# Parallel Empirical Evaluations: Resilience despite Concurrency

**Johannes K. Fichte**[1], **Tobias Geibinger**[2], **Markus Hecher**[3], **Matthias Schlögel**[2]
[1] AIICS, IDA, Linköping University, Sweden
[2] KBS Group, Institute for Logic and Computation, TU Wien, Austria
[3] Massachusetts Institute of Technology, USA
johannes.fichte@liu.se, {tobias.geibinger,matthias.schloegel}@tuwien.ac.at, hecher@mit.edu

## Abstract

Computational evaluations are crucial in modern problem-solving when we surpass theoretical algorithms or bounds. These experiments frequently take much work, and the sheer amount of needed resources makes it impossible to execute them on a single personal computer or laptop. Cluster schedulers allow for automatizing these tasks and scale to many computers. But, when we evaluate implementations of combinatorial algorithms, we depend on stable runtime results. Common approaches either limit parallelism or suffer from unstable runtime measurements due to interference among jobs on modern hardware. The former is inefficient and not sustainable. The latter results in unreplicable experiments.

In this work, we address this issue and offer an acceptable balance between efficiency, software, hardware complexity, reliability, and replicability. We investigate effects towards replicability stability and illustrate how to efficiently use widely employed cluster resources for parallel evaluations. Furthermore, we present solutions which mitigate issues that emerge from the concurrent execution of benchmark jobs. Our experimental evaluation shows that – despite parallel execution – our approach reduces the runtime instability on the majority of instances to one second.

## Introduction

“Can we please do science again” was a highly provocative catchphrase by Karem Shakallah in a roadmap talk on his perspective for the next stage of research in solving the Boolean satisfiability (SAT) problem (Sakallah 2023). He argued that we have a limited understanding of certain aspects of modern solving techniques, that understanding could be purely driven by empirical competitions catching for slightly improving another technique or implementation, and that models and techniques are too complex for grasp.

Still, not just in the SAT community but also in various combinatorial-solving communities (Bartocci et al. 2019), empirical evaluations are at heart and certainly a tedious part of scientific work (McGeoch 2012). Competitions (Bartocci et al. 2019), obtaining conclusions about practical algorithms (Elffers et al. 2018), verifying proof traces (Heule and Kullmann 2017), or estimating the benefits of some new algorithm or technical improvement (Müller-Hannemann and Schirra 2010) all require in-depth, breadth, and reliable empirical experiments. The sheer amount of required resources often makes it impossible to execute the experiments on a single computer. Instead, we use clusters of computers for experiments, which cannot be used exclusively by single persons or groups due to cost-efficiency demands. In engineering, complexity, but there referring to complicatedness, is often seen as an enemy of reliability (Geer et al. 2003). On computer clusters, the complicatedness of the underlying architecture results in drawbacks and pitfalls during operations for combinatorial solving. Parallel executions, diverging runtimes of instances and solvers, shared resources (network or local), and scheduling efficiency are drawbacks that might result in unreliable execution and hence irreproducible empirical results.

In engineering (Henderson and Patel 2002; Force 1993), standardization is used to control complicatedness and reduce uncontrolled parts’ potential side effects. Unfortunately, to our knowledge, no standardization on configuring hardware and software is available in the combinatorial-solving community. Mistakes are common and far from easy to spot. Reliable empirical experiments on individual systems and performance improvements are well investigated (Georgiou et al. 2014; Beyer, Löwe, and Wendler 2019; Vercellino et al. 2023), and tools focusing on precise measurements, repeatability, optimal throughput, and efficiency of participating systems are available for installation (Beyer, Löwe, and Wendler 2019). However, a primary disadvantage of most dedicated tools is exclusive and very permissive system access, which is often hard to establish or requires dedicated resources for a limited number of people resulting in extremely unsustainable and inefficient usage of computer hardware.

Still, cost efficiency demands from a management or resource availability perspective, and sustainability require that resources are shared among many applications. High-performance computing (HPC) data centers have resources in the form of computation clusters widely available and are often very well maintained (Green500 Authors 2022; Strevell et al. 2019). Fortunately, when applying the right restrictions, widely developed technology from HPC environments can be highly useful for computational experiments in various combinatorial communities. Furthermore, these systems allow us to “maximize” throughput and use of shared resources to increase cost efficiency and sustainability.

**Contributions** Our contributions are as follows:

1. We investigate foundations for *stable, parallel, repeatable empirical evaluations* of computational experiments. In contrast to previous works, we focus on the memory design in modern computer architecture, which is a significant factor for variation and irreproducibility.
2. We provide a novel technique for stable and repeatable experiments. Our main ingredient *cache partitioning* enables us to eliminate issues that arise from modern memory architectures. To our knowledge, cache partitioning as yet not been suggested for repeatable experiments.
3. Our overall approach fits well into standard high-performance computing (HPC) environments, which encourages the use of modern cluster environments that was previously seen highly problematic for replicability.

**Related Works.** McGeoch (2012) provides extended insights into setting up experiments. Müller-Hannemann and Schirra (2010) discuss basic algorithm engineering. Researchers developed techniques to optimize scheduling within computer clusters (Bridi 2018; Galleguillos et al. 2019). The influence of hardware efficiency and measurements is well established (Georgiou et al. 2014; Beyer, Löwe, and Wendler 2019; Vercellino et al. 2023). Unfortunately, effective, specialized resource limitation and benchmarking tools are often impracticable. Highly effective tools such as BenchExec (Beyer, Löwe, and Wendler 2019) must be installed deep into the system, require very permissive access, maintaining of different user groups, and additional finetuning. These tools often require to run an entire benchmarking toolchain making testing and debugging hard. Complications of modern hardware on combinatorial solvers have been studied (Koopmann, Hähnel, and Turhan 2017; Fichte et al. 2021). For algorithm configuration, pitfalls in empirical work have been previously identified (Bocchese et al. 2018) and best practices to avoid them suggested (Eggensperger, Lindauer, and Hutter 2019). Recent initiatives on reproducible research focus on transparent research artifacts with Guix, a system that enables the building of computation environments (Vallet, Michonneau, and Tournier 2022). Software heritage projects aim at preserving source code and binaries from research software (Cosmo and Zacchiroli 2017; Audemard, Paulevé, and Simon 2020). Experiments on SAT solver development over time and hardware influence exist in the literature (Biere et al. 2023; Fichte, Hecher, and Szeider 2023).

## Basics of Empirical Evaluations

Empirical evaluations are crucial to modern combinatorial problem-solving when we reach beyond theoretical algorithms or bounds. In practice, we often start from an algorithm, its implementation, and hypotheses about the behavior of the implementation on certain inputs, called instances, and variations of parameters of the implementation, called configurations. Then, we decide on a design for the experiment (DOE), which contains information about the implementations, configurations, instances, and appropriate measures to evaluate our hypotheses.

**Requirements.** When we execute a designed experiment, we need to ensure basic principles to obtain a scientific value. Two fundamental principles are repeatability and replicability. The goal of *repeatability* is to obtain the same result on the same computer reliably. When repeating a computation of a solver in the same configuration and with the same instances multiple times, we aim for the identical outcome assuming that the algorithm is deterministic. *Replicability* or *recomputability* encompasses the principle that we can obtain the same results confidently given the original artifacts and comparable hard- and software. To ensure these fundamental principles, we are interested in *deterministic hard- and software* platforms for our evaluation, which ensures repeatability and allows us to estimate random errors or study non-deterministic algorithms.

**Structuring Work.** Since tasks can frequently take up much work and the sheer amount of required resources makes it often impossible to execute the experiments on a single computer, we need *scalability* of the experiment to multiple computers. Therefore, we describe the instances, solvers, and conditions under which the experiment will be expected to run successfully. We call the execution of this description a *job*, which is more generally an allocation of resources for a specific amount of time to execute a dedicated task. A *job* may consist of one or more steps, each consisting of one or more process using one or more CPUs. Later, we refer to a *process* by exactly one sequential process. This leads us to *automated job execution* for which we can formulate natural requirements. Since we are interested in fast scheduling and high throughput of our experiments, the system needs to be *resource efficient*. We need *stability* and *resilience* of the job execution system, as we are interested in repeatability and recomputability.

**Automating Job Execution.** Many different job execution systems exist (Wasik et al. 2016; Stump, Sutcliffe, and Tinelli 2014; Beyer, Löwe, and Wendler 2019; Ceri et al. 2003; Chappell 2004; Ibsen and Anstey 2018; Luksa 2017). In academia, the largest available computation power is in *high-performance computing (HPC)* (Eijkhout 2022), which aims for fast, energy efficient, highly parallel, scalable, and isolated execution of computation tasks (Sterling 2002; Jette 2012; Green500 Authors 2022). HPC uses a set of loosely coupled computers acting as one system, called *cluster*, to solve problems that are computationally hard or highly data intensive. A single computer of a cluster is usually referred to as a *node*. Since the size of a cluster can reach thousands of machines, tools for maintainability, scalable cluster management, and job scheduling are necessary. Today's most popular software is the Simple Linux Utility for Resource Management (`SLURM`) (Yoo, Jette, and Grondona 2003; Auble et al. 2023), which contains a scheduling component where jobs describe all details of the execution. For combinatorial experiments, we require certainty and replicability (Beyer, Löwe, and Wendler 2019), which is quite orthogonal to the HPC's goals of fast and energy efficient

| Type | Issue | Example | Reference | Solution |
|---|---|---|---|---|
| System | Kernel | Unexpected performance behavior | (Kocher et al. 2019) | Monitor |
| | CPU throttling | System heats up and reduces performance | (Fichte et al. 2021) | Governors |
| | CPU load | Use of desktop operating systems | (Li, Ding, and Shen 2007) | Avoid |
| | Overhead | Use of virtual machines | (Joy 2015) | Avoid |
| Design of Experiment | Documentation | Options not documented | (McGeoch 2012) | Care |
| | Measurements | core/wall time; virtual/actual memory | (Fichte et al. 2021) | Decision |
| | Slow runs | Proof logging onto slow storage | (Beyer, Löwe, and Wendler 2019) | shm |
| | Parameters | Incomparable parameters | (McGeoch 2012; Bocchese et al. 2018) | Care |
| Execution | Isolation | Resource enforcement fails | (Beyer, Löwe, and Wendler 2019) | cgroups |
| | Memory | Memory paging | (Beyer, Löwe, and Wendler 2019) | cgroups |
| | Slow I/O | Read/write large amounts of data | (IBM Team 2021) | shm |
| | Cacheline | Cores share L2/L3 cache | This paper | resctrl |

Table 1: Listing of pitfalls when running experimental evaluations.

computation and complex storage and memory architectures in clusters. Thus, we need to be extremely careful when setting up and running an experiment.

**Common Pitfalls.** When we are interested in reliable empirical evaluations of combinatorial experiments, we can easily run into numerous pitfalls. We list standard issues that frequently show up in combinatorial evaluations and provide references to the literature for more details in Table 1. We separate these issues into three types depending on the type or phase when these issues show up: the system, the design of experiment, and the execution.

**Measuring.** We use the Linux performance events subsystem (`perf`) to measure runtime, memory, and extended system events (Zijlstra, Molnar, and de Melo 2009; Weaver 2013). `perf` is part of the Linux kernel and allows to monitor both hardware and software at a fairly low overhead. For example, `perf stat -e cycles -I 1000 cat /dev/urandom > /dev/null` measures the number of cycles, which state the CPU frequency at the time of measurement and can be used to quickly spot performance degeneration originating in varying CPU frequency. The CPU frequency is usually adjusted by Linux performance governors (Brodowski et al. 2016). Specialized tools for stressing the system, test initial system performance, and detect silent performance degeneration of hardware are `sysbench` (Zaitsev, Kopytov et al. 2020), `stress-ng` (Haleem et al. 2023), and `GeekBench` (Primate Labs Inc. 2023). Discrepancies in hardware parameters can be spotted using tools such as `hardinfo` (Pereira et al. 2023), `dmidecode` (Cox et al. 2023), and `lshw` (Vincent et al. 2023). Resource limits can be enforced in multiple ways, `BenchExec` (Beyer, Löwe, and Wendler 2019) `runsolver` (Roussel 2011), and `cgroups`. We suggest to use a combination of `runsolver` and `cgroups`, as `BenchExec` can oftentimes not be employed in HPC environments. Using the above mentioned tools, we can tune the system to the best possible performance. If we run a purely sequential execution where each process has exclusive access to the entire hardware and avoid solving multiple instances, many issues can be circumvented (Beyer, Löwe, and Wendler 2019).

**Resource Enforcement.** Many issues that we presented above can already be circumvented by a precise setup of `SLURM` in combination with a dedicated job generators and resource monitoring tooling (Auble et al. 2023). In `SLURM`, we can isolate each individual executions of a solver, configuration, and instance using the `cgroupsv1` plugin (Jackson and Lameter 2006; Auble et al. 2022). The `cgroups` restriction makes sure that the solver sees only the assigned amount of resources (cores and memory) and cores are pinned automatically when the cluster scheduler spawns a task on a node. We can strictly restrict cores by setting the `ConstrainRAMSpace` option. Then, no oversubscription is possible. When we enforce memory limits using the `ConstrainRAMSpace` option in the cgroups plugin, the kernel triggers an out-of-memory if the memory limit is reached and terminates processes.

## Concurrency and Resilience

Literature on empirical evaluation oftentimes suggests to run experiments sequentially where each process has exclusive access to an entire computer to obtain stable and repeatable conditions (Beyer, Löwe, and Wendler 2019; Eggensperger, Lindauer, and Hutter 2019). However, there are two major shortcomings from this approach (i) even in a sequential setting, runtime variations can be significant; and (ii) cost, time, and resource-sustainable perspective still calls for solving multiple instances in parallel since modern computers provide many cores. In this section, we tackle these shortcomings and stabilize hardware for replicable sequential runs and enable concurrent execution of processes by a resilient configurations for cluster schedulers. We take advantage of the maximum available resources resulting in the execution of multiple processes.

Before addressing the shortcomings, we need to identify its origins. The major contributing effect is modern hardware architecture, which is fairly complicated. While modern processors consist of many physical cores access to certain parts of the memory might be fast, slow, or vary in terms of speed. Combinatorial solvers and, in particular, SAT solvers have extensive memory requirements and demand fast memory access (Zhang and Malik 2004; Fichte et al. 2023b). Over

90% of the runtime of a modern SAT solver attributes to a process called unit propagation, which depends on fast memory access. Less known but well-studied is the effect of memory cachelines and mapping between virtual and physical memory to memory bound combinatorial solvers for the same reasons (Hölldobler, Manthey, and Saptawijaya 2010; Fichte et al. 2020). Now, a process may have access to a small or large portion of the memory depending on the current resource allocation. If multiple sequential solvers are executed in parallel, runtime can degenerate quickly.

**Processors and Memory Access.** To design stable and repeatable experiments, we need to revisit basics in hardware architecture and to understand the structure of modern CPUs and the interconnection to memory. Modern computer architecture employs a hierarchy between different types of memory (Hennessy and Patterson 2011). While this is folklore for persistent (HDD, SDD) and volatile memory (DRAM), it is less known that modern processors have only indirect access to dynamic random-access memory (DRAM). The reason can be found in the balance between access time and costs. For DRAM, we see access times of around $10^{-8}$s. But the effective base frequency of a modern processor is around 2GHz, i.e., one cycle per $5 \cdot 10^{-10}$s. In consequence, DRAM is not fast enough to provide data to a processor core. To avoid wasting cycles, faster but more expensive memory is employed, which are called registers. A register provides the fastest way to access data but is usually very small. For example, floating point registers of modern processors contain 512 bit. Behind the registers, we find caches that are slower but larger and still faster than DRAM. Usually, there are multiple cache levels, L1, L2, and sometimes L3 or L4 that occupy notable parts of the actual microchip. In practice, the amount of data that can be reused from a cache (hit) plays an important role in performance. Decreasing the access time to a cache also boosts performance. However, before a core can access data from DRAM, data needs to be fetched into the cacheline. Here, latency matters since a core that runs out of required data needs to wait for the data (stalling). In addition, cores have usually fast access only to a certain part of the DRAM meaning that DRAM on the local socket is faster accessible than DRAM that is wired to another socket. This memory design is called non-uniform memory access (NUMA) and present in multicore architectures (Majo and Gross 2011).

**Example 1.** *Fig. 1 provides a hierarchical map of involved elements in the memory hierarchy of a system with an Intel E5-2650v4. The system contains two physical sockets and 12 physical cores on each socket. Hyperthreading is disabled. The L1 cache can store 32KB data, the L2 256KB data, and L3 30MB data. In total, the 12 cores per socket share one L3 cache and from the specification we see that each socket has 4 memory channels (Intel 2016).*

## Memory-aware Concurrent Job Scheduling

Many modern combinatorial solvers are extremely memory demanding. Additionally, modern computer architecture consists of multiple processor cores, fairly slow main memory, and elaborate memory cachelines to compensate

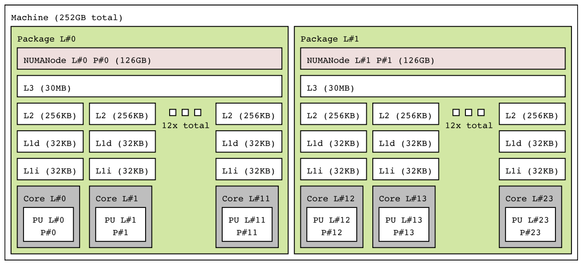

Figure 1: Illustration of a memory cacheline obtained by `lstopo`. We see a hierarchical map of the key computing elements, e.g., NUMA memory nodes, shared caches, processor sockets, and processor cores (threads).

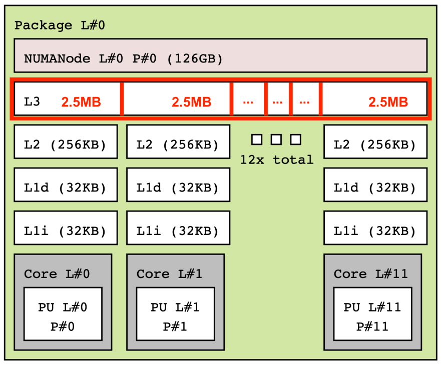

Figure 2: Cache partitioning of the L3-cache from Figure 1.

for slow memory access. Hence, when aiming for repeatable experiments, we need to eliminate unpredictable memory access patterns. Furthermore, when solving multiple instances in parallel, we cannot accidentally block caches and memory, given that modern cores are notably faster than access to the memory. Therefore, we construct configurations for resilient shared caches and ensure that non-uniform memory access (NUMA) is reduced to a minimum.

**L3 Cache Partitioning.** In most modern CPUs, the L3 cache is shared among cores on the same socket. For example, assume that two processes, solver A and solver B with an input instance each, are executed on separate cores but share a cache, e.g., the L3 cache. If solver A runs in a highly memory-intensive phase, solver A might cause all shared caches that are also allocated by solver B to be evicted. Then, the performance of solver B will very likely be reduced. Even worse, if solver B is also highly memory-intensive, both solvers frequently evict each others required caches. In consequence, when running multiple independent jobs on a single socket, we observe severe interference among the cache allocations of those jobs. We tackle this issue by *cache partitioning*, which is a technique to make the cache behavior more predictable. Therefore, we partition a shared cache into smaller parts to which so-called dedicated resource groups or dedicated cores have access. Figure 2 illustrates an equal partition of the L3-cache from Example 1. We assign a process to a resource group by cache access patterns. For replicable empirical evaluations, we define non-overlapping parts for all concurrently running processes and ensure exclusive access to the cache. Thereby, we enforce more predictable memory access, which in turn makes the runtime more repeatable. Note that on certain hardware, the shared caches may have shared buffers or queues that are not part of the partitioning. In addition, smaller caches result in increased cache misses, but more predictable runtime among varying runs.

**Enforcing Non-overlapping Caches.** Recent Linux kernel implementations allow to manually partition shared caches by a hardware control component called *Resource Control (`resctrl`)* (Yu, Luck, and Shivappa 2023; Intel 2023). L3 cache partitioning allows us to manually assign a particular part of the cache memory to a specific process. Unfortunately, we cannot simply split the cache into equal parts or directly assign equal parts to particular cores. Instead we construct a bitmask consisting of $\ell$-bits that specify how to divide the cache. The number of available bits depends on the actual hardware, namely, the physical limitation to access the cache, so-called *cache ways* (Intel 2019). When the mask is set, a set of cores has access to a defined part. To that end, let $n, \ell > 0$ be positive integers representing the number $n$ of cores on the socket and the number $\ell$ of L3 cache ways, respectively. Now, the idea is that we partition the L3 cache in chunks corresponding to the greatest common divisor of $n$ and $\ell$. Therefore, let $\gcd(n, \ell)$ refer to the greatest common divisor, which is the largest positive integer dividing both integers, between $n$ and $\ell$. This results in $k := \gcd(n, \ell)$ partitions of the cache and each of those partitions will be assigned $n/\gcd(n, \ell)$ cores. In more technical details, we set the bitmask as follows. For each integer $1 \leq j \leq n$ representing the core number, we define the bitmask $b_1 \ldots b_m$ where

$$b_i := \begin{cases} 1 & \text{if } \frac{(i-1)\cdot n}{\gcd(n,\ell)} \leq j < \frac{i \cdot n}{\gcd(n,\ell)}, \\ 0 & \text{otherwise.} \end{cases} \qquad (\text{for } i \leq \ell)$$

The full cache bitmask is the disjunction of the bitmasks of its assigned CPU cores. Whenever a core from a partition is used by a process (solver), the corresponding cache partition is made available. By ensuring that processes get assigned to non-overlapping partitions of cores, we also ensure that the L3 cache partitions do not overlap.

**Example 2.** *Consider the memory layout as given in Example 1 and illustrated in Figure 1. When splitting the L3-cache according to our formula above, we obtain 4 partitions containing 5 cache ways and 3 cores each.*

**Core Binding and Memory Channels.** Since our cache partitioning relies on assigning solvers to particular CPU cores, we bind running processes to a specific set of cores within a cluster node (Auble et al. 2023). Furthermore, the available memory channels can have a notable impact on the runtime of solvers on the exact same instances. Hence, we limit the number of running solvers by the available memory channels to avoid over-committing.

## Scheduling in Practice

In the previous section, we introduced concepts and methods to reduce uncertainty in memory access. To obtain actual replicable experimental results, we need to put these insights into practice. Therefore, we establish the technical part next.

**Scheduling Jobs.** We employ the HPC software `SLURM` to describe and execute experiments. We compile a detailed description of the job that is supposed to run on the cluster including a configuration to enforce resource requirements and wrappers to monitor occupied system resources. The cluster scheduler enables us directly to isolate runs, memory, and avoid oversubscribing of resources when properly configured. However, cache-aware scheduling is not available and needs additional effort. We employ the `runsolver` tool (Roussel 2011) to sample memory consumption and patiently terminate a processes hierarchy that rans out-of-memory allowing the solver to write logs and statistics. In addition, we measure the performance of the task during the execution with `perf`. We provide our cluster configuration including additional comments in the supplement (Fichte et al. 2023a). We use `ansible` to deploy configurations onto nodes (Hochstein and Moser 2017).

**Cache Partitioning.** We implement caching partitioning by the `resctrl` Kernel feature. To set up the configuration for each `SLURM` job, we utilize a custom *prolog* script, which runs prior to the job execution. The script creates a new `restcrl` resource group, sets the bitmask according to the formula stated in the previous subsection, and inserts the identifier of the process into that group. Child processes inherit the same restrictions. For more technical details on `resctrl` we refer to the documentation (Intel 2023).

**Memory Channels as Resource.** Since memory channels can have notable practical impact on the runtime of solvers on the exact same instances, we introduce additional features for the cluster scheduler to enable exclusive access also for memory channels. We establish this by a fairly unconventional approach. `SLURM` (`gres.conf`) allows to specify and configure arbitrary Generic RESources (GRES). On each compute node, we employ GRES and create mock-resources, which we call *memch-resources*. These are simply empty files `memch0`, `memch1`, . . . in the directory `/opt/gres/`. GRES recognizes these files as resources. Then, a job can request the virtual resource and the `SLURM` scheduler ensures that no other job accesses our "memch-resources" while it is in use. In the configuration, we link the resource to cores according to the hardware system specification. If a user requests the number of cores that matches with a multiple of the number of cores in a memch-resource, `SLURM` assigns all cores according to the memch-resource. Thereby, we ensure that job is given cores consecutively according to the memch-resource definition, which we produce according to the memory layout of our hardware. We can employ a similar approach for caches, if we want to take L2 or L3 caches into our consideration. For simplicity, also a combined resource capturing stable behavior for cacheline and memory channels can be defined. In that way, users can request both cachelines or memory channels as a resource.

**Example 3.** *For our system described in Section and Figure 1, we have consecutively numbered cores with L1 and L2 caches each, one shared L3 cache on each socket and 4*

```
NodeName=node[1-11] Name=memch File=/opt/gres/memch0
    ↪ Cores=0,1,2
...
```

Figure 3: Scheduler cacheline-aware configuration.

*memory channels, where 4 memory modules are present for each socket. So technically, we could assign set a memch-resource as each multiple of* $24/4/2 = 3$. *However, we align the resource with our L3-cache partitioning from Example 2. Hence, we define the memch-resource* `memch0` *on cores 0–2 and so on (see Figure 3 for details).*

**Task/affinity.** To ensure consistent and stable memory access, we employ the `task/affinity` plugin in `SLURM`, which allows us to bind a task to a specific set of cores within a node (Auble et al. 2023). Using extended parameters, we can schedule tasks according to present NUMA regions `-cpu-bind=rank_ldom` or specify specific orders in which we aim to use cores. Regardless of specific requests in jobs, the plugin ensures that a default memory placement policy is enforced automatically according to the requested cores.

**Storage Access.** In order to avoid issues that originate in slow I/O and that file system caches are employed implicitly, which might slow down or speed up a subsequent execution of on the same instance, we provide each task with an individual copy of the input via the shared memory file system (`/dev/shm`). We copy input files into the temporary folder and provide the actual input from the shared memory file system.

**Copperhead.** The construction of jobs and configurations, which implement the techniques established above, can easily consume precious time for each experiment. In practice, we aim at reducing user overhead as much as possible. Therefore, we designed a tool, called `copperbench`[1] that generates jobs from compact descriptions for experiments. Our tool creates a script that wraps the experimental task to resolve the aforementioned issues. After the job finished, we collect data, parse the output files, and compile a summary. In that way, experimenting can uniformly be automatized.

## Experiments: Concurrent Benchmarking

To investigate effects on memory caches and the execution of solving multiple instances in parallel, we design a small experiment. Binaries, instances, and logs can be found in the supplement (Fichte et al. 2023a). We consider SAT solvers, which are known to be highly memory demanding. For simplicity, we take the solvers `glucose` (Audemard and Simon 2019), `CaDiCaL` (Biere 2019) and `Kissat` (Biere et al. 2020), which show robust performance. We take the instance set *set-asp-gauss*, which contains 200 publicly available SAT instances from a variety of domains with increasing practical hardness (Hoos et al. 2013). We set timeouts to 900s and memouts to 10GB, as we are primarily interested in repeatability on individual instances. In contrast, SAT competitions restrict the runtime to 5,000s and memory to 128GB, however, require certificates.

[1]github.com/tlyphed/copperbench

| #cores | relative time diff [%] | | | | | |
|---|---|---|---|---|---|---|
| | `CaDiCaL` | | `Kissat` | | `glucose` | |
| | off | on | off | on | off | on |
| 1 | -0.0 | -0.0 | -0.0 | -0.0 | -0.0 | -0.0 |
| 2 | 0.1 | 7.0 | -0.0 | 7.5 | 0.1 | 4.8 |
| 4 | -1.6 | 4.6 | -1.7 | 5.1 | -0.7 | 3.4 |
| 6 | -3.5 | – | -3.6 | – | -1.6 | – |
| 8 | -5.3 | -0.4 | -5.4 | -0.3 | -2.6 | -0.4 |
| 24 | -18.9 | – | -20.1 | – | -11.0 | – |

Table 2: Relative wall clock time difference to baseline for each solver in relation to 1 occupied core. Lower is better. on/off refers to cache partitioning. We mark unavailable runs due to bitmask limitations by "–".

| Solver | $t_{\text{off}}[h]$ | $t_{\text{on}}[h]$ | $\Delta[\%]$ |
|---|---|---|---|
| `CaDiCaL` | 8.38 | 9.01 | 7.0 |
| `Kissat` | 6.55 | 6.87 | 4.7 |
| `glucose` | 7.12 | 7.69 | 7.4 |

Table 3: Comparing the total runtime when a solver has exclusive access to the entire node (1 core occupied). $t[h]$ states the total wall clock time and on/off refers to L3-cache partitioning and limiting the size of the partition to a quarter of the total cache; $\Delta[\%]$ gives the relative change in percent.

**Environment.** We run on a cluster consisting of 11 nodes. Each node is equipped with two Intel Xeon E5-2650v4 processors consisting of 12 physical cores running at a base frequency of 2.2GHz, 256GB shared RAM in total. Hyperthreading is disabled. The operating system is a Ubuntu 22.04.2 LTS running a 5.19.0-41-generic Linux kernel.

**Design of Experiment.** Next, we test the effect of the memory cacheline. To this end, we run multiple settings between purely sequential runs with no interference and multiple instances solved in parallel. We repeat each instance 5 times per solver. The considered solvers are sequential and execution means executing a solver on the command line, i.e., the combination of solver, configuration, instance, and repetition. We compare varying number of occupied cores together with activated and deactivated cache partitioning. Both setups implement all other techniques illustrated in the previous sections.

**Expectations.** Our expectations are as follows:

E1 The runtime difference between several repetitions of each instance is low (average), but can be high on certain instances regardless of parallel runs.

E2 Parallel execution degenerates the total runtime, number of solved instances, and results in higher deviation between repetitions.

E3 If memory requirements are chosen according to available memory cacheline and channels, variation in the total runtime and number of solved instances is minimal.

**Observation.** In Table 2, we find a summary of the results for the different solvers and settings comparing the relative

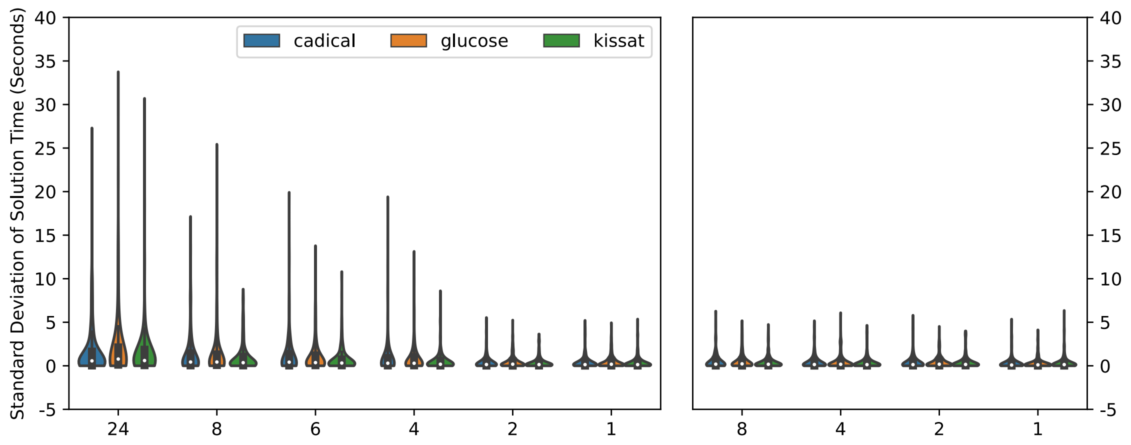

Figure 4: The standard deviation of solving time among the 5 repetitions for the considered settings without (left) and with L3-cache partitioning (right). Besides on the very right, the x-axis refers to the number of active cores, meaning, on each node in total x instances are run in parallel while each solver instance occupies one core. The right most setting "S" shows the results for running a single solver exclusively on one node but with the same cache restriction as running 8 in parallel.

runtime difference. Figure 4 shows the standard deviation of the solving times among the 5 runs in the different settings. On the left side, we illustrate the situation without any modifications to the L3-cache. Whereas, on the right side, we visualize the results when restricting the L3-cache according to our formula in Section  and the setting "S" which has the same cache restriction as the 8 solver setting and shows that differences in the current load have no impact. Table 3 illustrates the performance loss when activating cache partitioning and assigning the same amount of cache as in the setting of 8 parallel solvers in wall clock time for the baseline, that is, occupying only one core and having exclusive access to an entire node (the best possible performance for an individual solver run). Restricting the cache significantly affects performance. However, as we have already stated above this setting produces the same performance as running 8 solvers in parallel. Hence, the performance is marginally worse but stable. We observe that without cache partitioning the overall performance of solvers degrades with the number of simultaneously committed cores (Table 2). Moreover, the variance in runtime for the exact same solver and input instance 4 increase significantly. The runtime increases up to 5% when running 8 solvers and 18% when running 24 solvers in parallel on one node and the standard deviation for the exact same instances can be up to a factor of 5 to 7 higher (from 5s to 25 or 35s, respectively). If we enable cache partitioning, total runtime is stable even when simultaneously committing multiple cores, i.e., running multiple solvers in parallel on a node. Runtime degeneration is slightly worse than without cache partitioning (c.f., Table 2 and Table 3). Note that the employed instance set contains instances with varying runtimes. The effect increases when the total runtime of an instance increases. Hence, we expect an even more problematic behavior in experiments, where instances that were solved extremely fast, are excluded.

**Summary.** Our experiments show that running multiple jobs in parallel can severely influence performance and thus repeatability. As shown in Table 2, running one job per core (24) can lead to drastically longer solving time and thus also less solved instances. Further, depending on the current load when a job was run, its performance can differ. By careful cache partitioning, we obtain stable, resilient, and replicable experiments. Partitioning increases the individual runtime slightly, but we obtain much more sustainable hardware usage. To this end, we run 8 processes in parallel, which is the expected technical maximum on our setting due to the 4 memory channels per socket. Furthermore, if we select a uniform cache size regardless of the actual occupied cores, see Figure 4 (right) 8 vs. S, we ensure stable results independent of the current hardware load.

## Conclusion

We investigate how sustainable, replicable empirical experiments can be designed and established. In contrast to previous work, we suggest conditions for parallel execution. By exploring the factor system memory, we eliminate a major issue that is often neglected as solving focuses on processors only. We illustrate how widely employed cluster schedulers can be fruitfully employed for combinatorial evaluations. Finally, we emphasize that a proper setup of empirical work should not be trivialized. The effect of a problematic execution can easily destroy the scientific value of an experiment.

Our work opens up multiple directions for future research. We believe that an interesting question is to evaluate whether task isolation, frequency scaling, and further methods that involve system memory can be employed to design abstract and reliable execution environments that provide repeatability beyond the hardware where it was executed.

## Acknowledgments

Authors are ordered alphabetically. The work has been carried out while Fichte and Hecher visited the Simons Institute at UC Berkeley. Research is supported by ELLIIT funded by the Swedish government; the Austrian Academy of Sciences (ÖAW), DOC Fellowship; the Austrian Science Fund (FWF), grant J4656, and the Society for Research Funding in Lower Austria (GFF) grant ExzF-0004.

## References

Auble, D.; et al. 2022. Control Group in Slurm. https://slurm.schedmd.com/cgroups.html.

Auble, D.; et al. 2023. Slurm Workload Manager. https://slurm.schedmd.com.

Audemard, G.; Paulevé, L.; and Simon, L. 2020. SAT Heritage: A Community-Driven Effort for Archiving, Building and Running More Than Thousand SAT Solvers. In *SAT'20*, 107–113. Springer Verlag.

Audemard, G.; and Simon, L. 2019. Glucose in the SAT Race 2019. In *Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions*, 19–20. University of Helsinki.

Bartocci, E.; Beyer, D.; Black, P. E.; Fedyukovich, G.; Garavel, H.; Hartmanns, A.; Huisman, M.; Kordon, F.; Nagele, J.; Sighireanu, M.; Steffen, B.; Suda, M.; Sutcliffe, G.; Weber, T.; and Yamada, A. 2019. TOOLympics 2019: An Overview of Competitions in Formal Methods. In *TACAS'19*, 3–24. Springer Verlag.

Beyer, D.; Löwe, S.; and Wendler, P. 2019. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1): 1–29.

Biere, A. 2019. CaDiCaL Simplified Satisfiability Solver. http://fmv.jku.at/cadical/.

Biere, A.; Fazekas, K.; Fleury, M.; and Heisinger, M. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *SAT COMPETITION 2020*.

Biere, A.; Fleury, M.; Froleyks, N.; and Heule, M. J. 2023. The SAT Museum. In *Proceedings of the 14th International Workshop on Pragmatics of SAT (PoS'23)*, volume 3545. CEUR Workshop Proceedings (CEUR-WS.org).

Bocchese, A. F.; Fawcett, C.; Vallati, M.; Gerevini, A. E.; and Hoos, H. H. 2018. Performance robustness of AI planners in the 2014 international planning competition. *AI Communications*, 31(6): 445–463.

Bridi, T. 2018. *Scalable optimization-based Scheduling approaches for HPC facilities*. Ph.D. thesis, Universida di Bolonga.

Brodowski, D.; Golde, N.; Wysocki, R. J.; and Kumar, V. 2016. CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

Ceri, S.; Fraternali, P.; Bongio, A.; Brambilla, M.; Comai, S.; and Matera, M. 2003. *Designing data-intensive Web applications*. Morgan Kaufmann.

Chappell, D. A. 2004. *Enterprise service bus: Theory in practice*. O'Reilly.

Cosmo, R. D.; and Zacchiroli, S. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES'17*.

Cox, A.; Delvare, J.; Delvare, J.; et al. 2023. dmidecode(8) - Linux man page. https://linux.die.net/man/8/dmidecode.

Eggensperger, K.; Lindauer, M.; and Hutter, F. 2019. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, 861–893.

Eijkhout, V. 2022. *The Art of HPC*. Lulu Press. https://github.com/VictorEijkhout/TheArtofHPC_pdfs/.

Elffers, J.; Giráldez-Cru, J.; Gocht, S.; Nordström, J.; and Simon, L. 2018. Seeking Practical CDCL Insights from Theoretical SAT Benchmarks. In *IJCAI'18*, 1300–1308. IJCAI.

Fichte, J. K.; Geibinger, T.; Hecher, M.; and Schlögel, M. 2023a. Dataset: Parallel Empirical Evaluations: Resilience Despite Concurrency. Zenodo. doi.org/10.5281/zenodo.10400972.

Fichte, J. K.; Hecher, M.; Le Berre, D.; and Szeider, S. 2023b. The Silent (R)Evolution of SAT. *Communications of the ACM*, 66(6): 64–72.

Fichte, J. K.; Hecher, M.; McCreesh, C.; and Shahab, A. 2021. Complications for Computational Experiments from Modern Processors. In *CP'21*, 25:1–25:21. Dagstuhl Publishing.

Fichte, J. K.; Hecher, M.; and Szeider, S. 2023. A Time Leap Challenge for SAT-Solving. *CoRR*. arxiv.org/abs/2008.02215. A preliminary version appeared in CP'20.

Fichte, J. K.; Manthey, N.; Schidler, A.; and Stecklina, J. 2020. Towards Faster Reasoners by using Transparent Huge Pages. In *CP'20*, 304–322. Springer Verlag.

Force, I. E. T. 1993. IETF Online Proceedings. https://www.ietf.org/old/2009/proceedings_directory.html.

Galleguillos, C.; Kiziltan, Z.; Sîrbu, A.; and Babaoglu, O. 2019. Constraint Programming-Based Job Dispatching for Modern HPC Applications. In *CP'19*, 438–455. Springer Verlag.

Geer, D.; Bace, R.; Gutmann, P.; Metzger, P.; Pfleeger, C. P.; Quarterman, J. S.; and Schneier, B. 2003. CyberInsecurity: The Cost of Monopoly. https://cryptome.org/cyberinsecurity.htm.

Georgiou, Y.; Cadeau, T.; Glesser, D.; Auble, D.; Jette, M.; and Hautreux, M. 2014. Energy Accounting and Control with SLURM Resource and Job Management System. In *Distributed Computing and Networking*, 96–118. Springer Berlin Heidelberg.

Green500 Authors. 2022. The Green500 Supercomputers. https://www.top500.org/lists/green500/.

Haleem, A.; et al. 2023. stress-ng (stress next generation). https://github.com/ColinIanKing/stress-ng.

Henderson, J.; and Patel, S. 2002. The Role of Market-based and Committee-based Standards. Technical report, Babson College.

Hennessy, J. L.; and Patterson, D. A. 2011. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition.

Heule, M. J. H.; and Kullmann, O. 2017. The Science of Brute Force. *Communications of the ACM*, 60(8): 70—79.

Hochstein, L.; and Moser, R. 2017. *Ansible: Up and Running: Automating configuration management and deployment the easy way*. O'Reilly Media.

Hölldobler, S.; Manthey, N.; and Saptawijaya, A. 2010. Improving Resource-Unaware SAT Solvers. In *LPAR'16*, 519–534. Springer Verlag.

Hoos, H. H.; Kaufmann, B.; Schaub, T.; and Schneider, M. 2013. Robust Benchmark Set Selection for Boolean Constraint Solvers. In *LION'13*, 138–152. Springer Verlag.

IBM Team. 2021. IBM Debugging disk I/O on Linux servers. https://www.ibm.com/docs/en/ioc/5.2.0?topic=resources-debugging-disk-io-linux-servers.

Ibsen, C.; and Anstey, J. 2018. *Camel in action*. Simon and Schuster.

Intel. 2016. Specification: Intel® Xeon® Processor E5-2650 v4 30M Cache, 2.20 GHz. https://www.intel.com/content/www/us/en/products/sku/91767/intel-xeon-processor-e52650-v4-30m-cache-2-20-ghz/specifications.html.

Intel. 2019. Use Intel® Resource Director Technology to Allocate Last Level Cache (LLC). https://www.intel.com/content/www/us/en/developer/articles/technical/use-intel-resource-director-technology-to-allocate-last-level-cache-llc.html.

Intel. 2023. User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat/wiki/resctrl/.

Jackson, P.; and Lameter, C. 2006. cgroups - Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

Jette, M. 2012. Slurm Workload Manager Architecture, Configuration and Use. https://www.open-mpi.org/video/slurm/Slurm_EMC_Dec2012.pdf.

Joy, A. M. 2015. Performance comparison between Linux containers and virtual machines. In *ICACEA'15*, 342–346.

Kocher, P.; Horn, J.; Fogh, A.; ; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; Schwarz, M.; and Yarom, Y. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P'19*.

Koopmann, P.; Hähnel, M.; and Turhan, A.-Y. 2017. Energy-Efficiency of OWL Reasoners—Frequency Matters. In *JIST'17*, 86–101. Springer Verlag.

Li, C.; Ding, C.; and Shen, K. 2007. Quantifying the Cost of Context Switch. In *ExpCS'07*, 2–es. Association for Computing Machinery, New York.

Luksa, M. 2017. *Kubernetes in action*. Simon and Schuster.

Majo, Z.; and Gross, T. R. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *SYSTOR'11*. Association for Computing Machinery, New York.

McGeoch, C. C. 2012. *A Guide to Experimental Algorithmics*. Cambridge University Press.

Müller-Hannemann, M.; and Schirra, S., eds. 2010. *Algorithm Engineering*. Springer Verlag. ISBN 978-3-642-14866-8.

Pereira, L. A. F.; et al. 2023. HARDINFO. https://github.com/lpereira/hardinfo.

Primate Labs Inc. 2023. GeekBench. https://www.geekbench.com/download/linux/.

Roussel, O. 2011. Controlling a Solver Execution with the runsolver Tool. *J. on Satisfiability, Boolean Modeling and Computation*, 139–144.

Sakallah, K. 2023. A Roadmap for the Next Phase of SAT Research. https://simons.berkeley.edu/talks/karem-sakallah-university-michigan-2023-04-18.

Sterling, T. L. 2002. *Beowulf cluster computing with Linux*. MIT Press.

Strevell, M.; Lambiaso, D.; Brendamour, A.; and Squillo, T. 2019. Designing an Energy-Efficient HPC Supercomputing Center. In *ICPP Workshops'19*. Association for Computing Machinery, New York.

Stump, A.; Sutcliffe, G.; and Tinelli, C. 2014. StarExec: A Cross-Community Infrastructure for Logic Solving. In *IJCAR'14*, 367–373. Springer Verlag.

Vallet, N.; Michonneau, D.; and Tournier, S. 2022. Toward practical transparent verifiable and long-term reproducible research using Guix. *Scientific Data*, 9(1): 597.

Vercellino, C.; Scionti, A.; Varavallo, G.; Viviani, P.; Vitali, G.; and Terzo, O. 2023. A Machine Learning Approach for an HPC Use Case: the Jobs Queuing Time Prediction. *Future Generation Computer Systems*, 215–230.

Vincent, L.; et al. 2023. lshw: HardWare LiSter for Linux. https://github.com/lyonel/lshw.

Wasik, S.; Antczak, M.; Badura, J.; Laskowski, A.; and Sternal, T. 2016. Optil.Io: Cloud Based Platform For Solving Optimization Problems Using Crowdsourcing Approach. In *CSCW'16*, 433–436. Association for Computing Machinery, New York.

Weaver, V. M. 2013. Linux perf_event features and overhead. In *The 2nd international workshop on performance analysis of workload optimized systems (FastPath'13)*, 5.

Yoo, A. B.; Jette, M. A.; and Grondona, M. 2003. SLURM: Simple Linux Utility for Resource Management. In *JSSPP'03*, 44–60. Springer Verlag.

Yu, F.; Luck, T.; and Shivappa, V. 2023. The Linux Kernel: User Interface for Resource Control feature. https://docs.kernel.org/arch/x86/resctrl.html.

Zaitsev, P.; Kopytov, A.; et al. 2020. sysbench. https://github.com/akopytov/sysbench.

Zhang, L.; and Malik, S. 2004. Cache Performance of SAT Solvers: a Case Study for Efficient Implementation of Algorithms. In *Theory and Applications of Satisfiability Testing*, 287–298. Springer Berlin Heidelberg.

Zijlstra, P.; Molnar, I.; and de Melo, A. C. 2009. Performance Events Subsystem. https://github.com/torvalds/linux/tree/master/tools/perf.