A runtime infrastructure for the Continuum of Computing

Edoardo Tinto*
edoardo.tinto@phd.unipd.it
University of Padova
Padova, Italy

Tullio Vardanega†
tullio.vardanega@unipd.it
University of Padova
Padova, Italy

ABSTRACT

Devices at the Edge of the network are experiencing a considerable increase in computational resources. At the same time, connectivity becomes more pervasive. These phenomena jointly facilitate the emergence of a new computational model, increasingly referred to as the *Continuum of Computing*. This model aims at including Edge resources in Cloud-like (and Cloud-inclusive) resource pooling to accommodate computations that need reduced latency, increased privacy, and general mobility. This model has the potential to enhance the power and the reach of high-performance computing (HPC) applications, making them extend up to the Edge of the network. However, managing a pool of resources that span across both Cloud and Edge nodes poses new challenges. Moving data across the network generates latency and security issues, while national policies may outright limit data mobility. This suggests moving computation towards data instead of the usual opposite. Enabling migrating computation is one of key traits of the envisioned Continuum of Computing. The vast heterogeneity in the technological stacks and the lack of uniform standards, however, hinder the deployment of applications in the Continuum. The availability of a common runtime environment across all host nodes of the Continuum is an obvious way to circumvent those problems, reviving the *write-once-run-anywhere* promise in that context. The ability to move computations opportunistically after user-specific performance objectives is another key trait of the Continuum model, which also is a foundation to *spatial computing*, a context-aware and space-aware computing paradigm. How to effectively orchestrate migrating computations so that they can deliver value added to their users is still an open question. There is a general understanding that Cloud-native orchestrators perform poorly when shifting towards the Edge, due to exceedingly restrictive (Cloud-centric) assumptions underneath their orchestration model. The matter of efficient orchestration in the Continuum is paramount in the envisioned model. To showcase the feasibility and viability of a Continuum-worthy runtime infrastructure, we singled out two emerging technologies: Rust and WebAssembly. The Rust programming language's highlight is its statically-checked memory safety. WebAssembly's highlights are solid guarantees of isolation and a portable bytecode format for applications compiled for its Instruction Set Architecture (ISA). To this project, WebAssembly components written in Rust constitute the candidate building blocks for the Continuum infrastructure, centred on memory-safe and sandboxed execution capsules. In addition to that, this project aims to develop and deploy Continuum-worthy orchestration capabilities that leverage seamless migration.

The initial results of this project suggest that applications written in Rust and executing as WebAssembly components offer greater isolation and memory safety compared to containerized applications. Moreover, this novel approach might easily support *live migration*, consisting of the migration of an executing application into a different hosting node, preserving the state of the computation. Live migration prevents re-execution, and, at the time of writing, is largely unsupported in modern industrially applied containerized solutions. Supporting migrating computations might benefit multiple application scenarios. For example, the ability to migrate computation instead of freezing it during a low-energy phase may be of interest to energy-harvesting systems, Similarly, urgent science and Internet of Things (IoT) applications might want to move across Cloud and Edge nodes opportunistically, seeking optimal trade-offs between heavy and low-latency types of computation.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Continuum of Computing, WebAssembly, Rust, Migration

ACM Reference Format:
Edoardo Tinto and Tullio Vardanega. 2024. A runtime infrastructure for the Continuum of Computing. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3625549.3658832

1 THE VISION: A COMPUTE CONTINUUM

The Continuum we envision entails the following key traits: (1) Resources should be accessed as a single pool, encompassing both the Cloud and the Edge. On top of this infrastructure, (2) computations are expected to migrate, striving to maximize (or minimize when needed) user-declared metrics, opportunistically, and not just as a contingency response to system faults and overload situations. The interactions between components and resource access (3) should be web-based, leveraging HTTP-and-above protocols, in a manner that resembles how Cloud providers handle Cloud resources. Figure 1 offers a graphical representation of the envisioned model and its key traits. When destined to perform possibly frequent and certainly fast migrations (live redeployments), applications should be developed as orchestrated aggregates of relatively small migrating

*Corresponding author.
†Supervisor.

This work is licensed under a Creative Commons Attribution International 4.0 License.
HPDC '24, June 3–7, 2024, Pisa, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0413-0/24/06.
https://doi.org/10.1145/3625549.3658832

computations. Aware of the user-defined metrics it is expected to improve, the orchestrator of such an application should take migration decisions that accord with user-defined metric objectives. Applications deployed on top of the envisioned infrastructure will benefit from the following three facts:

(1) Adopting the Continuum model will simplify application development. The heterogeneity afflicting previous models, specifically in the Edge portion of the resource pool, could be confronted with the aid of a common runtime layer, allowing applications compiled once to run on virtually every node in the systems.
(2) Migrating components will boost application performance. This effect is due to (1) increased flexibility in orchestration decisions, (2) centrality of user-defined metrics, instead of a rule-based approach, and (3) the emerging capability of closing the distance between data and computation, without requesting the migration of possibly large amount of data.
(3) Potentially, the Continuum model could proffer application developers a vast number of computing nodes, many of which were previously considered just sensors and actuators in the Edge. Targeting this sort of resource enforces a frugal runtime footprint. Even if unfit for heavy-weight computations, those nodes might still contribute to parallel and real-time applications due to their proximity to sensed data and interacting users.

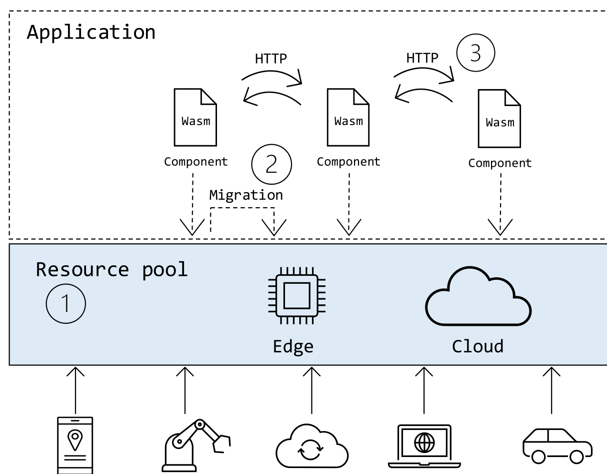

Figure 1: The envisioned model for the Continuum of Computing. It offers (1) resource pooling across Cloud and Edge, (2) migration for application components, and (3) web-based interactions.

1.1 Technologies selection

Two emerging technologies promise to play a key role in the envisioned model: WebAssembly and Rust. WebAssembly [1], Wasm for short, is a virtual ISA, supported as a compilation target for several languages, such as C, C++, and Rust. Wasm bytecode executes within a sandboxed environment. Using a sandbox allows to isolate the executing component from the host environment, and from other running applications. Wasm bytecode is also suited for network transfer. In the words of [2], WebAssembly could be adequate for implementing a seamless execution environment across the entire Continuum. Rust [3], on the other hand, offers statically assessed memory-safety guarantees thanks to its ownership approach. According to [4], the Rust programming language is a promising technology for embedded programming in the Edge. Using Rust promotes the development of memory-safe applications, reducing memory-related vulnerabilities, as invited in [5]. Together, these two technologies enforce a safe-by-design approach to application development, while contributing to the construction of a seamless runtime infrastructure amenable to the Continuum.

1.2 Scenarios of interest

Pooling Cloud and Edge resources seamlessly might benefit high-performance computing (HPC) applications. In this scenario, the use of containers is limited. The reasons, according to [6], are the following. Containerize solutions usually require root privilege for execution, which is not feasible for HPC components sharing a distributed file system. Secondly, building container images for heterogeneous nodes is complex. It requires central support and cross-compilation, which are unusual requirements for HPC applications, where compilation happens *on-site*, pursuing high optimization. Moreover, access to licensed libraries and compilers is generally not possible at the local level, imposing centralized compilation. When containers are in use, an advanced framework is beneficial to face the complexity of such an infrastructure.

Conversely, compiling a component once, and being able to execute it on every node of the resource pool would overcome those limitations. The work of [6] provides encouraging insights on the use of Wasm in MPI-based (Message-Passing Interface) HPC applications. Similarly, [7] discusses the use of Rust for HPC applications. This second study highlights some aspects of Rust that are particularly suited to HPC contexts, in addition to the already-mentioned safety concerns.

2 PROBLEM STATEMENT

This thesis strives to realize the envisioned model for the Continuum of Computing. To this end, we single out three objectives for our investigations, each of which consolidates a portion of the runtime infrastructure described in Section 1 and constitutes a research contribution:

(1) Develop a migration-capable runtime infrastructure. Migration should preserve the state of a running component, upon resuming. Existing WebAssembly runtimes do not support it. Hence, introducing the core functionalities to support computing mobility is an objective of this work. Also, components transiting in the network might have active connections, acting as either server or client, as discussed in [8]. The capability of performing a connection-preserving migration, regardless of the role in the connection, is paramount.
(2) Orchestrate migrating computations. Modern state-of-the-art Cloud orchestrators show their limitations when applied in the Edge, according to [9]. Moreover, we expect the orchestrator to be able to coordinate the migration of computations,

respectfully to user-defined metrics. Enriching an existing orchestration engine to tackle both the performance issue and migration handling, thus exploring the orchestration policies underneath, is an objective of this project.

(3) Assess the runtime and management cost of the proposed solution. This experimentation should take place along three distinct axes:
- Benchmarking between existing Wasm runtimes and the migration-capable solution emerging in this thesis.
- Compare the performance of the Continuum-native orchestration strategy against a state-of-the-art Cloud orchestrator. A reasonable candidate for the latter is Kubernetes.
- Assess the applicability of said Continuum infrastructure in a real-world High-Performance Computing (HPC) application.

3 PRELIMINARY RESULTS

Computation mobility is already part of previous computing models to some extent. For instance, the Fog model includes an *offloading* mechanism to delegate an execution to a more resourceful node. In that context, computations usually happen within containers. We do not inherit this constraint. In the envisioned model, computations should happen outside of any virtualized environment, and on top of a common runtime. WebAssembly has the potential to embody this runtime layer. There is another important difference between migration and offloading. The latter happens due to necessity, in an attempt to prevent downtime. Instead, migration is triggered by orchestration-level decisions. Migration happens opportunistically, while pursuing user-defined performance metrics. In this process, orchestration plays a key role, which should be explored in depth as the second objective of this project.

Migration could take place in two manners. *Offline* migration requires re-execution. The migrating component is interrupted, in the source node, moved towards the destination node, and then executed again. Doing so does not preserve any previous progress in the computation. With *live* migration instead, the runtime performs a snapshot of the component status, often referred to as *checkpoint*. With a checkpoint, it is possible to resume the computation after a migration occurs. While beneficial in terms of performance, live migration could be challenging. In the case of container technologies, support for live migration is limited, often just as an experimental feature, as noticed in [10]. At the time of writing, neither of those manners is supported in WebAssembly runtimes. Aiming to extend a Wasm runtime to enable migration, we could take at least two routes. One contemplates the migration of interpreted components, leveraging a runtime that offers interpreting capabilities. The other enables the migration of compiled components, which instead requires compile-time support.

3.1 Interpreter-based migration

According to the general understanding, compiled programs are superior to interpreted ones, in terms of run-time performance. On the other hand, interpreter benefit from the ease of deployment, due to their simpler structure. The startup delay for an interpreter is generally shorter than Just-in-Time (JIT) techniques, which consist of compiling the application code straight before executing it. An interpreter might start executing as soon as the migration ends. Also, [10] suggests that interpreters could present less memory footprint compared to compilers. These reasons motivate the investigation towards interpreter-based migration.

To assess the viability of this path, we have selected the only maintained Wasm runtime offering interpreter capabilities. Wasm Micro Runtime [11], WAMR in short, targets Edge devices, with potentially constrained resources. It could be deployed on top of a Real-Time operating system (RTOS), and features two types of interpreter. The `classic_interprter` works as a standard bytecode interpreter. It executes each operand of the invoked Wasm function, without performing any conversion to an intermediate representation (IR). The `fast_interpreter` instead performs *rewriting*, namely, it converts the Wasm bytecode into a format more suitable for interpretation. The initial delay suffered by a rewriting interpreter compared to a standard one, should be motivated by performance improvement in the long run. For this investigation, a standard (without rewriting) interpreter was chosen, to benefit from the minimal initial delay.

The solution proposed consists of three additional mechanisms:

- A runtime mechanism to signal a pending migration request. This procedure should terminate the interpreter in a consistent state, aiming to resume from there once the migration is completed.
- A mechanism to produce a snapshot of the context of the running function. It involves preserving the function call stack.
- A procedure to resume the interpretation from the migrated context, without re-execution.

This solution is under active development. An experimental assessment of the overall result is part of our future work.

3.2 Migrating compiled components

Migrating a compiled component across different ISAs is challenging. Two issues arise here. The first is interrupting the execution of the component in a consistent state. The second is resuming the execution from this consistent state but in a different host, featuring a possibly different ISA. To tackle those criticisms, we could adopt a checkpoint-based mechanism. The idea is to enforce an idiomatic structure in the Wasm component. In our work, we focus on components written in Rust. Within each Wasm module, an initialization and a checkpoint procedure are defined. The runtime is responsible for ensuring that the body of the checkpoint procedure is executed atomically, preventing migration-caused termination. Also, upon resuming the computation in the destination host, the runtime should invoke the initialization procedure using the checkpoint data, and then restart the computation. The checkpoint is located in the component linear memory. Supporting those two facilities would be enough to offer live migration capabilities. Note that, instead of porting the execution context from one ISA to another, as is common practice for other migration technologies, such as CRIU [12], we rely mainly on the Wasm runtime. The former is an error-prone procedure, and, at the time of writing, finds little use in real-world applications, except as a debugger tool.

Modern WebAssembly runtimes already offer the functionalities required for offline migration. To prove this claim, we considered the case of Wasmtime. Wasmtime [13] is a Rust-based runtime, supporting Ahead-of-Time (AOT) and Just-in-Time (JIT) compilation for Wasm modules. This solution could be used for embedding Wasm components in Rust-written applications. In our work, we have shown that offline migration could be achieved with the following steps:

- Take a snapshot of the linear memory. This could be achieved by using dynamic allocation for data that need to be preserved in a migration. For instance, global variables.
- Serialize (and deserialize, once in the destination node) the linear memory. Note that Wasm bytecode is already in a network-friendly format, but that is not the case for the linear memory segment.
- Restart the execution. This step requires (1) restoring the migrated memory, (2) compiling the Wasm component for the new target, and (3) invoking the desired function.

This approach permits migrating components across different ISAs but does not preserve the computation advancement yet.

4 PROJECT'S OUTCOME

The expected outcome of this PhD research includes the following two contributions. Firstly, a viable runtime infrastructure for the Continuum of Computing, characterized by (1) a common execution layer based on WebAssembly and supporting (2) live migration of components with active connection, (3) a space and context-aware orchestration engine for the opportunistic manoeuvring of those computations. This infrastructure will be capable of executing applications developed as an orchestrated set of components, developed in Rust, thus enjoying the language memory safety, and compiled to Wasm bytecode.

Secondly, to support the claim that the Continuum model is capable of contributing to HPC applications, this thesis should study the application of said model in conjunction with a state-of-the-art framework for HPC. The nature of the resulting use case, and the kind of HPC model applied (e.g., MPI), should be formalized over the course of the project. The resulting experimentation will provide empirical evidence and experience on the impact of the Continuum model for application developers.

From these premises, two main research directions arise. The first one consists of exploring the dimension of predictability. Edge devices are used in various contexts, often requiring guarantees on the execution itself. Some of them, such as the Industrial Internet of Things (IIoT) and the automotive sectors, present high criticality requirements. Here, predictability and timeliness are essential. Exploring how to support real-time execution in the Continuum would be a valuable contribution.

The other research direction consists of investigating other use cases for the Continuum of Computing. For instance, the capability of migrating computations might be appealing for energy harvesting systems (EHS). Those systems are made of battery-less devices, with intermittent execution and frequent interruption due to energy shortages. Enriching EHS with migration capabilities might enable them to migrate computations to other devices just before a low-energy phase. Doing so would enable these systems to achieve higher levels of utilization.

REFERENCES

[1] Webassembly, 2022. https://webassembly.github.io/spec/core/index.html.
[2] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '22, page 3–8, New York, NY, USA, 2022. Association for Computing Machinery.
[3] Rust programming language, 2024. https://www.rust-lang.org/.
[4] Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. Tighten rust's belt: shrinking embedded rust binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, page 121–132, New York, NY, USA, 2022. Association for Computing Machinery.
[5] United States Cybersecurity and Infrastructure Security Agency, United States National Security Agency, United States Federal Bureau of Investigation, Australian Signals Directorate's, Australian Cyber Security Centre, Canadian Centre for Cyber Security, United Kingdom National Cyber Security Centre, New Zealand National Cyber Security Centre, and Computer Emergency Response Team New Zealand. The case for memory safe roadmaps. Technical report, United States Cybersecurity and Infrastructure Security Agency, 2023.
[6] Mohak Chadha, Nils Krueger, Jophin John, Anshul Jindal, Michael Gerndt, and Shajulin Benedict. Exploring the use of webassembly in hpc. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 92–106, New York, NY, USA, 2023. Association for Computing Machinery.
[7] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. Gpu programming in rust: Implementing high-level abstractions in a systems-level language. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 315–324, May 2013.
[8] Carlo Puliafito, Luca Conforti, Antonio Virdis, and Enzo Mingozzi. Server-side quic connection migration to support microservice deployment at the edge. *Pervasive and Mobile Computing*, 83:101580, 2022.
[9] Giovanni Bartolomeo, Simon Bäurle, Nitinder Mohan, and Jörg Ott. Oakestra: an orchestration framework for edge computing. In *Proceedings of the SIGCOMM '22 Poster and Demo Sessions*, SIGCOMM '22, page 34–36, New York, NY, USA, 2022. Association for Computing Machinery.
[10] Ben L. Titzer. A fast in-place interpreter for webassembly. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022.
[11] Webassembly micro runtime, 2022. https://bytecodealliance.github.io/wamr.dev/.
[12] Checkpoint/restore in userspace, 2023. https://criu.org/Main_Page.
[13] Wasmtime, 2024. https://wasmtime.dev/.

Received 02 March 2024; revised 17 April 2024; accepted 09 April 2024