CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D4.2 Cloud-Edge Cells Release Candidate and Specifications

Due date of deliverable: 30-06-2024
Actual submission date: 28-06-2024

Start date of project: 01-01-2023 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	23
WP/Task related to this document	WP4
WP/Task responsible	IMP
Leader	Peter Pietzuch (IMP)
Technical Manager	Carlos Segarra (IMP)
Quality Manager	Marc Sanchez-Artigas (URV)
Author(s)	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP), and Ardhi Putra Pratama Hartono (TUD)
Partner(s) Contributing	IMP, TUD
Document ID	CloudSkin_D4.2_Public.pdf
Abstract	First prototype of a system using C-Cells to optimize resource usage and performance by exploiting C-Cell migration and elastic scaling. This reference implementation can be used to implement other systems on top of C-Cells.
Keywords	WebAssembly, MPI, OpenMP, orchestration, scientific applications, Azure Batch, Slurm, cluster, high-performance-computing

History of changes

Version	Date	Author	Summary of changes
0.1	24-05-2024	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP)	First draft's skeleton.
0.2	28-05-2024	Ardhi Putra Pratama Hartono (TUD)	Adding confidential computing perspectives
0.3	28-05-2024	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP)	Fill in main sections of the document.
0.4	30-05-2024	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP)	First full version ready for review.
0.5	26-06-2024	Marc Sanchez-Artigas	Provide feedback on deliverable.
0.6	26-06-2024	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP)	Pass on the paper after feedback.
1.0	28-06-2024	Carlos Segarra (IMP), Huanzhou Zhu (IMP), Guo Li (IMP), Peter Pietzuch (IMP)	Final version.

Table of Contents

List of Abbreviations and Acronyms

AI	Artificial Intelligence
AOT	Ahead-of-Time
API	Application Programmable Interface
C-Cell	Cloud-edge Cell
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
EPC	Enclave Page Cache
GB	Giga Byte
GPU	Graphic Processing Unit
IoT	Internet of Things
JIT	Just-in-Time
JS	JavaScript
JVM	Java Virtual Machine
MEE	Memory Encryption Engine
MPI	Message Passing Interface
OMP	Open MP
OS	Operating System
PRM	Processor Reserved memory
SDK	Software Development Kit
SFI	Software Fault Isolation
SGX	Software Guard eXtensions
TCB	Trusted Computing Base
TDX	Trust Domain eXtensions
TEE	Trusted Execution Environment
TPU	Tensor Processing Unit
VM	Virtual Machine
WASI	WebAssembly System Interface
WASM	WebAssembly
WAST	WebAssembly Text Representation

1 Executive Summary

This deliverable presents GRANNY, the first full-system implementation using C-Cells as its execution unit. GRANNY focuses on one particular workload: scientific applications using MPI and OpenMP. We first motivate why these applications are relevant to CLOUDSKIN. Then we describe GRANNY, and show how by using C-Cells, and C-Cell-migration, we can improve the performance and resource utilization of large clusters of VMs running these applications. In addition, we also provide a first prototype of extending GRANNY with higher privacy and security guarantees. In our future work we describe how GRANNY can be used as a reference implementation for future systems using C-Cells as their execution unit.

2 Introduction

CLOUDSKIN's goal is to build an environment for the Cloud-edge continuum. This work package, WP4, is responsible for the design and implementation of a key piece in the Cloud-edge continuum: its execution unit. We have named the execution units of the cloud-edge continuum Cloud-Edge Cells (or C-Cells, for short). In D4.1 we presented the first prototype implementation for C-Cells. In this deliverable, D4.2, we present GRANNY, the first implementation of a complete distributed system using C-Cells as its executing unit.

There are several application domains that would benefit from executing in CLOUDSKIN's cloud-edge continuum. For the first reference implementation, in D4.2 we pick a domain that is suited to the current limitations of C-Cells. As previously described in D4.1, C-Cells currently only support execution in X86 [1] enviornments. Support for AArch64 [2] is experimental, and not ready for a production environment, and do not have accelerator (e.g. Graphic Processing Unit (GPU) [3], TPU [4], FPGA [5]) support. Consequently, AI workloads (other than a simple inference), are out of the scope. On top of that, we want an application domain that can benefit from C-Cells' high-performance execution and low-overhead live migration. Our application domain of choice is, thus, long-running compute-intensive scientific applications.

To support different flavours of scientific applications, or multi-tenant execution, C-Cells also need to be compatible with different software and hardware techniques for ensuring high-performance execution while keeping up the isolation between applications. Here we also present the state-of-the-art isolation support for C-Cells by using a Trusted Execution Environment, namely Intel SGX. Enveloping C-Cells execution by Intel SGX has two-fold benefits: application can have isolation from both other applications as well as the privileged entities such as the Operating System (OS) while having minimal performance degradation in the long-running compute-intensive scientific application.

In this deliverable, we present GRANNY, a system for granular management of compute-intensive applications using C-Cells. To make our system as generic as possible, and given C-Cells' language and API independence, we tackle a very broad class of scientific applications. GRANNY supports any application written using the popular OpenMP [6] or MPI [7] APIs. These two APIs have for decades been the *de-facto* standard to implement multi-threaded (for scale-up) and multi-process (for scale-out) applications. We also believe that targeting scientific applications using OpenMP/MPI will facilitate the adoption of GRANNY (and C-Cells) by our use-cases, and the scientific community more broadly. We also present our prototype of equipping C-Cells with Intel SGX using SCONE framework.

The rest of the document is structured as follows. §3 provides background and motivation into why we have chosen scientific applications and OpenMP/MPI. §4 presents GRANNY, the first distributed system that uses C-Cells to execute unmodified applications. §5 presents a thorough and exhaustive evaluation of different policies that we can build based on C-Cells, and C-Cell migration. To finish-up, we outline future lines of work in §6.

3 Background: Compute-Intensive Applications and Why They Matter

Compute-intensive applications are common in many domains including machine learning [8], weather forecasting [9], hydrodynamics [10], genomics [11], simulation, and modeling [12]. These applications must exploit parallelism, and therefore, they often use programming models that employ multiple threads (with shared memory) and/or multiple processes (with message passing), such as OpenMP [6] and MPI [7]. These programming models allow applications to scale to hundreds or thousands of CPU cores across nodes.

To accommodate their resource demands, such applications are deployed on shared clusters with high node and CPU core counts, either on-premise or, increasingly, in the cloud. Clusters are managed by resource managers, *e.g.* Slurm [13] or Azure/AWS/Google Batch [14, 15, 16], which monitor a queue of submitted application jobs and allocate them to resources (*e.g.* CPU cores).

Existing resource managers, however, cannot alter an application's resource allocation after it has started executing. This is because existing shared memory/message passing runtimes [17, 18] do

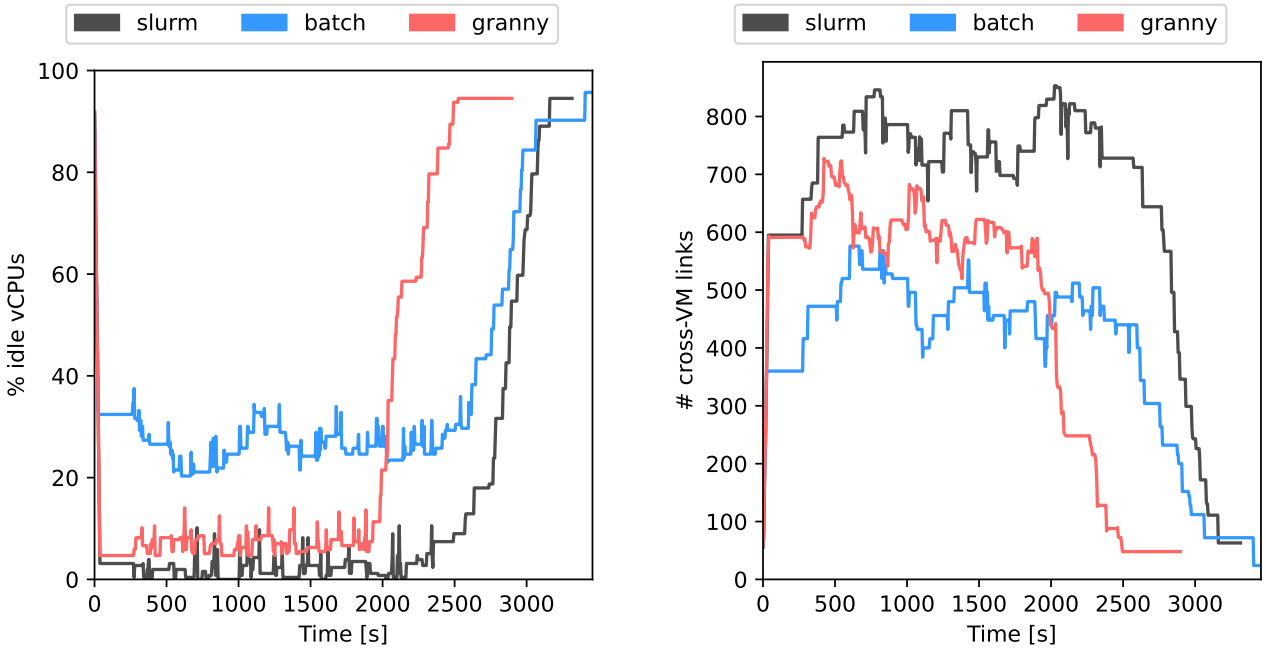

Figure 1: Time-series of idle vCPUs (left) and cross-VM network links (right) when executing a trace of 100 MPI jobs on a 32-VM cluster. We compare GRANNY, with two state-of-the-art batch schedulers: Azure Batch [14] and Slurm [13].

not support modifications to distribution or parallelism at runtime. Such changes could violate the consistency of message-passing or shared data: *e.g.* when changing distribution in MPI, messages may be in-flight; when changing parallelism in OpenMP, shared data structures may be inconsistent pending synchronization. This lack of flexibility prevents these compute-intensive applications from harnessing the benefits of elastic fine granular resource management in the cloud.

As an example, consider the tension between compute/ data locality and resource utilisation in clouds [19, 20]: high utilisation can be achieved by allocating resources at fine-grained granularity (*e.g.* allocating CPU cores to applications, as they become available), but this leads to fragmented resources, and thus worse communication performance. Different schedulers handle this tension differently: when executing MPI applications, the Azure Batch [14] scheduler allocates resources at VM granularity. Azure Batch thus achieves good locality at the cost of resource utilization, because idle CPUs cores in VMs cannot be used to execute other applications; in contrast, Slurm [13], another popular scheduler, allocates at CPU granularity. Thus, Slurm has high utilization but incurs fragmentation. While high utilization is desirable for cloud providers, high locality, as a proxy for performance, is desirable for users. We observe that an ideal cloud scheduler must thus navigate this trade-off by, *e.g.* compacting applications at runtime as new cluster resources become available.

Fig. 1 shows the execution of 100 MPI jobs on a 32 VM cluster (§5). We present the time-series of idle vCPUs as a proxy for resource utilization and the number of cross-VM links as a proxy for locality. We compare GRANNY with Azure Batch [14] and Slurm [13]. As previously introduced, Azure Batch allocates resources at VM granularity, so it therefore achieves optimal locality (in terms of cross-VM links) at the cost of leaving 25% of resources (in terms of vCPUs) idle. On the other hand, Slurm allocates resources at CPU granularity, achieving high utilization at the cost of fragmentation and poor locality, and therefore poor performance. With GRANNY, we can navigate the trade-off to achieve optimal performance and utilization. In this experiment we configure GRANNY to target a threshold utilization (95% in this case) and use the remaining idle vCPUs to compact applications as they become fragmented. As a consequence GRANNY can improve performance on both baselines by up to 25%.

Domain	Name	Language	MT	MP
Molecular dynamics	LAMMPS [26]	C++	✗	✓
	MDAnalysis [27]	Python	✓	✗
Bio-informatics	BioPython [28]	Python	✓	✗
	gatk [29]	Java	✓	✗
Fluid dynamics	OpenFOAM [10]	C++	✗	✓
	SU2 [30]	C++	✓	✗
Deep learning	OpenCV [31]	C++	✗	✓
	Tensorflow [32]	Python	✓	✗

Table 1: Github's most-starred projects in compute-intensive domains use multi-threading (MT) or multi-processing (MP)

In addition, no matter how effective a scheduler's bin-packing approach is, it cannot completely avoid idle resources. This is particularly true for multi-threaded applications, which cannot be distributed across VMs. Our experiments show that, when deploying OpenMP applications, Azure Batch and Slurm consistently leave 60% and 40% of CPUs idle, respectively, even when there are still pending applications to be scheduled (§5.2). Cloud providers have developed elastic execution models such as serverless [21, 22, 23] to increase utilization, but these are not well-suited for executing existing shared memory or message passing applications without major changes.

Finally, failures and evictions impact the resources that long-running applications have in cloud environments. For example, spot VMs [24] have gained prominence as a cost-effective way to run cloud workloads, but any spot VM may be evicted after a short grace period (*e.g.* 1 min on Azure [25]). Existing schedulers, such as Azure Batch, are forced to re-start execution after an eviction was detected, because current shared memory and message passing runtimes cannot handle such a fine-grained resource change.

We argue that all of the above challenges can be addressed through executing threads and processes as C-Cells, enabling fine-granular resource management at runtime with low-overhead migration. In the rest of this section, we characterize compute-intensive applications (§3.1), describe associated resource managers (§3.2), and explain why current shared memory and message passing runtimes fail to support the flexible and fine granular resource management needed in cloud environments (§3.3).

3.1 Compute-intensive applications

Compute-intensive applications include large-scale data analytics [33], video processing [34], and deep learning training [32], and also typical high performance computing (HPC) workloads, such as fluid dynamics [10], molecular simulation [26], and weather forecasting [9]. These applications require plentiful hardware resources, and they must parallelize the computation by distributing it across many CPU cores, both within nodes and across nodes.

To handle increasing problem sizes without increasing execution time or exhausting memory, compute-intensive applications make use of scale-up and scale-out patterns. Such patterns are implemented using multi-threading (with shared memory) and/or multi-processing (with distributed message passing), as offered by programming models such as OpenMP [6] and MPI [7]. Tab. 1 shows the most starred open-source GitHub repositories in several application domains, and how all of them use either or both of these programming models.

3.2 Cluster resource managers

Compute-intensive applications are deployed on large clusters (either on-premise or in the cloud) with high CPU core and node counts. Users submit applications as jobs to a work queue managed by a cluster resource manager. The resource manager allocates jobs to compute resources (*e.g.* CPU cores) according to a job's demand, *e.g.* as indicated by `mpirun`'s `np` flag [35], or the `OMP_NUM_THREADS`

	High utilization	High locality	Eviction resilience
Azure, AWS, GoogleBatch [15, 14, 16]	✗	✓	✗
Slurm [13], Volcano [37], Kube-Batch [38]	✓	✗	✗
GRANNY	✓	✓	✓

Table 2: Comparison of cluster resource managers

environment variable [36]. Tab. 2 summarizes existing cluster resource managers and their associated trade-offs.

```
int[] weights = initWeights();

for (int i = 0; i < numSteps; i++) {
#pragma omp parallel shared(weights) {
    int threadNum = omp_get_thread_num();
    int nThreads = omp_get_num_threads();
    updateWeights(
        weights, threadNum, nThreads);

#pragma omp single
    applyWeights(weights);
}
```

Listing 1: Pseudocode for machine learning training using OpenMP's `parallel` abstraction (Within the `parallel` block, the OpenMP runtime controls the degree of parallelism, and ensures access to and synchronisation of the shared variable *weights*.)

```
int worldSize, rank;
MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int[] weights = initWeights();

for (int i = 0; i < numSteps; i++) {
  updateWeights(weights, rank, worldSize);
  MPI_Allreduce(
    MPI_IN_PLACE, weights, nWeights,
    MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank == 0) applyWeights(weights);
}
```

Listing 2: Pseudocode for machine learning training using MPI's `MPI_Allreduce` function (The MPI runtime controls the degree of parallelism, data partitioning and messaging topology.)

Cloud providers use resource managers such as AWS Batch [15], Azure Batch [14], and Google Batch [16] to execute shared memory and message passing applications. These resource managers have a pool of VMs that can be scaled up or down on-demand. They schedule jobs once sufficient VMs are available, thus optimizing for locality. By scheduling to as few VMs as possible, they minimize inter-VM communication, thus improving per-job performance, but this comes at the cost of utilization: idle CPU cores in VMs cannot be allocated to other pending jobs.

For finer-grained control over the scheduling logic and the underlying VM pool, some cloud providers support general-purpose resource managers such as Slurm [13], KubeBatch [38], or Volcano [37], which are also often used for on-premise clusters. Slurm maximizes utilization by allocating resources at CPU core granularity, but this comes at the cost of locality, as the VMs become fragmented over time.

Existing resource managers must therefore choose between high utilization or high locality, because they cannot change the distribution or parallelism of jobs at runtime. In addition, when the underlying resources changes, *e.g.* because a spot VM is evicted from the pool, an affected job must be restarted.

3.3 Shared memory/message passing runtimes

Multi-threaded applications with shared memory or multi-process applications with message passing require runtime support. Runtimes such as OpenMP [6] provide functionality to manage threads, coordinate access to shared state, and provide synchronization primitives; runtimes such as MPI [7] also handle inter-node messaging, message synchronization, and data partitioning.

Consider two sample implementations of stochastic gradient descent (SGD) [39], a core algorithm in machine learning training—one with OpenMP (Listing 1) and another with MPI (Listing 2). As shown in Listing 1, OpenMP's `omp parallel` construct means that the `for` loop can be executed in parallel with access to a single shared variable, the `weights` vector. The OpenMP runtime then synchronizes writes to the shared variable from multiple threads. Similarly, in Listing 2, the `MPI_Allreduce()` operation allows concurrent processes to execute the same function to transform and aggregate results on multiple nodes, sending messages to synchronize.

Such runtimes, however, were not designed with features that would allow a cloud provider to manage job resources dynamically and control parallelism at a fine granularity. In particular, the execution abstractions used by runtimes, namely threads and processes, are tied to resources (CPU cores) and do not support dynamic reallocation. Runtimes decide on the allocated resources (CPU cores and VMs) and degree of parallelism at deployment time: in Listing 1 (lines 5–6), the program queries the runtime to obtain the current thread identifier and the total number of threads; in Listing 2 (lines 2–3), the program is given the number of nodes (`worldSize`) and the current process identifier (`rank`) from the runtime. Any change in the resource allocation would require the migration of execution threads or processes, which is unsupported.

While compute-intensive applications could use existing process-level (*e.g.* CRIU [40]) or VM-level migration techniques [41], such techniques incur significant overheads [42], which are well understood [43] *e.g.* when migrating during a parallel section in OpenMP, the memory of all threads must be transferred. For MPI applications, these challenges are further exacerbated: since MPI applications are distributed, a single-process checkpointing approach cannot ensure consistent checkpoints across all processes. Applying existing migration techniques [44] thus requires extensive changes to applications and OS kernels.

We observe that what existing runtimes for compute-intensive applications lack is a single abstraction that unifies thread- and process-based parallelism. Such an abstraction could then enable efficient fine granular resource management in cloud environments, including elastic scaling and dynamic migration. In the next section we argue why C-Cells are the right implementation of this abstraction.

4 GRANNY

In this deliverable, we present the design of GRANNY, a new cloud runtime for compute-intensive applications (§4.1). At the core of GRANNY are C-Cells (§4.2). C-Cells support efficient migration, because their full state can be captured using snapshots (§4.3). GRANNY controls the execution of Granules at well-defined control points (§4.4). At each control point, GRANNY may decide to spawn a C-Cell to add a new thread or process (§4.5), or migrate a C-Cell to another VM (§4.6). Executing threads and processes as C-Cells, GRANNY can implement a variety of fine grain resource management policies (§4.7), as well as providing C-Cells with additional security guarantees with Intel SGX (§4.8).

4.1 Overview

With GRANNY, each shared memory and message passing application is executed as one or more C-Cells (shown as circles in Fig. 2). GRANNY executes one runtime instance per VM, and each instance manages the set of C-Cells executing on that VM. The managed C-Cells can belong to multiple applications (indicated by different colors in Fig. 2). GRANNY employs a logically centralized resource manager, the Planner , to schedule C-Cells on nodes and implement policies for managing C-Cells (§4.7). Each application has a compute demand in terms of CPU cores (indicated by a number in the figure).

Fig. 2 shows how GRANNY implements a policy for defragmenting applications to increase locality: it migrates C-Cells between VMs when new idle resources become available in order to consolidate the application onto fewer VMs. In Fig. 2-a, GRANNY eagerly schedules the yellow application from the submission queue when there are at least six compute slots available in the VM pool. When the red application completes (Fig. 2-b), the Planner can use the newly freed slots to migrate the

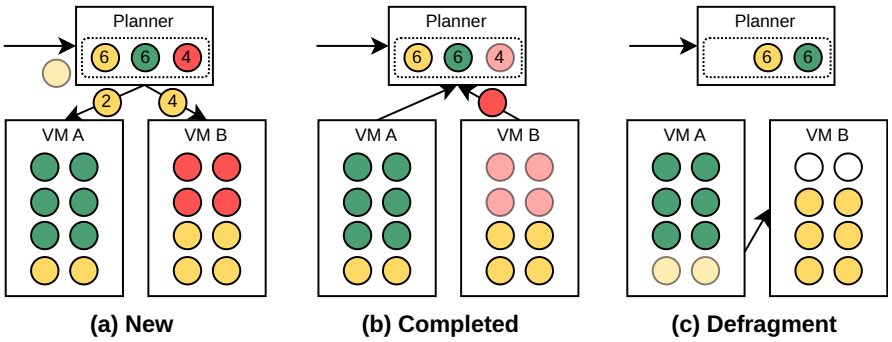

Figure 2: Management using GRANNY's distributed runtime. Each GRANNY instance runs on a VM and controls a variable-sized pool of C-Cells. A Planner defragments applications to increase locality when other applications have completed.

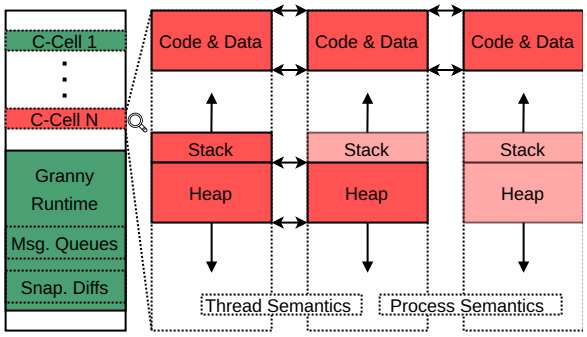

Figure 3: C-Cells on a VM execute within a single virtual address space. Each C-Cell has a simple memory layout that can be used to spawn new child granules with thread or process semantics.

C-Cells of the yellow application between VMs, thus defragmenting the application (Fig. 2-c).

4.2 C-Cell abstraction

In this section, we re-introduce C-Cells in the context of GRANNY. C-Cells unify multi-threaded and multi-process execution, and C-Cells can interact with each other through regular shared memory/message passing APIs.

For this, C-Cells can share memory directly with each other (i) to implement thread semantics and (ii) to exchange messages efficiently when implementing inter-process communication. C-Cells are therefore implemented as WebAssembly (WASM) modules [45]: application code is cross-compiled to WASM, as a platform independent binary instruction format. The use of WASM allows C-Cells in a VM to execute side-by-side within a single virtual address space, together with the GRANNY runtime (Fig. 3, left), while enforcing memory safety [46].

To preserve thread/process semantics, each C-Cell is spawned from a parent C-Cell: a C-Cell executing a thread shares the static sections (code and data) and heap area with its parent (Fig. 3, thread); a C-Cell executing a process only shares the static section with its parent (Fig. 3, process).

To exchange messages between C-Cells, which is needed for message passing, C-Cells use shared memory queues and the GRANNY runtime manages cross-VM delivery over the network. C-Cells have consistent addresses that remain unchanged, even when a C-Cell is migrated between VMs. When two C-Cells are located on the same VM, message delivery by the runtime exploits efficient local shared queues.

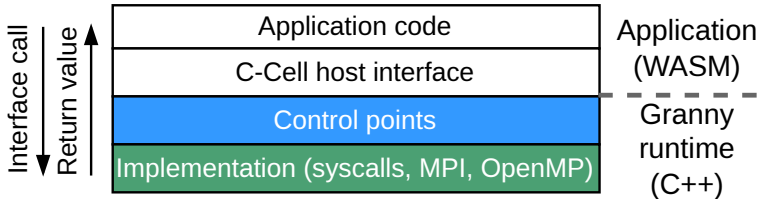

Figure 4: Control points are triggered when application code calls certain APIs, before their implementation is executed.

4.3 C-Cell snapshots

WASM's linear memory model [45] means that C-Cells have a simple memory layout (Fig. 3, right). Therefore, a C-Cell's entire execution state can be encapsulated in a snapshot . A snapshot contains the linear memory, with its stack and heap and function tables. It also includes C-Cell state that is stored in the host's runtime like messaging queues and open file descriptors.

Having a concise snapshot representation for C-Cells is an essential requirement for migration. Since all state is contained in a snapshot, GRANNY does not require OS kernel modifications to obtain the full execution state of a C-Cell. This is in contrast to general process checkpointing [40], which must also extract process state from the OS kernel.

Since C-Cells are spawned from parent C-Cells, a new C-Cell shares memory with its parent. This enables GRANNY to implement optimizations that improve the performance of C-Cell spawning (§4.5) and migration (§4.6). In particular, GRANNY can efficiently represent the difference between the snapshots of two C-Cells through byte-wise diffs . A byte-wise diff is a list of memory spans that describe regions in which the two snapshots differ. GRANNY uses byte-wise diffs to reduce the amount of memory transferred between VMs when C-Cells are spawned on remote VMs or migrated.

4.4 Interrupting C-Cells at control points

GRANNY takes control over C-Cell execution at control points . A control point is triggered by a system call or a call to a shared memory/message passing API. There are two types of control points: (i) regular control points and (ii) barrier control points. At regular control points, such as system calls, the application state is not guaranteed to be consistent, but this still allows GRANNY to send point-to-point messages or operate on shared memory. In contrast, barrier control points guarantee that the application state is consistent, *e.g.* no messages must be in-flight, and no shared memory areas must be pending to be synchronised. Therefore, GRANNY can only trigger C-Cell migration at barrier control points.

Creating barrier control points requires semantic knowledge of the shared memory/message passing API. For example, the implementation of `MPI_Barrier` has a reduce phase in which all C-Cells send messages to the C-Cell with the lowest MPI rank, and a broadcast phase in which all C-Cells are notified that the barrier has completed. After the reduce phase, the C-Cell with the lowest MPI rank can rely on the fact that there are no outstanding messages, and has thus reached a consistent state.

Fig. 4 shows the call stack when application code triggers a control point. We refer to the set of all possible such APIs that application code may call as the C-Cell's host interface. The C-Cell host interface combines system/POSIX, MPI and OpenMP APIs: for the system/POSIX interface, GRANNY uses WASM's system interface (WASI [47]); for MPI, it uses the standard `MPI_*` APIs, *e.g.*`MPI_Reduce` [7]; for OpenMP, it uses the underlying LLVM OpenMP runtime interface, *i.e.* once pragmas have been transformed to function calls, *e.g.*`__kmpc_fork_call` [48].

GRANNY must intercept calls for control points without a large runtime overhead. The symbols corresponding to these calls are left undefined when cross-compiling the application to WASM, and marked as function imports [49]. The symbols are then implemented by the GRANNY runtime and linked to C-Cells at runtime. While all of these calls trigger a WASM context switch, these context

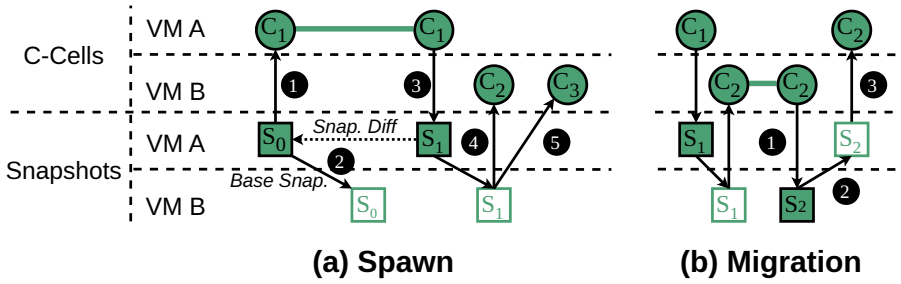

Figure 5: Snapshots spawn new C-Cells. Each C-Cell is spawned from a parent snapshot. All C-Cells in an application have a common base snapshot, and all subsequent snapshots are expressed as byte-wise diffs from the base. Byte-wise diffs for child C-Cells (with process semantics) ignore the heap and stack sections; byte-wise diffs for migration do not.

switches are fast (tenths of cycles, similar to a function call [50]).

4.5 Spawning C-Cells from byte-wise diffs

As described in §4.2, C-Cells are spawned from a snapshot of their parent C-Cell. When both parent and child are co-located in the same VM, and thus in the same virtual address space, GRANNY spawns a C-Cell by mapping its memory regions to the parent's snapshot. When parent and child are not co-located, *e.g.* in a distributed MPI application, GRANNY sends a snapshot (or a byte-wise diff) over the network and spawns the child from the parent's snapshot.

By sending a byte-wise diff instead of a full snapshot, GRANNY greatly reduces the amount of data transferred, *e.g.* avoiding sending the code and data sections, which can be up to hundreds of MBs for large compute-intensive applications, thus reducing remote spawn time.

Fig. 5-a shows local and remote C-Cell spawning in more detail. All C-Cells in an application share a base snapshot with the code and data sections. The first C-Cell is spawned locally from the base snapshot (Fig. 5-a, ❶). In the background, base snapshots can be distributed to other VMs ❷. When a C-Cell starts a remote spawn, GRANNY takes a snapshot of the C-Cell ❸ and sends it to the destination VM. Note that this second snapshot is expressed as a byte-wise diff of the base snapshot (§4.3). Finally, the new C-Cell can be spawned locally from the copy of the snapshot ❹. Subsequent remote spawns from the same C-Cell result in local spawns from the snapshot copy ❺.

4.6 Migrating C-Cells

C-Cell migration is similar to a remote C-Cell spawn with one difference: when spawning remote C-Cells with process semantics (*e.g.*`MPI_Init`), the byte-wise diff must ignore regions in the snapshot that do not need to be synchronized, such as the stack and the heap. When migrating a C-Cell, the byte-wise diff does not ignore the stack and the heap, as execution must resume from where it stopped.

Fig. 5-b shows the steps of C-Cell migration. After a C-Cell reaches a barrier control point, the Planner decides if the C-Cell must be migrated based on its policy (not shown). If the C-Cell is to be migrated, GRANNY takes a new snapshot (Fig. 5-b, ❶), and sends it, as a byte-wise diff, to the destination VM, together with any missing prior snapshots ❷. Note that, under a defragmentation policy that co-locates C-Cells as resources become available (Fig. 2), prior snapshots will already likely exist at the destination VM due to previous migrations, thus reducing costs. Finally, the migrated C-Cell is spawned locally from the local snapshot ❸.

4.7 Granular Application Management

The unified abstraction of C-Cells, which offers migratable threads and processes, enables new polices for fine granular resource management in cloud environments. State-of-the-art resource managers such as Azure Batch [14] and Slurm [13] are limited to resource allocation decisions at deploy-

ment time: they must decide on the specific resources and their number when a job is removed from the job queue, and any decision cannot be changed even if it turns out to be suboptimal.

GRANNY, on the other hand, is not limited to making scheduling decisions only when jobs are removed from the queue. Instead, C-Cells can be spawned, interrupted, checkpointed, and migrated at runtime without violating shared memory or message passing semantics. This ability allows for new dynamic scheduling policies. In this section, we describe different policies supported by our GRANNY prototype.

4.7.1 Improving locality

Existing resource managers can either optimize for locality, by delaying job allocation until sufficient unfragmented resources have become available, or for utilization, by allocating jobs to any available resource. In contrast, GRANNY's Planner can allocate resources greedily and then defragment (*i.e.* compact) the cluster when other resources become available (as shown in Fig. 2). We refer to this as a compaction policy.

A compaction policy is useful for message passing applications that benefit from being co-located, because it minimizes the number of cross-VM messages that must be communicated over the network. When an application reaches a barrier control point, the Planner checks if it can benefit from a change of distribution that improves locality by reducing the number of cross-VM network links. Especially, for network-bound applications, this improvement in co-location leads to an improvement in performance. For other cluster sizes or heterogeneous clusters, we envision other policies that consider co-location, *e.g.* in terms of NUMA awareness or rack placement.

We show experimentally in §5.1 that, using a compaction policy, GRANNY maintains high cluster utilization while keeping fragmentation low, improving performance.

4.7.2 Improving resource utilization

Even with a compaction policy, some resources in the cluster may still be idle due to the nature of the bin-packing scheduling. For example, each cluster node may have some spare capacity that is insufficient to deploy a new application. With GRANNY, we can utilize this capacity by leveraging C-Cells that are part of multi-threaded (OpenMP) applications. Such applications contain `parallel` code sections that can be scaled elastically at runtime.

To benefit from elasticity, an elastic policy greedily assigns extra available local CPU cores to applications whenever an application reaches a barrier control point. Such a policy may slightly delay the scheduling of pending jobs, but it maximizes cluster utilization and improves job performance by exploiting extra parallelism.

In §5.2, we show experimentally that this policy improves end-to-end execution time while decreasing idle CPU cores.

4.7.3 Eviction from ephemeral resources

Compute-intensive applications may be deployed on a pool of spot VMs [24] to benefit from lower costs. Here, application processes face eviction when spot VMs are withdrawn by the cloud provider after a short grace period. Despite this, the application must make steady progress even in the presence of these partial failures.

To achieve this goal, it is possible to implement a policy in GRANNY that follows a two-part migration approach: (1) when the cloud provider notifies the Planner of an upcoming spot VM eviction from the pool, it stops the scheduling of C-Cells on that VM; (2) when the existing C-Cells on the to-be-evicted VM reach a barrier control point, the Planner tries to reschedule them on the remaining VMs. If there are insufficient available compute resources, snapshots are taken of all affected C-Cells, send to the Planner, and the C-Cells are terminated. Jobs with these interrupted C-Cells are then added to the beginning of the job queue.

In §5.3, we show experimentally that, using this policy, GRANNY can reduce end-to-end execution time when deploying compute-intensive applications using spot VMs.

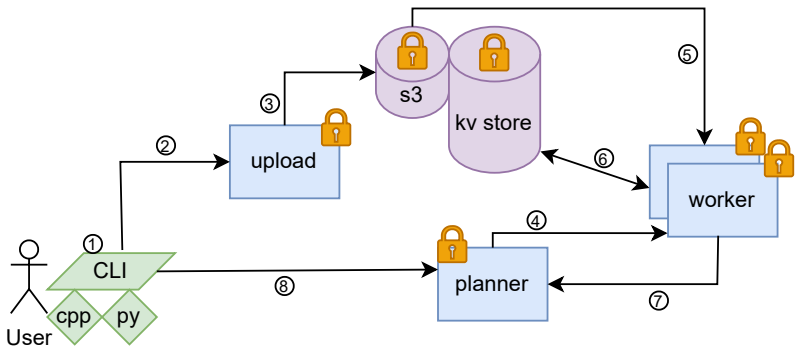

Figure 6: Initial Confidential C-Cells architecture

4.8 Confidential C-Cells with Intel SGX

To support confidential computing within in Cloudskin architecture, we use SCONE to lift-and-shift not only local C-Cells runtime but also other components as well. Currently, SCONE supports Intel SGX as the hardware-backing Trusted Execution Environment (TEE). Intel SGX provides a powerful tool for developers to create secure, trusted, and performant applications that protect sensitive data and computation.

SCONE is able to convert user application to be SGX-compatible, either by recompilation or automatic image generation. However, some protection that is outside of the Intel SGX's scope are still needed. In this case, SCONE also provides a mechanism to ensure data-at-rest integrity through its file system shield as well as protecting network interaction via its network shield.

4.8.1 The importance of confidential computing on Cloudskin architecture

As mentioned in Deliverable D4.1, Cloudskin's (and GRANNY's) architecture is based on Faasm, a stateful serverless runtime. In Fig. 6 shows the overview of its architecture with respect to confidentiality. We will then elaborate on what is the importance of enveloping some of the components with a TEE.

A user could compile its scientific application written in its language of choice, for example, CPP or Python. This component lies on the user side and hence, does not need to be protected. A user, then, can upload the program to an *upload* server. *Upload* server' task is to receive all function, state, and shared file uploads. It can then forward necessary and suitable binary to the S3-compatible storage. As a user, it is important to trust the *upload* server since it will transform a user program to be compatible with workers capabilities. Similarly, a user needs to trust S3-compatible storage either by encrypting the data, relying to a third-party, or self-hosting it with TEE. Here we focus with the self-hosted solution by deploying TEE-powered MinIO.

Then, at some point, the central *planner* will delegate task execution to one of the *worker*s. Since the adversary could take control of, or impersonate both, it is important to make sure that both are trusted. Compromised *planner* can, for example, fake the workload scheduling in such a way that sensitive computation method or data could be sniffed and manipulated. A compromised *worker* combined with a powerful adversary controlling the orchestrator could fake and alter the workload execution, making the user have incorrect perception.

4.8.2 Implementation of Confidential C-Cell

We have implemented C-Cells, which is based on Faasm and Faaslet, to be compatible with Intel SGX supported by SCONE framework. Most of the components described in Fig. 6 is now TEE-protected, except the one on the user side as we deemed it is not necessary. In general, the implementation could be described in a few phases:

In the first phase, we need to ensure the compatibility between Faasm runtimes and Intel SGX. Currently, Faasm support two runtimes: WAVM and WAMR. We focus on WAMR since WAVM has not been actively developed for the last two years. This decision also limit the language the user can use, restricting them to deploy only CPP-based workload. We made minor changes to WAMR runtime as well as set up default workload configuration that is compatible with Intel SGX.

In the next phase, we involved both SCONE and Faasm developers. We adapted the runtime's dependencies to be compatible with SCONE. SCONE is based on musl libc instead of a more popular but rather bloated GNU libc. Therefore, some libraries do not support musl out-of-the box. Our contribution is not only beneficial to confidential adaptation but also to the upstream to support more cases in the future. One of the examples of such case is the lack of a *backtrace* function in musl libc.

In a separate issue, we also disabled hardware bound checks with the WAMR runtime. This is due to how SCONE allocates memory which causes heap fragmentation. WAMR, as one of the WebAssembly runtimes, allocates linear virtual memory at the beginning. Although this virtual memory allocation will be rarely used due to the nature of bound checking, this is an acknowledged problem inside an SGX enclave. Enabling manual memory allocation using `malloc` and checking it afterwards is still supported. Since SCONE is still actively developed, hopefully this issue will be resolved in the future. Alternatively, one could implement a garbage-collection like approach on the runtime, although that might introduce complications as hinted in the upstream.

Last important phase is to also support other components in Cloudskin architecture, namely Redis and MinIO. Although users have an option to trust a third-party service, we believe deploying both services on-premise offers more control and gives more relevance to the project as a whole. We in collaboration with SCONE, add experimental support on gcc compiler as the Go compiler. In this experimental support, we replace all syscall instructions in Go to use the SCONE syscall interceptor. Moreover, we introduce a dedicated runtime stack for system calls due to the expansive nature of the Go stacks. Initially, SCONE relies on dynamic linking to enable applications running on top of Intel SGX. We also provide an alternative approach for Go in such a way that it is also compatible with the statically linked executable.

5 Evaluation

Our evaluation explores the benefits of using GRANNY to run compute-intensive applications in the cloud in terms of: (i) improving performance and locality while maintaining a target utilization with compaction (§5.1); (ii) improving performance and utilization by allocating extra CPU cores (§5.2); and (iii) ensuring efficient fault-tolerant execution with ephemeral spot VMs (§5.3).

To break down where these benefits come from, we also analyze: (i) the overhead of executing MPI applications with C-Cells instead of processes (§5.4); (ii) the overhead of executing OpenMP applications with C-Cells instead of threads (§5.5); (iii) the cost of migrating C-Cells (§5.6); and (iv) the cost of scaling-up to use extra CPU cores (§5.7).

Lastly, we also analyze the baseline costs of making C-Cells confidential by executing them inside SGX enclaves (§5.8).

5.1 Improving performance and locality with defragmentation

This experiment explores the benefits of using GRANNY to improve application performance and locality while maintaining a target cluster utilization. We use a compaction policy in the Planner that migrates C-Cells of an MPI application at runtime to reduce fragmentation (§4.7.1). Each application job executes the LAMMPS [12] molecule dynamics simulator, running the Lennard-Jones (LJ) benchmark with a varying number of MPI processes. Jobs are executed in order, and may wait in a queue until sufficient resources become available in the cluster.

For our baselines, we use the Azure Batch (`batch`) and Slurm (`slurm`) schedulers, as described in §3.2. Given that the performance benefits stem from more effective co-location, we make both baselines use Granny's MPI implementation for a fair comparison.

Fig. 7 shows various performance metrics, as we increase the number of VMs and the number of

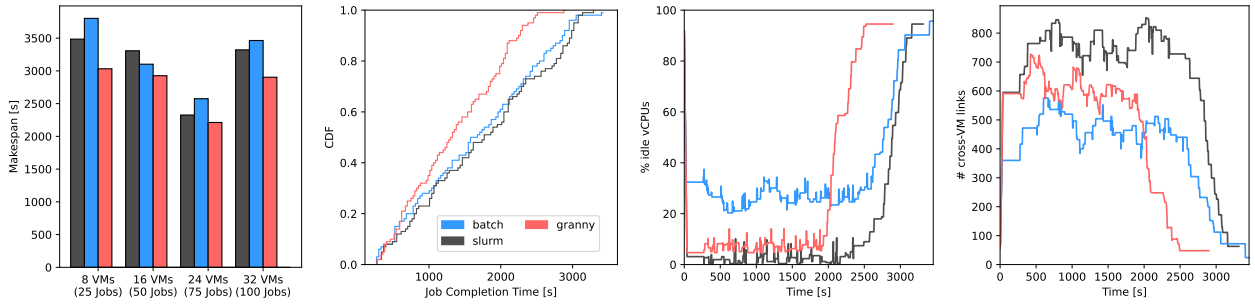

Figure 7: Defragmenting message passing applications using migration. We compare GRANNY with Azure Batch and Slurm. We report the total end-to-end execution time (left-most), and the CDF of job-completion time (left). For the (32 VMs, 100 Jobs) execution we also report the time-series of idle vCPUs (right), and number of cross-VM network links (right-most).

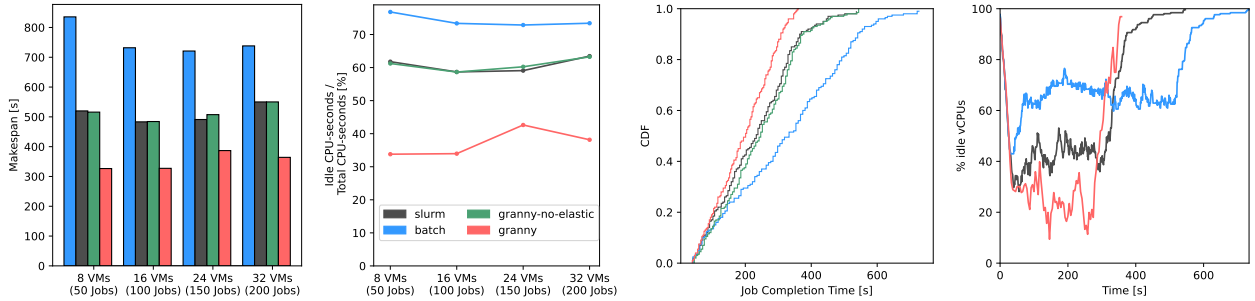

Figure 8: Elastically scaling-up multi-threaded applications to use extra CPU cores. We compare GRANNY with Azure Batch, Slurm, and GRANNY without elastic scaling. We report the makespan (leftmost), and an aggregate metric for cluster utilization (left). For the (32 VMs, 200 Jobs) experiment we also report the CDF of JCT (right), and the time-series of idle CPU cores (rightmost).

jobs. Fig. 7-leftmost shows that GRANNY improves end-to-end execution time (makespan) by up to 20%. Using compaction, GRANNY always improves makespan across all baslines and cluster sizes.

To show the compaction policy in action, Fig. 7-right and Fig. 7-rightmost plot the time-series of idle vCPUs (as a proxy for cluster utilization) and cross-VM network links (as a proxy for locality) for the (32 VMs, 100 Jobs) execution. We see that, differently to Slurm, GRANNY deliberately leaves a percentage of vCPUs idle corresponding to a target utilization (5% in this experiment). GRANNY can use these spare vCPUs to defragment applications at runtime, achieving consistently 25% less fragmentation than Slurm. In fact, GRANNY is closer in terms of fragmentation to Azure Batch, which behaves optimally with respect to this metric, with only 5% idle vCPUs, whereas Azure Batch leaves 30% of vCPUs unused.

Fig. 7-left shows that this reduction in fragmentation at high cluster utilization has a direct impact on job completion time (JCT). For the (32 VMs, 100 Jobs) execution, GRANNY improves median JCT and tail JCT by up to 20%. We conclude that GRANNY with compaction can navigate the utilization-locality trade-off in a more fine-grained way compared to Azure Batch and Slurm.

5.2 Elastically scaling CPU cores

This experiment explores the benefits of using GRANNY to improve per-job performance and cluster utilization by elastically scaling-up the parallelism of shared memory jobs using OpenMP (§4.7.2). Each job is a multi-threaded application that runs a large-scale version of the `p2p` ParRes OpenMP kernel [51], which performs a compute-intensive pipelined parallel algorithm on a large matrix. In this experiment, the native baselines (`batch` and `slurm`) execute jobs using LLVM's OpenMP runtime [18].

Fig. 8 shows a variety of performance metrics, as we scale both the number of VMs in the cluster

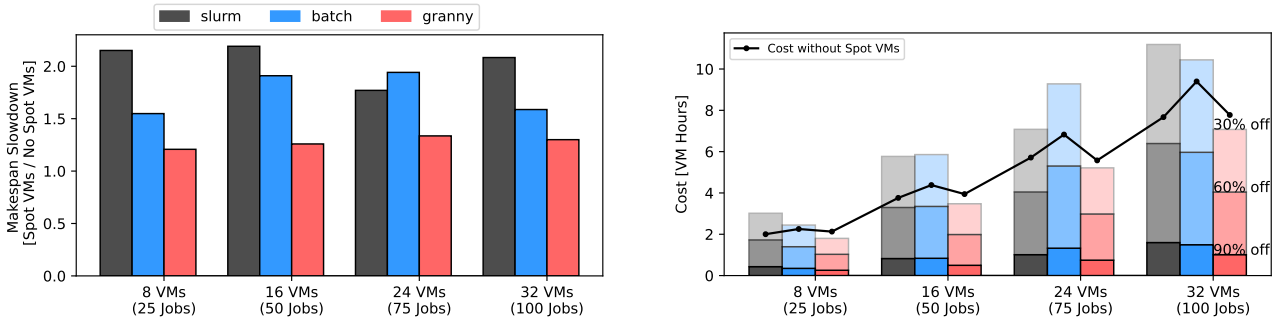

Figure 9: Execution of applications on spot VMs. We report the makespan slowdown (left) and the monetary cost assuming a unit cost per VM-hour (right). With GRANNY, it is always more cost-effective to use spot VMs.

and the number of submitted jobs. Fig. 8-leftmost shows that GRANNY improves makespan by up to 60% compared to the native baselines and GRANNY without elastic scaling (`granny-no-elastic`). This performance improvement stems from the fact that GRANNY can reduce the idle CPU cores in the cluster by up to 30% (Fig. 8-left), and use these extra cores to improve median JCT and tail JCT by up to 50% (Fig. 8-right).

This large performance gap can be understood when considering how many computing resources are left idle by the native baselines. Fig. 8-rightmost shows a time-series of the percentage of idle vCPUs when running 200 jobs on a 32-VM cluster. Azure Batch and Slurm, even when the job queue is not empty, consistently leave 60% and 40% of vCPUs idle. This is due to multiple reasons: (i) OpenMP jobs have fixed parallelism; (ii) it is not possible to distribute jobs across different VMs; and (iii) in the case of Azure Batch, it is not possible to run jobs from multiple users on the same VM. By harvesting these extra resources, GRANNY maintains the fraction of idle vCPUs at around 20%, while there are still pending jobs in the queue.

5.3 Faul-tolerant execution on spot VMs

This experiment explores the performance benefits of using GRANNY to execute compute-intensive applications on a cluster with ephemeral spot VMs. The submitted jobs and the native baselines are the same as in §5.1. In this experiment the native baselines execute jobs using OpenMPI [17]. To emulate the behaviour of spot VMs, while making our findings reproducible, we withdraw VMs at a pre-defined rate with a 1 min grace period.

The eviction rate for this experiment is 25% of the VMs, selected at random, each minute. We choose this eviction rate after measuring the eviction rate for `Standard_D8_v5` spot VMs using Azure's Resource Graph Analyzer [25] (25% per hour) and scaling it to our MPI jobs' length (minutes, instead of hours, to make the experiments reproducible).

Fig. 9-left shows the reduction in makespan when comparing each baseline to itself without evictions. We see that, across cluster and batch sizes, all OpenMPI baselines experience a minimum of a 50% slowdown, and a maximum of a 2× slowdown. This is because the native baselines must restart jobs each time they fail due to an evicted VM. Instead, GRANNY uses the eviction-aware policy described in §4.7.3. As a consequence, its slowdown is at most 25%, which is consistent with the eviction rate.

The native slowdowns of 50%–100% can potentially thwart the cost benefits of using spot VMs. Indeed, Fig. 9-right shows the normalized cost of each execution for the range of discounts based on Azure's spot VM price list [52]: 30%, 60%, and 90%. We calculate the cost by assuming a unit price per VM-hour and applying the corresponding discount. We also overlay the cost of not using spot VMs.

We observe that, for the native baselines, the effectiveness of spot VMs in terms of cost savings depends on the discount rate at which they are offered. Counterintuitively, for many discount ranges,

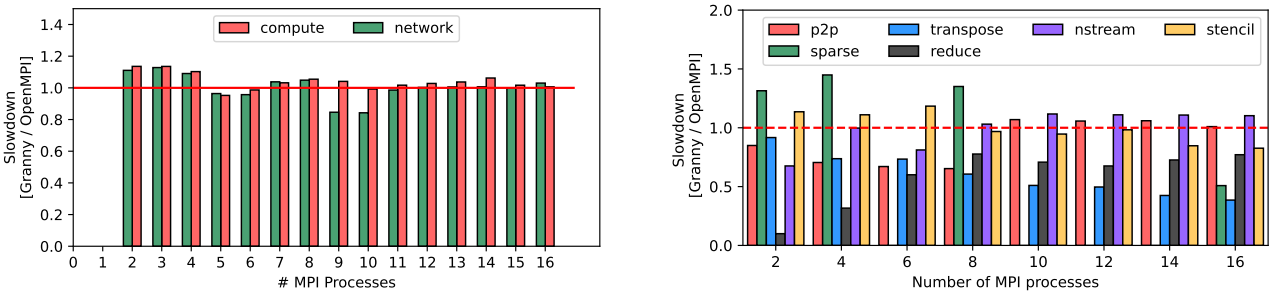

Figure 10: Performance executing MPI applications. As workloads we use the LAMMPS simulator (left) and a subset of the ParRes Kernels (right).

it is not cost-effective to run compute-intensive applications using OpenMPI on spot VMs, because the re-execution costs outweigh the price discount. In contrast, with GRANNY, it is always cost-effective to use spot VMs, because the eviction slowdown is lower than the smallest cost discount.

5.4 Message passing performance

This experiment investigates GRANNY's performance when executing message passing applications. We run the same MPI application as in §5.1. To stress GRANNY's communication layer, we update the benchmark and increase the synchronisation steps, resulting in three orders of magnitude more cross-VM messages. We refer to the vanilla LJ benchmark as `compute`, and the modified one as `network`. We also execute a subset of the ParRes kernels [51] to evaluate specific parts of GRANNY's MPI implementation.

Fig. 10-left shows the slowdown in execution time of GRANNY compared to OpenMPI when executing the two LAMMPS simulations with different levels of parallelism across two VMs. We observe that the overhead introduced by GRANNY is within 10% and often negligible. GRANNY occasionally introduces minor performance gains due to the benefits of intra-process co-location of C-Cells.

Fig. 10-right shows the slowdown when executing the ParRes kernels with different levels of parallelism. For this workload, the performance of GRANNY and OpenMPI varies more than for LAMMPS, because these parallel kernels execute particular MPI operations in a tight-loop. As a consequence, the performance benefit of intra-process co-location of C-Cells becomes much more significant, leading to a substantial performance benefit for GRANNY. In these cases, GRANNY can replace the sending of messages (`reduce`) with reducing shared memory variables.

5.5 Shared memory performance

Next, we measure GRANNY's performance when executing OpenMP jobs. We execute the ParRes kernels [51] in their OpenMP implementation. We execute each kernel with a varying number of threads, and take the average execution time over 5 runs.

Fig. 11 shows the slowdown in execution time of GRANNY compared to LLVM's OpenMP implementation for a variety of kernels. We observe that, for most kernels, GRANNY performance matches the native baseline. For the `dgemm` kernel, GRANNY introduces an 80% slowdown. This kernel performs a dense matrix multiplication, and the overheads come from WebAssembly's less efficient floating-point operations [53].

5.6 C-Cell migration

This experiment explores GRANNY's performance overhead when migrating C-Cells at runtime, and the potential benefits of improved co-location. As workloads, we use the compute-bound LAMMPS simulation, and a network-bound all-to-all kernel, which performs synchronisation over a vector in a loop. For each experiment, we force the Planner to fragment the 8 MPI processes across two VMs, and migrate half of them to the other VM at 20%, 40%, 60%, or 80% of execution time.

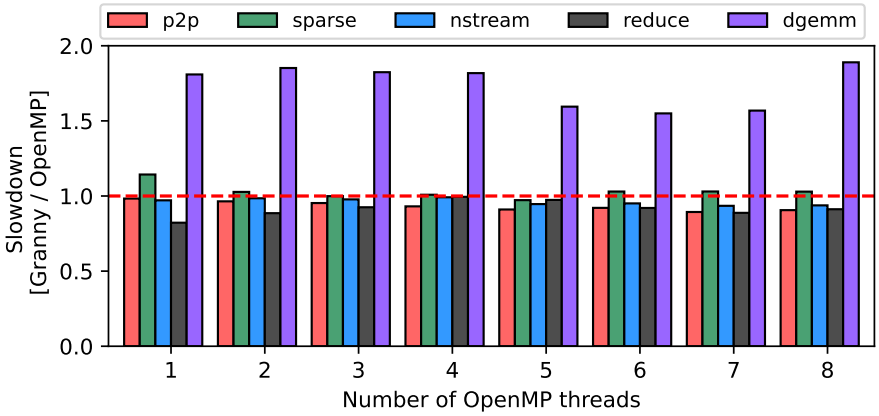

Figure 11: Performance executing OpenMP ParRes Kernels.

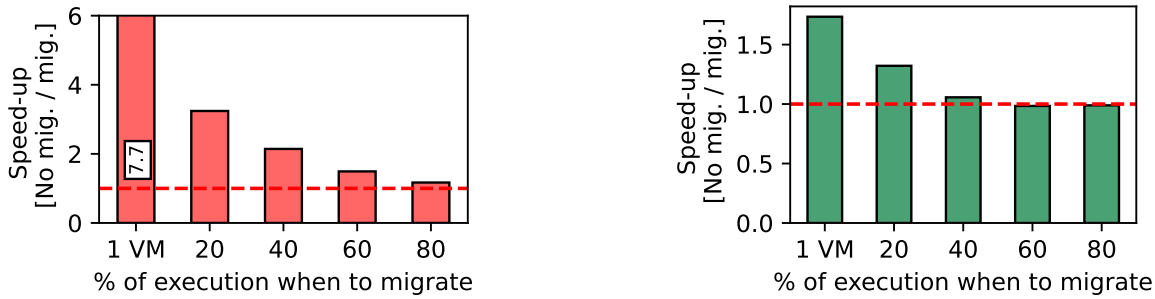

Figure 12: Speedup when migrating C-Cells. We deploy 8 MPI processes across two VMs and migrate 4 at runtime. We report the speedup compared to not migrating. As workloads we use a network-bound *all-to-all* kernel (left) and a compute-bound LAMMPS simulation (right).

Fig. 12-left shows the speedup when migrating a purely network-bound application compared to not migrating at all. For such an application, fragmentation has a high cost: the speedup for running in one VM (`1 VM`) is 7.7×. By migrating after 20%, 40%, 60%, and 80%, of execution, GRANNY achieves speedups of 3.5×, 2.2×, 1.5×, and 1.1×, respectively. We conclude that it is always worth to migrate C-Cells at runtime for network-bound applications.

Fig. 12-right shows the speedup when migrating a compute-bound application. For such an application, fragmentation is less of an issue: the speedup for running in one VM is 1.7×. By migrating after 20% and 40% of execution, we achieve speedups of 1.3× and 1.1×, respectively. We observe no benefit when migrating later during execution. GRANNY's migration mechanism introduces a negligible overhead, enabling cloud providers to optimize for locality.

Migration time is dominated by the time to transfer the snapshot (or byte-wise diff) from one VM to another. For a 4 MB snapshot, corresponding to the all-to-all kernel, C-Cell migration is on the order of 30 ms. From this, only 3 ms corresponds to creating the snapshot, which is almost an order of magnitude faster than a highly-optimized version of CRIU [43].

5.7 Elastic scale-up

Next, we examine the benefits of elastically scaling up the execution of multi-threaded applications to use extra available CPU cores. As a workload, we deploy the same OpenMP application as in §5.2. We initially use a varying number of OpenMP threads, and scale up to all available CPU cores (8) after 50% of execution. We report the speedup compared to not scaling at all.

Fig. 13 shows that, by scaling-up to use more CPU cores, GRANNY achieves a speedup of up to 60% when scaling from 1 to 6 threads. With more than 7 initial threads, we do not observe a benefit,

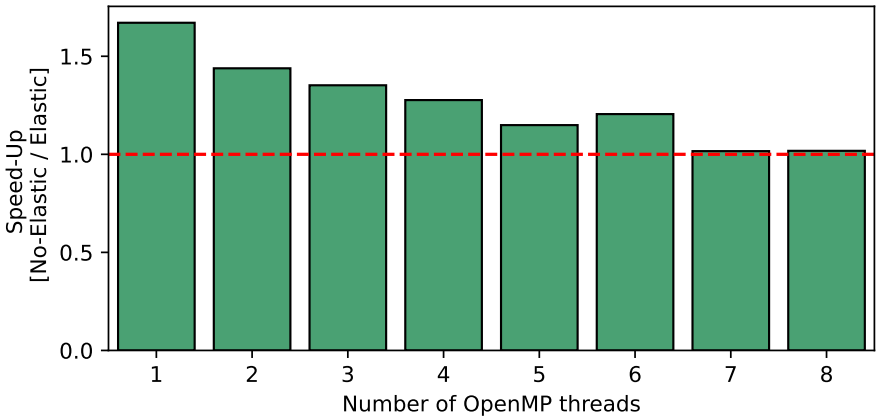

Figure 13: Speedup when elastically scaling to more vCPUs (We deploy a varying number of OpenMP threads, and elastically scale up to all CPU cores after 50% of execution. We report the speedup compared to not scaling up).

as we have exhausted the application's parallelism. We conclude that elastically scaling-up does not slow down execution, because elastic scaling integrates naturally as part of OpenMP's fork-join semantics.

5.8 Initial performance measurement on Confidential C-Cells

Our next experiment is to analyze and provide a base line of running an C-Cells compatible application on top of Intel SGX, specifically after using lift-and-shift approach by SCONE runtime. We performed Polybench benchmark suite on Xeon machine with a E-2186G CPU that has 3.80GHz frequency. We set the SCONE heap for each worker to 8 GB, the worker threads to 2 threads, and left the rest as default.

In Fig. 14, we show various programs that are included in Polybench performance relative to its native execution counterparts. We draw a line at 1.0000 to show that program runs on both execution (Native and Trusted) have exactly the same performance. We also provide different tuning alternatives as comparison. *Tuning 1* involves very high spinning, which improves system call handling at the expense of higher CPU cycles. Meanwhile, *Tuning 2* not only has higher spinning than default although less than *Tuning 1* but also very low system call sleep back-off time. Note that typically a user want to adjust those kinds of tuning profiles based on a specific use case. For completeness, we also include a case where we use a native worker.

Based on the experiment, we found out that having the Polybench benchmark running inside the C-Cells and on top of Intel SGX does not degrade performance. The average overhead by introducing confidentiality with default tuning compared to native execution is roughly 0.05%. Note that in this experiment, all components are trusted, unless the 'Native worker' where the worker is in native environment. Both of our tuning alternative improve the performance further.

One of the nature of the serverless application is its short-duration and burstiness. Although based on previous studies, application can still run while retaining its native performance on top of Intel SGX, we are aware that the typical challenge is the enclave creation. Therefore, we examined our current approach with respect to startup time on multiple environments. In this experiment, we run one simple IO program which only retrieves some string from user, allocates memory, and prints the string back.

In Fig. 15, we show boot duration on the worker for the workload we described above. As expected, doing it with SCONE runtime, regardless whether it is run on top of Intel SGX or not, introduces significant delay. In Fig. 15, *HW* and *SIM* means that it runs with and without Intel SGX, respectively. In SCONE, *SIM mode* refers to an environment where SCONE simulates how Intel create an enclave. In SIM mode, a developer can deduct whether the overhead comes from the hardware

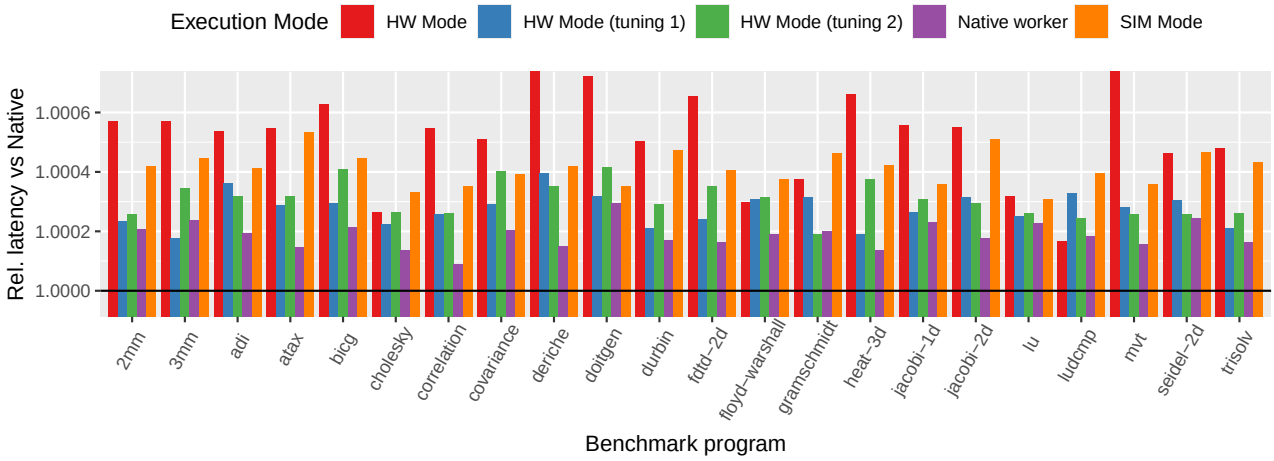

Figure 14: Confidential C-Cells benchmark with polybench.

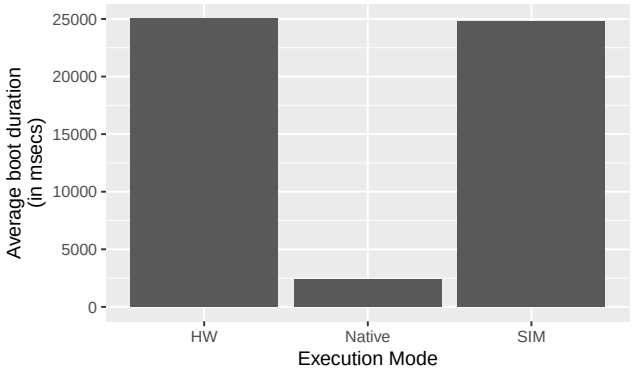

Figure 15: Confidential C-Cells startup time on worker.

or the enclave mechanism. Here, we conclude that since each workload needs to have and spawn its own enclave, each then will have significant startup delay due to memory movement and encryption in an enclave. Here, the startup delay introduced by SCONE could be 10 times more than the native one.

Lastly, we examined again with the same simple workload its duration. Consistent with previous performance experiment, the degradation with respect to performance is minimal. This pictured in Fig. 16. Invoke time here means the time time needed for a single workload invocation to be successful. After the worker booted up, we started to measure the time and recorded it again after the invocation finished, before worker teardown.

Based on those experiments, we draw several conclusions. First, the introduction of confidential approach to C-Cells does not degrade the performance. In all cases, we get less than 1% overhead. This applies to both simple workloads and the Polybench benchmark suite. However, we notice that reducing the startup time could be challenging. Current approach shows significant overhead with 10 times larger delays. Although, as we mentioned earlier in §2, our domain of choice is long-running compute-intensive scientific application. This domain fits well our current limitation and we left the challenge of reducing startup time as future work.

6 Future Work

This deliverable marks the halfway point of the CLOUDSKIN project. For the second half of the project, the main tasks to do in WP4 are the following.

Figure 16: Confidential C-Cells general overhead.

First, extend, and evaluate, the usage of GRANNY to some of the use-cases of the project. In D2.3 we list some of the ways in which we are going to do so. Secondly, we want to address some core limitations of C-Cells, like hardware (ISA) independence, as well as more front-end programming languages support (specially managed runtimes like Python). Third, in terms of confidentiality, we would also like to address the startup time when spawning a confidential worker. There are multiple approaches, namely having a *warm* worker, or make the enclave creation efficient by having some state. Lastly, we also want to evaluate with more (different) applications to further motivate the extensibility and usability of C-Cells.

7 Conclusions

In this deliverable we have presented GRANNY, the fist reference implementation of a system using C-Cells to improve performance and resource utilization of real-world scientific applications.

Scientific applications often rely on shared memory and message passing programming models to harness the parallelism of CPU cores in large clusters. We observe that existing resource management approaches in cloud environments are, however, incompatible with such workloads, as resource managers are unable to balance resource locality and utilization effectively in shared VM pools. Our evaluation shows that by executing multi-threaded OpenMP and multi-process MPI applications using C-Cells, GRANNY can implement a variety of policies to improve job execution time, or idle resource usage.

From the confidential perspective, we have added support to C-Cells in Cloudskin architecture one of the most popular TEE, namely Intel SGX with the support of SCONE. The implementation itself was not trivial, but we managed to integrate all components with Intel SGX enclave. We also evaluate our prototype with Polybench benchmark suite. The result is exceptional as there is very minimal performance degradation.

References

[1] Wikipedia, "x86," 2023.

[2] Wikipedia, "Aarch64," 2023.

[3] Wikipedia, "Graphical processing unit," 2023.

[4] Wikipedia, "Tensor processing unit," 2023.

[5] Wikipedia, "Field-programmable gate array," 2023.

[6] OpenMP, "The OpenMP API specification for parallel programming." https://www.openmp.org/specifications/, 2021.

[7] MPI, "MPI Forum." https://www.mpi-forum.org/, 2022.

[8] B. Li, R. B. Roy, T. Patel, V. Gadepally, K. Gettings, and D. Tiwari, "Ribbon: Cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, Association for Computing Machinery, 2021.

[9] Press Office, "Up to 1.2 billion for weather and climate supercomputer." https://www.metoffice.gov.uk/about-us/press-office/news/corporate/2020/supercomputer-funding-2020, 2022.

[10] OpenFOAM, "Github - OpenFOAM." https://github.com/OpenFOAM/OpenFOAM-dev, 2022.

[11] AWS, "Genomics in the cloud." https://aws.amazon.com/health/genomics/, 2022.

[12] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," Journal of Computational Physics, 1993.

[13] SchedMD, "Slurm Workload Manager." https://slurm.schedmd.com/overview.html, 2024.

[14] Microsoft, "Azure Batch." https://azure.microsoft.com/en-us/services/batch/, 2021.

[15] Amazon Web Services, "AWS Batch." https://aws.amazon.com/batch/, 2021.

[16] Google, "Google Cloud Batch." https://cloud.google.com/batch, 2023.

[17] OpenMPI, "OpenMPI: Open Source High Performance Computing." https://www.open-mpi.org/, 2021.

[18] LLVM Project, "LLVM/OpenMP documentation." https://openmp.llvm.org/, 2022.

[19] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in Proceedings of the 6th International Conference on Supercomputing, ICS '92, (New York, NY, USA), p. 323–334, Association for Computing Machinery, 1992.

[20] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13, (New York, NY, USA), p. 351–364, Association for Computing Machinery, 2013.

[21] Amazon Web Services, "AWS Lambda." https://aws.amazon.com/lambda/, 2021.

[22] Microsoft, "Azure Functions." https://docs.microsoft.com/en-us/azure/azure-functions/, 2021.

[23] Google, "Google Cloud Functions." https://cloud.google.com/functions, 2021.

[24] Azure, "Use Spot VM Instances." https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms, 2024.

[25] Azure, "Use Spot VM Instances - Eviction Rate and Pricing History." https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms#pricing-and-eviction-history, 2024.

[26] Sandia National Laboratories, "Github - LAMMPS." https://github.com/lammps/lammps, 2020.

[27] MDAnalysis, "Github - mdanalysis." https://github.com/MDAnalysis/mdanalysis, 2022.

[28] BioPython, "Github - BioPython." https://github.com/biopython/biopython, 2022.

[29] Broad Institute, "Github - gatk." https://github.com/broadinstitute/gatk, 2022.

[30] su2code, "Github - SU2." https://github.com/su2code/SU2, 2022.

[31] OpenCV, "Github - OpenCV." https://github.com/opencv/opencv, 2022.

[32] TensorFlow, "Github - TensorFlow." https://github.com/tensorflow/tensorflow, 2022.

[33] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, (USA), p. 10, USENIX Association, 2010.

[34] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in Proceedings of the ACM Symposium on Cloud Computing, SoCC '18, (New York, NY, USA), p. 263–274, Association for Computing Machinery, 2018.

[35] OpenMPI, "mpirun - Man Page." https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php, 2024.

[36] OpenMP Api Specification, "OMP_NUM_THREADS." https://www.openmp.org/spec-html/5.0/openmpse50.html, 2022.

[37] Volcano, "Cloud native batch scheduling system for compute-intensive workloads." https://volcano.sh/en/, 2024.

[38] kube batch, "A batch scheduler of kubernetes for high performance workload, e.g. AI/ML, BigData, HPC." https://github.com/kubernetes-retired/kube-batch, 2024.

[39] H. E. Robbins, "A stochastic approximation method," Annals of Mathematical Statistics, 2007.

[40] CRIU, "Checkpoint-Restore in Userspace." https://www.criu.org/Main_Page, 2021.

[41] VMWare, "Migrating Virtual Machines." https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vcenterhost.doc/GUID-FE2B516E-7366-4978-B75C-64BF0AC676EB.html, 2024.

[42] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman, "Optimized pre-copy live migration for memory intensive applications," in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, (New York, NY, USA), Association for Computing Machinery, 2011.

[43] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), (Boston, MA), pp. 497–517, USENIX Association, July 2023.

[44] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in hpc environments," in SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12, 2008.

[45] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017.

[46] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Veriwasm: Sfi safety for native-compiled wasm," 01 2021.

[47] WASI, "WebASsembly System Interface." `https://wasi.dev/`, 2024.

[48] LLVM Project, "LLVM OpenMP Runtime Library Interface." `https://openmp.llvm.org/doxygen/index.html`, 2024.

[49] FFmpeg Contributors, "Webassembly lld port." `https://lld.llvm.org/WebAssembly.htmlimports`, 2022.

[50] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, (New York, NY, USA), p. 266–281, Association for Computing Machinery, 2023.

[51] ParResKernels Team, "Parallel Research Kernels." `https://github.com/ParRes/Kernels`, 2021.

[52] Azure, "Pricing - Linux Spot VMs." `https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/#pricing`, 2024.

[53] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in USENIX Annual Technical Conference (USENIX ATC), USENIX Association, 2020.