Computation-Centric Networking

Yuhan Deng, Angela Montemayor, Amit Levy*, and Keith Winstein
Stanford University, *Princeton University

Abstract

We propose putting *computation* at the center of what networked computers and cloud services do for their users. We envision a shared representation of a computation: a deterministic procedure, run in an environment of well-specified dependencies. This suggests an end-to-end argument for serverless computing, shifting the service model from "renting CPUs by the second" to "providing the unambiguously correct result of a computation." Accountability to these higher-level abstractions could permit agility and innovation on other axes.

Act 1, Scene 1

CECILY: Gwendolen, I love that paper you wrote last year! How did you make Figure 3? I'd like to reproduce it.

GWENDOLEN: Happy to help, Cecily! Reproducibility is my jam. Just go to GitHub, clone our repo, and it's all in there.

CECILY (to computer): `git clone repo; cd repo; ls`

The computer prints:
```
data.csv          graph.svg          make-graph.py
result-new.csv    graph.tiff         real-data.csv
graph.eps         gwen-grapher.py    results-dir
extract-old.sh    draw-plot.py       result1.txt
```

CECILY: Hmm, this isn't very helpful. It might all be here somewhere, but I don't know what to run, which of these files is the real data, or how anything relates to anything else. Gwendolen, do you remember how you generated this figure?

GWENDOLEN: Huh, I forget. It's weird how Git is so good at recording *compositional* relationships—what files are in which directory trees, what tree each commit points to—but there's no way to record *computational* relationships like "this file is the output of *this* computation, given *this* input." Hmm.

Scene 2

LADY BRACKNELL: Algernon, I was amazed to learn how networked filesystems use complex algorithms to losslessly recompress image and video files on the backing storage [3, 12]. Aren't they worried that a bug, or pathological file, could make them unable to recover the exact original contents?

ALGERNON: Indeed, which is why they don't throw away the original file until after they've decompressed their own compressed output and checked that the result is identical.

BRACKNELL: But what if the decompression program is nondeterministic? They could get the right answer once, but never be so lucky again. It's hard to enforce reproducibility on an arbitrary program without a big slowdown [19]. Hmm.

Scene 3

LANE: Oh no, we've been pwned! Three months ago, attackers broke in and modified our code.

CHASUBLE: What are the consequences? If we had been running the right code, how would our output have differed?

LANE: How should I know? There's no way to tell. Hmm.

Scene 4

JEFF BEZOS: Man, AWS Lambda is bumming me out. Our users will invoke a function with an HTTP request, and we feed that to a Linux program they uploaded. We can get that running within 100 ms, and burst to thousands of concurrent executions, which is great. Except: most functions spend 90% of their time idling the CPU, waiting on dependencies from S3 and elsewhere. We're wasting a lot of expensive infrastructure, which hurts us and ultimately our customers.

MERRIMAN: Why? Can't we just overprovision by 10×?

BEZOS: Maybe, but that's still inefficient—we'll tie up a ton of RAM while the functions block on I/O—and still slow.

MERRIMAN: Can we cleverly schedule functions to run on the same nodes as their dependencies? This could be a great place for us to innovate and outperform the other clouds.

BEZOS: Not easily, because we don't know the dependencies; the function fetches whatever it wants after it starts. And we aren't strongly incentivized to make jobs complete earlier when users pay us for each millisecond of runtime. Hmm.

Scene 5

JACK WORTHING: Man, AWS Lambda is bumming me out. My function spends 90% of its billed time blocked on I/O.

PRISM: What do you expect? It's an opaque program to them; they have no ability to make it run faster.

JACK: I just want to tell them what I'm trying to compute, with what inputs, and let them be smart about how it's done. We could split any efficiency savings. All I *really* care about is getting the right answer! It doesn't even have to be AWS, as long as somebody reputable stands behind the result. Hmm.

CCS Concepts: • **Networks** → **Cloud computing**; **Keywords:** serverless computing

HotNets '22, November 14–15, 2022, Austin, Texas
© 2022 Copyright held by the owner/author(s). Publication rights licensed to the ACM.
ACM ISBN 978-1-4503-9899-2/22/11. https://doi.org/10.1145/3563766.3564106

ACM Reference Format:
Yuhan Deng, Angela Montemayor, Amit Levy, and Keith Winstein. 2022. Computation-Centric Networking. In *Proceedings of The 21st ACM Workshop on Hot Topics in Networks (HotNets '22).* ACM, New York, NY, 7 pages. https://doi.org/10.1145/3563766.3564106

2 Introduction

The stories you just read were a fib, but the issues are real. While classical "infrastructure-as-a-service" cloud computing involves renting a virtual server and paying by the second, current "function-as-a-service" offerings provide almost the same service model: renting an x86 or ARM worker and paying by the tenth of a second until a task completes. Providers have little visibility into client dataflow, which translates into inefficient placement and poor utilization. When most jobs spend most of their time waiting for bytes to arrive from across the network, even a clever provider has little ability (or incentive) to improve the situation.

In this paper, we argue that the root issue behind each of these vignettes is an underconstrained notion of networked computation. We propose a research agenda centered around what we call "computation-centric networking"[1]: the idea that a networked service's job is primarily to provide *answers to computations* and would benefit from (1) fine-grained visibility into application dataflow, (2) an objective, common notion of correctness, and (3) a separation between I/O and compute, with delineated nondeterminism.

In our view, successfully realizing this vision would:

- let networked systems track the computational relationships between artifacts, so that sharing a reproducible pipeline is as simple as a `git push`/`git pull`/"`git reproduce`" (Scene 1),
- guarantee reproducibility of server-side algorithms that process data on a user's behalf (Scene 2),
- allow rerunning a computational pipeline with modified code or data, to discover the consequences of, and clean up after, an intrusion (Scene 3), and
- benefit "serverless" providers and customers (Scenes 4 and 5). Providers would have the flexibility to schedule and place jobs in a way that minimizes dataflow and maximizes utilization, as long as they reach the correct answer. If the customer chooses to double-check a result and finds the provider was mistaken, they'd be able to collect from the provider's insurance. That, in turn, might free the customer to bid jobs out to competing providers. Our theory here is akin to an end-to-end argument [21]: *accountability* to one high-level abstraction (correctness) can create *agility* on other axes.

[1]We call it "computation-centric" by analogy to content- or information-centric networking [10, 14], which argues for refactoring the network service away from messaging and towards retrieval of identified content, however sourced. We propose a similar refactoring, towards evaluation of named *computations* however executed. (We don't take a position on what layer of the stack should be modified to achieve this and aren't seeking to replace IP.)

Summary of results. We have begun to design and implement a framework for computation-centric networking, which we call Fixpoint. We are defining a low-level, lightweight representation for deterministic computations-on-named-data, known as "Fix." To represent the relationships between code and data, Fix defines an addressing scheme that allows data to be identified either in terms of its contents (similar to systems like Git, BitTorrent, and IPFS) or by referring to a deterministic computation that computes it. The Fixpoint system includes a compiler that transforms Fix into raw machine codelets, and runtime engines that evaluate such codelets on various platforms: multicore computers, clusters, and serverless computing platforms.

Our preliminary benchmarks have found that these abstractions are lightweight enough to let Fixpoint provide isolation and reproducibility with overhead close to an ordinary virtual function call. On a recent x86-64 CPU and Linux kernel, Fixpoint's invocation overhead is about 37× faster than vforking a process, and about 531× faster than record-replay techniques such as `rr`. The raw invocation overhead is roughly 50 ns, about 5× as slow as a virtual function call in C++.

Computation-centric networking is about constraining computation and exposing its dataflow, in order to free the network to innovate in *how* it produces results. We hypothesize that it will be possible to fit *most* software into these strictures without significant penalty, to create a world where most computations are reproducible-by-default and amenable to efficient outsourcing to oceans of cores in the cloud, supplied by competing providers who bid for the work, innovate to find better ways to run customer jobs, and guarantee correctness. Whether we're right about that remains to be seen.

3 Central hypotheses of the research

Before we dive into Fixpoint's design, we'd like to discuss four assumptions that will probably need to hold for the dream of computation-centric networking to be realized.

Separating I/O and compute is widely achievable. A key assumption is that many useful programs can be feasibly separated into I/O and compute, so that each stage of a computation can declare its dependencies before execution, and then execute deterministically (given those inputs) to make some forward progress before discovering a need for additional inputs. This model works well for many tasks (exploratory data analysis, compilation and testing, 3D rendering, machine learning), but is less applicable to software that involves multiple users interacting concurrently with a service with tight synchronization (e.g., a concurrent RDBMS or lockserver).

Nondeterminism can be delineated. When we refer to "I/O," we expect that almost all inputs can be identified deterministically, that is, pre-specified in a way that locks down their contents. This can occur if inputs are named by a hash of their contents or by a deterministic way to compute them.

Of course, real-world programs sometimes desire nondeterministic I/O, e.g., to gather a random seed, the current time, keyboard input, or arbitrary information from a remote sensor or service. In Fixpoint’s world, this nondeterminism needs to be delineated from the rest of the program, similar to an “unsafe block” in some programming languages. The expectation is that these occasions are infrequent, and the rest of a program operates reproducibly *given* the input from a nondeterministic procedure. (E.g. given the random seed, a Monte Carlo simulation or encryption tool runs predictably.)

Programs won’t pay a big performance penalty. Fix has advantages in allowing better placement decisions and fast context switching, but also imposes slowdowns compared with native code. We will need to gather more information about how this nets out in real-world use.

Increased efficiency is mutually beneficial. If infrastructure providers are able to execute client jobs more efficiently based on visibility into their dataflow, this may translate into a mixture of greater return on investment for the provider and more-competitive pricing for users, incentivizing deployment.

4 Fix: a language of computation on data

“It has been said that the principal function of an operating system is to define a number of different names for the same object, so that it can busy itself keeping track of the relationship between all of the different names.” —David Clark, RFC 814 [5]

Fix is a way to represent computations on data, built around abstractions inspired by Git. The key idea is to name every computation and piece of data in a way that fully describes its contents, in a manner both lightweight enough to permit tens-of-nanosecond overheads, yet general enough to support arbitrary applications, including ones that require control of strictness/laziness, to support functions like a short-circuiting “and,” and higher-order functions like “curry.”

4.1 Types

Objects. Fix describes three types of object: a Blob (a vector of bytes), a Tree (a vector of Names, each marked strict or non-strict), or a Thunk, which refers to a computation and stands in for the value that computation will produce. A Thunk “refers to a computation” by giving the Name of an Encode, described below. Each object is immutable and has a canonical representation.

Names. A Name is a 256-bit value that uniquely identifies an object. This can be a “canonical” name (the SHA-256 hash of the object’s canonical representation), a “literal” name (the canonical representation of the object itself), or a “local” name (a locally generated unique ID).

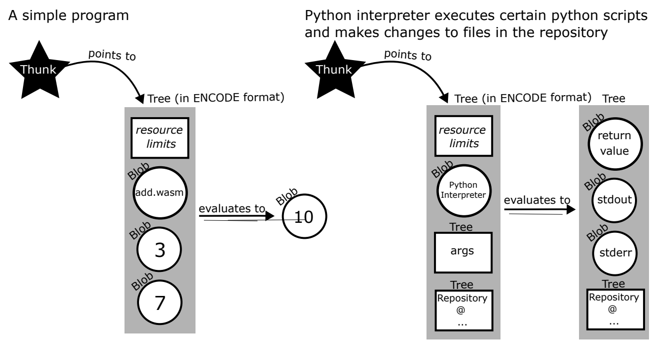

Figure 1: Representing a simple function (left) or Unix-like execution (right) in Fix’s abstractions.

Encodes. An Encode[2] is a Tree in a particular format that describes the application of a function to inputs, producing an object as output. The entries in the Tree are (a) resource limits that will govern the execution at runtime, (b) the procedure itself (represented as a Blob of code, or another Encode with captured data), and (c) any other data that should be available to the function at runtime.

4.2 Operations

Fix’s semantics are similar to a Lisp with variable strictness for each argument [2]. There are two fundamental operations: evaluation of an object, and applying code to data.

Evaluation. Evaluating a Blob is the identity. Evaluating a Tree evaluates each of its strict entries.[3] Evaluating a Thunk means running the referenced computation—applying the Encode’s procedure to its data—and continuing recursively until a Blob or Tree is produced.

Application of code to data. To run an Encode, the system first evaluates the Encode Tree itself. Then, Fixpoint runs the procedure, giving it access to the Encode’s strict entries and the ability to create new objects. The function’s return value is the Encode’s output.

Figure 1 shows this process for a simple “add” function (left side) and for a more complicated execution of the CPython interpreter on a Python script and filesystem (right side).

4.3 Representating the code

Fix ultimately represents the code as a Blob containing a WebAssembly (Wasm) module [11]. We chose this because:

- Wasm is deterministic, with limited exceptions.
- The popular clang/LLVM compiler includes a Wasm backend (alongside x86, ARM, etc.), so thousands of existing programs can already be compiled into it.
- It’s possible to compile Wasm ahead-of-time into native machine codelets that maintain Wasm’s guarantees [1],

[2]A recursive acronym for “Explicit Named COmputation on Data or Encodes.”
[3]Each entry can be evaluated in parallel, exploiting massive parallelism [7, 8].

with good performance: invoking a function becomes an ordinary function call, and programs run about 88% as fast as when source code is compiled directly [24].

We emphasize that the choice of Wasm isn't intrinsic to Fix: any language capable of guaranteeing deterministic safe execution could be used. Before runtime, the Wasm procedures will be compiled to native machine code, and at runtime, Fixpoint will jump to these codelets, oblivious to the language they came from.

5 Prototype implementation of Fixpoint

To test our design, we have implemented a "trusted toolchain" that compiles Fix programs into x86-64 machine codelets, and a prototype runtime engine that executes computations locally on a single core. These tools will be available as free (open source) software.

Given a Wasm module, Fixpoint compiles it by using (1) the WebAssembly Binary Toolkit's `wasm2c` tool [1] to convert it to C source code, (2) `libclang` to compile that to optimized x86-64 machine code, and then (3) an in-memory ELF linker that we developed to link the codelet with the Fixpoint API. (We contributed several changes back to `wasm2c` to support this approach. Using this pipeline is easier and simpler than writing and maintaining our own Wasm-to-x86 compiler.) These steps can be performed upfront before runtime. We intend for the toolchain to be an ordinary Fix program, run inside Fixpoint like any other program.

From the perspective of the original Wasm code, Fixpoint's API allows it to "map" Blobs and Trees into native Wasm data types (for Blobs, a read-only linear memory, and for Trees, an externref-typed table). This allows the procedure to have zero-copy access to the Encode and its strict entries. Names are represented as 256-bit vector types, which can be passed by value through AVX2 (YMM) registers.

We have also begun implementing a library to let existing Unix-style programs run on Fixpoint. The Wasm community has created a standard C library (wasi-libc) that implements the C/POSIX interface in terms of underlying system calls known as the WebAssembly System Interface (WASI). This allows many existing programs to be compiled to Wasm (e.g. the CPython interpreter, clang, ffmpeg, etc.). In turn, we implemented a library called Flatware (Figure 2) that implements the WASI interface in terms of the Fixpoint API—treating the Encode as containing a Unix-like filesystem. From Fixpoint's perspective, this translation layer is an ordinary unprivileged part of the procedure.

6 Evaluation

In this section, we measure Fixpoint's overhead by comparing the performance of various 8-bit `add` programs. We'd like to provide preliminary evidence that the principles of

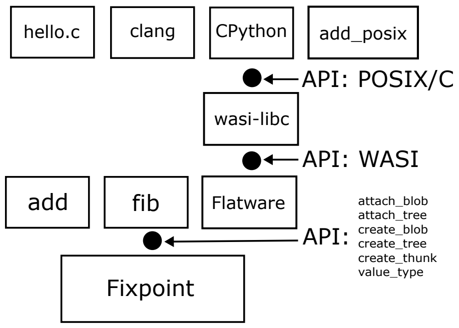

Figure 2: Fix can express "native" functions as well as Unix-style programs that manipulate a filesystem.

computation-centric networking (fine-grained visibility into application dataflow, accountability to a shared notion of correctness, and I/O-compute separation) can be realized with acceptable overhead.

6.1 Setup

Implementations: We compare the performance of six 8-bit `add` functions: three simple functions, and three POSIX programs with full libc initialization.

(1) **static**: Calling a statically linked function in C that adds two 8-bit integers. This provides no isolation, reproducibility, or declared dataflow.
(2) **virtual**: The same, called as a virtual function in C++.
(3) **Fixpoint**: The same, implemented in WebAssembly against the Fixpoint API. This provides isolation, reproducibility, and declared dataflow.
(4) **Flatware**: A full C/POSIX program with a `main` function, linked against the WASI libc and our Flatware library, then run in Fixpoint. This also provides isolation, reproducibility, and declared dataflow.
(5) **Process**: The same, linked against the system libc and run as a Linux process. This provides isolation.
(6) **Process in rr**: The same, inside a long-running invocation of the `rr` record-replay tool [19], which provides isolation and "replayability" (the same output can be achieved by replaying the trace recorded when `rr` was first run).

Benchmark: For the first four, we evaluate the add function 4,096 times, and report the average time per function call. For the last two, we `vfork` the add program and wait for its completion 4,096 times, and average the time per execution. We report the average of five benchmark runs.

Hardware: Our experiments were executed on a machine with two 64-core 2 GHz CPUs (AMD EPYC 7702).

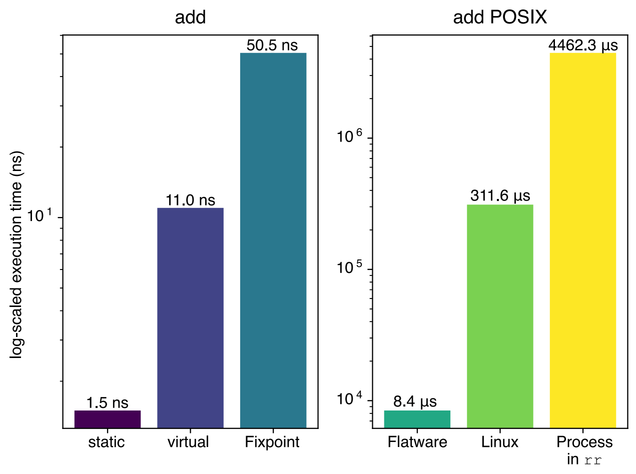

Figure 3: Benchmark Result

6.2 Analysis

As Figure 3 shows, implementing `add` as static or virtual function calls are the fastest implementations, taking 1.5 and 11.0 nanoseconds for execution, but do not provide isolation, reproducibility, etc. Executing `add` as a Linux process provides isolation, at the cost of a $> 300\,\mu s$ context-switching penalty. Running a process inside a long-running `rr` invocation (for replayability) incurrs a > 4-millisecond penalty.

By contrast, the two Fixpoint versions seem to compare promisingly with their conventional counterparts. The "simple" Fixpoint program is 5× slower than a virtual function call—an overhead of about 40 nanoseconds. We think this is a fair trade for enabling computation-centric networking, by enforcing that the program only accesses dependencies that have been declared to the underlying infrastructure and runs reproducibly. Realistically, it suggests that a practical Fixpoint function granularity will be one that takes at least 5× this duration (roughly 200 ns) to execute.

Fixpoint's execution of a "full" POSIX program (including libc setup, the `start` and `main` functions, etc.) compares well with conventional methods—it is 37× faster than a Linux `vfork`, and about 531× faster when that `vfork` happens inside `rr`. This mostly speaks to the advantages of WebAssembly's enforced isolation by use of a secure language, rather than relying on context switching, the MMU, and tracing sources of nondeterminism.

Of course we are measuring a microbenchmark of our choosing against the sophisticated and rich behavior of the Linux kernel; it's not clear these advantages will be maintained as Fixpoint and the Flatware layer grow inevitably more complicated. Still, these preliminary benchmarks give some hope that the architecture we have sketched is lightweight enough to provide a reasonable substrate for many real-world applications.

7 Towards computation-centric networking

Our hypothesis has been that by changing the model of networked computation and cloud computing away from "billing for effort" (renting CPUs) and towards "paying for results" (i.e., objectively correct answers), substantial benefits can be unlocked. We expect networked applications to change dramatically in ways that can't be feasibly realized when software is represented as it is today:

- **Massive burst parallelism.** Applications will burst to massive parallelism on a transient basis—e.g., spinning up 100,000 parallel computations for one second, with heterogenous dataflow between computations. This could transform many types of batch operations into interactive ones.
- **Reproducibility and determinism by default.** As software increases in complexity, users will want the ability to take any computed output and share the process needed to reproduce it, as easily as they can trace and share the history of source code in a version-control system (such as Git) today.
- **Security and information flow control.** Users who compute with both shared and private data will want to track the provenance of computed outputs and enforce policies about who can see, or use, the results.
- **Software will be natively distributed and accommodate heterogenous infrastructure.** As more software becomes distributed, applications will make their dataflow manifest to the operating system to let the infrastructure make good placement decisions to increase locality. The provider will benefit from the freedom to choose among "moving code to data," "moving data to code" or even "recompute data instead of moving it." Datacenters will include a diversity of machine types, accelerators, and power efficiency strategies.
- **Ultra-high-density multitenancy.** Compute infrastructure will need to pack as many applications as possible into limited memory and CPU resources. This will require fine-grained understanding of each application's current working set and the ability to swap out state that isn't currently used—or to discard state completely, and if needed later, recalculate it from a checkpoint.
- **Arm's length verifiable computation.** Computation will become a commodity, with software automatically bidding out its compute needs to competing infrastructure providers. To allow users to trust the results, the infrastructure will supply a *verifiable* certificate of correctness and an insurance policy that pays if the user chooses to verify and an error can be demonstrated.
- **Regulation.** Are there industries in which regulators might require that companies commit to the chain of reasoning behind their computational results, even if the

algorithm remains confidential for the foreseeable future? E.g. should each credit denial, or stock-exchange trade, or search results page, be accompanied by the hash of a Thunk that computed it?

We expect many open questions will need to be solved before these dreams might be realized, e.g.:

- How should nondeterministic I/O be handled?
- How and when should garbage collection occur?
- Should there be a second language of "execution strategy" hints to guide the network's placement and scheduling of evaluations, in order to maximize locality?
- How should users share reproducible computations with one another—through a Git-like repository?
- What's the right API for higher-level languages to expose constructs like "a parallel `map` that compiles into a Fix program that can run in parallel on 10,000 cores"?
- What should a "visual debugger" look like to aid incremental development and inspect the computation flow and provenance of computed outputs?
- How should this type of "serverless computation" be billed—perhaps based on CPU runtime (with no blocking for I/O, since it's the provider's job to position dependencies in place before execution begins) with discounts for collocation and other efficiencies?

8 Related work

This paper's vision of computation-centric networking has many antecedents: container frameworks such as Nix [6] and Docker [17]; cluster-computing systems like Hadoop [22], Dryad [13], Spark [23], CIEL [18], or Scanner [20]; burst-parallel software for "serverless" platforms, such as ExCamera [9], PyWren [16], gg [7], and R2E2 [8]; reproducible and deterministic execution systems such as rr [19] and Determinator [4]; and content-addressed and -centric networked systems, including BitTorrent, Git, IPFS, and NDN [10, 15, 25].

Given this substantial literature, the reader might reasonably wonder "what's new here," why "this time it's different," and what this has to do with networking as traditionally understood. In our view, none of the above systems is equipped to tackle the vignettes that opened this paper. The approach we propose here is well-predicated, but we suspect different *enough* to bear fruit in the areas we focus on.

An objective notion of correctness probably requires reproducibility, even with adversarial input. For a distributed version-control user to be able to ask "why" an artifact has the contents it does and be assured of a correct answer (Scene 1), or for a networked filesystem to throw away the original copy of a file, knowing it can reproduce it on demand (Scene 2), or for an insurance company to underwrite the correctness of a cloud service's results and pay out for wrong answers (Scene 5), we'd like the system to enforce reproducibility on arbitrary, perhaps buggy or adversarial, software. Few of the systems above are intended for this, the closest being replay debuggers (such as `rr`) that impose a substantial overhead, in terms of slowdown and space to record a trace of machine-generated nondeterministic inputs to be replayed later.

Fine-grained visibility into dataflow suggests an evolving AST, rather than a static DAG, may be necessary to express "everyday" computations. Many cluster-computing systems express jobs as a directed acyclic graph of functions, and work to schedule functions efficiently on a cluster. This model has proved extraordinarily helpful in large-scale data processing, but less-so for general software whose data-dependencies are only revealed at runtime—running such tasks requires "over-capturing" inputs of which only a fraction are used. We believe that fine-grained visibility into application dataflow will be crucial for forensic inquiry (Scene 3) and efficient use of shared infrastructure (Scene 4), which suggests a more fine-grained model of computation. Fix's evolving computation (a DAG at any snapshot, but similar to Lisp or lambda calculus in its dynamicity) is more amenable to expressing the gamut of heterogenous computing tasks.

Difference in overheads. The latency required to start a Docker container or Hadoop job is generally measured in milliseconds at least—meaning nobody is threading these types of modularity and isolation deeply into their software. Fixpoint's vastly lower overhead may help this sort of modularity become more pervasive. And Fixpoint's ability to run arbitrary machine codelets (most of which will have come from Wasm, itself a well-tested LLVM target alongside x86 and ARM) will likely make it easier to run arbitrary software.

9 Conclusion

In this paper, we presented a research agenda to put *computation* at the center of what networked computers and cloud services do for their users. We argue that providing infrastructure with fine-grained visibility into application dataflow, an objective notion of correctness, and a separation between I/O and compute will open up new models of serverless "accountability" and enable agility and innovation on other axes. Many open questions remain before these dreams might be realized, but we are excited about opportunities to make progress and hopeful that others will join in some of these directions.

Acknowledgments

We thank the HotNets reviewers for their thoughtful feedback, and Mark Handley, Gerald Sussman, Sam Clegg, Deian Stefan, and Justin Weiler for helpful conversations. This work was supported in part by NSF grants 2045714, 2039070, 2028733, and 1763256, DARPA contract HR001120C0107, a Sloan Research Fellowship, and by Google, Huawei, VMware, Dropbox, Amazon, and Meta Platforms.

References

[1] 2018. wasm2c (WebAssembly Binary Toolkit). https://github.com/WebAssembly/wabt/tree/main/wasm2c. (2018).

[2] Harold Abelson and Julie Sussman, G. J. with Sussman. 1996. *Structure and Interpretation of Computer Programs (exercise 4.40)* (2nd editon ed.). MIT Press/McGraw-Hill, Cambridge.

[3] Jyrki Alakuijala, Ruud van Asseldonk, Sami Boukortt, Martin Bruse, Iulia-Maria Comsa, Moritz Firsching, Thomas Fischbacher, Sebastian Gomez, Evgenii Kliuchnikov, Robert Obryk, Krzysztof Potempa, Alexander Rhatushnyak, Jon Sneyers, Zoltan Szabadka, Lode Vandevenne, Luca Versari, and Jan Wassenberg. 2019. JPEG XL next-generation image compression architecture and coding tools.

[4] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, USA, 193–206.

[5] D. D. Clark. 1982. *RFC814: Name, Addresses, Ports, and Routes*. Technical Report. USA.

[6] Eelco Dolstra. 2006. *The purely functional software deployment model*. Utrecht University.

[7] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[8] Sadjad Fouladi, Brennan Shacklett, Fait Poms, Arjun Arora, Alex Ozdemir, Deepti Raghavan, Pat Hanrahan, Kayvon Fatahalian, and Keith Winstein. 2022. R2E2: Low-Latency Path Tracing of Terabyte-Scale Scenes Using Thousands of Cloud CPUs. *ACM Trans. Graph.* 41, 4, Article 76 (July 2022), 12 pages. https://doi.org/10.1145/3528223.3530171

[9] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.. In *NSDI*. 363–376.

[10] Ali Ghodsi, Teemu Koponen, Jarno Rajahalme, Pasi Sarolahti, and Scott Shenker. 2011. Naming in Content-Oriented Architectures. In *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking (ICN '11)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2018584.2018586

[11] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[12] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Laas, and Keith Winstein. 2017. The Design, Implementation, and Deployment of a System to Transparently Compress Hundreds of Petabytes of Image Files for a File-Storage Service. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 1–15. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn

[13] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[14] Van Jacobson. 2006. A new way to look at networking. Google Tech Talks. (August 2006). https://www.youtube.com/watch?v=gqGEMQveoqg.

[15] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1658939.1658941

[16] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[17] D. Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[18] 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 *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing

[19] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[20] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. 2018. Scanner: Efficient Video Analysis at Scale. In *ACM Transactions on Graphics*. Software available from https://github.com/scanner-research/scanner.

[21] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (Nov. 1984), 277–288. https://doi.org/10.1145/357401.357402

[22] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[24] Alon Zakai. 2020. WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing. https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html. (July 2020).

[25] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 66–73. https://doi.org/10.1145/2656877.2656887