior. Whether each one of these techniques is required or not is a recipe-specific question, discussed in (§6); here, we merely show how Ignis implements each technique.

The techniques below require additional metadata to be attached on the serialized value. In Ignis, this is achieved by adding a new "hidden" __ignis property instead of embedding the entire value in another message.

Distributed References To facilitate cross-replica addressing, transformations at the replica boundary assign identifiers to non-primitive values. These IDs can be viewed as distributed, shared-memory pointers which RPCs include in their messages. Replicas then maintain a "decoding" hash table, mapping IDs to their in-replica pointers: whenever they receive a message, replicas use the table to route freshly deserialized values to the right function (or method).

Generating a fresh ID requires that the new pointer is different from all other pointers; otherwise, two pointers refer to the same location and should be assigned the same ID. This is achieved by maintaining a second "encoding" hash table from non-primitive values to IDs. In constant time, Ignis checks if an entry already exists (extracting the associated ID) or not (inserting a new (val, ID) entry).

The creation of copies during transformation and serialization breaks reference equality. To solve this, when an RPC leads to a new memory alias in a replica, Ignis attaches an alias entry to the serialized value. When receiving such a value, Ignis on the child will create and return a reference to an existing object. The sender side uses the "encoding" map to assign a distributed pointer to an object.

The same consideration applies to preserving reference equality for the root of the DAG between RPCs. A common pattern in many languages is to have methods that return self; such code would break if the return value of the RPC was a fresh copy of the method receiver.

Call Types Constructors, prefixed by new, are different from typical functions (*i.e.*, memory allocation outlives the function call). Ignis inserts additional logic into the RPC stubs to detect this case,[1] and augments RPC messages to signal that the target functions should also be called as constructors. The return value from a constructor is in turn transformed into an object whose methods are RPC stubs (§5 and Fig. 6): the true object lives within its own replica.

Similarly, standard garbage collection (GC) cannot "see through" boundaries. To solve this, Ignis also propagates garbage collection events by adding GC hooks to every object that is the result of a transformation. These "weak finalizers" fire when the object is about to be collected, causing Ignis to broadcast a message for removal of that object from any auxiliary tables. With no inbound pointers, objects in replicas will be collected in the next cycle.

Although state updates via method calls are redirected to remote objects, direct updates via property values require custom detection and propagation. Ignis wraps the transformed output DAG with an interposition mechanism that provides reflection capabilities and gets invoked upon property accesses.[2] This wrapper detects and records changes to any of the object's properties. Further nested wrappers monitor nested objects (Fig. 6). A similar mechanism is used to detect program- or user-initiated invalidation in the module cache (*e.g.*, to reload a module): a cache wrapper detects and broadcasts entry invalidation.

Although unusual, modules other than Ignis may dynamically rewrite module interfaces. If Ignis is loaded earlier, these modules will attempt to overwrite the RPC stubs instead of the encapsulated methods. The DAG interposition wrapper described above detects accesses and applies rewrites internally. Such rewrites are simplified by not having to cross channels, as they occur long before scale-out.

Environment Binding Generally, names defined by the programming language or standard libraries are valid on all replicas: language constructs such as the top-level Object are implemented in the runtime; stateless libraries such as crypto and location-agnostic OS-wrappers such as net are made available unmodified within a replica.

Certain global or pseudo-global[3] constructs may require redirection to the top-level process. For example, a replica's out and error streams must appear in the respective streams of the top-level process. Ignis on each replica transforms and shadows these methods with RPCs that redirect their arguments to the top-level process.

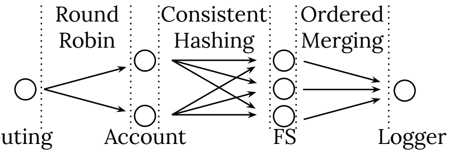

Fig. 7. Effects of recipes. Call distribution across replicas of the wiki's authentication submodules, as a result of recipes (*Cf.*§6).

Other built-in libraries are replaced by scalable, distributed versions. This replacement is achieved by transforming the DAG of the specified built-in module on every replica with functions that call into the distributed version. A notable example is the fs module, with a large and well-explored space of available trade-offs. At one end of the spectrum lies a partitioned fs with strong consistency guarantees that redirects accesses to a single authoritative node. At the other end of the spectrum lies a replicated fs that distributes accesses in an eventually consistent fashion across a subset of nodes, using a consistent-hashing scheme [39, 82].

Finally, objects may invoke methods inherited from classes higher in the hierarchy. These superclasses—or prototypes, for prototype-based languages such as Lua and JavaScript—may have been imported from a different module. A naive implementation of transformations to RPC stubs can thus lead to a series of nested round-trips until a call reaches the correct destination. Ignis detects class (prototype) hierarchy levels while traversing the DAG and creates a dispatch table with RPC stubs that route calls to the final destination.

Maintaining Ordering Although communication primitives across a single edge are reliable and in-order, messages that cross multiple edges may arrive out of order. To maintain ordering, serialized values are assigned an internal sequence number. Sequence numbers are generated at the call site and follow the value as it travels through the system. In cases where the value is changed or replaced by a new one, the sequence number is extracted and transferred along.

6 Distribution Recipes

Recipes are runtime configuration objects that give developers the ability to tune several trade-offs related to the resulting distribution [1, 30, 53] without requiring manual development. They are responsible for generating transformation parameters and configuring deployment details. The latter is important in order to identify which nodes can be used for scaling out a module. The large number of modules and their different requirements make this challenging (but we do not discuss it in more detail here).

Fig. 7 shows the (semantic) result of annotating the wiki's authentication subsystem (§2.3) with recipes. Module routing spreads calls homogeneously over the two account replicas. Module account spreads consistently over fs—that is, identical arguments at any of the account call sites will hit the same fs node. Module logger orders calls to avoid mixing logs from different requests.

[1] For example, __call metamethod in Lua and new.target in JavaScript.

[2] For example, metatables in Lua and Proxy objects in JavaScript.

[3] JavaScript implementations introduce objects that are not part of the EcmaScript specification into the global scope, such as process and console. Similarly, Lua's Luvit introduces its own globals, such as p() and exports.

Ordering highlights some interesting features. First, as it degrades performance, it can be enabled only for a limited amount of time—benefiting from recipes being *runtime* constructs. Second, order is not enforced for direct accesses, such as assignments that change logging levels. While stronger consistency can be guaranteed with different recipes, the developer here expects logger levels being set only once and probably at the start. This is a case where causal consistency between assignments and calls (but not between calls) can be exchanged for strong eventual consistency.

Writing recipes is equivalent to specifying a system's distribution properties. Reasoning about them, rather than the mechanisms by which these properties are implemented, reduces the chance of errors while scaling out a system.

Recipe Expressions Recipes can be expressed at the level of a program, propagating down to the rest of the dependency chain, or at that of individual modules. Ignis' built-in recipes are overridden by program-level recipes, which are in turn overridden by recipes accompanying individual modules.

System-wide recipes describe how to configure the distributed system, and include: profiling details such as queue depths and saturation levels, decomposition limits such as replica counts and module groups, expected semantics of augmented built-in libraries (*e.g.*, `fs`), and module priorities, such regex patterns for modules that should or should never scale out. Users also need to provide the details of the daemons running on remote nodes; Ignis can then configure the details by querying the daemons. Here is a typical program-wide recipe, extracted from an experiment:

```
1 nodes:[{ip: "128.30.2.133",  port: 8013}],
2 fs: ignis.fs.EVENTUAL,
3 cold: [/process/],
4 hot: [/ejs/, /.*dash/]
```

It configures Ignis so that an additional node can be used to launch replicas (1), the standard `fs` module is replaced by an Ignis-provided, eventually consistent one (2), the `process` module should not be replicated (3), and `ejs` and `lodash` should be distributed by default (4).

Module-specific recipes give developers fine-grained control over distribution, allowing them to express intuition about individual modules they import:

```
1 copies: [2, 10],
2 fs: ignis.fs.LOCAL,
3 order: false,
```

This module should inherit the global recipes specified earlier. Moreover, it should have a minimum of two and a maximum of 10 replicas running (1), use the local `fs` on each node (2), and not need any ordering (3).

Discussion Tab. 2 summarizes more recipes, a few non-obvious characteristics of which are worth clarifying.

Being dynamic objects, recipes are flexible. They can be re-generated at runtime and change during the lifetime of the program—even between different imports of the same module. For example, different branches of the control flow can load the same module with different recipes.

Tab. 2. Ignis recipes. Selected recipes and their default parameters (*Cf.*§6).

Recipe	Options	Default	Explanation
Nodes	`[{ip:. . . }]`	`localhost`	Nodes running Ignis
Level	`0, 1, ..`	`1`	Decomposition depth
Group	`subtree.json`	—	Group module subtrees
Hot	`["util"]`	`parse, crypto`	Always distribute module
Cold	`["fs", "os"]`	`process, fs`	Never distribute module
Copies	`true, [2, 8]`	`[0, CPU]`	Multiple replicas
Sched	`RR, MLF, Weigh`	`RR`	RPC scheduling policies
Q_Depth	`10--100K`	`Inf`	Inter-module queue depth
On_Fail	`(e) => {..}`	`throw`	Module failure hook
Saturation	`(lvl) => {..}`	—	Saturation level hook
Comm	`TCP, UDP`	`TCP`	Communication type
Preload	`true, false`	`false`	Load proactively, not lazily

Currently, Ignis defaults to recipes that are conservative, in the sense that they will never attempt distribution that breaks the semantics of the program. For example, purely functional built-in libraries such as `parse` and `crypto` default to permitting distribution, but `process` and `fs` do not.

Since recipes affect the semantics of the resulting program, an obvious question is whether developers can get them wrong. The answer is yes, but this is no worse than other approaches that aid distributed programming: for example, developers are free to introduce side-effectful computations in MapReduce's purely functional primitives [20]. Ignis makes reasoning about semantics easier, as it concentrates the decisions that could break semantics into the recipes, rather than forcing them to be interleaved with application logic. Our position is that there is *much* more room for error by avoiding aid from tools like Ignis and MapReduce altogether and building distributed systems from scratch.

7 Implementation

We built a prototype of Ignis for the JavaScript ecosystem, consisting of 3K lines of JavaScript code atop Andromeda [87] and available via `npm -i @andromeda/ignis`.

Andromeda is a distributed overlay environment provided as an extensible library of distributed services. Example services include distributed storage, inter-node communication, and task orchestration. Ignis builds on the primitives provided by Andromeda to implement monitoring, adaptive scaling, and wrappers for built-in libraries such as `fs`. Andromeda can be executed atop any JavaScript runtime, but on Unix it defaults to Node.js [19], a runtime that bundles (i) Google's V8, a fast JIT compiler, (ii) libUV, asynchronous cross-platform OS wrappers, and (iii) a small set of standard libraries (*e.g.*, `crypto`). For every node in the distributed system, Andromeda spawns a separate userspace process.

Applications interact with Ignis via JavaScript's built-in `require` function. Recipes are encoded as JavaScript objects and passed as the second parameter. Due to variadic arguments, recipe-infused codebases remain backward-compatible with Ignis-less runtimes.

8 Evaluation

At a high level, we are interested in understanding four aspects of Ignis: (i) the overheads of different Ignis elements, (ii) the fidelity of its runtime profiling, (iii) its behavior under load, and (iv) its benefits compared to manual application distribution. For the first, we use microbenchmarks that stress different parts of Ignis (§8.1). For the rest, we use a combination of synthetic (§8.2) and real (§8.3–8.5) applications.

Several takeaways are worth highlighting. Scaling out with Ignis can require minimal code changes, less than 0.001% of a complex codebase (§8.5). Even for simple applications and scaling goals, this represents 10-20× less effort than manual approaches and avoids the introduction of bugs (§8.3). In a case study where we scale a web crawler to better utilize a 60-core host, Ignis leads to speed-ups of 27× (§8.4). Aside from speed-ups, Ignis reduces memory requirements by up to 11× compared against typical whole-application replication, by only replicating bottlenecked components (§8.5).

Experiments were run on a network of five workstations connected by 1Gbps links: one large machine (a_1) with 251GB of memory and 64 2.1GHz Intel Xeon E5-2683 cores, and four smaller machines (q_{1-4}), each with 4GB of memory and two 3.33GHz Intel Core Duo E8600 processors. No special configuration was made beyond disabling hyper-threading—specifically, the network protocol stack was left unoptimized.[4] For our software setup, we use Node.js 6.14.04, bundled with V8 v.5.1.281.111, libUV v.1.16.1, and npm v.3.10, executing on top of Linux kernel v4.4.0-134.

8.1 Microbenchmarks

No-op Modules To understand Ignis' inherent startup and communication costs, we run a few microbenchmarks comparing Ignis with the unmodified module system as a function of the number of replicated modules. We remove orthogonal network concerns by running experiments locally on a_1, and minimize the effects of module sizes and transformations by creating "no-op" modules.

Tab. 3 rows 2 and 3 show the base startup time of modules that return a single integer. Ignis modules incur significantly higher startup times, but these overheads are amortized as the number of modules increases.

Vanilla JavaScript loads modules sequentially. However, Ignis-replicated modules can amortize their (much more expensive) startup costs by leveraging parallelism—an idea that (at least in part) motivated asynchronous spawning (§5.1). Asynchronous spawning raises concerns regarding interference with the main process on a single host. To investigate this, we start 5K modules in parallel (total time: 15.4s, avg: 3ms/module) while a main module "mines" SHA512 hashes. Hash rate, about 5.4 MH/s, essentially remains unaffected.

Tab. 3. Replication overheads. Replica (i) startup (rows 2, 3) and (ii) communication (rows 4–8) costs for different numbers of replicas (*Cf.*§8.1).

Module Replicas (Single Host)	5	50	500
Startup Overheads:			
Latency (Unmodified)	12.9ms	106.6ms	824.7ms
Latency (Ignis)	342.5ms	1.4s	6.2s
Communication Overheads:			
Latency (Unmodified)	6.5ns	90.18ns	294.3ns
Latency (Ignis)	27.15ms	388.55ms	11.05s
Throughput (Unmodified)	192.3GB/s	157.1GB/s	46.5GB/s
Throughput (Ignis)	158.1MB/s	134MB/s	20.9MB/s

Tab. 3 rows 5–8 show the cost of communication (remote invocation), by processing an in-memory stream of 1GB across a series of modules. The stream starts only after all connections have been established (*i.e.*, no TCP handshake costs included). As expected, copying the payload (rather than passing pointers) is significantly more expensive.

Real Modules To understand startup and communication costs on real modules, we run single-replica experiments on a diverse set of popular modules under three configurations: (i) vanilla, (ii) co-located distribution (as before), (iii) networked distribution. Their source-level aspects (Tab. 4) affect startup times, whereas call argument sizes affect invocation times.

Fig. 8 shows module startup costs. Distribution requires transforming modules (RPC stubs), communicating them to the remote node, and launching replicas there. The startup time of larger modules is dominated by file-system accesses (*e.g.*, `cash` takes 798.2–1049.1ms to load all of its files). The startup time of smaller modules, such as `verbs` and `pad`, is dominated by constant factors (*e.g.*, 138.5ms for process spawn and 17.6–35ms for TCP setup).

Fig. 9 shows inter-module communication costs over the same configurations. To bring these costs into perspective, we embed modules in "no-op" HTTP applications. By placing these modules behind an HTTP server, we can study Ignis' impact on the end-to-end latency an HTTP client would experience. Processing-heavy modules such as `pad` and `nacl` are tested under input of size 5B (`S`) and 5MB (`L`).

To understand the costs of boundary interposition, we measure the time to access deeply-nested properties of two versions of an object: unmodified and proxy-wrapped. Paths to the properties (*e.g.*, `a.b.c....`) are random but generated prior to running the experiment. We construct 500MB-sized objects, each with a fanout of 8 fields (DAG child nodes) nested for 12 levels. The proxy-wrapped version introduces interposition at every level. Traversing one million 12-edge paths (*i.e.*, root to leaves) averages 167.2ms and 595.7ms for the unmodified and proxy-wrapped versions, respectively.

Towards Practical Overheads The micro-benchmarks presented in this section highlight Ignis-inherent overheads

[4] Features such as kernel bypass [71] should yield significant improvements.

Tab. 4. Characteristics of the benchmarked modules. Module shorthand, size in terms of lines of code (LoC), number of files (Files), number of nodes in the import return object (DAG), depth of dependency graph (DD), average fan-out of the dependency graph (F-O), number of function values (fs), overview of its internals (Notes) (*Cf.*§8.1).

Module	LoC	Files	DAG	DD	F-O	fs	Notes
verbs	29	1	28	2	27.0	0	constant string-to-string map
pad	52	1	1	1	1.0	1	small, pure function
cash	451725	10839	75	7	314.0	49	large library with system calls
chalk	145706	9630	5	3	5.3	2	builder objects/cascading calls
debug	554746	8657	34	4	51.3	14	varargs; output to parent stream
ejs	59396	4950	25	4	12.0	11	extensive, pure, testing fixtures
dns	4826	1	60	3	34.0	16	built-in module, async calls
nacl	94686	5387	54	5	40.8	42	CPU crypto processing

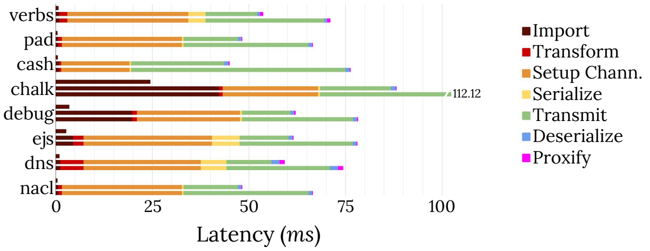

Fig. 8. Startup latency breakdown. Each module is measured under three configurations: unmodified (short bar, only `import`), co-located on the same host (medium bar), and scaled out across the network (*Cf.*§8.1).

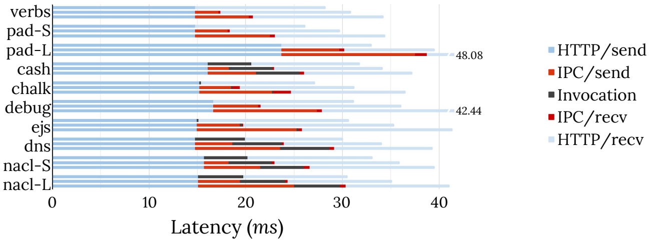

Fig. 9. Call latency breakdown. Same configurations as Fig. 8 (*Cf.*§8.1).

by using carefully-constructed, worst-case workloads that are nowhere near the ones seen in practice (§8.3–8.5): 1GB-sized function-call arguments, deeply-nested 0.5GB-sized module interfaces, and near-zero module-internal latencies (that in practice would not lead to scale out).

8.2 Synthetic Applications

To better understand Ignis' profiling and distribution in a controlled environment, we craft a four-module application with pre-defined bottlenecks. The application can perform one of three operations, depending on its current state: (i) invoke a call to a module it imports, (ii) busy-wait, or (iii) accept calls. State transitions are controlled by a probability distribution; and modules can introduce additional delays. The application accesses modules `m1` and `m2` with equal probability; `m1` accesses `m3` and `m4` similarly, but `m2` accesses `m3` three times more often than it does `m4`. The four modules add a delay of 10ms × their depth in the dependency graph. The application is part of Ignis' testing infrastructure, encoding well-understood scenarios with the goal of verifying that scale-out follows the developer's intuition.

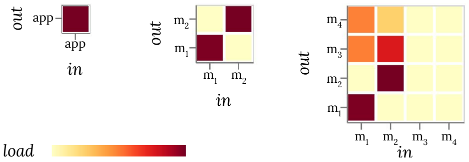

Fig. 10. Boundary pressure. The dependency graph of an application with four modules is presented as an adjacency matrix at different granularities. Instead of a single bit, cells of adjacent modules are labeled with the heat (load) of the boundary they represent, as recorded by the Ignis coordinator. The diagonal represents pressure internal to a module (*Cf.*§8.2).

Fig. 10 depicts the importance of fidelity in load attribution. Load is visible at the application level (left), but the lack of detail does not help determine which modules must be scaled out. Module-level data, collected by Ignis' coordinator (right), reveal that `m3` receives the majority of the load.

Manually deciding which module to scale out would require installing `linux-tools`, setting up `perf` [32], resolving JavaScript symbols with V8's `--perf-basic-prof`, recoding event samples, mapping samples to modules (using stack information), and visualizing results. Many of these steps would need to be repeated for every new bottleneck and, after scale-out, combine results from multiple replicas.

Ignis reaches the decision to scale out within a few hundred milliseconds, primarily because of lack of confidence due to cold-start effects. By observing long inter-module queues and a sufficient number of idle processors, Ignis launches three replicas of `m3` at once.

8.3 Macrobenchmark: A (very) Simple Weblog

Scaling out a system is often as easy as upgrading to a readily-available distributed storage system. This is the simplest case of bolt-on distribution, because it does not require thinking deeply about the structure of the computation. To compare with such minimum-effort rewrites, we plug a distributed key-value store into a simple blog application.

`KoaBlog` [36] (commit 1fd5316) is a small application for learning the `Koajs` web framework. The blog manipulation code totals 50 lines and imports six direct dependencies, for a total of 160 packages and 96 KLoC. Entries are indexed by an integer ID. They are stored on disk using methods from the built-in `fs` module (*e.g.*, `write`, `readdir`).

Even though `KoaBlog` is a trivial application, the manual effort required to use a NoSQL system such MongoDB [14] (v.3.4.10) is considerable. We first use `npm` to download and import the `monk` module for interfacing with MongoDB, similar in effort to importing Ignis. We then create a database schema, and configure the connection to the master node—including details such as binding address, port, and username. We also remove the import of the `fs` module, and rewrite all file-system operations such as `read` and `write` to make use of `monk`'s `find` and `insert`. All the above is expressed in JavaScript and within `KoaBlog`, in a diff that totals 11 lines (22% of end-developer code). However, it omits lines typed in the MongoDB and Unix shells (outside `npm` and requiring `sudo`): fetching and installing MongoDB, configuring administrative users, setting up one master and two slave nodes, and connecting MongoDB's startup with `KoaBlog`.

Adapting `KoaBlog` also required fixing two bugs that we unintentionally introduced. First, by replacing *all* `fs.writes` with `inserts`, we broke updates; `fs.writes` corresponding to updates should have been replaced by `updateById`, not `insert`. Second, we misconfigured the binding address and port of one node. These bugs were not difficult to fix, but illustrate the inherent dangers of manually scaling out an application, even when the modifications *seem* straightforward.

In contrast, with Ignis this scale-out requires a short recipe at the `fs` import: `require("fs", {copies: 3})` (aside from downloading and importing Ignis, à la `monk`). The recipe specifies a minimum and a maximum number of three replicas, similar to Mongo's setup. Ignis launches three replicas of the `fs` module (see "Environment Binding" in §5.2 for built-ins), traverses the return DAG of the original `fs`, and rewires methods such as `readFile` and `writeFile` to call Andromeda's distributed storage equivalents. Node.js's `fs` methods take optional arguments such as encoding (*e.g.*, UTF-8) and access mode (*e.g.*, `RW`). As Andromeda stores objects (rather than files), Unix flags such as `RW` were initially a concern; however, existing support for UTF-8 was enough to avoid breakage.

To evaluate performance, we pre-populate `KoaBlog` with 10K posts of 1.1MB each, and issue a 2-minute HTTP `GET` workload of 5K req/s. To saturate disk bandwidth and "force" Ignis to scale out, we initiate three local, parallel, long-running `cp` commands in the background. Distribution improves request latencies significantly (table below) as well as request throughput and transfer rates: 24 requests per second (28.2MB/s) become 88.22 (103.5MB/s). No significant difference between Ignis and MongoDB was noticed; this is expected because no advanced indexing, replication, and consistency features were used, where the two diverge.

Percentile	50%	90%	99%	99.9%
Baseline	154ms	1028ms	4731ms	8905ms
Ignis	65ms	231ms	318ms	1187ms

These performance improvements are on top of the base benefits of storage distribution—increased capacity (combined, q_{1-4}'s disk capacity is over 1TB) and availability (3× replication). (This availability is different from the fault-tolerance of Ignis *itself*, which is left for future work (§10).) The introduction of Ignis does not impact the memory consumption of the main node. This is because the `koa-*` modules dominate, whereas the newly-introduced `ignis` module is comparable in size to `fs`.

8.4 Macrobenchmark: Document Ranking

For a more complex application, we consider a custom natural-language processing (NLP) pipeline that is used as part of a larger web crawler application. As documents arrive, the NLP pipeline extracts word stems, removes stop-words, normalizes terms, creates *n*-grams of sizes 2–5, and runs frequency analyses. The pipeline is built around v0.6.3 of `natural`, a third-party package for NLP, and applied on 200 books, each averaging 1MB, from the Project Gutenberg corpus [34].

Scale-out depends on the relative overheads of different NLP stages. Skipping the manual effort of profiling (described at the end of §8.2), manual scale-out would need to extract the interfaces of used modules, generate RPC stubs (*e.g.*, gRPC [83]) and load them remotely, start communication servers, and balance load at runtime. We did not attempt this; instead, we added `require("ignis", {hot: ["natural"]})`.

The centralized version processes at a rate of 22.2 documents per minute (2.7s per document). Ignis improves throughput by a factor of 27× to 612.8 documents per minute (97.9ms per document). While Ignis sees gains in launching more processes on the local host, it does not see any gains in further distributing across physical hosts. The reason is that the crawler does not feed the NLP pipeline with documents at a rate that is high enough to benefit from networked distribution (in our setup).

8.5 Macrobenchmark: Wiki Engine

For a complex application, we turn to `wiki.js` (2.0.0-dev), a popular wiki engine that imports 130 top-level modules; counting recursive imports, the total jumps to 1640 modules and about 597K lines of code. We augment `wiki.js` with `uri-js v2.1.1`, an extensible URI parsing and validation library that is fast in the average case but can sometimes spend upwards of 400ms per URI in pathological edge cases. This leaves `wiki.js` susceptible to denial-of-service attacks [17] (ReDoS) and slowdown in certain non-adversarial workloads. We use Ignis to mitigate this with a simple two-line recipe (*i.e.*, less than 0.0005% of the codebase) that scales out `uri-js`.

Under normal operation, when no URIs in the workload invoke the edge cases in `uri-js`, Ignis introduces little runtime overhead. Issuing an HTTP load of 5Kreqs/s on `wiki.js`'s sample dataset, the unmodified wiki responds with an average latency of 34.1ms (σ: 2.1ms). Introducing Ignis bumps latency to 34.3ms (σ: 2.8ms). Changes in memory consumption, which averages around 100MB, remain below 0.01%.

With a workload that contains even a small fraction of pathological URIs, the benefits of Ignis become significant. Servicing a workload with 99% benign URIs and 1% pathological URIs, the throughput of the unmodified wiki drops to 197req/s (15.2s per request, σ: 11.04s). Ignis, on the other hand, achieves a throughput of 208req/s (8.1s per request, σ: 4.9s) when distributing to q_{1-4}'s four network replicas. When distributing to 60 replicas on a_1, it achieves a throughput of

1,880req/s (42.66ms per request, σ: 1.35ms). Ignis detects the load pressure applied on uri-js within the first few pathological requests, scaling out in under half a second.

Memory consumption of each replica is about 17.3MB, the vast majority of which comes from the Node.js runtime and libraries. For comparison, naive application replication leads to a total memory footprint of 1.1GB (on top of the "master" wiki.js consumption).

9 Related Work

Our techniques are related to a large body of previous work in several distinct domains.

Automated Parallelization There is a long history of automated parallelization starting from explicit DOALL and DOACROSS annotations [12, 51] and continuing with compilers that attempt to automatically extract parallelism [33, 65]. These systems operate at a lower level than Ignis (*e.g.*, that of instructions or loops instead of module boundaries) and typically do not exploit runtime information.

More recent work focuses on extracting parallelism from domain-specific programming models [28, 31, 46] and interactive parallelization tools [38, 40]. While these tools simplify the expression of parallelism, programmers are still involved in discovering and exposing parallelism. Moreover, the insights behind these attempts are significantly different from ours, as they extract parallelism statically during compilation instead of dynamically during runtime.

Distributed Environments A plethora of systems assist in the construction of distributed software. At one end of the spectrum, distributed operating systems [4, 22, 58, 64, 67, 69, 72, 73, 76, 89] and programming languages [25, 43, 77, 88] provide a significant amount of flexibility in the resulting application. However, they involve significant manual effort using the provided abstractions, which are strongly coupled with the underlying operating or runtime system. Light-touch distribution does not make assumptions about the underlying operating system, and makes only minimal assumptions about the language runtime.

At the other end of the spectrum, distributed computing frameworks [20, 60, 61, 68, 92] and domain-specific languages [2, 6, 23, 56, 57] simplify certain patterns, but do not offer the flexibility of a full-fledged environment. Developing under these frameworks differs quite significantly from how developers normally compose applications. In Ignis, developers write general programs as they would do normally—only sprinkling them with "control-plane" insights.

Object-based Distribution Several language-based approaches attempt to provide a single system image (SSI), either under new [9, 13, 49, 52] or existing languages [3, 5, 94]. The latter are closer to Ignis, but focus on SSI rather than component replication (except pure-function replication for cJVM [3]), and do not support dynamic profiling-based scale-out. They also impose a cluster-aware version of the JVM, whereas Ignis comes as a third-party module running on a completely unmodified V8.

Taurus' policies [54] and Terracotta's annotations [10] share the same flexibility-automation philosophy as Ignis' recipes, albeit at different levels. Taurus does not transform non-distributed applications, but is complementary to Ignis: using a holistic runtime would have helped coordinate just-in-time compilation, module spawning, and garbage collection across nodes. Terracotta's approach (see AOP below) requires significantly more developer effort, and does not support distributing the standard library (à la fs for Ignis).

More generally, distributed operating systems, programming languages, and language-based run-times are closer to Andromeda [87]—the platform upon which Ignis was developed—than Ignis itself.

Application Partitioning Automated application partitioning [37, 91] and mobile code offloading [15, 18, 24, 45, 74, 90] introduce (opaquely) the network into the application. Applications are split into a small number of parts, typically two: one runs on the server with nearly unlimited resources, while the other runs on the client with very constrained resources. There is no runtime profiling, and often no performance-oriented component replication [37, 91], as the goal is to hide the network and offer a continuum to the end-user. Wishbone [63] partitions sensornet programs automatically, but only if written in a custom stream-processing language and with predictable input patterns.

Circus [16] and ISIS [7] exploit development-time module structure, but their replication focuses on fault-tolerance instead of scalability. Circus forwards calls to all replicas, whereas ISIS uses a primary-backup scheme. In contrast to Ignis, they operate in a static environment and without runtime introspection, decomposing applications at compile-time. They also assume knowledge of module requirements and deterministic, idempotent modules whose semantics remain locked; in Ignis, such domain-specific information is expressed via recipes and can change at runtime.

Security-oriented compartmentalization [8, 11, 44] decomposes software into multiple isolated components with the goal of improving its security properties—and often at the boundaries of (third-party) modules [47, 59, 84, 86]. However, it does not leverage runtime profiling, and is usually static, targeting privilege reduction rather than performance increase. DeDoS [21] includes profiling for denial-of-service attacks, but requires users to structure their applications in terms of minimum schedulable units (MSUs).

Component Architectures Lambda [26, 35] and microservice [27, 62] architectures build server applications as sets of loosely-coupled components. While in principle small and light, both are vastly more coarse-grained than language-level modules. Whereas a single multi-hundred-package microservice can scale out independently, light-touch distribution can scale out individual components of a *single* microservice. Moreover, communication between services is request-response style and is made explicit to the application. Most importantly, such decomposition is a manual process that requires careful design, including agreeing on the interfaces, prior to development.

Transformations Aspect-oriented programming (AOP) is a programming model that maps program join-points to advice, actions to be taken at these points [41]. For light-touch distribution, these would be "calls at the module boundary" and "wrap with profiling", respectively. Some cross-cutting aspects around the program could be transformed to their distributed versions (*e.g.*, built-in `fs` module). In contrast to AOP, Ignis does not inject dependencies, therefore control flow is not obscured. Moreover, developers do not need to understand different concerns—that is, program structure is not affected and developers do not alter or introduce code.

More generally, AOP is related to metaobjects [42] that enable a program to access to its own internal structure, including rewriting itself as it executes. They are examples of runtime reflection, of which Ignis makes extensive use to traverse, understand, and rewrite interfaces; but developers using Ignis do not need to provide their own metaobjects.

There has been a recent emergence of unsound program transformations [66, 70, 78, 79] that attempt to alter the semantics of the original program in principled ways. Light-touch distribution can be seen as introducing programmer-guided, "control-plane" semantic hints at the module boundaries. Using these hints, programmers effectively guide the principles behind how the semantics of a program change.

10 Discussion

Ignis' key enablers—*i.e.*, dynamic languages, module ubiquity, and the programming model (§2.4)—are also its key limiting barriers for wider applicability. Making Ignis applicable more broadly would require lifting these barriers—for example, decomposing an application into components even in the absence of explicit modules and providing bolt-on scalability without any of the features offered by dynamic languages.[5] Aside from these limitations there are several potential avenues for future research.

Fault-Tolerance The distributed versions of built-in modules provide a degree of fault-tolerance—*e.g.*, `fs` for persistent state. However, the current version of Ignis does not adequately handle replica failures or network partitions in the general sense, which includes handling calls that were scheduled on failed replicas. Call scheduling could be prefixed with a form of check-pointing so that, upon failure, Ignis reschedules the dropped call on a different replica (potentially jumping the queue, to compensate for the lost time). Moreover, failure-aware scheduling would adapt scale-out to the unaffected part of the deployment. Ignis would also need to protect against partial global changes, where only a subset of nodes receives an update (*e.g.*, write to a global variable), using some form of consensus. Work on failure-tolerant light-touch distribution will lead to new recipes for tuning to failure-related trade-offs.

Profiling Models More sophisticated prediction logic for profiling and coordination (§4) is critical for elasticity. It should combine longer call history with the ability to quickly detect sudden changes in workload characteristics. Moreover, it should attempt to capture resource heterogeneity—even the distinction between locally available processors and distributed hosts. The main challenge is combining sub-linear space growth at the module boundaries with prediction latency low enough to be useful in online decision-making.

Recipe Inference The inference of recipes—the last barrier to automation—would be extremely useful, but is not trivial. Static analysis is difficult in environments with few-to-zero type annotations and modules written in multiple languages, some of which come compiled. Dynamic analysis is equally challenging, as it requires carefully tracing pre-runs that are still not guaranteed to cover all possible (non-deterministic) interleavings. A combination of tooling (for space exploration) with some form of learning could lead to conservative recipes that improve performance without breaking semantics.

11 Conclusion

This paper introduces *light-touch distribution*, a first step towards automating the generation of distributed systems from distribution-oblivious programs. This is achieved using a set of programmatic *transformations* parametrizable by optional *distribution recipes*. These operate at module boundaries during runtime to collect profiling information, detect bottlenecked components, and dynamically separate and coalesce parts of the application.

The presented design shows that it is possible to get distribution benefits out of programs that were not explicitly developed to support distribution with minimal developer effort.

Acknowledgments

We would like to thank Arthur Azevedo de Amorim, Bill Cheswick, Radoslav Ivanov, James Mickens, Leonidas Lampropoulos, Nik Sultana, and the anonymous reviewers for helpful thoughts. We are indebted to our shepherd, Martin Maas, whose detailed comments and suggestions significantly improved our work and the exposition of our ideas. This research was funded in part by National Science Foundation grant CNS-1513687.

[5] Recent advances show promise: C-Strider [75] provides type-aware heap traversal for C programs; LImpMod/LLibcheri [84] and Sandcrust [47] attempt module-level decomposition in compiled languages.

References

[1] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. 249–260.

[3] Yariv Aridor, Michael Factor, and Avi Teperman. 1999. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the 1999 International Conference on Parallel Processing (ICPP '99)*. IEEE Computer Society, Washington, DC, USA, 4–. http://dl.acm.org/citation.cfm?id=850940.852885

[4] Amnon Barak and Oren La'adan. 1998. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems* 13, 4 (1998), 361–372.

[5] John K. Bennett. 1987. The Design and Implementation of Distributed Smalltalk. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*. ACM, New York, NY, USA, 318–330. https://doi.org/10.1145/38765.38836

[6] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. 2013. Distal: A Framework for Implementing Fault-tolerant Distributed Algorithms. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '13)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/DSN.2013.6575306

[7] Kenneth P. Birman. 1985. Replication and Fault-tolerance in the ISIS System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP '85)*. ACM, New York, NY, USA, 79–86. https://doi.org/10.1145/323647.323636

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322. http://dl.acm.org/citation.cfm?id=1387589.1387611

[9] Andrew P Black, Norman C Hutchinson, Eric Jul, and Henry M Levy. 2007. The development of the Emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 11–1. http://www.emeraldprogramminglanguage.org/authorsVersion.pdf

[10] Jonas Bonér and Eugene Kuleshov. 2007. Clustering the Java virtual machine using aspect-oriented programming. In *AOSDâĂŹ07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*.

[11] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1251375.1251380

[12] Michael Burke and Ron Cytron. 1986. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN '86)*. ACM, New York, NY, USA, 162–175. https://doi.org/10.1145/12276.13328

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, Vol. 40. ACM, 519–538.

[14] Kristina Chodorow. 2013. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.".

[15] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 301–314. https://doi.org/10.1145/1966445.1966473

[16] Eric C. Cooper. 1985. Replicated Distributed Programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP '85)*. ACM, New York, NY, USA, 63–78. https://doi.org/10.1145/323647.323635

[17] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks.. In *Usenix Security*, Vol. 2.

[18] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 49–62.

[19] Ryan Dahl and the Node.js Foundation. 2009. Node.js. https://nodejs.org Accessed: 2017-06-11.

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[21] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Nik Sultana, Bowen Wang, Jingyu Qian, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. 2017. A Demonstration of the DeDoS Platform for Defusing Asymmetric DDoS Attacks in Data Centers. In *Proceedings of the SIGCOMM Posters and Demos (SIGCOMM Posters and Demos '17)*. ACM, New York, NY, USA, 71–73. https://doi.org/10.1145/3123878.3131990

[22] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. 1997. The Inferno operating system. *Bell Labs Technical Journal* 2, 1 (1997), 5–18. http://www.vitanuova.com/inferno/papers/bltj.pdf

[23] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

[24] Janick Edinger, Dominik Schäfer, Martin Breitbach, and Christian Becker. 2017. Developing distributed computing applications with Tasklets. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. IEEE, 94–96.

[25] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 118–129. https://doi.org/10.1145/2034675.2034690

[26] Marius Eriksen. 2013. Your Server As a Function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS '13)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2525528.2525538

[27] Martin Fowler and James Lewis. 2014. Microservices. http://martinfowler.com/articles/microservices.html Accessed: 2015-02-17.

[28] Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices* 33, 5 (1998), 212–223.

[29] Nicolas Giard and Wiki.js Contributors. 2018. wiki.js. https://wiki.js.org/ Accessed: 2018-09-18.

[30] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (2002), 51–59.

[31] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. 2002. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, Vol. 36. ACM, 291–303.

[32] Brendan Gregg. 2013. *Systems Performance: Enterprise and the Cloud* (1st ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[33] Mary W Hall, Jennifer M Anderson, Saman P. Amarasinghe, Brian R Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S Lam. 1996. Maximizing multiprocessor performance with the SUIF compiler. *Computer* 29, 12 (1996), 84–89.

[34] Michael Hart. 1971. *Project Gutenberg*. https://www.gutenberg.org/

[35] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[36] TJ Holowaychuk and the Koa.js Developers. 2009. Koajs Examples: Blog. https://github.com/koajs/examples/tree/master/blog Accessed: 2019-03-01.

[37] Galen C. Hunt and Michael L. Scott. 1999. The Coign Automatic Distributed Partitioning System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Berkeley, CA, USA, 187–200. http://dl.acm.org/citation.cfm?id=296806.296826

[38] Makoto Ishihara, Hiroki Honda, and Mitsuhisa Sato. 2006. Development and implementation of an interactive parallelization assistance tool for OpenMP: iPat/OMP. *IEICE transactions on information and systems* 89, 2 (2006), 399–407.

[39] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[40] Ken Kennedy, Kathryn S Mckinley, and C-W Tseng. 1991. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (1991), 329–341.

[41] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

[42] Gregor Kiczales and Jim Des Rivieres. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA.

[43] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 179–188. https://doi.org/10.1145/1250734.1250755

[44] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*. 273–284.

[45] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*. IEEE, 945–953.

[46] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. 2007. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices* 42, 6 (2007), 211–222.

[47] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*. ACM, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[48] Hugh C. Lauer and Roger M. Needham. 1979. On the Duality of Operating System Structures. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 3–19. https://doi.org/10.1145/850657.850658

[49] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. 1981. The Architecture of the Eden System. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*. ACM, New York, NY, USA, 148–159. https://doi.org/10.1145/800216.806603

[50] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. 1975. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles (SOSP '75)*. ACM, New York, NY, USA, 132–140. https://doi.org/10.1145/800213.806531

[51] Amy W. Lim and Monica S. Lam. 1997. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 201–214. https://doi.org/10.1145/263699.263719

[52] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. 1987. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/41457.37514

[53] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions.. In *OSDI*. 135–150.

[54] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 457–471. https://doi.org/10.1145/2872362.2872386

[55] Neil McAllister. 2012. Twitter survives election after Ruby-to-Java move. https://www.theregister.co.uk/2012/11/08/twitter_epic_traffic_saved_by_java/ Accessed: 2019-04-11.

[56] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: a language for distributed, eventually consistent computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 7.

[57] Adrian Mizzi, Joshua Ellul, and Gordon Pace. 2018. D'Artagnan: An Embedded DSL Framework for Distributed Embedded Systems. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. ACM, New York, NY, USA, Article 2, 9 pages. https://doi.org/10.1145/3183895.3183899

[58] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53. https://www.cs.cornell.edu/home/rvr/papers/Amoeba1990s.pdf

[59] Derek G. Murray and Steven Hand. 2008. Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes. In *Proceedings of the 1st European Workshop on System Security (EUROSEC '08)*. ACM, New York, NY, USA, 40–46. https://doi.org/10.1145/1355284.1355292

[60] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[61] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 113–126. http://dl.acm.org/citation.cfm?id=1972457.1972470

[62] Sam Newman. 2015. *Building Microservices*. O'Reilly Media, Inc.

[63] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. 2009. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 395–408. http://dl.acm.org/citation.cfm?id=1558977.1559004

[64] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite network operating

system. *Computer* 21, 2 (1988), 23–36. http://www.research.ibm.com/people/f/fdouglis/papers/sprite.pdf

[65] David A Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. 1993. Polaris: A new-generation parallelizing compiler for MPPs. In *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign.*

[66] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 87–102. https://doi.org/10.1145/1629575.1629585

[67] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. 1990. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference.* 1–9. http://css.csail.mit.edu/6.824/2014/papers/plan9.pdf

[68] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. 2014. Aggregation and Degradation in JetStream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).* USENIX Association, 275–288.

[69] Richard F Rashid and George G Robertson. 1981. *Accent: A communication oriented network operating system kernel.* Vol. 15. ACM. https://cseweb.ucsd.edu/classes/wi08/cse221/papers/rashid81.pdf

[70] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. 2004. Enhancing Server Availability and Security Through Failure-oblivious Computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04).* USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251254.1251275

[71] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12).* 101–112.

[72] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. 1991. Overview of the chorus distributed operating systems. In *Computing Systems.*

[73] Jan Sacha, Henning Schild, Jeff Napper, Noah Evans, and Sape Mullender. 2013. Message passing and scheduling in Osprey. (2013). http://sfma13.cs.washington.edu/wp-content/uploads/2013/04/sfma2013-final6.pdf

[74] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).

[75] Karla Saur, Michael Hicks, and Jeffrey S. Foster. 2016. C-strider: Type-aware Heap Traversal for C. *Softw. Pract. Exper.* 46, 6 (June 2016), 767–788. https://doi.org/10.1002/spe.2332

[76] Malte Schwarzkopf, Matthew P Grosvenor, and Steven Hand. 2013. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems.* ACM, 9. http://www.cl.cam.ac.uk/~ms705/pub/papers/2013-apsys-dios.pdf

[77] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2005. Acute: High-level Programming Language Design for Distributed Computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05).* ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1086365.1086370

[78] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. 2009. ASSURE: Automatic Software Self-healing Using Rescue Points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV).* ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1508244.1508250

[79] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. 2005. Building a Reactive Immune System for Software Services. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05).* USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1247360.1247371

[80] Joel Spolsky. 2000. Things You Should Never Do, Part I. https://tinyurl.com/j5ml4gg Accessed: 2017-12-11.

[81] Gerald Jay Sussman and Guy L. Steele, Jr. 1998. The First Report on Scheme Revisited. *Higher Order Symbol. Comput.* 11, 4 (Dec. 1998), 399–404. https://doi.org/10.1023/A:1010079421970

[82] D. G. Thaler and C. V. Ravishankar. 1998. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking* 6, 1 (Feb 1998), 1–14. https://doi.org/10.1109/90.663936

[83] The gRPC Authors. 2018. gRPC. https://grpc.io/ Accessed: 2019-04-16.

[84] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards automatic compartmentalization of C programs on capability machines. In *Workshop on Foundations of Computer Security 2017 (FCS'17).* 1–14.

[85] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2017. Towards Fine-grained, Automated Application Compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17).* ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/3144555.3144563

[86] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18).* https://doi.org/10.14722/ndss.2018.23131

[87] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. 2015. From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV).* USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/vasilakis

[88] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2Nd Ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[89] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. 1983. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, Vol. 17. ACM, 49–70.

[90] Xiaojuan Wei, Shangguang Wang, Ao Zhou, Jinliang Xu, Sen Su, Sathish Kumar, and Fangchun Yang. 2017. MVR: An Architecture for Computation Offloading in Mobile Edge Computing. In *Edge Computing (EDGE), 2017 IEEE International Conference on.* IEEE, 232–235.

[91] Cliff Young, Yagati N Lakshman, Tom Szymanski, John Reppy, David Presotto, Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse. 2001. Protium, an infrastructure for partitioned applications. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on.* IEEE, 47–52.

[92] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12).* USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[93] Jamie Zawinski. 1999. Resignation and Postmortem. https://www.jwz.org/gruntle/nomo.html Accessed: 2018-02-12.

[94] John N Zigman, Ramesh Sankaranarayana, et al. 2002. dJVM-A distributed JVM on a Cluster. (2002).