To Pin or Not to Pin: Asserting the Scalability of QEMU Parallel Implementation

M. Badaroux, Saverio Miroddi, Frédéric Pétrot

To Pin or Not to Pin: Asserting the Scalability of QEMU Parallel Implementation

Marie Badaroux
Univ. Grenoble Alpes, CNRS,
Grenoble INP†, TIMA
Grenoble, France
marie.badaroux@univ-grenoble-alpes.fr

Saverio Miroddi
Open Source and Research Program
Ticketsolve
Berlin, Germany
smiroddi@ticketsolve.com

Frédéric Pétrot
Univ. Grenoble Alpes, CNRS,
Grenoble INP†, TIMA
Grenoble, France
frederic.petrot@univ-grenoble-alpes.fr

***Abstract*—Due to its speed in cross-executing sequential code, dynamic binary translation is the unchallenged technology for full system-level simulation. Among the translators, QEMU has become the de facto solution. It introduced parallel host execution of the target cores a few years ago for the ARM instruction set architecture and this support is now also available, among others, for RISC-V. Given the popularity of these instruction sets in multi and many-core systems, assessing the scalability of their parallel implementation makes sense. In this paper, we use a subset of the PARSEC benchmark to measure the execution time of QEMU's parallel implementation, to which we added the ability to pin a target processor to a host core or hardware thread. We report the results of a wealth of experiments we performed on a 16-core/32-thread x86-64 SMP machine. They show that the support of parallelism in QEMU scales well, and that, somewhat counter intuitively, pinning does not improve performance.**

I. Introduction

The momentum around the RISC-V [1] has led many academic and industrial groups into designing systems based around this instruction set architecture. Among them, quite a few are designing and implementing multi and manycore platforms (e.g. [2, 3, 4]), some being actually industrialized[1].

Instruction set simulation is the primary tool for validating instruction set architecture (ISA) choices, retargeted compilers, operating systems (OS) ports and pre-silicon validation of hardware/software systems. It also plays a major role in guiding system-level design space exploration. Booting an OS on a platform and running significant applications requires a full system simulator that is both fast and faithful. Classical instruction per instruction interpretation is too slow, while native simulation [5, 6, 7] or static binary translation are ruled out by the necessity of handling self modifying code as used when, e.g., loading dynamic libraries (let alone run jited code like java or .NET bytecodes) [8]. This leaves us with cross-ISA dynamic binary translation (DBT) as solution.

QEMU [9] might not be the fastest DBT engine today [10], but it has unparalleled stability, support for many targets and hosts CPUs, is open-sourced, and has a large and very active community supported by the industry. It has become the reference for many users and developers, which makes it the de facto standard for cross-ISA emulation. In particular, the support for the RISC-V ISA was quickly added, and so did the support for parallel execution of multicore RISC-V target systems on multicore hosts. QEMU vanilla does not provide the ability to pin emulated processors on harts, a feature we added so as to evaluate the benefit it could provide.

In this paper, we report the work we did to experimentally evaluate the scalability of the QEMU dynamic binary translator emulating the RISC-V and ARM architectures on a x86-64 host with 16-core/32-hart[2], with and without pinning. To that aim, we use the subset of the PARSEC benchmark [11] that can be cross-compiled without too much pain and that is stable enough to work consistently on the emulator.

The paper is organized as follows. We first give in Section II some insights on how parallel emulation is handled in QEMU and present works related to ours. Section III presents our implementation of pinning in the emulator, and how we map the emulated CPUs on the host architecture. Section IV details the strategy we have been using to conduct our simulations. Section V relates the experiments we made and provides an analysis of the results, and we finally wrap-up in Section VI.

II. Background and related work

When it comes to running parallel workloads on a multi or manycore platform, having a simulator able to take benefit from the parallel nature of the host is a must. Even parallelized, many simulators, usually in their quest for evaluating performance accurately, incur too many synchronizations to be scalable. This means that processor simulation should progress until an actual synchronization is needed by the application, to ensure maximum lookahead while guaranteeing correctness.

Work on QEMU parallelization was initiated by [12] and [13]. The approach they propose is conceptually intuitive: it consists of emulating the target CPU's (also called vCPU) atomic instructions using the host CPU's atomic instructions. Said so, this seems relatively straightforward, however slight semantic differences between the target and the host memory models lead to generating complex code snippets that are also difficult to ensure correct [14]. But this is only the tip of the iceberg, and many complicated technical details have to be solved, among which parallel target code generation and caching, choice of the next emulated processor to execute, interprocessor communication, etc. These efforts where not upstreamed in QEMU, and rapidly became unusable given the pace at which its code evolves. The idea was taken over by [15] and [16] that ended up in a solution accepted by the community. It was given the name *mttcg*, for multi-thread tiny code generator. The primary host was x86-64, and the first target was,

†Institute of Engineering Univ. Grenoble Alpes
[1]https://github.com/riscv/riscv-cores-list#socs lists current silicon products.
[2]Hardware threads.

without surprise either, the ARM ISA. Since then, the approach has been ported to a handful of hosts and targets, enhanced and generalized [17], and fully integrated into QEMU. A great care has been taken to minimize locking [18], and whenever possible, shared structures are accessed using atomic operations. To avoid useless re-translations, a single code generation buffer is shared by all vCPUs, however code generation itself takes place in a local buffer to avoid locking the translation cache.

These papers generally focus, and rightly so, on the specific points that they aim at improving, but without providing detailed simulation results. Among those, the most relevant one is [17], first because it reflects the current implementation of QEMU, and second because the authors used the PARSEC benchmarks to experiment. They use Aarch64 as target, and report results for user emulation, not for full system emulation, the target of this work.

The RISC-V ISA defines 9 atomic operations and the load-reserved and store-conditional instructions, for 32-bit and 64-bit. In addition, two bits can be used to accommodate release consistency in these 11 instructions. A fence instruction is also available for memory ordering.

Our goal in this work is to evaluate how the emulation of such a complex ISA scales. Additionally, we add support for pinning vCPUs to host CPUs in QEMU, which to the best of our knowledge was never reported in the literature.

III. Pinning in QEMU

Pinning might improve simulation speed, as by assigning a vCPU to a CPU we prevent migrations and thus cache trashing. To devise a pinning strategy, we must study how QEMU handles parallelism and how the host machine provides parallelism:

- QEMU allows to specify the number of vCPUs and a target non-uniform memory access (NUMA) architecture. However, although the NUMA description allows the target operating system to gain insight on the topology, using typically `hwloc` [19], it does not influence performance-wise the execution. QEMU *mttcg* creates as many software threads as vCPUs. QEMU also create threads for emulated I/Os handling peripherals, and for a few other bookkeeping tasks,
- All server hosts today are multicore, NUMA, and generally also support simultaneous multithreading [20] (SMT), two harts being the norm on x86-64 architectures. Modern operating systems have the ability to enable and disable, at run time, the support for harts, leaving only cores to execute on. A program either sees all CPUs, or half of them, and does not know if they are cores or harts. From an identification perspective, it is possible to know which CPU(s) belong to a core, which cores belong to a socket, and which sockets belong to a NUMA node.

As our focus is on processor simulation scalability, we will have very few I/O devices in the system, and these will be seldom active, so we do not dedicate them a processor.

We assign the vCPUs in such a way that they first share a core, then the cores of a socket, and only then we move to the next socket until all sockets of a node are used, and then we move to the next node. When all nodes are exhausted, if there are still vCPUs to assign, we start again at the first hart of the first core of the first socket of the first node. For example, on the machine whose topology is given Figure 1, we would assign the first vCPU

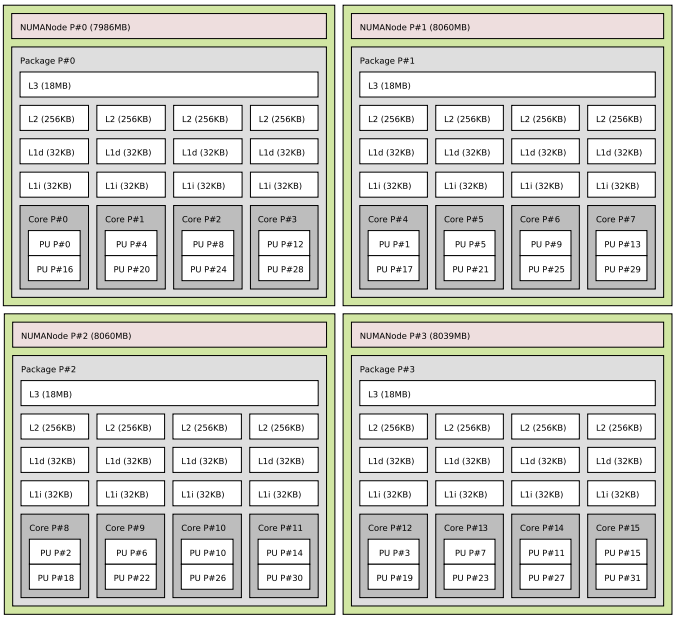

Fig. 1. Mid-end host server: Dell PowerEdge R910 (`lstopo` picture).

on `PU#0`, the second one on `PU#16`, the 9th one on `PU#1`, and so on, back to `PU#0` for the 33rd.

The rationale behind this approach is that the closer the vCPUs on the host, the better the sharing of the resources. Indeed, the current servers predominantly only share their last level cache in a socket, the other levels being private to a core, as illustrated Figure 1 (all caches are 8-way set associative). We also experimented without harts, in the hope that although cache sharing is postponed at socket level, not sharing a core could be beneficial. For the same server, the assignment would be simply sequential modulo 16. The implementation is done classically by setting a single bit in a variable of type `cpu_set_t` to indicate the CPU to use, and passing it to `pthread_setaffinity_np` to perform the assignment.

To isolate vCPUs from other threads, including kernel ones, we use the Linux kernel option `isolcpus`, which specifies physical CPUs to be excluded from scheduling. This avoids any thread from executing on the same CPU where a QEMU vCPU thread is running, with the intended outcome of avoiding context switches. While this strategy is intuitive, it makes comparisons with standard setups possible but not exact. With the natural choice of isolating `Cores 1` to `15` (which allows kernel scheduling only on `Core 0`), it is possible to run a number of threads that is a power of 2 only until half of the number of CPUs; after that, only integral multiples of $n_c - 1$ are possible.

IV. Methodology

Our goal is to measure the scalability of the emulator with presumably scalable non-synthetic workloads. To that aim, we use the PARSEC benchmark with the "simlarge" inputs for all programs, using Linux implementation of the POSIX thread library as support to parallelism. We are aware that scalability issues in the PARSEC programs have been pointed out [21]. However, for lack of a better benchmark, we will use it.

The programs are cross-compiled on the host. Some of them are not meant to be cross-compiled, so they fail and we ignore them, some others fail at run-time, due to either initialization or later memory allocation problems, we also skip them. This is why our plots contain a subset of the benchmark, namely `blackscholes`, `bodytrack`, `cholesky`, `ferret`, `fft`, `fluidanimate`, `freqmine`, `lu_cb`, `lu_ncb`, `ocean_cp`, `radix`, `swaptions`, `water_nsquared` and `water_spatial`. Some of these programs have additional restrictions. Five of them cannot be executed if the number of threads is not a power of 2: `fft`, `fluidanimate`, `ocean_cp`, `radix` and `swaptions`. Furthermore, `swaptions` cannot be executed when the number of threads is higher than the `swaptions` parameter value. We also cross-compiled Linux 5.9.6 and OpenSBI 0.8, configured with up to 128 processors, as run-time for the guest.

For each program we measure the wall clock time for the full execution, T_{full}, and the execution of the region-of-interest (ROI), T_{roi}, as defined by the PARSEC benchmark. The sequential nature of the initializations and of some others parts might have a non negligible influence on the actual scalability of the programs, as Amdahl's law teaches us [22].

Figure 2.(a) presents a high level view of the typical structure of the programs included in the PARSEC benchmark: 1) sequential allocation and initialization of data structures, parsing of files, etc, 2) activation of the ROI timer, 3) start the compute threads. They execute more or less independently, as there are generally some (number heavily depends on the program) variables that need to be accessed concurrently, and thus protected by locks, 4) synchronize all threads at a barrier, 5) iterate to point 3 or deactivate ROI timer and perform something similar to points 1, 2, 3 and 4, and, when done, continue and gather results, 6) deactivation of ROI timer, 7) housekeeping and exit.

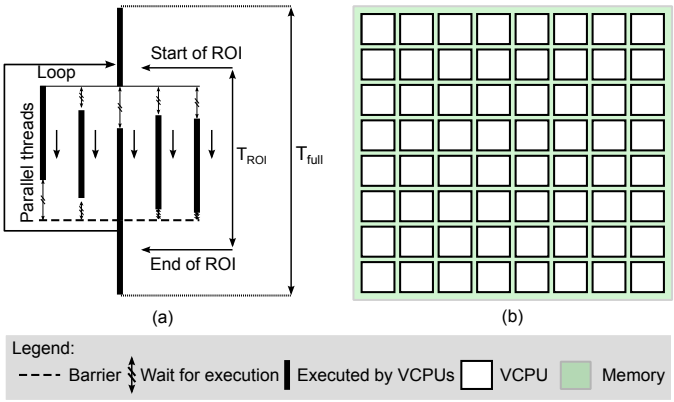

Fig. 2. (a) Parallel programs structure and (b) Target architecture organization

We measure T_{full} and T_{roi} in the programs running on the target using the `clock_gettime` system call using the real-time clock `CLOCK_REALTIME`. Eventually this system call ends up accessing RISC-V status register `instret` which, under QEMU's hood, reads the x86-64 time stamp counter using `rdtsc`. On our host machine the CPUs honor the `constant_tsc` and `nonstop_tsc` flags, which although not perfect make these measurements reproducible and trustworthy.

Figure 2.(b) gives QEMU's view of the target architecture. Each emulated processor can potentially access the whole memory, with synchronization taking place only on instructions performing atomic operations and fences.

The measures are done on a R910/0P658H Dell PowerEdge server. It contains 4 Nehalem-EX 7520 Intel Xeon Processors, each integrating 4 cores with 2 harts, for a total of 16 cores/32 harts, as detailed in [23] (Figure 1). We are sole users of the machine during the experiments. Arguably this machine is not really representative of today's state of the art in terms of servers. However, we are doing fair comparisons since all workloads are run on the same machine, and we believe the overall scalability trends are representative of what could be observed on more up to date servers.

Given what we presented above, we have 6 main parameters:

- number of CPUs of the host machine: 16 or 32,
- number n_c of vCPUs. We run simulations for 1, 2, 4, 8, 16, 24, 32, 48, 64, 96, and 128 vCPUs,
- number n_t of threads that the programs fork during execution: between 1 and 128 included. Several programs need the number of threads to be an integral power of 2, which explains what could be seen as missing values in our plots,
- PARSEC thread affinity. The PARSEC hooks library adds support for thread affinity. When used, the threads are assigned to CPUs (to QEMU vCPUs in our case),
- pinning QEMU vCPUs to physical CPUs. When pinning, we assign the vCPUs the CPUs as detailed in Section III,
- strictly separating the physical CPUs allocation between QEMU vCPU threads and the rest, presented in Section III too.

In addition, we noted that the Linux timer rate heavily impacts the boot time of the vCPUs. As in QEMU the timer is synchronized with the host real-time clock, the slower the boot the higher the number of timer interrupts. So we opted for a value of 100 Hz, classical for server workloads. Furthermore, we force the frequency of the physical cores to be 1862 MHz to get comparable results, and we ensure only the base Linux kernel, an ssh connection and QEMU are running on the machines while doing the experimentations. Finally, we measure 10 executions of each program for a given set of parameters and give as "wall-clock time" the mean.

V. Performance Results

We report in this section the measured execution times. The vertical line in the plots marks the number of host processors. We have chosen to plot wall-clock times rather than speedups for two reasons. First, a practical reason: several programs have a similar scaling behavior, and it would have been difficult to present results that would feature many overlaps. Second, because it gives an order of magnitude of the programs' run-times on QEMU, which is a meaningful information in itself. We have split the presentation in four parts. First, we run natively the 14 programs on our R910 machine. All programs were run with the number of threads $n_t \in \{1, 2, 4, 8, 16, 24, 32, 48, 64, 96, 128\}$. Second, we launch QEMU without vCPUs pinning. We have done lots of variations around this configuration: with $n_c = 128$, with $n_c = n_t$, i.e. there is one vCPU per thread, thus no inactive vCPUs, and with or without SMT activated on the R910 machine. Third, we do the same experiments but with vCPUs pinning and, additionally, with CPUs isolation. Finally, we analyse our results.

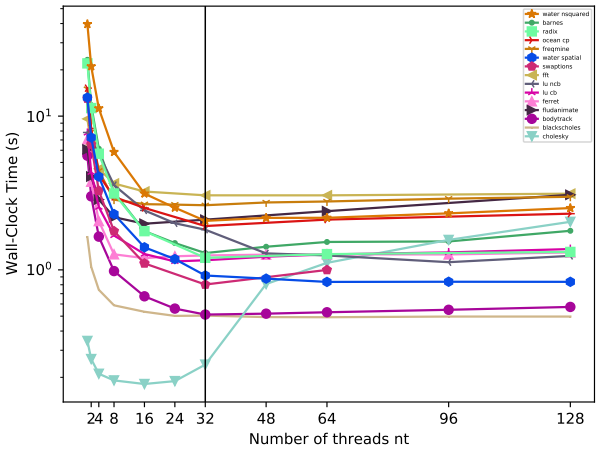

Fig. 3. Full execution time in **x86 without thread affinity**

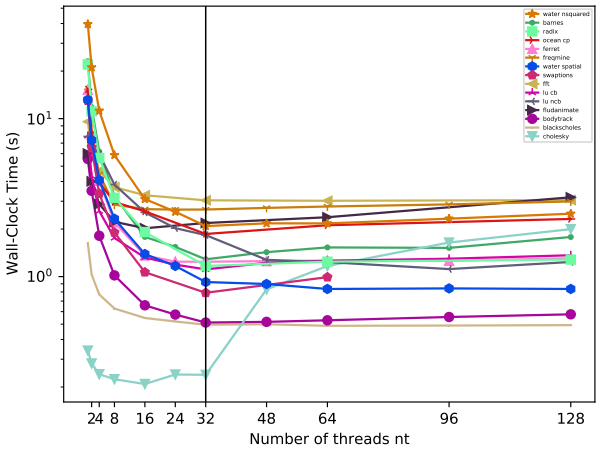

Fig. 4. Full execution time in **x86 with thread affinity**

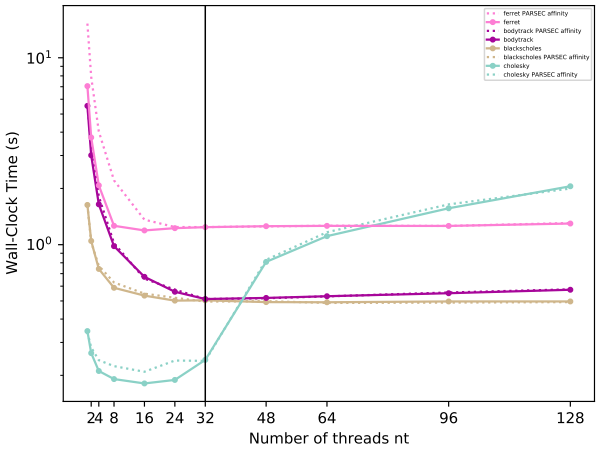

Fig. 5. Comparison full execution time in **x86**

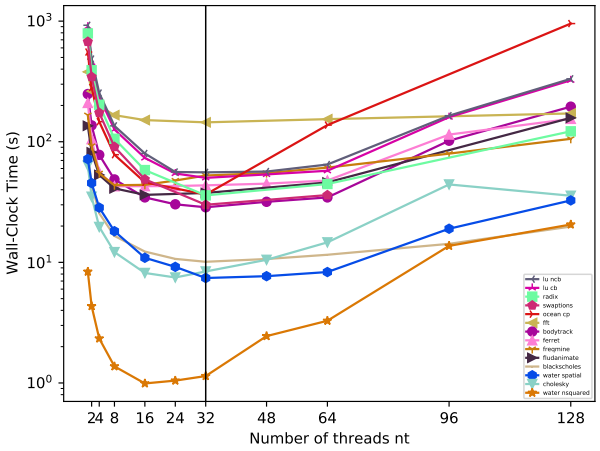

Fig. 6. Full execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning**

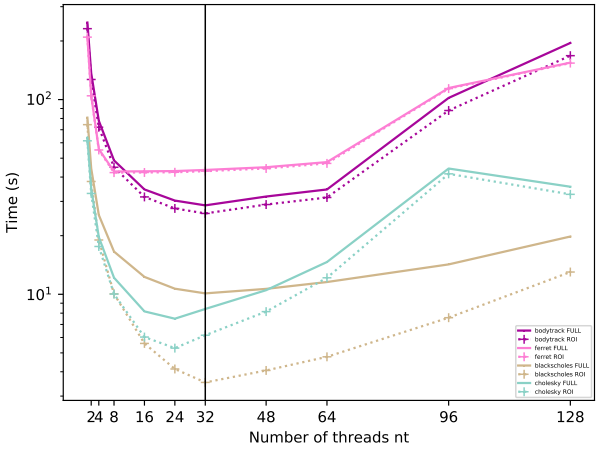

Fig. 7. Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning**

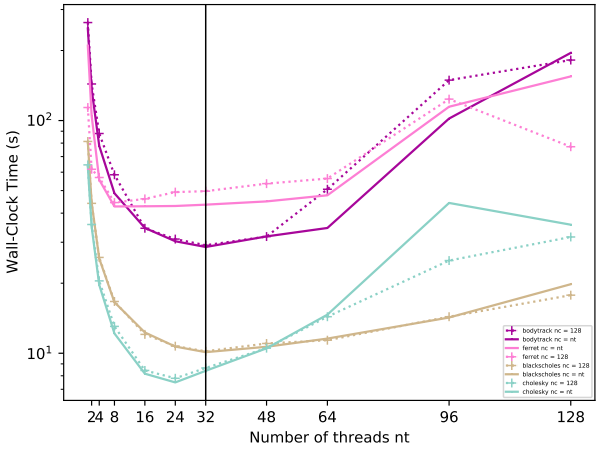

Fig. 8. Comparison full execution time in QEMU **RISC-V without pinning** with $n_c = n_t$ and $n_c = 128$

A. x86-64 native execution

Figures 3 and 4 report T_{full} without and with the PARSEC thread affinity. On both figures, T_{full} decreases as the number of threads increases and then stays almost constant beyond 32 threads. Figure 5 compares the curves with and without PARSEC thread affinity of four programs. The behaviour is the same for our program subset but for the sake of clarity, we represent only four of them. Cholesky is the fastest program and its execution takes less than one second with a number of threads less than 64, so its behaviour is different from the others in the figures. One can see that the thread affinity does not modify the execution time a lot. Some programs are less stable than others but we notice that for the majority of the programs, the execution with and without the PARSEC thread affinity is more or less the same. We will not take this parameter into account for the experiments that follow.

B. QEMU RISC-V without vCPUs pinning

Figure 6 reports T_{full} without vCPUs pinning and with $n_c = n_t$. It is visible that the programs benefit very well from the parallel execution of QEMU. Indeed T_{full} decreases constantly when the number of threads increases up to 32, and increases again as the number of threads goes beyond 32 (the number of host hardware CPUs).

Figure 7 shows the comparison between T_{full} and T_{roi}. Here again, for the sake of clarity, we compare only four programs. According to [21], `blackscholes` exhibits the better scalability for the ROI among the PARSEC, as the previous figure confirms. For the majority of the programs, T_{full} is similar to T_{roi}, which means that the parallel section for our benchmark subset represents an important part of the global execution time.

Figure 8 compares T_{full} with $n_c = n_t$ and $n_c = 128$. The idea behind the experiment is to investigate whether inactive vCPUs in QEMU induce an important overhead or not. For a number of threads less than 96, we observe that for some programs, the execution time with $n_c = 128$ is a little bit higher than with $n_c = n_t$. But overall there is no significant difference in wall-clock time so we can conclude that unused vCPUs in QEMU do not slow down simulation significantly.

Figure 9 compares T_{full} with $n_c = n_t$ without pinning for the host machine with SMT enabled, providing 32 harts, or disabled, providing 16 cores. Figure 9 shows that between 1 and 16 threads, the four programs are faster when SMT is disabled, with a maximal gain of 28% on average at 16. Beyond 16 threads, the two curves SMT enabled/disabled tend to get closer to each other with a slight gain without SMT.

After all, for all configurations of the 14 programs that we used, QEMU for RISC-V scales well.

C. QEMU RISC-V with vCPUs pinning

We now plot the same measures with vCPUs pinning as detailed in Section III. That is to say, assuming vCPUs pinning, Figure 10 reports T_{full} for $n_c = n_t$, Figure 11 shows the comparison between T_{full} and T_{roi}, Figure 12 compares T_{full} with $n_c = n_t$ and $n_c = 128$ and Figure 13 compares T_{full} with $n_c = n_t$ with SMT enabled or disabled. We draw the same conclusions than previously: the programs benefit from the parallel execution of QEMU and QEMU scalability is good.

D. Isolating physical CPUs for QEMU vCPU threads

For this test, we have isolated the physical Cores 1-15 (PU 1-15 and 17-31), leaving Core 0 (PU 0 and 16) to the kernel. The rationale for leaving a whole Core to the kernel is twofold:

- allow some level of multithreading to the kernel processes, in order to avoid (or at least mitigate) bottlenecks,
- prevent the kernel from competing with QEMU vCPU threads for execution units.

Having one Core less available prevents tests that require a number of threads equal to a power of 2 to run: `fft`, `fluidanimate`, `ocean_cp`, `radix`, `swaptions`.

The analysis of the comparison with the non-pinned setup can be divided in three sections:

- from 1 to 15 threads: virtually no difference for all programs except for `lu_ncb` which shows an improvement of 7% for 15 threads compared to non-pinned setup.
- from 60 and upwards: differences in both directions, mostly regressions.
- for 30 threads: a non-negligible improvement for 4 programs compared to non-pinned setup in the range of 4 to 16%, with some regressions.

The comparison between non-pinned, previous pinned and the isolcpus pinned setups for 32 threads is presented Figure 15. For the isolcpus pinned setup, we were only able to run the programs with 30 threads as the Core 0 is dedicated to kernel threads. We see that for `cholesky`, `freqmine`, `lu_cb` and `lu_ncb`, the execution time is smaller when using pinning with isolcpus. For the rest of the programs, there is either no improvement or a performance degradation.

Within the limits exposed above and the experimentation performed, pinning and isolating CPUs for a number of threads not greater than the harts yields a non-negligible improvement for some programs.

E. QEMU performance for a large number of vCPUs

A significant difference found between the x86 and emulated performance is the steep increase in execution time above 32 threads. In order to investigate this phenomenon, we've monitored with the Linux Perf tool [24] the number of cpu cycles spent for a single execution of `lu_ncb`'s ROI section, on a recompiled QEMU with the optimizations disabled. This benchmark was chosen for the high ratio between the execution time with 128 vCPUs versus 16, and because it runs long enough to isolate the ROI section.

The result showed that the vast majority of the time (approximately 70% of the profiled cycles) was spent in the Linux `futex` syscall, used by QEMU's global mutex. The code generator's interrupt handling function requires a lock on this mutex, causing resource starvation on setups with a large number of vCPUs. This also explains why the Linux kernel interrupt frequency was correlated to the boot time.

F. Is Pinning Helpful?

Figure 14 shows the curves of T_{full} for 4 programs without and with vCPUs pinning when $n_c = n_t$. Except for Cholesky which is not stable when n_t is high, the curves without and with pinning are really close. We made the same observation for the rest of our programs. Much to our surprise, the execution times are similar and pinning vCPUs does not seem to have a

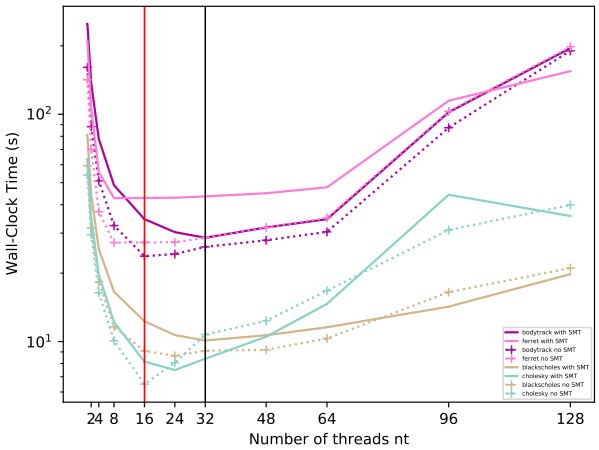

Fig. 9. Comparison of full execution time in QEMU **RISC-V without pinning** with $n_c = n_t$ for the host machine with and without **SMT**

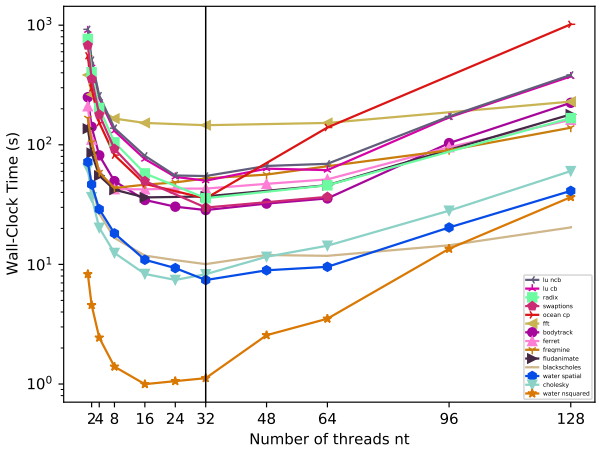

Fig. 10. Full execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning**

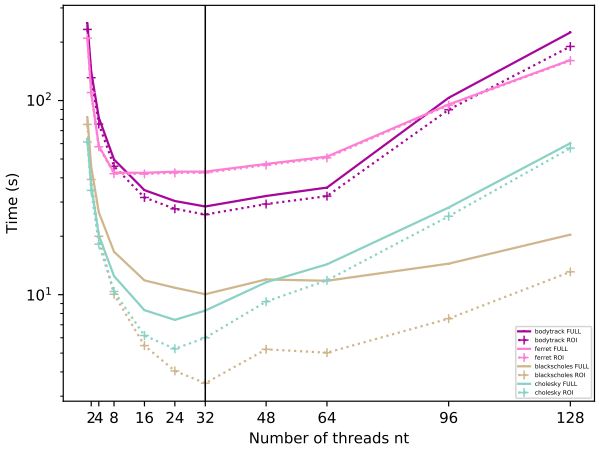

Fig. 11. Comparison of **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning**

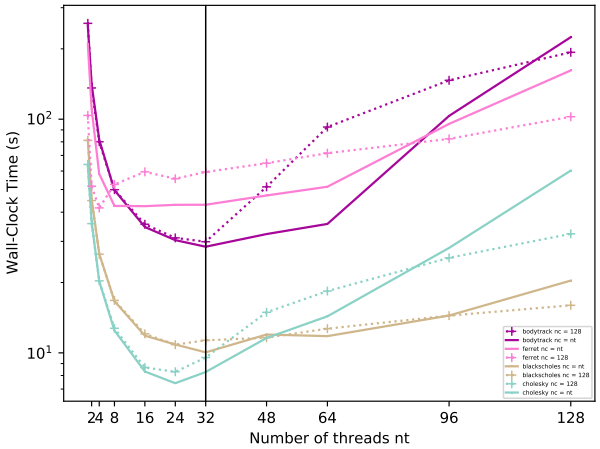

Fig. 12. Comparison of full execution time in QEMU **RISC-V with pinning** with $n_c = n_t$ and $n_c = 128$

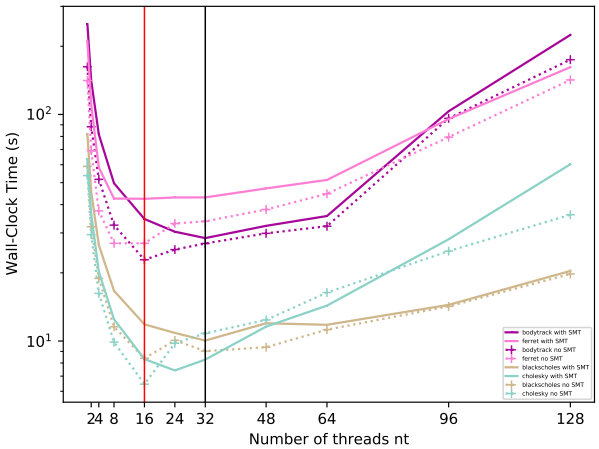

Fig. 13. Comparison of full execution time in QEMU **RISC-V with pinning** with $n_c = n_t$ for the host machine with and without **SMT**

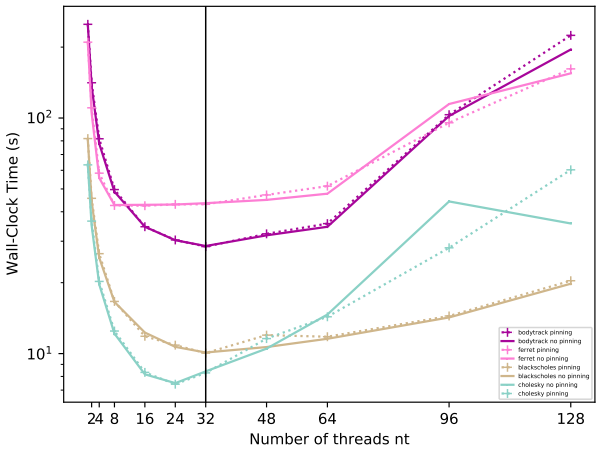

Fig. 14. Comparison of full execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** and **with pinning**

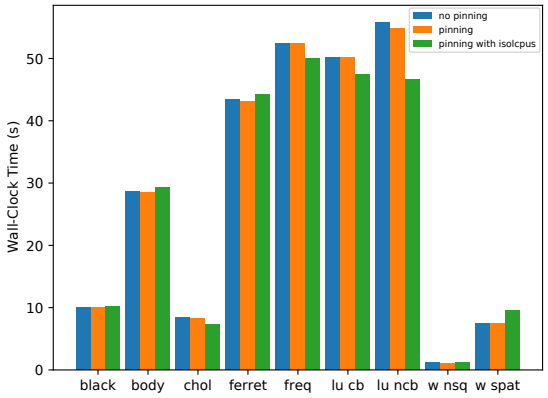

Fig. 15. Comparison of the full execution times for 32 threads without pinning, with pinning and with isolcpus pinning

TABLE I
PERF FOR QEMU RISC-V WITHOUT AND WITH VCPUS PINNING

	CPU migrations		**L1-dcache-load-misses** (10^6)	
n_t	**no pinning**	**pinning**	**no pinning**	**pinning**
1	1 014	1 155	79 469	81 340
2	884	799	82 478	97 332
4	2 706	705	86 350	102 624
8	7 930	84 687	95 995	114 283
16	23 692	91 564	114 800	135 526
24	29 404	96 639	60 700	64 071
32	1 965 662	1 345 667	168 547	164 196
48	8 411 897	260 084	96 999	91 701
64	17 141 497	722 055	258 957	254 168
96	62 427 446	689 736	500 362	465 388
128	324 423 980	2 958 824	2 089 470	2 347 334

positive impact on T_{full} for our programs. We now conduct an analysis to understand why pinning does not improve performance using Linux Perf, which reports statistics based on the processors' performance counters. In addition, to make sure that what we observe are not artefacts due to QEMU RISC-V implementation, we also experimented with the ARM ISA.

1) Perf: As we saw previously, $n_c = n_t$ and $n_c = 128$ have the same behaviour so we focus here on the experiments with $n_c = n_t$. We recorded the result of Perf applied to QEMU as a whole for the 14 programs each run 10 times. Table I shows the evolution of the number of thread migrations and L1 cache misses in QEMU with and without vCPUs pinning. Remember that peripherals and a few other tasks are threads in QEMU, this explains that migrations exist even though the vCPUs are pinned.

In both cases, the number of migrations increases with the number of threads n_t. When n_t is not a power of 2, only 9 programs can be run that is why the values of the metrics can be lower than others. Overall, the number of CPU migrations is smaller when using QEMU RISC-V with vCPUs pinning (more than 100 times smaller with $n_t = 128$) so pinning vCPUs has an impact on the CPU migrations. When we have $n_t \geq 32$, the pinning drastically reduces the number of CPU migrations. The order of magnitude the *L1-dcache-load-misses* metric are similar, which can explain why the execution times remain more or less the same without and with vCPUs pinning.

2) ARM: To see if the behaviour of the programs with and without pinning is only specific to RISC-V or is generalized to other architectures, we experimented with the ARM implementation of QEMU, with $n_c = n_t$ and without thread affinity. QEMU ARM with machine `virt` limits the number of cores so we were able to run it up to only 96 vCPUs. As in RISC-V, each program is run 10 times. Additionally, we were able to run the program `barnes` in ARM unlike in RISC-V. We decided to put it as information in our QEMU ARM graphs.

Figures 16 and 17 report T_{full} on the benchmark subset for QEMU ARM $n_c = n_t$ without and with vCPUs pinning. As with QEMU RISC-V, T_{full} decreases constantly as the number of threads is between 1 and 32 and begins increasing beyond 32 threads.

Figure 18 shows the comparison between QEMU ARM without and with vCPUs pinning. One can see that there is no big difference, the behaviour is the same without and with pinning. We can do the same observation than in RISC-V: pinning vCPUs does not improve the execution time of our program subset. Table II presents the results of Perf.

TABLE II
PERF FOR QEMU ARM WITHOUT AND WITH VCPUS PINNING

	CPU migrations		**L1-dcache-load-misses** (10^6)	
n_t	**no pinning**	**pinning**	**no pinning**	**pinning**
1	253	300	254 493	263 479
2	630	112	251 762	321 809
4	2 131	76	282 065	371 609
8	8 579	726	329 442	417 983
16	84 640	438 471	450 250	565 866
24	341 051	33 784	495 140	508 461
32	26 675 217	5 189 901	1 046 410	975 366
48	354 436 238	3 027 408	1 165 892	913 236
64	744 645 854	7 170 919	1 911 263	1 589 535
96	919 101 918	4 913 936	1 809 297	1 454 464

When comparing the number of CPU migrations for QEMU ARM without vs with vCPUs pinning, we again notice that overall, we have much less migrations with pinning, but as in RISC-V, this doesn't lead to significantly lower *L1-dcache-load-misses*.

3) Analysis: The metrics given by Perf show an improvement when pinning vCPUs. The number of *CPU migrations* is reduced but the number of *L1-dcache-load-misses* is close without and with vCPUs pinning. In the end, the execution times of our program subset without and with pinning remain similar and for some programs it even appears to be a little slower when pinning vCPUs. According to [25], CPU-bound applications, such as the PARSEC ones, do not improve the performance when pinning virtual machines. The current Linux scheduler attaches a thread to a CPU as long as it does not degrade performance as estimated at run time. The CPU affinity is thus based on the actual workloads, and recent works have shown that pinning benefits only certain applications [26].

VI. CONCLUSION

Simulation is often useful during the design, implementation and even use of processor centric systems. The wide adoption of the RISC-V ISA and its use in current and future multi and manycore systems makes the evaluation of QEMU, the most stable and used parallel processor emulator, a lively topic.

In this work, we have simulated shared-memory target systems with up to 128 processors on a 16-core/32-thread host, using Linux SMP and the PARSEC benchmark. As the result of this comprehensive experimental study, we observed that QEMU

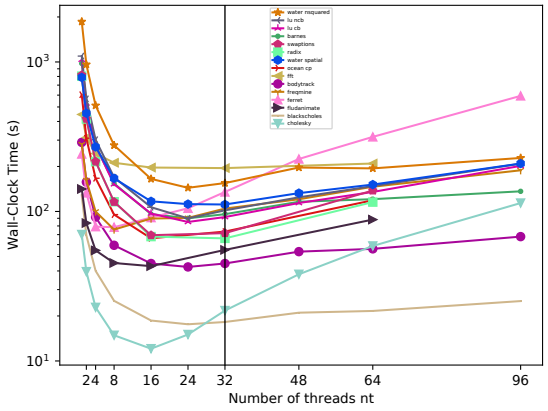

Fig. 16. Full execution time in QEMU **ARM** $n_c = n_t$ **without pinning**

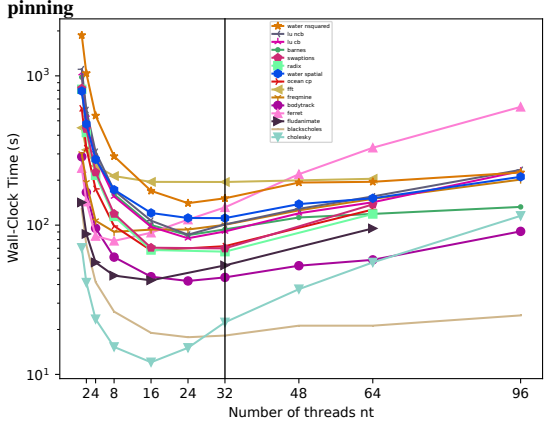

Fig. 17. Full execution time in QEMU **ARM** $n_c = n_t$ **with pinning**

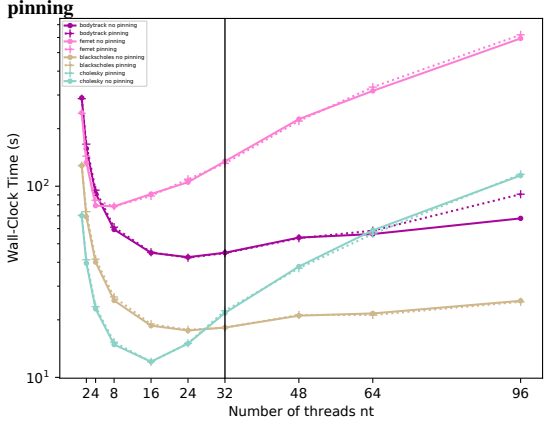

Fig. 18. Comparison of full execution time in QEMU **ARM**

scalability is rather good, which makes it suitable for supporting the software development of, in particular, new RISC-V based platforms. We added in QEMU the ability to pin vCPUs to host processors, and much to our disappointment, forcing the affinity did not improve performances.

Acknowledgment

We would like to thank the partners of the ANR Rakes project and acknowledge the financial support of the French Agence Nationale de la Recherche (ANR-18-CE25-0017, https://anr.fr/Project-ANR-18-CE25-0017).

References

[1] A. Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UC Berkeley, May 2011.

[2] S. Davidson et al. "The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips". In: *IEEE Micro* 38.2 (2018), pp. 30–41.

[3] A. Kurth et al. "HERO: An open-source research platform for HW/SW exploration of heterogeneous manycore systems". In: *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. 2018, pp. 1–6.

[4] E. Flamand et al. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–4.

[5] H. Shen, M.-M. Hamayun, and F. Pétrot. "Native simulation of MPSoC using hardware-assisted virtualization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.7 (2012), pp. 1074–1087.

[6] L. Díaz et al. "VIPPE, parallel simulation and performance analysis of multi-core embedded systems on multi-core platforms". In: *Design of Circuits and Integrated Systems*. 2014, pp. 1–7.

[7] A. Nicolas and P. Sanchez. "Parallel native-simulation for multi-processing embedded systems". In: *2015 Euromicro Conference on Digital System Design*. 2015, pp. 543–546.

[8] F. Pétrot et al. "On MPSoC Software Execution at the Transaction Level". In: *IEEE Design & Test of Computers* 28.3 (2010), pp. 2–11.

[9] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[10] T. Spink, H. Wagstaff, and B. Franke. "A Retargetable System-Level DBT Hypervisor". In: *2019 USENIX Annual Technical Conference*. 2019, pp. 505–520.

[11] C. Bienia et al. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 72–81.

[12] Z. Wang et al. "COREMU: a scalable and portable parallel full-system emulator". In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 2011, pp. 213–222.

[13] J.-H. Ding et al. "PQEMU: A parallel system emulator based on QEMU". In: *1st International QEMU Users' Forum*. 2011, pp. 35–38.

[14] M. Kristien et al. "Fast and correct load-link/store-conditional instruction handling in DBT systems". In: *CASES'20: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 2020.

[15] G. Delbergue et al. "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0". In: *8th European Congress on Embedded Real Time Software and Systems*. 2016.

[16] A. Rigo, A. Spyridakis, and D. Raho. "Atomic instruction translation towards a multi-threaded QEMU". In: *30th European Conference on Modelling and Simulation*. 2016, pp. 1–9.

[17] E. G. Cota et al. "Cross-ISA machine emulation for multicores". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization*. 2017, pp. 210–220.

[18] A. Bennée. *Multi-thread tiny code generator*. https://github.com/qemu/qemu/blob/master/docs/devel/multi-thread-tcg.rst. 2020.

[19] F. Broquedis et al. "hwloc: A generic framework for managing hardware affinities in HPC applications". In: *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE. 2010, pp. 180–186.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism". In: *Proceedings of the 22nd annual international Symposium on Computer Architecture*. 1995, pp. 392–403.

[21] G. Southern and J. Renau. "Deconstructing PARSEC scalability". In: *Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking*. 2015.

[22] G. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: (Apr. 1967), pp. 483–485.

[23] P. Gepner et al. "Evaluating new architectural features of the Intel (r) Xeon (r) 7500 Processor for hpc workloads". In: *Computer Science* 12 (2011), pp. 5–17.

[24] A. C. De Melo. "The new linux "perf" tools". In: *Linux Kongress*. Vol. 18. http://vger.kernel.org/~acme/perf/lk2010-perf-acme.pdf. 2010.

[25] D. Ghatrehsamani et al. "The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms". In: *49th International Conference on Parallel Processing*. 2020, pp. 1–11.

[26] A. Podzimek et al. "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 1–10.