`libgomp` Optimizations for Para-virtualized Scheduling Guided OpenMP Execution

`OMP_DYNAMIC_POLICY=pvsched`

Himadri CHHAYA-SHAILESH[1,2] Jean-Pierre LOZI[2] Julia LAWALL[2]

[1]Whisper, Inria Paris
[2]KrakOS, Grenoble INP

FOSDEM 2026

Agenda

Context

The Triad

1. **Parallelization**
   $\rightarrow$ Using GCC's implementation of OpenMP (`libgomp`).
2. **Scheduling**
   $\rightarrow$ Using Linux's Earliest Eligible Virtual Deadline First Scheduler (`EEVDF`).
3. **Virtualization**
   $\rightarrow$ Using the Quick EMUlator (`QEMU`) and the Kernel-based Virtual Machine (`KVM`).

Background: OpenMP

What is OpenMP? [1]

- An Application Program Interface (API) useful for achieving multi-threaded, shared memory parallelism.
- It provides:
  1. Compiler Directives,
  2. Runtime Library Routines,
  3. Environment Variables.
- It is implemented as a runtime library by the compiler (e.g., `libgomp` in GCC)
- The programmer has the full control over parallelization.

[1] https://hpc-tutorials.llnl.gov/openmp

Parallelism using OpenMP

A simple sequential loop in C

```c
for (i = 0; i < N; i++)
    work();
```

A simple loop in C parallelized using OpenMP

```c
#pragma omp parallel for
for (i = 0; i < N; i++)
    work();
```

In the previous example, OpenMP took care of...

- Determining how many workers to use for parallelization
  i.e. the Degree of Parallelism (DoP).
- Creating the worker threads using the `pthread` library
- Distributing the loop iterations evenly among the workers.
- Launching the workers to execute the loop iterations concurrently.
- Synchronizing the workers at the end of the loop.
- Terminating the workers.

OpenMP 101: Threads

- A fork-join model, where a master thread and several worker threads form a team to execute parallel work.
- The value of DoP is resolved at the beginning of a parallel region.
- OpenMP creates, re-uses, and terminates threads as and when needed.

OpenMP 101: Barriers

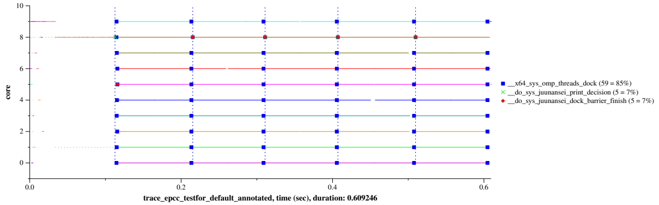
- A barrier marks a point where all worker threads must finish their work before any can proceed.
- Some threads finish earlier than others, so early arrivals at the barrier must wait for the stragglers.
- Threads can wait at a barrier either by spinning or by blocking.
- Barrier usage in OpenMP:
  - Inter-region barriers (ThreadsDock), which marks the beginning of a new parallel region.
  - Intra-region barriers (TeamBarrier), which synchronizes threads within a parallel region.

Parallelism using OpenMP

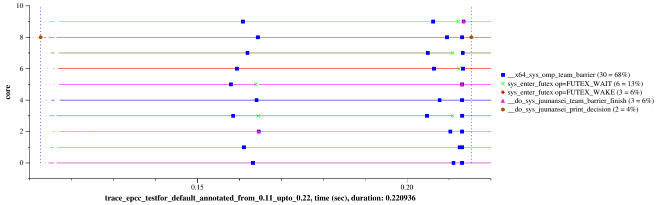
A simplified version of the `testfor()` from EPCC

```c
void testfor(void) {
    int i, j, k;
    /* Five parallel regions */
    for (k = 0; k < 5; k++) {
        #pragma omp parallel private(j)
        {
            /* Two parallel loops per parallel region */
            for (j = 0; j < 2; j++) {
                #pragma omp for
                for (i = 0; i < 1000; i++)
                    delay(250);
            }
        }
        delay(1000);
    }
}
```

OpenMP Threads & Barriers in the Previous Example



Execution of five parallel regions



Zoom on a single parallel region

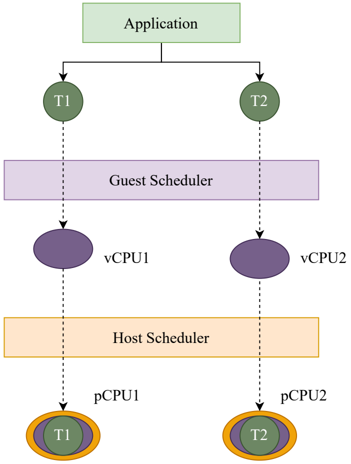
Visualizations produced using schedgraph-tools [FOSDEM'23]

Background: Scheduling

Scheduling 101

- The `task_struct` data structure represents a task using numerous fields.
  e.g. `pid`, `comm`, `state`, `priority`, `policy`, etc.
- A runqueue is a per-CPU data structure that holds all tasks eligible to run on that CPU.
- Stopping a task from running is called preemption.
- Switching between two tasks is called a context switch.

Dual level of task scheduling in the cloud



Life of a vCPU

- A vCPU is created when a VM is launched.
- A vCPU is destroyed when the VM is terminated.
- The host scheduler treats vCPUs like any other regular task.
- **A vCPU can be preempted at any time on the host**

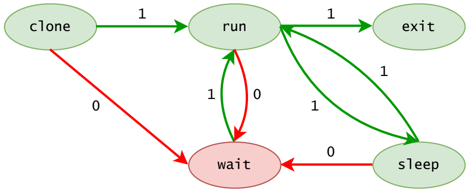


Figure: vCPU state transitions on the host

Thesis Overview

Motivation

- OpenMP execution in cloud VMs is commonplace.
- Oversubscription is a popular cost-cutting practice in cloud deployments.
- OpenMP performance heavily relies on task scheduling and barrier synchronization.
- The relevant default choices in `libgomp` are static and virtualization oblivious.

Thesis

Scheduler Guided OpenMP Execution in Cloud VMs:

- Combine task-scheduling insights from both the schedulers.
- Use these insights to guide OpenMP runtime choices regarding:
  - Degree of Parallelism (DoP) per parallel region.
  - Barrier synchronization mechanism per barrier.
- Improve performance inside oversubscribed Linux VMs.

Contributions

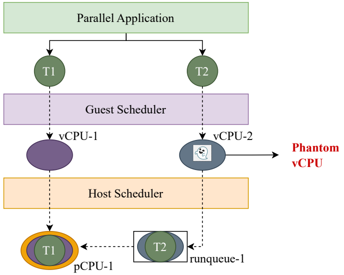
1. **Phantom Tracker**:
   An algorithm that tracks vCPU states during OpenMP execution in VMs.
2. **pv-barrier-sync**:
   A paravirtualized barrier synchronization mechanism.
3. **Juunansei**:
   An extension for `libgomp`, which uses Phantom Tracker and pv-barrier-sync with few added optimizations.

Contribution-1: Phantom Tracker

Phantom vs. Viable vCPUs Under Oversubscription

A vCPU is a phantom if:

1. It is currently waiting in the runqueue of a pCPU on the host.
2. A guest task that was running on this vCPU is now stalled because the vCPU is not executing.



Phantom Tracking

1. Enable communication between the host and guest schedulers.

Phantom Tracking

1. Enable communication between the host and guest schedulers.
2. Register OpenMP threads in the guest Scheduler.
3. Monitor guest scheduler's decisions for the registered threads.

Phantom Tracking

1. Enable communication between the host and guest schedulers.
2. Register OpenMP threads in the guest Scheduler.
3. Monitor guest scheduler's decisions for the registered threads.
4. Register target VM's vCPUs in the host Scheduler.
5. Monitor host scheduler's decisions for the registered vCPUs and keep track of phantom vCPU occurrences.

Phantom Tracking

1. Enable communication between the host and guest schedulers.
2. Register OpenMP threads in the guest Scheduler.
3. Monitor guest scheduler's decisions for the registered threads.
4. Register target VM's vCPUs in the host Scheduler.
5. Monitor host scheduler's decisions for the registered vCPUs and keep track of phantom vCPU occurrences.
6. Compute **`phantom_average`** - a metric that captures the impact of host overload on the OpenMP execution inside the guest at the granularity of a scheduler tick.

Contribution-2: `pv-barrier-sync`

To Spin or Not to Spin

Huang et al.'s finding reveal that for barrier synchronization

- Using spinning can lead to cascading performance loss and a significant waste of CPU cycles under oversubscription.
- Using blocking is inefficient because Linux kernel's task sleep and wake-up process involves multiple queue locking and thread state transitions, making it too expensive under oversubscription.

[1] HPDC'21: "Towards Exploiting CPU Elasticity via Efficient Thread Oversubscription"

To Spin or Not to Spin

Kontothanassis et al. propose using scheduler information for making optimal choices between spinning and blocking at barriers.

- For N threads trying to get through the barrier using P CPUs where $N >= P$:
- The optimal policy would force the first $N - P$ threads to block, so that the remaining P can finish their work.

[1] PPOPP '93: "Using scheduler information to achieve optimal barrier synchronization performance"

Phantom-Guided Spinning with `pv-barrier-sync`

1. Extends Kontothanassis et al.'s proposal to cloud VMs using paravirtualized scheduling insights generated by Phantom Tracker.
2. Respects Huang et al.'s findings:
   - Maximize spinning as long as there are no phantoms.
   - Minimize blocking and use it as a last resort if phantoms are detected.
3. Decided at each barrier on a per-thread basis:
   "Should I block at this barrier?"
4. Upon detecting phantoms,
   blocks as many threads as there are phantoms.

libgomp implementation

Algorithm pv-barrier-sync

```
1: procedure DO_SPIN
2:     loop
3:         if $phantoms\_detected$ then
4:             if $atomic\_sub(to\_block, 1) > 0$ then
5:                 $block()$
6:             end if
7:             if $phantoms\_increasing$ then
8:                 $should\_i\_block = check\_to\_block\_more()$
9:                 if $should\_i\_block$ then
10:                    $block()$
11:                end if
12:            end if
13:        else if $itr\_count \% onems\_spins == 0$ then
14:            $phantom\_average = read\_ivshmem()$
15:            $update\_pv\_barrier\_sync\_stats(phantom\_average)$
16:        end if
17:    end loop
18: end procedure
```

Contribution-3: Juunansei

A three part solution

1. Phantom Tracker guided DoP adaptations at the beginning parallel regions.
2. `pv-barrier-sync` optimizations for ThreadsDock and TeamBarriers.
3. A lightweight task-affinity mechanism to guide the guest scheduler.

[1] In addition to `phantom_average`, Juunansei extends Phantom Tracker to produce per-tick `idle_average` as well.

Phantom-guided DoP Adaptation

The DoP resolution is handled by the master thread upon entering the `gomp_dynamic_max_threads()` function:

- `phantom_average` > 0
  $\rightarrow$ scale down DoP
- `phantom_average` $= 0$ and `idle_average` > 0
  $\rightarrow$ scale up DoP
- `phantom_average` $= 0$ and `idle_average` $= 0$
  $\rightarrow$ maintain current DoP

Stability requirement

In order to be conservative while scaling up the DoP:
Juunansei only consumes the minimum of reported `idle_average` values over the last `stability_requirement` ticks.

- Default value: `stability_requirement` = 2 ticks
- If phantoms are detected after scaling up the DoP,
  → `stability_requirement *= 2;`

`juunansei_cpuset`

- At the beginning of each new parallel region:
  If $DoP = N$
  $\rightarrow$ the last thread to reach ThreadsDock applies `juunansei_cpuset` $=$ [0, N-1] on the entire team.

`juunansei_cpuset`

- At the beginning of each new parallel region:
  If $DoP = N$
  $\rightarrow$ the last thread to reach ThreadsDock applies `juunansei_cpuset` = [0, N-1] on the entire team.
- In case of phantom detection with `pv-barrier-sync`:
  If P phantoms are detected at a barrier
  $\rightarrow$ one of the spinning threads adjusts
  `juunansei_cpuset` = [0, N-P-1] and updates the entire team.
  If more phantoms are detected during the same region
  $\rightarrow$ no further updates to `juunansei_cpuset`.

Juunansei in Action

The next figure refused to be compressed into a slide!

Evaluation

Evaluation Goals

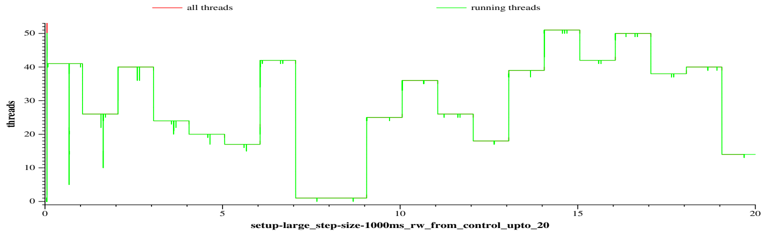
- Quantify the performance improvements in oversubscribed virtualized environments.
- Analyze the behavior under varying levels of resource contention caused by a competing VM.
- Use three different experimental set-ups (Small, Medium, Large) to test the scalability.

Target VM runs NAS Parallel Benchmarks

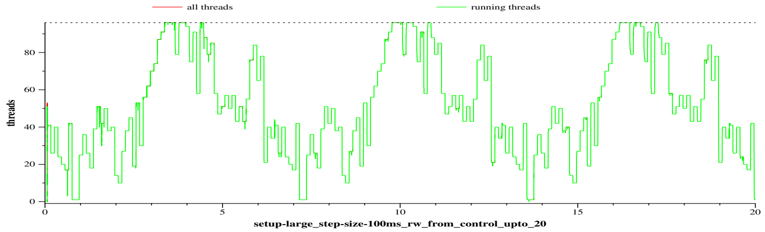
Table: Benchmark characteristics

Application	Input Size	Parallel Regions	Barriers
BT	Class B	1022	2234
CG	Class B	35	10123
FT	Class B	112	224
LU	Class B	516	2810
MG	Class B	1281	3071
SP	Class B	3616	7644
UA	Class B	38769	80503

Competing VM runs a Random Spinners Workload



Step size = 1000 ms (Set-up: Large)



Step size = 100 ms (Set-up: Large)

Visualizations produced using schedgraph-tools [FOSDEM'23]

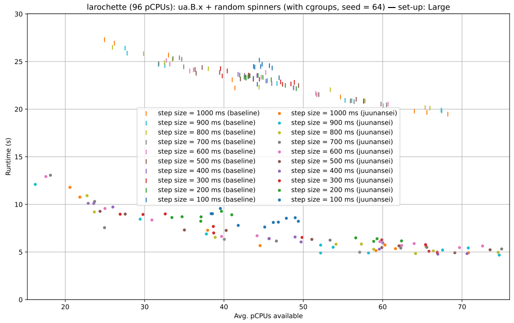
Baseline vs. Juunansei set-up

- **Baseline**:
  - `OMP_NUM_THREADS` = Number of vCPUs of the target VM
  - `OMP_WAIT_POLICY` = passive
  - Host, Guest run vanilla kernels
  - NAS benchmarks load default `libgomp` from GCC branch-releases/gcc-14, commit-569f826774a

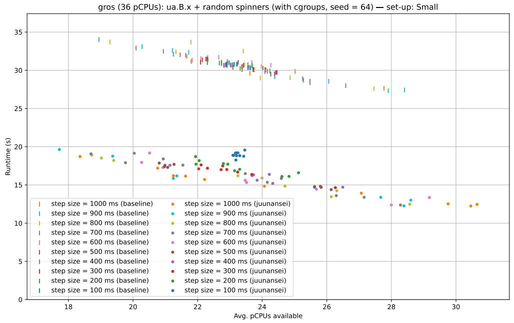
Baseline vs. Juunansei set-up

- **Baseline**:
  - `OMP_NUM_THREADS` = Number of vCPUs of the target VM
  - `OMP_WAIT_POLICY` = passive
  - Host, Guest run vanilla kernels
  - NAS benchmarks load default `libgomp` from GCC branch-releases/gcc-14, commit-569f826774a
- **Juunansei**:
  - `OMP_DYNAMIC` = true
  - Host, Guest run Linux kernels with Phantom Tracker patches
  - NAS benchmarks load a custom `libgomp` build with Juunansei patches

Benchmark: ua.B.x (38769 Parallel Regions & 80503 Barriers)

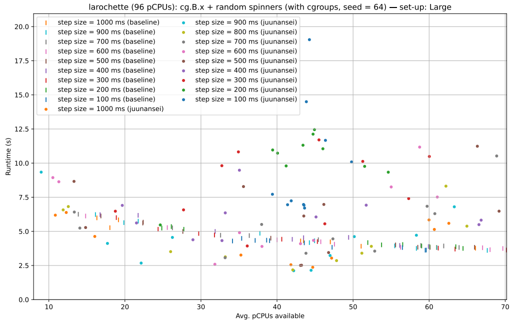


Set-up: Large (96 vCPUs)

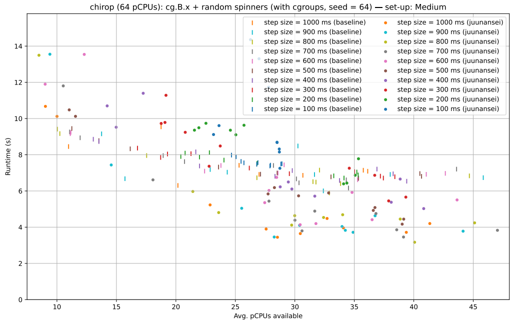


Set-up: Small (36 vCPUs)

Benchmark: cg.B.x (35 Parallel Regions & 10123 Barriers)



Set-up: Large (96 vCPUs)



Set-up: Medium (64 vCPUs)

Next things

Implementation Proposal for `libgomp`

- Extend the current `OMP_DYNAMIC` mechanism.
- Introduce a new icv: `OMP_DYNAMIC_POLICY`.
- Usage:

```
OMP_DYNAMIC=true OMP_DYNAMIC_POLICY=pvsched ./your-parallel-app
```

I can use extra sets of eyes and more hands!

Seeking...

1. feedback on the implementation proposal.
2. code reviewers for the eventual PR(s).
3. funding for hiring an intern via Outreachy / Summer of Code.
4. collaborators and a co-mentor for the internship.

Contact: himadrics@pm.me