Complications for Computational Experiments from Modern Processors*

Johannes K. Fichte
TU Dresden, Germany

Markus Hecher
TU Wien, Austria and University of Potsdam, Germany

Ciaran McCreesh
University of Glasgow, Scotland, UK

Anas Shahab
TU Dresden, Germany

Abstract

In this paper, we revisit the approach to empirical experiments for combinatorial solvers. We provide a brief survey on tools that can help to make empirical work easier. We illustrate origins of uncertainty in modern hardware and show how strong the influence of certain aspects of modern hardware and its experimental setup can be in an actual experimental evaluation. More specifically, there can be situations where (i) two different researchers run a reasonable-looking experiment comparing the same solvers and come to different conclusions and (ii) one researcher runs the same experiment twice on the same hardware and reaches different conclusions based upon how the hardware is configured and used. We investigate these situations from a hardware perspective. Furthermore, we provide an overview on standard measures, detailed explanations on effects, potential errors, and biased suggestions for useful tools. Alongside the tools, we discuss their feasibility as experiments often run on clusters to which the experimentalist has only limited access. Our work sheds light on a number of benchmarking-related issues which could be considered to be folklore or even myths.

2012 ACM Subject Classification Computer systems organization → Multicore architectures; Hardware → Temperature monitoring; Hardware → Impact on the environment; Hardware → Platform power issues; Theory of computation → Design and analysis of algorithms

Keywords and phrases Experimenting, Combinatorial Solving, Empirical Work

Digital Object Identifier 10.4230/LIPIcs.CP.2021.55

Funding *Johannes K. Fichte*: Google Fellowship at the Simons Institute, UC Berkeley.
Markus Hecher: FWF Grants Y698 and P32830 and Grant WWTF ICT19-065.

Acknowledgements Authors are given in alphabetical order. The work has been carried out while the first three authors were visiting the Simons Institute for the Theory of Computing.

1 Introduction

"Why trust science?" is the title of a recent popular science book by Naomi Oreskes [74]. We can ask the same question of combinatorial sciences, algorithms, and evaluations: *Why trust an empirical experiment?* Roughly speaking, in science, we try to understand why things happen in the real world and investigate them with the help of scientific methods. One important aspect to make an empirical evaluation trustworthy is reproducibility. This topic has been the subject of much recent scrutiny, with some arguing there is a reproducibility crisis in areas fields of computer science [33, 31] and even a replicability crisis in other scientific

* This work was done in part while the first three authors were participating in a program at the Simons Institute for the Theory of Computing.

fields of research [81]. Luckily in combinatorial problem solving, replicability is often already indirectly addressed in public challenges, which many combinatorial solving communities organize in order to foster implementations and evaluations [47, 82, 80, 83, 86, 20, 21]. The challenges provide a place for empirical evaluations, feature shared benchmarks, and support long-term heritage [3, 17, 22, 54]. It is therefore often assumed that everything should be judged with respect to these benchmarks or latest solvers [5]. However, benchmarks featured in competitions are not necessarily robust [40, 49] and might bias towards existing solving approaches and heuristics. On that account, one can argue that non-competitive evaluations are quite helpful for papers that are orthogonal to classical improvements over one particular solving technique or algorithm [19, 30]. There, one can often see a strong focus on algorithm engineering and their evaluation [63], which might not always be desired from a theoretical perspective. In particular, it makes reproducibility far less obvious than one would expect from theory. While reproducibility initiatives are becoming fashionable [73, 59], aspects are often left out in practical algorithm engineering and when testing combinatorial implementations: (i) the test-setup is not given (*no protocol*) or error prone (*no failure analysis/considerations*), (ii) modern hardware is simplified to the von-Neumann model (Princeton architecture) [87] and considered deterministic, and (iii) underlying software is neglected.

In this work, we summarize a list of topics to consider that might be folklore to an experienced engineer, but are often only mentioned between the lines while being crucial to actual reproducibility (Section 2.1). We include a list of *system and environment parameters* that are impactful when carrying out empirical work (Section 2.3). We summarize useful tools and list *practical problems* that repeatedly occur when experimenting (Sections 3.1, 3.2, and 3.3). In the main part of our paper, we provide an initial *list of issues* caused by modern consumer hardware that can have a notable impact if setup and configurations are not carefully designed (Table 1). We show by example that *one can achieve different results* in the number of solved instances ranging from 5%–40% on the same hardware, depending on the setup (Experiment 1, Table 3). This could suggest that it is not always meaningful to only prefer solvers that beat the "best" solver, but to aim for *clean benchmark settings* and elaborate discussions that highlight both the solver's advantages and disadvantages.

Related Works. There are various works that address aspects of reproducibility [3, 7, 8, 15, 16, 56, 79, 94, 98] and experimental design [39, 63, 71], including micro-benchmarking, which requires special attention in terms of statistical analysis [42], [71, Ch.8]. Previous works neglected effects of modern parallel hardware on experimenting and some aspects have only been addressed in the background by the community. We put attention on certain issues arising on modern machines, updating outdated assumptions on measures, and illustrating how certain problems can be omitted. In the sequel, we revisit some of these related works.

2 Evaluating Combinatorial Algorithms

Natural sciences have a long tradition in designing experiments (DOE). Practical experiments date back to the ancient Greek philosophers such as Thales and Anaximenes with empirically verifiable ideas. There, *methodology is key* and has a long tradition with formal approaches existing since the late 1920s [25, 26]. Methodology not only involves the experiment itself, but also observation, measurement, and the design of test aiming to reduce external influence. Already in 1995, Hooker [39] discussed challenges in competitive evaluations of heuristics. A variety of these challenges are still of relevance in today's combinatorial solving community. In particular, emphasis on competitions tells which algorithms/implementations are better, but not *why*; this remains a particularly big challenge in the SAT community [27, 91]. If a novel implementation wins, it is accepted; otherwise, it is considered as failure, resulting in a high incentive to find the best possible parameter settings. The challenge of designing an experiment (DOE) has meanwhile been addressed by a more experimental community in broad guides [63] or algorithm engineering works [71]. In contrast, theoretical computer scientists often neglect environmental considerations due to the assumption that modern hardware behaves similar to simplified mathematical machine models [90] or classical hardware models [93]. Unfortunately, this is no longer the case for modern architectures. There is a list of concepts and external influences that can interfere, some of which are discussed below.

2.1 Repeatability, Replicability, and Reproducibility

When conducting a study or experiment, a central goal is to reduce inconsistencies between theoretical descriptions and actual experiments. Three major principles play a central role: repeatability, replicability, and reproducibility. Unsurprisingly, these topics are also critically discussed in other scientific fields [65] and sometimes confused with each other.

Repeatability requires repeating a computation by the same researcher with the same equipment at reliably the same result. The main purpose is often to estimate random errors inherent in any observation. When evaluating combinatorial solvers, repeatability translates to running the same solver with the same configuration on a given instance multiple times, maybe even on different hardware. Some publicly accessible evaluation platforms for combinatorial competitions address repeatability to a certain extent, as for example, StarExec [84] and Optil.io [94]. Effective tools to measure and control the execution of combinatorial solvers are runsolver [79] and BenchExec [7, 8].

Replicability, sometimes also called method reproducibility, refers to the principle that if an experiment is replicated by independent researchers with access to the original artifacts and same methodology, that then outcomes are the same with high confidence. When evaluating combinatorial solvers, replicability translates to running the same solver with the same configuration and instance on a different system by independent researchers, which is sometimes also called *recomputability*. Relevant aspects relate to works such as the Heritage projects [3, 16, 15], which preserve access to old solvers and making sources accessible to a broad community, or Singularity, which aims for easy an setup on high-performance computing (HPC) systems with few prerequisites on the environment [56, 98]. Another initiative (Guix) aims for a dedicated Linux distribution that provides highly stable system dependency configurations [1, 96]. Already in 2013, the recomputation manifesto postulated that one can only build on previous work if it can properly be replicated as a first step [28]. In addition, it makes research more efficient, similarly to how high quality publications can benefit other researchers. In contrast, some researchers argue that replicability is not worth considering, since sharing all artifacts is a non-trivial activity, which in consequence wastes efforts of the researchers [18]. Still, replicability is getting solid attention within the experimental algorithmics community [73], since it supports quality assurance.

Reproducibility aims for being able to obtain the same outcome using artifacts, which independent researchers develop without help of the original authors. When evaluating combinatorial solvers reproducibility roughly refers to another group constructing a second solver that implements the same algorithmic ideas. For example, Knuth re-implemented SAT algorithms from several epochs [52]. More experimental directions are investigations into robustness of benchmark sets and their evaluation measures [49]. Reproducibility can also be interpreted fairly vaguely [32]. Interestingly, the literature on experimental setup [63] and algorithm engineering [71] already contains a variety of suggestions to obtain reproducibility.

Repeatability and to some extent also replicability are the focus of our paper. Our aim is to make researchers aware of potential problems caused by modern computer systems, illustrate how to detect and reduce them without spending hours of debugging or overvaluing small improvements. Before we go into details, we briefly discuss principles and tools that support both repeatability and replicability. Since recent works on reproducibility provide various helpful suggestions on replicability in terms of environment [75], we focus only on aspects that might degrade long-term repeatability and replicability, resulting in over-engineering or over-tooling. We argue in favour of reviving an old Unix philosophy: build simple, short, clear, modular, and extensible code [64] both for the actual solver as well as the evaluation. Always keep dependencies low and provide a statically linked binary along with your code [89, 60] or a simple virtual environment to reproduce dependencies if you use interpreted languages. Even if the source code does not compile with newer versions, binary compatibility is mostly maintained for decades. The primary focus of container-based solutions, such as Singularity [56], is current accessibility of scientific computing software that requires extensive libraries and complex environments. It is quite useful if the software is widely used, requires complicated setup on high performance computing environments, and is continuously maintained. Container-based solutions can also be useful for building source code on old operating systems [3]. However, they introduce additional dependencies, increase conceptual complexity, can have notable runtime overhead under certain conditions [100], require additional work for a proper setup (both hosts as well as containers), and increase chances that the software does not out run of the box in 3 years. A practical observation illustrates this quite well: already since 2010, a meta software (Vagrant) tries to wrap providers such as VirtualBox, Hyper-V, Docker, VMWare, or AWS. While virtualization can be tempting to use, chances are high that some of these providers upgrade functionality or disappear entirely resulting in useless migration efforts.

2.2 An Experiment

In the beginning of Section 2, we stated classical experimental viewpoints: fixed solver or fixed instance set. In contrast, we take a third perspective by fixing both the instance set and the solver and focus on differences in hardware configurations. Therefore, we turn our attention to a recent experiment on SAT solvers (time leap challenge) [22]. We repeat the experiment with the solver `CaDiCal` on other hardware to investigate side effects of experimental setup and hardware. Also, we use *set-asp-gauss* as instances, which contains 200 publicly available SAT instances from a variety of domains with increasing practical hardness [40][1]. We take a timeout of 900 seconds, but would like to point out that recent SAT competitions restrict the total runtime over all instances to 5,000 seconds. We run experiments on the following environments: Comet Lake (i7 Gen10): Intel i7-10710U 4.7 GHz, Linux 5.4.0-72-generic, Ubuntu 20.04; Haswell (Xeon Gen4): 2x Intel Xeon E5-2680v3 CPUs, Linux 3.10.0-1062, RHEL 7.7; Rome (Zen2): 2x AMD EPYC 7702, Linux 3.10.0-1062, RHEL 7.7; and Skylake (Xeon Gen6): Xeon Silver 4112 CPU, Linux version 4.15.0-91, Mint 19. We explicitly include cheap mobile hardware by using a Comet Lake (i7 Gen10) CPU, since not every group can afford expensive server hardware or spend valuable research time on setting up stable experiments on a cluster.

Table 1 illustrates the results of the experiment on varying hardware. Unsurprisingly, the modern hardware running at 4.7 GHz solves the most instances. Somewhat unexpected is that two potentially faster processors solve fewer instances. Namely, the Rome CPU which

[1] The benchmark set is available for download at `https://www.cs.uni-potsdam.de/wv/projects/sets/set-industrial-09-12-gauss.tar.xz`

Processor (CPU)	f	p	$s(15)$	$t[h]$
Skylake	3.0	1	190	5.12
Haswell	3.3	1	189	3.89
Rome	3.4	1	190	3.79
Comet Lake	4.7	1	191	3.81
Comet Lake	4.7	6	189	6.13
Comet Lake	4.7	12	176	7.18

Table 1 Number of solved instances out of 200 SAT instances running the solver `CaDiCal` on varying platforms. Column $s(15)$ contains the number of solved instances when timeout is 15 minutes; f and p refer to the CPU frequency in GHz and number of solvers running in parallel, respectively. The t column contains the total runtime in hours for all instances solved within 15 minutes.

is faster than the Skylake CPU solves fewer instances and similarly the Haswell solves fewer instances than the Skylake. Since both processors are different generations one might expect that the AMD CPU is simply slower. While the 5% fewer solved instances might seem not much comparing the results to the ones of the time leap challenge, it would mean that a ten year old solver solves almost the same number of instances on a modern hardware as `CaDiCal` on very recent hardware. Below, we explain that this is clearly not the case and illustrate details of the experimental setup that contribute to the low number of solved instances. In contrast, when comparing the number of solved instances for the Comet Lake configurations, it is obvious to an experienced reader that while the Comet Lake CPU exposes 12 software cores, due to multithreading (MT) only 6 physical cores are available. Still, when using all physical cores, we have more than 30% higher runtime, which can be particularly problematic when comparing to settings that prefer total runtime as measure. To avoid this, we could simply not run solvers in parallel; however, this seems quite impractical and inefficient. In the following sections, we clear up which problems in the setup may have caused the differences and illustrate how to avoid such issues.

2.3 Uncertainty on Modern Hardware

Modern processors can do many calculations at the same time by using multiple cores on each processor and each core also has built-in a certain parallelism. While this can be exploited explicitly in terms of parallel programming frameworks, some features are already done by on-board circuits or firmware, which is a low level software layer between the CPU hardware and the operating system. Compile time optimizations such as *automated parallel execution optimization* and *cache performance optimization* [4] can then automatically employ specific features. For example, *loop optimization* tries to automatically rewrite loops in programs such that the loop can be executed in parallel on multiprocessor systems. Scheduling splits loops so that they can run concurrently on multiple processors. Vectorization optimizes for running many loop iterations on parallel hardware that supports single instruction multiple data (SIMD). Therefore, instead of processing a single element of a vector N times, m elements of a vector are processed simultaneously N/m times. In fact, modern CPUs have so-called *vector instruction sets* such as SSE, AVX, NEON, or SVE depending on the architecture, which makes them SIMD hardware. Loop vectorization can have a significant impact on the runtime due to effects on pipeline synchronization or data-movement timing. Usually, dependency analysis tries to optimize these operations. But depending on the compiler (GCC, Intel, or LLVM) different runtimes of the resulting binary can be observed [92].

Processor specific features add to less pre-calculable behavior. *Turbo Boost*, which was introduced around 2008, allows to dynamically overclock the CPU if the operating system requests the highest performance state of the processor [66]. *Thermal design power (TDP)*, which was established around 2012, allows to scale the power (energy transfer rate) variably between 50W and 155W [70] to save energy depending on the system load. In particular, this is active on laptop systems that are not connected to an electrical outlet or if certain system sensors detect high temperature. *Turbo Boost 2.0* was introduced around 2011 and it uses time windows with different levels of power limits, so that a processor can boost its frequency beyond its thermal design power, which can thus only be maintained for a few seconds without destroying the CPU [2]. *Huge Pages*, which were increased to 1GB, can reduce the overhead of virtual memory translations by using larger virtual memory page sizes which increases the effective size of caches in the memory pipeline [24]. *Branch prediction*, whose early forms already date back to the 1980s in SPARC or MIPS [68], speculates on the condition that most likely occurs if a conditional operation is run. Modern CPUs have a quite sophisticated branch prediction system, which executes potential operations in parallel [48]. The CPU can then complete an operation ahead of time if it made a good guess and significantly speed up the computation. This often depends on how frequently the same operation is used. Otherwise, if the branch predictor guessed wrong, the CPU executes the other branch of operation with some delay, which can be longer than expected as modern processors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. The situation gets more complicated when substantial architectural bugs are mitigated or patched, as this can notably slow down the total system performance [58, 53].

Clearly, we need practical empirical evaluations of algorithms and techniques and oftentimes it is not useful to just restrict an evaluation to existing benchmarks used in competitions, if they even exists. But just the “complications” or, more formally, source for an error in measurement mentioned above, could make the outcome of an experiment far less deterministic than one would expect. For that reason, we suggest a more rigorous process when evaluating implementations, including the understanding of measurements and effects of potential errors on the outcome as well as approaches to reduce unexpected and not entirely deterministic effects. In a way, the following sections provide a modern perspective on simple measures (runtime) that incorporate state-of-the-art in hardware and operating system technology.

3 Measurements and Hardware Effects

In the following, we discuss measurements used when evaluating runtime of empirical work. Along with the measures, we recap useful measuring and controlling tools. Since most of the tools are highly specific to the kernel in the used operating system, we restrict ourselves to recent versions of Linux and widely used distributions thereof.

3.1 Runtime

When evaluating algorithms, a central question is how long its implementation actually runs on the input data (runtime). There are five main measures that are interesting in this context: real-time, user-time, system-time, CPU-usage, and system load. The *real-time*, frequently just called *wallclock time*, measures the elapsed time between start and end of a considered program (method entry and exit). In contrast, *CPU-time* measures the actual amount of time for which a CPU was used when executing a program. More precisely, the *user-time* measures how much CPU-time was utilized and *system-time* how much the operating system has used the CPU-time due to system calls by the considered program. Both measures neglect waiting times for input/output (I/O) operations or entering a low-power mode due to energy saving or thermal reasons. There are more detailed time measures on time spent in user/kernel space, idle, waiting for disk, handling interrupts, or waiting for external resources if the system runs on a hypervisor. *CPU-usage* considers the ratio of CPU-time to the CPU capacity as a percentage. It allows for estimating how busy a system is, to quantify how processors are shared between other programs. The *system load* indicates how many programs have been waiting for resources, e.g., a value of 0.05 means that 0.05 processes were waiting for resources. The system load is often given as *load average* which states the last average of a fixed period of time; by default, system tools report three time periods (1, 5, and 15 minutes). If the load average goes above the number of physical CPUs on the system, a program has to idle and wait for free resources on the CPU.

Suggested Measure for Runtime. When measuring runtime, the obvious measure is to use elapsed time, so as to measure the real-time of a program. However, when setting an experiment, we aim to (i) reduce external influences, (ii) conduct reasonable failure analysis, or (iii) use an alternative measure in the worst-case. Real-time can be unreliable on sequential systems as a program can be influenced by other programs running on the system and the program competes on resources with the operating system. For that reason, dated guides on experimenting suggested to run a clean system and obtain a magic overhead factor, which follows Direction (ii) replacing an expected failure analysis. More recent guides, follow Direction (iii) and suggest to use CPU-time [63, 71], mainly arguing that real-time minus unwanted external interruption should roughly equal used CPU-time when evaluating sequential combinatorial solvers that use a CPU close to 100%. However, we believe that the best approach for an experimental setup is always to follow Direction (i) and reduce external influences. Suggestions on CPU-time are outdated as modern hardware is inherently parallel. Even small single-board computers such as the Raspberry Pi have multi-core processors. This allows to run programs and the operating system simply in parallel. Still, CPU-time might prove useful to estimate a degree of parallelism or debug unexpected behavior.

Expected Errors. Real-time is measured by an internal clock of the computer. Nowadays, hardware clocks are still not very accurate. Expected time drifts are about one second per day [95], which is often negligible for standard experiments as micro-benchmarking is anyways rarely meaningful. But, time drift can be far higher, for example, when system load is very high [72] and systems run within virtual machine guests [44, 6, 85]. Since modern cryptography still requires exact system times, all state-of-the-art operating systems synchronize the system clock frequently. Unfortunately, many widely used tools do not incorporate time drifts and corrections by time synchronization utilities. Thus, if time drifts are high (virtual machines) or a misconfiguration of the synchronization service occurs, measures can be completely unreliable. Note that we can expect difference between CPU-time and real-time in cases where heavy or slow access to storage occurs, slow network is involved, or unexpected parallel execution happens. However, this should be ground to investigate details and either eliminate problems in the experimental setup or update problematic program parts, if possible. A classical example occurs when using the ILP solver `CPLEX`, which sets by default a number of threads equal to the number of cores or 32 threads (whichever number is smaller). An issue, which can especially happen when measuring CPU-time, is due to the operating system and specific tooling. Namely, a program starts multiple processes, e.g., the program calls a SAT solver, but the monitoring tool captures only one process.

Tools to Measure Runtime. A standard system tool is `GNU time` [50], which provides CPU-time, real-time, and CPU usage of an executed program when run with the command-line flag *-v*. Note that time refers to a function in the Linux shell whereas `GNU time` can be found at `/usr/bin/time`. `GNU time` suffers from issues with time skew. A compact, free, and open source tool with extended functionality is `runsolver` [79]. It can be easily compiled and requires only few additional packages, but also suffers from issues with time skew. An extensive monitoring tool is `perf`, which is available in the linux kernel since version 2.6.31 (2009) [101]. Perf provides statistical profiling of the entire system when run with flag *stat*. It is easy to use and well documented, but requires installation of an additional package, an additional kernel module, and setting kernel security parameters (*perf_event_paranoid*, *nmi_watchdog*) [55, 61]. However, `perf` is usually available on maintained HPC environments.

Restricting Runtime. Oftentimes when running experiments, we are interested in setting an upper bound on the runtime, let the program run until this time, then terminate and measure how many inputs have been solved successfully. Classical tools to impose a timeout are `timeout` [11], `prlimit` [13], and `ulimit` (obsolete [88]). These tools use a kernel function (*timer_create*) to register a timer. The tools notify the considered program about the occurred timeout by sending a signal to terminate the program, but only to the started program that is responsible to handle potentially started children (entire process hierarchy). For that reason, these tools are often useless or require to build additional wrapper scripts when running academic code, which often omit proper signal handling. A popular tool in the research community that circumvents these problems is the already above mentioned tool `Runsolver` [79], which uses a sampling based approach. It monitors and terminates the entire hierarchy of processes started by the tested program. However, signals are sent to child processes first, which may need additional exception handling in the tested program. Furthermore, the sampling-based approach may cause measurable overhead in used resources. `runexec` is modern and thorough tool for imposing detailed runtime restrictions. It can be found within the larger framework for reliable benchmarking and resource measurement (`BenchExec`) [14]. `runexec` uses kernel control groups (cgroups) to limit resources [46, 36]. Cgroups are precise, but cause a certain overhead and are fairly quite hard to use manually. Unfortunately, `BenchExec` does not directly support commonly used schedulers in HPC environments (except AWS), requires administrative privileges during setup, specially configured privileges at runtime, and fairly new distributions and kernels. It is only widely available on Ubuntu or systems running kernels of version at least 5.11.

Suggested Tooling. In principle, we find `runexec` quite helpful when restricting runtime. It is reliable and has very helpful features such as warning the user about unexpected high system loads. However, it has strong requirements, both in terms of privileges and dependencies, and can be hard to setup, especially in combination with existing cluster scheduling systems. `GNU time` and `timeout` are both system tools available out of the box. Though, when using `timeout` we require additional tools (e.g., `pstree`) and a bit of scripting to handle an entire process hierarchy. Still, both tools might be the best choice if only standard system resources are available and no libraries can be installed. For older systems that are well-maintained or where additional libraries can be installed, we suggest `runsolver` (enforcement) in combination with `perf` (measurement). Both tools keep setup and handling at a minimum. Issues on potential time-skew and sampling-based issues are minimized and more detailed statistics (memory) can be outputted if needed. However, using this tooling requires to check carefully if the system is over-committed or if `runsolver` terminated a program too late. If required kernel modules or security parameters for `perf` cannot be installed/set, `runsolver` in combination with `GNU time` can be a reasonable alternative.

3.2 CPUs and Scaling

Modern hardware has features to dynamically overclock the CPU, which then can run at high frequency for a short period of time (*Turbo Boost*). Frequency scaling can save energy (*Thermal power design*) when processes do not require full capabilities of the system. These

Processor	f_a	f_e	p	$s(15)$	$t[h]$
Coppermine	0.5	0.5	1	98	8.93
Comet Lake	4.7	0.5	1	160	9.99
Comet Lake	4.7	*0.8	1	174	9.09
Comet Lake	4.7	1.5	1	177	7.12
Comet Lake	4.7	2.0	1	189	5.13
Haswell	3.3	*2.5	1	189	3.89
Skylake	3.0	†3.0	1	190	5.12
Comet Lake	4.7	3.9	1	191	3.81
Rome	3.4	2.0	1	190	3.79

Table 2 Number of solved SAT instances running the solver `CaDiCal` on varying platforms. Column $s(x)$ contains the solved instances when the runtime is cut off after x minutes. f_a, f_e, and p refer to the available and effective frequency of the CPU in GHz and number of solvers running in parallel, respectively. The $t[h]$ column contains the total runtime in hours for all instances solved within 15 minutes. We enforced limits using kernel governor parameters. Frequencies marked by $\star$ are CPU base-frequencies. † we could not enforce frequencies due to administrative restrictions. For Coppermine (PIII), we directly list the results by Fichte et al. [22].

features can significantly impact performance and uncertainty on modern hardware [34]. We provide a brief experiment in Section 3.3 to illustrate effects. Within the operating system, the concept is known as dynamic CPU frequency scaling or CPU throttling, which allows a processor to run at frequency that is not its maximum frequency to conserve power or to save the CPU from overheating if the frequency is beyond its thermally save base frequency. In fact, modern operating systems have options to manually set performance states. In Linux, the CPU frequency scaling (CPUFreq) subsystem is responsible for scaling. It consists of three layers, namely, the core, scaling governors, and scaling drivers [97]. Available capabilities to modify the CPU frequency depend on the available hardware and driver [97]. A *scaling governor* implements a scaling algorithm to estimate the required CPU capacity [12]. However, minimum and maximum frequency can also be fixed by modifying kernel values. Specifications of modern CPUs detail the safe operating temperature (*Thermal Velocity Boost Temperature*) that still allows to boost the cores to their maximum frequency.

Tools to Modify the CPU Frequency. The tool `cpupower` provides functions to gather information about the physical CPU and set the scaling frequency. The flag *frequency-info* lists supported limits, activated governor, and current frequency. The tool `turbostat` allows to obtain extended information about base frequency, the maximum frequency, and the maximum turbo frequency depending on how many cores are active. The program `frequency-set` allows to set the maximum and minimum scaling frequency using flags *-u* and *-d*, respectively. However, the values can also be manually read/set in the kernel by modifying a text file. The turbo needs to be manually modified depending on the driver [97]. The current frequency can be tested explicitly by running the command: `perf stat -e cycles -I 1000 cat /dev/urandom > /dev/null`.

Revisiting the Experiment. With the knowledge of frequency scaling at hand, we focus our attention to Table 2. There, we state runtime results and number of solved instances in dependence of platform and CPU frequency. More precisely, the maximum CPU frequency and the chosen frequency scaling. Obviously, the runtime and number of solved instances significantly depends on the frequency scaling of the CPU, which already explains why CPUs that permit a higher frequency show less solved instances. From the number of solved instances for Comet Lake (i7 Gen10) CPU and Coppermine (PIII) CPU, we can also see that an increase in CPU frequency alone is clearly not the reason for modern solvers running faster on modern hardware than on old hardware.

Suggested Setup. When handling thermal management for experiments, one usually balances between three objectives (i) stability and repeatability of the experiment; (iia) maximum speed vs (iib) throughput; and (iii) low effort or no access to thermal management functions of the operating system while aiming to balance (i) and (ii). If we focus our setup on Objective (i), a conservative choice is to set the CPU frequency to its base frequency and limit the parallel processes according to available NUMA regions. Then, the thermal management has limited effects on an experiment. Running the same experiment another system, where the CPU frequency was fixed to the same value and where the memory layout is comparable, shows similar results for CPU-intensive solvers. Such an approach could simplify certain aspects of repeatability. However, then the number of solved instances is lower than the actual capabilities of the hardware, the experiment takes longer, and fewer instances are solved. If we balance towards Objective (iia) obtaining maximum speed of the individual solvers, we ignore thermal management, run at maximum speed, and execute all runs sequentially. However, then throughput is low, only a low number of instances are solved, and vasts of resources on typical server CPUs are wasted. If we balance towards Objective (iib) obtaining maximum throughput during the experiment, we run a number of solvers in parallel for which there is low effect on the turbo frequency. We can obtain the value by the tool `turbostat`. For example, a turbo frequency of 3.9GHz might be acceptable over 4.7GHz if 4 additional solvers can be run in parallel. In fact, one could also simply try to repeat the experiments often to avoid balancing between Objective (iia) and (iib), which would however often require plenty of resources. If we are in Situation (i) with no access to modify the CPU thermal management capabilities or we just want to keep tuning efforts low while still having a reasonable throughput at low solving time, we can just test a reasonable setup. We lookup the thermal velocity boost (TVB) temperature, e.g., [45]. Then, we execute a run with parallel solvers and sample CPU temperature. After evaluating several parallel runs, we favor a configuration where the median temperature is below the TVB temperature and the maximum temperature rarely exceeds TVB temperature.

3.3 CPUs and Parallel Execution

In the 2000s, the end of Moore's law [69] seemed near as CPU frequency improvements for silicon-based chips started to slow down [9, 78]. Parallel computation started to compensate for this trend and multi-core hardware found its way into consumer computers around 2004. In 2021, parallel hardware is widespread, for example, standard desktop hardware regularly has 8 cores (Intel i9 or Apple M1) or 12 cores (AMD Ryzen) and server systems go up to 64 cores (AMD Rome) or even 128 cores (Ampere Altra) per CPU where multiple sockets are possible. Still, parallel solving is rare in combinatorial communities such as SAT solving [35, 62] or beyond [23]. So a common question that arises in empirical problem solving is whether one can execute sequential solvers meaningful in parallel and speed-up the solution of the overall set of considered instances for an empirical experiment. While it clearly makes sense to carry out an experiment in parallel, one needs some background understanding on the hardware architecture of multi-core systems and on how to gather information about the actual system on which experiments are run. Modern systems with multiple processors on multiple sockets and processors that have multiple cores use a special memory design, namely *Non-uniform memory access (NUMA)*. There, access time to RAM depends on the memory location relative to the physical core. Each processor is directly connect to separate memory; access to "remote" RAM is still possible, but the requests are much slower since they pass through the CPU that controls the local RAM. If the operating system supports NUMA and the user is aware of the NUMA layout of the used system, the hardware architecture can help to eliminate performance degeneration that can occur due to allocation of RAM that is associated with another socket [37, 57]. The effect can be measurable, if consecutive pages are used by exactly one process as done in combinatorial solving. NUMA hardware layout also effects the cache hierarchy (L1, L2, often L3) and address translation buffers (TLB). Recall that caches can have a measurable effect on effectiveness of combinatorial solvers [24].

Evidently, if running an experiment a modern operating system does not solely execute the program under test. It runs function of the operating system itself, events from the hardware such as input from disk, network, user-interfaces or output to graphics devices. Further, programs or functions to control or monitor the program under test are running. These functions might interrupt the execution of the program under test and are often triggered by a mechanism called interrupt. In system programming an interrupt service routine (ISR) handles a specific interrupt condition and is often associated with system drivers or system calls. A common urban legend among students in the combinatorial solving community is that interrupt handling happens on CPU Core 0 (monarch core) and hence no solver should be scheduled on Core 0. However, this is only true when booting the system when firmware hands over control to the operating system kernel. Then, only one core is running, which usually is Core 0, takes on all ISR handling, initializes the system and starts all other cores. In old operating systems load was not distributed to other cores by default and hence the core that started the system would handle all ISRs. However, since version 2.4 Linux supports a concept called SMP affinity, which allows to distribute interrupt handling [67]. The actual balancing and distribution of hardware interrupts over multiple cores is then done by a system process, namely irqbalance [41]. Depending on the Linux distribution the balancing is done one-shot at system start, during runtime, or entirely omitted. Nonetheless, it might be helpful to understand the configured system behavior [76].

Tooling for Information on the CPU. Often, we need information on the CPU as starting point for setting up parallel execution of an experiment. Linux reports information on the CPU in the proc filesystem as text (`/proc/cpuinfo`) [10]. Among the information is data about the CPU model, microcode, available cores, and instruction sets. The tool `lscpu`, which is part of util-linux in most distributions, reports more details on the CPU such as architecture, cache sizes, number of sockets, number of virtual or physical cores, number of threads per core, details on NUMA regions, and active flags. More detailed information on NUMA regions can be obtained by running the tool `numactl` with flag *–hardware*, using `lscpu`, or by manually listing details in the cpulist. Note that NUMA regions and core numbering can be a bit tricky as cores and NUMA regions are often not in consecutive order.

Restricting NUMA, CPU, and IRQ affinity. When running a program on a multicore system, the scheduler in the operating system decides on which core the program runs. In principle, this depends on the current load and on a memory placement policy of the system. Some enterprise distributions have automated processes running (`numad`), which automatically estimate or balance NUMA affinity. Primary benefits are reported for long-running processes with high resource load, but degeneration for continuous unpredictable memory access patterns. The core and allowed memory regions can also be manually restricted. The tool `numactl` provides functionalities to force the execution of a program to certain NUMA nodes or cores, including strict settings [51]. The tool `runsolver`, which we already mentioned above, allows for setting the NUMA and CPU affinity. On modern distributions, these settings can also be set when running a program by `systemd`. Literature on manually tuning NUMA regions and CPU affinity reports both positive and negative effects, but less than 5% performance gain on full core CPU loads [38, 43]. Hence, detailed manual tuning might have a far less effect than what is usually anticipated within the community. Since combinatorial solvers often rely on fast access to caches, it might be more important to ensure that caches are accidentally shared between several running solvers. In principle,

p	$t_r[h]$	f_o[GHz]	f_{std}	θ_o[°C]	$\theta_{\max}$	s(1)	s(5)	s(10)	s(15)	s(25)	$t_s[h]$	s5k
1	7.37	3.90	0.26	53.4	64.0	132	179	190	191	193	3.82	161
2	4.06	3.69	0.29	60.8	72.5	125	179	189	191	193	4.27	158
4	2.49	3.30	0.28	74.2	92.0	120	175	183	190	192	5.01	150
6	1.85	2.95	0.30	76.6	94.5	111	171	181	189	191	6.03	142
8	1.77	2.81	0.46	74.5	94.0	98	160	176	183	190	6.05	131
10	1.77	2.71	0.57	74.0	92.0	88	155	174	181	189	6.78	123
12	1.59	2.59	0.51	87.0	72.5	80	147	170	176	187	10.82	117
14	1.47	2.51	0.28	91.5	72.5	88	155	174	181	189	11.17	111

Table 3 Overview on frequency scaling, thermal observations, and the number of solved instances (out of 200) on an Intel Comet Lake (i7 Gen10) processor for different number of parallel runs of the solver `CaDiCal`. The column "p" refers to an upper bound on the number of instances that are solved in parallel and "$t_r[h]$" refers to the total runtime of the experiment in hours. While the maximum CPU frequency is 4.7 GHz, the column "f_o" states the observed frequency in GHz and f_{std} to its standard deviation. Column "θ_o" lists the observed CPU temperature; $\theta_{\max}$ to the maximum temperature in °C. The column "$s(x)$" contains the number of solved instances when the runtime is cut off after x minutes. The column "t_s" refers to the total runtime (real-time) of the solved instances in hours at maximum runtime of 1500s for each instance. Finally, "s5k" how many instances can be solved in 5000s if instances are ordered by hardness and each run has at most 1500s. We used a simple python wrapper to start the parallel runs.

the IRQ affinity can be managed manually by setting dedicated flags for the system service `irqbalance`. However, time might be better spent on avoiding over-committing CPUs.

Suggested Tooling and Setup. Experiments that involve measuring runtime need exclusive access to the machine on which experiments are run, i.e., no other software interferes in the background (e.g., running a system update, database, file server, browser, GUI with visual effects) and no other users access the system in the meantime. If the hardware is used for other purposes, runtime differences of 30% and more are common. If an experiment runs on an HPC environment, a uniform configuration is indispensable, i.e., all nodes have the same CPU, microcode, and memory layout. The number of scheduled solver resources should never equal the number of cores on the system, since almost all combinatorial solvers use CPU(s) at full load and operating system and measurement tools require a certain overhead. If NUMA layout details are missing, one can take a rough estimate. Assume that controlling and monitoring software as well as the operating system need one core per tested program, add the expected number of occupied cores of the tested solver, and for a safe buffer multiply the result by two. However, a better approach is to gather detailed information and test whether an anticipated setup is stable. Information on the available CPUs and NUMA regions can be obtained by using the tools `lscpu` and `numactl`. Modern operating systems implement NUMA scheduling already well. However, it is still important to report details of the system within logs of the experiments. If manual NUMA region enforcement is needed, each running solver should only access the NUMA region on which it is pinned to [77]. Solvers requiring fast caches should not be scheduled in parallel on cores sharing L1 and L2 cache.

Effects of Parallel Runs, CPU Scaling, and Timeouts in Practice

In the previous section, we listed complications that may occur from technical specifications of modern processors and techniques present in modern operating systems. Next, we present a detailed experiment on parallel execution of solvers incorporating effects of actual processor frequency, stability of parallel runs, thermal issues, in combination with runtime and number

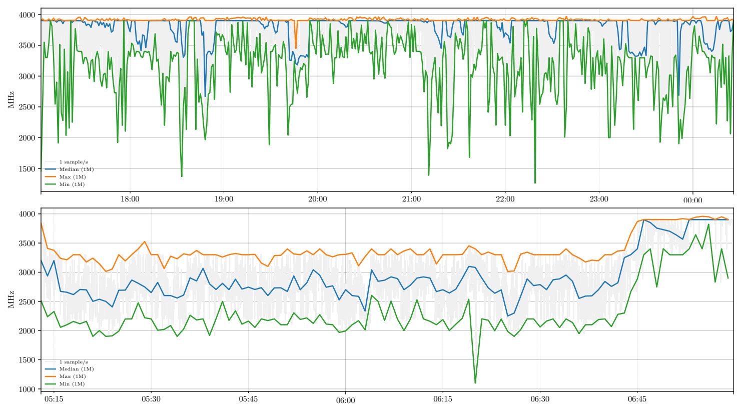

Figure 1 Illustration of the CPU frequency scaling when running the sequential solver `CaDiCal` on the considered instance set by solving in parallel 1 instance (upper) and 8 instances (lower).

of solved instances. We specify the setup, used measures, and common expectations of which some might be contradictory. In order to obtain a better view on effects of timeouts, we increase the maximum runtime per instance to 1500 seconds.

▶ **Experiment 1** (Parallel Runs). *We investigate complications of solving multiple instances in parallel with one sequential SAT solver on a fixed hardware.*

- *Setup: solve 200 instances by one SAT solver (`CaDiCal`) on Comet Lake (i7 Gen10), maximum runtime per instance (timeout) 1500 seconds.*
- *Measures: Runtime (real-time) [h], number of solved instances, temperature (median of sampling each 1s the average temperature over all cores) [°C], and CPU frequency (median of sampling each 1s the average over all cores) [GHz].*
- *Expectation 1a: Solving should never be executed in parallel on one machine as the runtime and number of solved instances significantly differ otherwise.*
- *Expectation 1b: Full parallel capabilities should be employed as long as runtime and number of solved instances remains similar.*
- *Expectation 2: Relying on multithreading degrades runtime.*
- *Expectation 3: Measures are stable over small runtime changes.*

Observations: Results of the first experiment are illustrated in Table 3. The number of solved instances for 1, 5, 10, 15, and 25 minutes provide an overview on how many instances can be solved quickly. Unsurprisingly, the total runtime of an experiment depends on the number of parallel processes running. More precisely, the total runtime of the experiment varies between 7.37 hours and 1.77 hours when running 1 or 10 instances in parallel. Just by running 4 instances in parallel instead of 1 we cut runtime down to 33% of the original runtime and still to 55% for 2 instances. However, the total real-time of the solved instances varies between 3.82 hours and 6.78 hours (44%). The number of solved instances varies by 2% at 25 minutes and 5% at 15 minutes, 13% at 5 minutes, and 33% at 1 minute timeout. When comparing the effect on the measure how many instances can be solved within 5000s, we obtain a notable 24% decrease. Surprisingly, the median CPU frequency never reached 4.7GHz even when running only one instance. The actual frequency reduced significantly when more instances are running. Figure 1 illustrates the changes of the CPU frequency over time for 1 and 8 instances solved in parallel. We see that the frequency is hardly consistent and increases significantly as soon as most instances are finished and less processes run in parallel. When using multiple cores, the median CPU temperature increases significantly and may even spike (94°C) close to the maximum operating temperature of the CPU (100°C).

Interpretation. On the considered set of instances, the number of solved instances and real-time over all solved instances decreases with an increasing number of instances run in parallel. The effect is particularly high, if the timeout was set very low or if the measure is number of instances solved within 5000s. This is not entirely surprising, since instances in the considered set were selected by Hoos et al. [40] using a distribution of instance hardness leading to many instances of medium hardness and a few easy and hard instances. Then, if the considered timeout is low, a small constant improvement by hardware effects can increase the number of solved instances notably. In contrast, there is only a 2% difference between number of solved instances when timeouts are higher. The measure of solved instances within 5000s is particularly runtime dependent and hence configuration of the experimental setup has notable effects. Regarding runtime, we can see that the real-time over all solved instances almost doubles when running almost as many instances as cores are available. However, the entire experiment finishes significantly faster, i.e., about 24% of the original runtime. Surprisingly, the CPU frequency was far below the potential 4.7GHz. If we check more details on the specification of the Comet Lake (i7 Gen10) CPU or by running the tool `turbostat`, we observe that the maximum frequency of the CPU is only 3.9GHz if 6 cores are active, i.e., not explicitly suspended. While our considered system has 12 MT cores, it has only 6 physical cores. Hence, we observe a measurable degeneration in number of solved instances when running more instances in parallel than present physical cores are present. When considering runtime, we observe a considerable increase when more than 2 instances run in parallel, as CPU frequency measurably drops and temperature increases significantly.

Outcome: After summarizing observations and interpretation of our experiment, we briefly evaluate phrased expectations from above. In theory, we would expect that Expectation 1a is true for real-time and number of solved instances within 5000s, which is also quite sensitive for runtime influences. Indeed, there is a measurable influence in runtime, but only slightly decrease in number of solved instances, while the experiment finishes much faster. If we take higher timeout, the number of parallel executions affects the runtime only if already known rough estimates are exceeded. Still, the number of parallel executions is influenced by throttling of the processor. Expectation 1b clearly does not hold. All measures are influenced by a higher system load and hence by solving several instances in parallel. While we can confirm Expectation 3 in the experiment, multithreading is not the only reason. Clearly, already when using all available cores runtime and number of solved degenerate. Unfortunately, our experiment does not fulfill Expectation 3. All considered measures are influenced by parallel execution. Especially, limiting the total solving time is prone to hardware effects and might accidentally over-highlight constant runtime improvements. Since the frequency is also not stable when running only one instance, fixing the frequency might be a reasonable approach during experimenting. However, if the base-frequency is exceeded, a stable frequency should be estimated and experimentally verified before comparing runtime and number of solved instances with multiple solvers. In our case, operating the CPU at fixed 3GHz showed stable frequency results when running 1–2 instances in parallel. Under the light of the mentioned complications, we fear that a single measure incorporating runtime, number of solved instances, and a cutoff time is problematic if setup is neglected.

3.4 Input/Output

Input and output performance, *I/O* for short, talks about read or write operations involving a storage device. On a desktop computer storage is usually restricted to local disks. On cluster environments, nodes have access to a central storage over network, fast temporary storage (over network), and local disks. Here, a variety of different topics are involved, for example, hardware (storage arrays/network), network protocols, and file systems, which can make it inherently complicated. Therefore, we provide only a brief and simple suggestion: keep external influence as low a possible. When reading input and writing output, use a shared memory file system (shm) to avoid external overhead. Before starting the solver under test, input files are copied in-memory. Then, measuring runtime starts when executing the solver, which takes as input the temporary files on the memory and outputs only to a shared memory file system. The measurement ends when the solver is terminated and afterwards temporary files are copied to the permanent storage and deleted from the temporary storage. This approach minimizes side effects from slow network devices and avoids side effects that may occur with large files and system file caches, especially when running multiple solvers on the same input. However, if files are too large or solvers need the entire RAM, temporary in-memory cannot be used and fast local disks (e.g., NVMe) can provide an alternative.

4 Conclusion

Empirical evaluations are essential to confirm observations in algorithmics and combinatorics beyond theory. Many evaluations typically focus on comparing runtimes and number of solved instances, since both measures are easy targets for comparison and probably roughly reflect needs of end users. However, the number of solved instances is sensitive to the chosen benchmark, so one has to be cautious about it. Playing devils advocate, we can even ask to what extent runtime is even a meaningful measure on modern hardware. If one solver is a factor of ten faster than another, we are fairly confident in it, but does modern hardware allow for accurate comparisons at a range of, say, 10%, which might be the contribution of an individual feature or optimization towards the hardware? Similar to experimental physics, we can simply repeat an experiment often or repeat in different environments. However, in combinatorial solving this is not always possible if many solvers need to be tested or a reasonably high number of hard instances have to be considered. Hence, we believe that an experimental setup should still be carried out thoroughly. Future work could consider up to what extent certain aspects can be neglected and how repetition can circumvent minor issues. In fact, our work only explains and illustrates certain complications from modern hardware to make researchers aware of potential issues. In a way, we also show that complications do not just concern CPU frequency, but also the experimental setup (timeouts, cutoffs, parallel running processes). Clearly, there is no reason to forbid the use of certain platforms, if we are aware of complications. On the meta level, we believe that clearly marking strengths and weaknesses of solvers provides more insights than finding scenarios where one solver is best.

An interesting question for future research is the boarder topic of SIMD and branch prediction, which could affect repeatability, replicability, and reproducibility. Both features are quite relevant for how a good solver author can write code, but it is unclear whether they can even change the overall results when comparing two solvers. In practice, one could maybe investigate issues by taking different versions of a CPU (or different firmware).

Further, we think that papers presenting experimental evaluations could provide a simple benchmark protocol as appendix, similar to literature as part of reproducibility work. Best practices and checklists could be developed in a community effort after thorough discussions and more detailed works. This can also include detailed guides or suggested configurations for standard cluster schedulers such as Slurm [99]. Having a list of common parameters to report or even practical tools could prevent manual repetitive labor. Thereby, we leave room for actual scientific questions, e.g., why implementations are efficient for certain domains [91, 29].

Finally, our experiments focused on consumer hardware, detailed investigations with server hardware are interesting for future investigations to confine limits of parallel execution.

References

1 Altuna Akalin. Scientific data analysis pipelines and reproducibility. `https://aakalin.medium.com/`, 2018. Retrieved 2021-05-29.

2 Chris Angelini. Intel's second-gen core CPUs: The Sandy bridge review. `https://www.tomshardware.com/reviews/sandy-bridge-core-i7-2600k-core-i5-2500k,2833-3.html`, 2011.

3 Gilles Audemard, Loïc Paulevé, and Laurent Simon. SAT heritage: A community-driven effort for archiving, building and running more than thousand sat solvers. In Luca Pulina and Martina Seidl, editors, *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT'20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 107–113, Alghero, Italy, July 2020. Springer Verlag.

4 David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994. `doi:10.1145/197405.197406`.

5 Daniel Le Berre, Matti Järvisalo, Armin Biere, and Kuldeep S. Meel. The SAT practitioner's manifesto v1.0. `https://github.com/danielleberre/satpractitionermanisfesto`, September 2020.

6 Donald Berry. Avoiding clock drift on VMs. `https://www.redhat.com/en/blog/avoiding-clock-drift-vms`, 2017.

7 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In Bernd Fischer and Jaco Geldenhuys, editors, *Proceedings of the 22nd International Symposium on Model Checking of Software (SPIN'15)*, volume 9232 of *Lecture Notes in Computer Science*, pages 160–178. Springer Verlag, 2015. `doi:10.1007/978-3-319-23404-5_12`.

8 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. `doi:10.1007/s10009-017-0469-y`.

9 Steve Blank. What the GlobalFoundries' retreat really means. *IEEE Spectrum*, 2018. URL: `https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means`.

10 Terrehon Bowden, Bodo Bauer, Jorge Nerin, Shen Feng, and Stefani Seibold. The /proc filesystem. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`, 2009.

11 Padraig Brady. Timeout(1). `https://www.gnu.org/software/coreutils/timeout`, 2019.

12 Dominik Brodowski, Nico Golde, Rafael J. Wysocki, and Viresh Kumar. CPU frequency and voltage scaling code in the Linux(tm) kernel. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`, 2016.

13 Davidlohr Bueso. Util-linux 2.37. `https://www.kernel.org/pub/linux/utils/util-linux/`, 2021.

14 Alejandro Colomar et al. Ulimit(3). `https://www.kernel.org/doc/man-pages/`, 2017.

15 Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Identifiers for digital objects: the case of software source code preservation. In *iPRES 2018 - 15th International Conference on Digital Preservation*, 2018. `doi:10.17605/OSF.IO/KDE56`.

16 Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, Kyoto, Japan, 2017. URL: `https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdfhttps://hal.archives-ouvertes.fr/hal-01590958`.

17 Roberto Di Cosmo, Stefano Zacchiroli, Gérard Berry, Jean-François Abramatic, Julia Lawall, and Serge Abiteboul. Software heritage. `https://www.softwareheritage.org/mission/heritage/`, 2020.

18 Chris Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, Montreal, Canada, 2009.

19 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1291–1299. International Joint Conferences on Artificial Intelligence Organization, 7 2018. `doi:10.24963/ijcai.2018/180`.

20 Johannes K. Fichte and Markus Hecher. The model counting competition 2021. `https://mccompetition.org/past_iterations`, 2021.

21 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *ACM Journal of Experimental Algorithmics*, 2021. In press.

22 Johannes K. Fichte, Markus Hecher, and Stefan Szeider. A time leap challenge for SAT-solving. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, pages 267–285, Louvain-la-Neuve, Belgium, September 2020. Springer Verlag. `doi:10.1007/978-3-030-58475-7_16`.

23 Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted model counting on the GPU by exploiting small treewidth. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16. Dagstuhl Publishing, 2018. `doi:10.4230/LIPIcs.ESA.2018.28`.

24 Johannes K. Fichte, Norbert Manthey, André Schidler, and Julian Stecklina. Towards faster reasoners by using transparent huge pages. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 304–322, Louvain-la-Neuve, Belgium, September 2020. Springer Verlag. `doi:10.1007/978-3-030-58475-7_18`.

25 Ronald Fisher. *Arrangement of Field Experiments.* 1926.

26 Ronald Fisher. *The Design of Experiments.* 1935.

27 Vijay Ganesh and Moshe Y. Vardi. On the unreasonable effectiveness of sat solvers. 2021.

28 Ian P. Gent. The recomputation manifesto. *CoRR*, abs/1304.3674, 2013. URL: `http://arxiv.org/abs/1304.3674`, `arXiv:1304.3674`.

29 Ian P Gent and Toby Walsh. The SAT phase transition. In Anthony G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'94)*, volume 94, pages 105–109, Amsterdam, The Netherlands, 1994. PITMAN.

30 Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-boolean solving. In Sarit Kraus, editor, *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, pages 1711–1718. International Joint Conferences on Artificial Intelligence Organization, 7 2019. `doi:10.24963/ijcai.2019/237`.

31 Odd Erik Gundersen. The reproducibility crisis is real. *AI Magazine*, 2020. `doi:10.1609/aimag.v41i3.5318`.

32 Odd Erik Gundersen. AAAI'21 reproducibility checklist. `https://aaai.org/Conferences/AAAI-21/reproducibility-checklist/`, 2021.

33 Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 2018.

34 Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the Intel Haswell processor. In Jean-Francois Lalande and Teng Moh, editors, *Proceedings of the 17th International Conference on High Performance Computing & Simulation (HPCS'19)*, 2019.

35 Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, pages 2120–2125, Toronto, Ontario, Canada, 2012. The AAAI Press.

36 Tejun Heo. Control group v2. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, 2015.

37 Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. The effect of numa tunings on cpu performance. *Journal of Physics: Conference Series*, 2015.

38 Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. The effect of NUMA tunings on CPU performance. *Journal of Physics: Conference Series*, 664(9):092010, dec 2015. `doi:10.1088/1742-6596/664/9/092010`.

39 John N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:33–42, 1995. `doi:10.1007/BF02430364`.

40 Holger H. Hoos, Benjamin Kaufmann, Torsten Schaub, and Marius Schneider. Robust benchmark set selection for Boolean constraint solvers. In *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION'13)*, volume 7997 of *Lecture Notes in Computer Science*, pages 138–152, Catania, Italy, January 2013. Springer Verlag. Revised Selected Papers.

41 Neil Horman, PJ Waskiewicz, and Anton Arapov. Irqbalance. `http://irqbalance.github.io/irqbalance/`, 2020.

42 Sascha Hunold, Alexandra Carpen-Amarie, and Jesper Larsson Träff. Reproducible MPI micro-benchmarking isn't as easy as you think. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 69–76, New York, NY, USA, 2014. Association for Computing Machinery, New York. `doi:10.1145/2642769.2642785`.

43 Satoshi Imamura, Keitaro Oka, Yuichiro Yasui, Yuichi Inadomi, Katsuki Fujisawa, Toshio Endo, Koji Ueno, Keiichiro Fukazawa, Nozomi Hata, Yuta Kakibuka, Koji Inoue, and Takatsugu Ono. Evaluating the impacts of code-level performance tunings on power efficiency. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 362–369, 2016. `doi:10.1109/BigData.2016.7840624`.

44 VmWare Inc. Timekeeping in VMware virtual machines. `http://www.vmware.com/pdf/vmware_timekeeping.pdf`, 2008.

45 Intel. Intel product specifications: Processors. `https://ark.intel.com/content/www/us/en/ark.html#@Processors`, 2021.

46 Paul Jackson and Christoph Lameter. cgroups - Linux control groups. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, 2006.

47 Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazin*, 33(1), 2012. URL: `http://www.aaai.org/ojs/index.php/aimagazine/article/view/2395`.

48 D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA'01)*, pages 197–206, 2001.

49 Armin Biere Katalin Fazekas, Daniela Kaufmann. Ranking robustness under sub-sampling for the SAT competition 2018. In Matti Järvisalo and Daniel Le Berre, editors, *Proceedings of the 10th Workshop on Pragmatics of SAT (POS'19)*, 2019.

50 Michael Kerrisk. *The Linux Programming Interface.* No Starch Press, 2010. URL: `https://man7.org/tlpi/`.

51 Andi Kleen, Cliff Wickman, Christoph Lameter, and Lee Schermerhorn. numactl. `https://github.com/numactl/numactl`, 2014.

52 Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley, 2015.

53 Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

54 Michael Kohlhase. The theorem prover museum – conserving the system heritage of automated reasoning. *CoRR*, abs/1904.10414, 2019. `https://theoremprover-museum.github.io/`. `arXiv:1904.10414`.

55 Divya Kiran Kumar. Installing and using Perf in Ubuntu and CentOS. `https://www.fosslinux.com/7069/installing-and-using-perf-in-ubuntu-and-centos.htm`, 2019.

56 G. M. Kurtzer, V. Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PLoS ONE*, 12, 2017. `doi:10.1371/journal.pone.0177459`.

57 Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.

58 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

59 Zhiyuan Liu and Jian Tang. IJCAI 2021 reproducibility guidelines. `http://ijcai-21.org/wp-content/uploads/2020/12/20201226-IJCAI-Reproducibility.pdf`, 2020.

60 Eric Ma. How to statically link C and C++ programs on Linux with gcc. `https://www.systutorials.com/how-to-statically-link-c-and-c-programs-on-linux-with-gcc/`, August 2020. Accessed 20-April-2021.

61 Kamil Maciorowski and meuh. Run perf without root-rights. `https://superuser.com/questions/980632/run-perf-without-root-rights`, 2015.

62 Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel sat solving. *Constraints*, 17(3):304–347, 2012. `doi:10.1007/s10601-012-9121-3`.

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

64 Doug McIlroy. Unix time-sharing system: Foreword. *The Bell System Technical Journal*, pages 1902–1903, 1978.

65 Marcin Miłkowski, Witold M. Hensel, and Mateusz Hohol. Replicability or reproducibility? on the replication crisis in computational neuroscience and sharing only relevant detail. *Journal of Computational Neuroscience*, 2018.

66 Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, 2009. `doi:10.1109/PACT.2009.22`.

67 Ingo Molnar and Max Krasnyansky. Smp irq affinity. `https://www.kernel.org/doc/Documentation/IRQ-affinity.txt`, 2012.

68 Matteo Monchiero and Gianluca Palermo. The combined perceptron branch predictor. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 487–496, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

69 Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 04 1965.

70 Hassan Mujtaba. Intel Xeon E7 'Ivy Bridge-EX' lineup detailed – Xeon E7-8890 V2 'Ivy Town' chip with 15 cores and 37.5 mb LLC. `Wccftech.com`, February 2014.

71 Matthias Müller-Hannemann and Stefan Schirra, editors. *Algorithm Engineering*, volume 5971 of *Theoretical Computer Science and General Issues*. Springer Verlag, 2010. Bridging the Gap Between Algorithm Theory and Practice. `doi:10.1007/978-3-642-14866-8`.

72 Steven J. Murdoch. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 27–36, New York, NY, USA, 2006. Association for Computing Machinery. `doi:10.1145/1180405.1180410`.

73 Gonzalo Navarro. RCR: Replicated computational results initiative. `https://dl.acm.org/journal/jea/rcr-initiative`, 2021.

74 Naomi Oreskes. *Why Trust Science?* The University Center for Human Values Series, 2021.

75 Jeffrey M. Perkel. Challenge to scientists: does your ten-year-old code still run? *Nature*, 584:656–658, 2020. `doi:10.1038/d41586-020-02462-7`.

76 Red Had Developers. Optimizing Red Hat Enterprise Linux performance by tuning IRQ affinity. https://access.redhat.com/articles/216733, 2011.

77 Benoit Rostykus and Gabriel Hartmann. Predictive CPU isolation of containers at Netflix. https://netflixtechblog.com/predictive-cpu-isolation-of-containers-at-netflix-91f014d856c7, 2019. accessed 20-May-2021.

78 David Rotman. We're not prepared for the end of moore's law. *MIT Technology Review*, 2020.

79 Olivier Roussel. Controlling a solver execution with the runsolver tool. *J. on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

80 Olivier Roussel and Christophe Lecoutre. XCSP3 competition 2019. http://xcsp.org/competition, July 2019. A joint event with the 25th International Conference on Principles and Practice of Constraint Programming (CP'19).

81 Marta Serra-Garcia and Uri Gneezy. Nonreplicable publications are cited more than replicable ones. *Science Advances*, 7(21), 2021. doi:10.1126/sciadv.abd1705.

82 Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2002 competition (preliminary draft). http://www.satcompetition.org/2002/onlinereport.pdf, 2002.

83 Helmut Simonis, George Katsirelos, Matt Streeter, and Emmanuel Hebrard. CSP 2009 competition (CSC'2009). http://www.cril.univ-artois.fr/CSC09/, July 2009.

84 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR'14)*, volume 8562 of *Lecture Notes in Computer Science*, pages 367–373, Vienna, Austria, July 2014. Springer Verlag. Held as Part of the Vienna Summer of Logic, VSL 2014. doi:10.1007/978-3-319-08587-6_28.

85 SUSE Support. Clock drifts in KVM virtual machines. https://www.suse.com/support/kb/doc/?id=000017652, 2020.

86 Guido Tack and Peter J. Stuckey. The MiniZinc challenge 2019. https://www.minizinc.org/challenge.html, July 2019. A joint event with the 25th International Conference on Principles and Practice of Constraint Programming (CP'19).

87 Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization.* Prentice Hall PTR, USA, 4th edition, 1998.

88 The Open Group. The open group base specifications issue 7, 2018 edition ieee std 1003.1-2017 (revision of ieee std 1003.1-2008). https://pubs.opengroup.org/onlinepubs/9699919799/functions/ulimit.html, 2018.

89 Linus Torvalds. LKML archive on lore.kernel.org: Very slow clang kernel config. https://lore.kernel.org/lkml/CAHk-=whs8QZf3YnifdLv57+FhBi5_WeNTG1B-suOES=RcUSmQg@mail.gmail.com/, May 2021.

90 A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi:10.1112/plms/s2-42.1.230.

91 Moshe Y. Vardi. Boolean satisfiability: Theory and engineering. *Communications of the ACM*, 57(3):5, March 2014. doi:10.1145/2578043.

92 Felix von Leitner. Source code optimization. Linux Congress 2009. http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf, 2009.

93 J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. doi:10.1109/85.238389.

94 Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. Optil.io: Cloud based platform for solving optimization problems using crowdsourcing approach. In Eric Gilbert and Karrie Karahalios, editors, *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, CSCW '16 Companion, pages 433–436, New York, NY, USA, 2016. Association for Computing Machinery, New York. doi:10.1145/2818052.2869098.

95 Ulrich Windl, David Dalton, Marc Martinec, and Dale R. Worley. The NTP FAQ and HOWTO: Understanding and using the network time protocol. `http://www.ntp.org/ntpfaq/NTP-s-sw-clocks-quality.htm#AEN1220`, 2006.

96 Ricardo Wurmus, Bora Uyar, Brendan Osberg, Vedran Franke, Alexander Gosdschan, Katarzyna Wreczycka, Jonathan Ronen, and Altuna Akalin. PiGx: reproducible genomics analysis pipelines with GNU Guix. *GigaScience*, 7(12), 10 2018. giy123. `doi:10.1093/gigascience/giy123`.

97 Rafael J. Wysocki. intel_pstate CPU performance scaling driver. `https://www.kernel.org/doc/html/v5.12/admin-guide/pm/intel_pstate.html`, 2017.

98 Rengan Xu, Frank Han, and Nishanth Dandapanthula. Containerizing HPC applications with singularity. `https://downloads.dell.com/manuals/all-products/esuprt_solutions_int/esuprt_solutions_int_solutions_resources/high-computing-solution-resources_white-papers10_en-us.pdf`, 2017.

99 Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple Linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'03)*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, Seattle, WA, USA, 2003. Springer Verlag.

100 Zhenyun Zhuang, Cuong Tran, and Jerry Weng. Don't let Linux control groups run uncontrolled: Addressing memory-related performance pitfalls of cgroups. `https://engineering.linkedin.com/blog/2016/08/don_t-let-linux-control-groups-uncontrolled`, 2016. Accessed Apr-28-2020.

101 Peter Zijlstra, Ingo Molnar, and Arnaldo Carvalho de Melo. Performance events subsystem. `https://github.com/torvalds/linux/tree/master/tools/perf`, 2009.