# Towards Faster Reasoners by using Transparent Huge Pages*

Johannes K. Fichte[1][0000−0002−8681−7470], Norbert Manthey[2], Julian Stecklina[2], and André Schidler[3]

[1] School of Engineering Sciences, TU Dresden, Dresden, Germany
johannes.fichte@tu-dresden.de
[2] {norbertmanthey@googlemail.com,js@alien8.de}
[3] Algorithms and Complexity Group, TU Wien, Vienna, Austria
aschidler@ac.tuwien.ac.at

**Abstract.** Various state-of-the-art automated reasoning (AR) tools are widely used as backend tools in research of knowledge representation and reasoning as well as in industrial applications. In testing and verification, those tools often run continuously or nightly. In this work, we present an approach to reduce the runtime of AR tools by 10% on average and up to 20% for long running tasks. Our improvement addresses the high memory usage that comes with the data structures used in AR tools, which are based on conflict driven no-good learning. We establish a general way to enable faster memory access by using the memory cache line of modern hardware more effectively. Therefore, we extend the standard C library (glibc) by dynamically allowing to use a memory management feature called huge pages. Huge pages allow to reduce the overhead that is required to translate memory addresses between the virtual memory of the operating system and the physical memory of the hardware. In that way, we can reduce runtime, which in turn decreases costs of running AR tools and applications with similar memory access patterns by linking the tool against this new glibc library when compiling it. In every day industrial applications, runtime savings allow to include more detailed verification tasks, getting better results of any-time optimization algorithms with a bound execution time, and save energy during nightly software builds. To back up the claimed speed-up, we present experimental results for tools that are commonly used in the AR community, including the domains ASP, hardware and software BMC, MaxSAT, and SAT.

## 1 Introduction

Automated reasoning (AR) tools such as solvers for Answer Set Programming (ASP), Bounded Model Checking (BMC), and Boolean Satisfiability (SAT) are

* The work has been supported by the Austrian Science Fund (FWF), Grants P32441 and W1255. We would like to thank the anonymous reviewers for very detailed feedback and their suggestions. Special thanks go to the reviewer who provided comments on CP solvers and propagators. The implementation is available at github: daajoe/thp_docker_build.

widely used to test and verify software and hardware in industry [14, 18, 21]. When improving AR tools, which usually run in industry each day for a few hours, one is also interested in speeding up existing implementations. A classical way to improve such tools is to consider their memory dependency when designing algorithms and data structures. While algorithm engineering often focuses on the source code of each of the solvers, we believe that an entirely overlooked direction is to take the interplay between hardware and software into account and speed up solving by modifying standard libraries (glibc) that programmers use to handle memory management. In the following, we outline how and why one can gain considerable improvements for many solvers by modifying glibc.

The underlying algorithms for AR tools are often based on a technique called conflict driven no-good learning (CDNL) with watch lists whose efficiency highly depends on their memory consumption [13]. On modern hardware, the accessible memory is virtual and handled by a physical memory management unit (MMU). The mapping between virtual and physical memory is stored in page tables by the MMU. In order to reduce access time, recently used mappings are stored in a *translation lookaside buffer (TLB)*. In practice, when using watch lists extensively, high memory consumption results in unpredictable memory accesses, many cache misses, and so called page translation failures (TLB misses). A common way to speed up a CDNL-solver is to use advanced data structures, which improve the memory access for frequently running operations in the solver, for example, data structures for learnt clauses, two watched literals, and linear lookup tables. Another way to obtain a speed-up is to respect the memory caches of the underlying hardware in the implementation. For example, one can considerably improve the performance of a naive SAT solver by modifying the implementation in such a way that the memory cache is used as much as possible (reducing cache misses) as implemented in Riss [23], using cache pre-fetching as done in the solver CryptoMiniSat [43], or implementing very sophisticated data structures with the goal to optimize the overall memory usage as in the solver Lingeling [9]. The hardware related line of optimization opens the research question whether one can obtain further speed-up by exploiting other cache layers that are available in modern CPUs. One such cache layer is used for address translation, i.e., the translation look aside buffer (TLB). As an extension, we want to investigate whether this kind of improvement is applicable to multiple AR tools.

### New Contribution

In this paper, we introduce a simple and transparent approach to effectively reduce the number of TLB misses in order to speed up the execution of modern memory dependent solvers, in particular, unit propagation, sometimes also called Boolean constraint propagation. We employ a Linux memory management feature called *transparent huge pages (THP)*, which reduces the overhead of virtual memory translations by using larger virtual memory page sizes [45], effectively increasing the size of that cache. Our approach is based on modifying the standard C library (glibc), which is the default standard library in Linux systems [2]. Whenever a

solver allocates memory, we make sure that we additionally give the operating system kernel advice about the use of the memory (madvise). This feature can then be used for a solver simply by recompiling it and statically linking it against our modified glibc. In that way, we obtain a significant speed-up on benchmarks in model checking of up to 15% and for most other solvers of up to 10%. The approach is based on a hardware feature and thus generalizable to other operating systems and CPU architectures supporting large page sizes.

Our advances summarize as follows:

1. We propose an easily accessible way to reduce the number of TLB misses in combinatorial memory-dependent solvers by patching the glibc in a way that our modifications can be activated or deactivated at runtime.
2. We provide a build system to easily patch glibc and statically link a solver against the patched version. Our system is based on a setup that uses OS-level virtualization (Docker) [25] and is available to all modern Linux systems. We already provide various pre-compiled state-of-the-art reasoning tools.
3. We carry out extensive benchmarks and present detailed results for various reasoning tools.

**Related Work**

Chu, Hardwood, and Stuckey [16] as well as Hölldobler, Manthey, Saptawijaya [23] considered cache utilization in SAT solvers and illustrated how a resource-unaware SAT solver can be improved by utilizing the cache sensibly, resulting in reasonable speed-ups. The latter already hinted that using larger pages results in a speed-up of 10% for SAT solvers, and was a motivation for this work. The effect of huge pages has already been widely investigated in the field of operating systems, e.g., [41]. However, the focus was mostly on database systems, while an analysis of the effect for reasoning tools was not yet available. Recent research considered benchmarking system tools [32], selecting benchmarks to tune solvers [24], and treating input benchmarks for benchmarking [12]. These topics are orthogonal to our work. In contrast, we consider computational resources and memory management of solvers, in particular, its effect on the runtime. Bornebusch, Wille, and Drechsler [15] analyzed the memory footprint of SAT solvers and tried to improve them. However, they did not consider propagation and its data structures, which is reasonable from a complexity point of view due to large formula sizes.

## 2 Modern CDNL-based Solvers and Memory

Before we present our advances, we give a brief explanation on how modern SAT solvers are implemented and introduce components and mechanisms that are relevant for memory access. As many reasoners are based on SAT technology, e.g., [13, 29, 46], core concepts are very similar for various reasoners. First, we define (propositional) formulas and their evaluation in the usual way and assume

familiarity with standard notations, including satisfiability. For basic literature, we refer to introductory work [30]. We consider a universe $U$ of propositional variables. A *literal* is a variable or its negation and a *clause* is a finite set of literals. A (CNF) *formula* is a finite set of clauses. A *(partial) truth assignment* is a mapping $\tau : \mathrm{var}(X) \to \{0, 1\}$ defined for a set $X \subseteq U$ of variables. For $x \in X$, we put $\tau(\neg x) = 1 - \tau(x)$. For a formula $F$, we abbreviate by $\mathrm{var}(F)$ the variables that occur in $F$. We say that a truth assignment $\tau$ *satisfies* a clause $C$, if for at least one literal $\ell \in C$ we have $\tau(\ell) = 1$. We say that a truth assignment $\tau$ falsifies a clause $C$, if it assigns all its literals to 0. We call a clause $C$ *unit* if $\tau$ assigns all but one literal to 0. A truth assignment $\tau$ is *satisfying* if for each clause $C \in F$, the truth assignment $\tau$ satisfies $C$.

### 2.1 SAT Solvers

So far, there are two main contributing factors to advances in the efficiency of modern SAT solvers: (i) theoretical improvements in terms of more advanced algorithms and heuristics and (ii) algorithm engineering in terms of data structures. The core algorithm that drives search in modern SAT solvers is based on *conflict driven no-good learning (CDNL)* or also known as *conflict driven clause learning (CDCL)* [20, 36], which was widely extended by search heuristics [13, 28] and simplification techniques during search [27]. A key technique is *unit propagation*, which aims at finding clauses where all literals but one are already assigned and then setting the remaining literal to a value that satisfies the clause. Unit propagation is responsible for the vast majority of the overall runtime even in modern solvers [28]. Hence, algorithm engineering and efficient data structures are essential for practical solving, i.e., the *two-watched-literal* scheme for unit propagation [39] and fast lookup tables, which are also important for the used heuristics and learning techniques. The watched literal scheme reduces the number of steps in the algorithm and memory accesses, but decreases the efficiency of the memory access [23]. Still, this results in a considerable overall runtime improvement [28]. While lookup tables provide fast access to relevant clauses, they result in a much higher memory footprint and may yield unpredictable memory access [23].

### 2.2 CPUs, Virtual Memory, and Paging

Modern *operating systems (OSes)* provide the concept of *virtual memory* to applications. Thereby, the OS releases software developers from worrying about the actual physical memory layout and also allows for overcommitting resources. Virtual memory is managed at the granularity of *pages*. A page is a contiguous block of memory in the virtual address space. The OS can map a page to a *page frame*, which is a corresponding location in physical memory. On the Intel Architecture [26], page tables describe the mapping from pages to page frames and thus virtual to physical addresses. On 64-bit Intel systems, these page tables are trees with a depth of commonly up to four levels for $2^{48}$-byte address spaces.

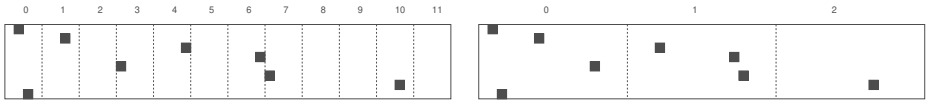


Fig. 1: This figure illustrates how pages cover a sequence of memory accesses for two different sizes of pages for a given amount of memory.

Walking these data structures to provide a translation for each memory access is infeasible, because it would add one page table read per level for each intended memory access. Instead, processors take advantage of spatial and temporal locality of memory accesses and cache translations in Translation Lookaside Buffers (TLBs). An Intel Skylake system has two levels of TLBs and the unified L2 TLB can hold 1536 entries [47]. Other recent CPUs have similar specifications. With 4KiB pages, this translates to holding translations for 6MiB of virtual memory in the TLB. A straight-forward way to increase the capacity of the TLB is for the processor architecture to allow for larger page sizes. On Intel 64-bit systems, in addition to 4KiB pages, the system also supports 2MiB and 1GiB pages. While 1GiB pages have (few) dedicated TLB entries, 2MiB pages share the same entries as 4KiB entries in the TLB on Skylake.

To benefit from large pages, the OS needs to make them accessible to applications [1] by constantly freeing memory (defragmenting) to obtain continuous blocks from which large pages can be allocated [40]. In more detail, fragmentation originates from applications that use 4KiB pages for which short blocks of memory are frequently allocated and freed. If many applications using 4KiB pages run in parallel, memory fragmentation is more likely. Then, it is harder to obtain free blocks of 2MiB memory. To still be able to use larger pages, the OS tries to restructure the memory mapping for future 2MiB requests.

Figure 1 illustrates the usage of memory with pages of different sizes. When using larger pages, fewer pages are required to cover the same area of memory. Hence, fewer TLB entries are occupied. In more detail, the black boxes in Figure 1 illustrate a sequence of accesses (from top to bottom). While for larger pages (right) it is sufficient to memorize the translation for three pages, smaller pages require seven pages (left). In case the TLB can only hold four entries, the entry of Page 0 would be evicted before it can be re-used to access the same clause again. When using larger pages, fewer initial translations have to be done, and only three pages are required to perform all accesses.

### 2.3 Large Pages in Linux

Large page sizes are supported in Linux by a feature called *transparent huge pages (THP)*, which offers both implicit and explicit use of large page sizes and was introduced in Linux 2.6.38 [45]. If THP is enabled, memory does not have to be statically provisioned for applications to use large pages, which is a clear advantage over previous attempts involving large pages [42]. Instead, the system is continuously compacting memory to free up contiguous space

to allocate large pages. The Linux kernel can then, depending on the system configuration, transparently allocate large pages for applications. If intended, a system administrator can still additionally provision large pages manually. THP can be globally enabled or configured as an opt-in feature. Both mechanisms degrade gracefully when no large pages are available and will instead back memory using the standard page size. When THP is configured for opt-in, an application can use the system call "madvise" with the "MADV_HUGEPAGE" flag to mark memory regions as eligible. If this is done for virtual memory regions that have not been backed by physical memory yet, e.g., directly after a "mmap" call, the kernel will try to allocate a large page on the first access to this memory. Otherwise, the kernel will occasionally scan virtual memory that is eligible for THP to create large pages. One downside of THP is that the kernel has to run scan and compact operations. Linux allows to configure this behavior to mitigate the impact by paying the cost for scanning and compacting at allocation time instead of doing it as a background job.

### 2.4 The Effect of THP

System workloads are known to speed up with huge pages. However, it may also reduce reproducibility, as huge pages have to be enabled in the kernel and globally for all applications on the system [42] and not every request might actually get a huge page. Hence, it is recommended to use small pages for benchmarking. Unfortunately, the StarExec cluster [44] has enabled THP by default for all executed programs making it incomparable to standard university computers, clusters, and industrial settings. The presented contribution is hence more targeted towards industrial use cases that want to solve a problem at hand as fast as possible. In case of virtualized machines, using small pages can result in almost 50% of the runtime being spent in address translation [33]. Using huge pages in both the guest and the host, which is 2M instead of 4k on the given architecture, reduces this value to about 4%. We expect similar savings for tools that are run in a virtualized environment, as virtualization typically uses huge pages internally. In the remainder of the paper, we investigate the actual effect on a bare metal setup for our experimental work.

## 3 TLB Misses in SAT Solvers

Typically, SAT solvers do not exhibit the memory access locality that caches or TLB are optimized for. While previous works considered caches [16, 23], memory translation and the TLB have not been taken into account. Hence, we focus on memory accesses in the most time consuming part of SAT solvers: unit propagation, also called Boolean constraint propagation.

Assume that a formula $F$ and a partial truth assignment $\tau$ is given and a two watched literal data structure is used [39]. Briefly, *unit propagation* works as outlined in Listing 1. Initially, for each clause $C \in F$, one selects two literals from $C$, which are not non-falsified by truth assignment $\tau$. Then, the truth

*UnitPropagate* (formula $F$, truth assignment $\tau$, literals $P$, watch lists $L$)

```
B1  while the list P of literals to propagate is not empty          //compute closure
B2    pick p ∈ P, and remove from P                                 //typically DFS
B3    access watch list L_p of clauses such that ¬p ∈ C             //propagate
B4    for all clauses C in L_p:
B5      if C ≠ ∅, x ∈ C, and x not falsified in assignment τ        //watchable literal
B6        remove C from L_p                                         //maintain lists
B7        add C to watch list L_¬x for ¬x                           //maintain lists
B8      else if C = (x) unit, extend P and τ with x                 //unit rule
B9      else if C is falsified, trigger conflict analysis(τ, C)     //conflict
```

Listing 1: Pseudo code for an implementation of unit propagation with the watched literal scheme. The state of the solver holds the formula $F$ as watch lists $L$, one list $L_x$ for each literal $x$, as well as a truth assignment $\tau$, and the list $P$ of literals to propagate. The result of the algorithm will either be an extension of the truth assignment, or a tuple truth assignment $\tau$ and a conflict clause that is falsified by truth assignment $\tau$.

assignment $\tau$ is extended by setting additional variables. Since assigning a literal $\ell \in C$ such that $\tau(\ell) = 1$ results in clause $C$ that is satisfied, which in turn allows to remove clause $C$ from the considered clauses right away, the only interesting case is if truth assignment $\tau$ sets literal $\ell \in C$ such that $\tau(\ell) = 0$. In that case, the clause $C$ might be falsified and be involved in a conflict or have unassigned literals, which can be used to imply the truth value of other literals. Then, UnitPropagate checks every clause $C$ that contains a literal which might be falsified during propagation. Therefore, the list $P$ of literals to propagate is traversed (Line B1), and each watch list $L_p$ for literal $p$ is processed (Line B3). Then, each clause $C$ in list $L_p$ contains $\neg p$, so that the new state of clause $C$ has to be evaluated by processing the other literals in clause $C$. Hence there are two cases: either (i) clause $C$ is satisfied by another literal, or (ii) clause $C$ contains another literal $x$ that is not yet falsified by the truth assignment $\tau$ (Line B5). Then, we watch literal $x \in C$ for being set to false instead of $\neg p$, and consequently have to update list $L_p$ (Line B6) and list $L_{\neg x}$ (Line B7). Otherwise, clause $C$ might be a unit clause (Line B8) or might be falsified by truth assignment $\tau$ (Line B9). In both cases, clause $C$ can remain in list $L_p$.

When considering the memory access pattern, the unit propagation algorithm has the following properties: the literals in list $P$ are not easily determined in advance. Hence, accesses in Line B3 to load the list are hard to predict. One could reduce the memory accesses in Lines B3 and B4 by pre-fetching data from memory in advance. This has been proposed in previous works [23]. Accessing the clauses in Line B4 are hard to predict, as the order of the clauses in list $L_p$ changes. In more detail, in Line B6 some clauses are removed and in Line B8 or B9 others are kept. To improve the access behavior in Line B5, Een and Sorensson [19] proposed for MiniSAT 2.1 an optimization to avoid the access

of clause $C$ for the satisfied case. There, another literal of clause $C$ is stored in list $L_p$. Since their introduction, blocking literals are commonly used in most modern solvers. Accessing literal $x$ in Line B5 is also unpredictable, as literal order in clauses also changes. Typically, the two watched literals are the first two stored literals and they change whenever the clause is moved to another list.

Our hypothesis is that unit propagation is the major source of memory accesses, as most runtime is spent in unit propagation, and many different memory locations, i.e., clauses are accessed non-linearly during unit propagation. This drives us to the following hypothesis:

**Hypothesis 1** *Accessing clauses during unit propagation as well as updating and accessing watch lists has a high impact on TLB misses.*

We support our hypothesis by the following observation. The two watched literals data structure allows to keep the number of overall accesses low, but has high memory footprint with additional data structures and lists. Further, the memory accesses for (i) clause to access next, (ii) literals of a clause to watch next, or (iii) list to place it are difficult to predict. Hence, Lines B3, B4, and B7 are prime candidates to access memory locations that have not been accessed recently, and hence, are not cached, nor served with current TLB entries.

*Analyzing Unit Propagation.* To back up Hypothesis 1 with data, we analyze the distribution of TLB misses in the SAT solvers MiniSat and Glucose[4]. When running MiniSat [19], we observe that 90% of TLB misses occur in unit propagation; thereof, about 10% when moving clauses to another (unpredictable) watch list and about 80% when accessing the first literal of the next watched clause. This data matches the assumption that Line B3 and B4 are responsible for most of the TLB misses. In addition, moving clauses to new watch lists contributes another 10%. When running Glucose version 4.2.1 [3], we can see similar results. 90% of the TLB misses happen in unit propagation. In the modern solver Glucose, unit propagation is split into (i) propagating binary clauses that contributed 5 % of all TLB misses, (ii) propagating during learned clause minimization [34] that contributes about 20 %, and (iii) propagating larger clauses and pushing them to watch lists that consume the majority of the TLB misses. Empirical observations for these two solvers confirm our hypothesis, unit propagation is the major source for TLB misses. The random memory accesses to check the next clause in the list for being unit, which can have an arbitrary memory location, as well as putting clauses into another watch list, are the major contributors. Unit propagation is responsible for a large fraction of the runtime of SAT solvers, which is actually spent in address translation. In Lingeling, Biere [8] places watch lists and its clauses closer to each other, to avoid TLB misses related to Line B4 (matching our observations in Section 5.1). Here, we present an orthogonal approach to avoid TLB misses, which allows to improve the implementation [8] further and can be applied to many other solvers.

[4] We use a sampling approach of CPU performance counters for TLB misses with the system tool perf.

*Unit Propagation Implementation Outlook.* We believe that additional data structure improvements along [16] are hardly feasible. Clauses would have to be even more compact and the changes require a huge effort for a single solver [7]. Changes to the underlying algorithms likely result in reduced performance and require tuning parameters again. On that account, we propose a general approach to THP, which can be easily used by many other tools.

## 4 Improving Unit Propagation with THP via Madvise

Modern Linux distributions provide native support for transparent huge pages. Usually, the systems allow a superuser to define the behavior via the configuration file "/sys/kernel/mm/transparent_hugepage/enabled" whose values "always" or "never" apply to all running processes. Because there might be applications running on the host that would suffer from larger pages, THP is usually disabled on physical systems and it is not advised to set the value to always. Fortunately, as described above, Linux also allows to use huge pages via the madvise system call. While this sounds fairly trivial, it requires (i) lots of manual adaption of the source code to mark memory regions as eligible and in turn makes the implementations of solvers (ii) fairly incomparable on an algorithmic level. When using "madvise" in combination with every malloc in a SAT solver, no speed-up is obtained, as most allocations are not aligned to the required large page size of 2M, and consequently are not backed by a huge page. In the following, we suggest an easily accessible way to reduce the number of TLB misses in combinatorial memory-dependent solvers.

### 4.1 Using More Huge Pages

In the previous section, we explained that using more huge pages seems to be a reasonable approach to speed up the memory access of modern solvers. This can be obtained by running a madvise system call to instruct the kernel to use transparent huge pages of 2M whenever the solver allocates memory. Then, we align all requested memory to 2M addresses and increase the size of the reservation accordingly, so that huge pages can actually be used. If we would not do so, two memory requests of the application can be in the middle of a 2M page, which results in not using a huge page. Compared to the system setting, this change results in using one more huge page per misaligned memory request.

### 4.2 Patching the Standard C Library (glibc)

In order to provide a transparent way to various solver developers, and offering a way for algorithms engineering to consider the effect of transparent huge pages on many AR tools, we want to avoid manual source code adaption as much as possible. To this end, we put our focus on the standard C system library glibc, which already provides standard functions to access the system memory. The library is used in Linux to compile most of the solvers. Instead of modifying

the source code of various solvers, we implement the above mentioned ideas into glibc[5]. Whenever a certain runtime flag is activated, our modified glibc takes care of the above mentioned changes. The current approach uses a system environment variable (GLIBC_THP_ALWAYS=1) that can be specified before calling a program. This way we allow setting the flag for a specific program instead of all running programs. Globally enabling transparent huge pages for all running applications on the system is usually forbidden both in industry and academia by administrators due to a variety of potential side effects, which might slow down a variety of programs. If the flag is not set, we disable the use of huge pages. The additional cost, compared to glibc, is a single if-statement in combination with the madvise system call. We implemented our patches into glibc 2.23 [2] and enable the feature without any source code modification of the various solvers themselves. In that way it is entirely sufficient for the user of a solver to recompile the solver and link it against our modified glibc.

### 4.3 Huge Pages in a Solver

In order to use the feature, there are two ways to proceed: (i) *link* the solver *statically* against our modified glibc or (ii) patch the system glibc and then *dynamically link* the solver against the new glibc. We provide an easy and accessible way for the former, since patching the system glibc is usually considered problematic due to side effects and as it requires superuser permissions, which makes it very unlikely that actual users of the reasoning tools will use this feature. We introduce a virtual environment that allows for easy compiling of the solver, in order to avoid problematic setups of a new secondary glibc.

Our system is based on the OS-level virtualization Docker [25], which isolates running programs entirely from each other. Docker itself is available on all modern OSes and allows to deliver software in packages, which are called containers. A running container is entirely isolated and can bundle its own software. We use this to not interfere with the system glibc. But we do not publish only a Docker container, instead we provide the scripts to build containers in which the compilation then runs. The user just needs to install Docker and we provide the tooling to link a solver with THP support. Along, we give many exemplary scripts to highlight how to run the tools and various pre-compiled state-of-the-art solvers.

We would like to emphasize that in our approach Docker is only used to compile the solver, not to actually run the solver. Compiling within a Docker environment can be done on a local machine or a trusted machine where the user has privileged access (as Docker often requires certain additional privileges) or is simply allowed to run Docker containers. Then, the resulting binary is transferred to the runtime environment. Since we compile files statically inside the Docker container to the modified glibc, there is absolutely no need to install a patched glibc or a Docker environment on the actual runtime system.

[5] Our latest implementation is publicly available at github:daajoe/thp_docker_build and the glib patch at github:conp-solutions/thp. An upstream to the glibc library is in progress and we are in contact with the glibc maintainers.

| | solver | $\#_n$ | $\#_{thp}$ | $t_n$ | $t_{thp}$ | $s$[%] | $\text{TLB}_n$ | $\text{TLB}_{thp}$ | $r_{tlb}$ | $\text{tlb}[s]_n$ | $\text{tlb}[s]_{tlb}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Winner19 | 194 | 197 | 7.38 | 6.13 | 16.97 | 3.4e+11 | 1.3e+10 | 3.94 | 5.35 | 0.25 |
| 2 | MergeSAT | 190 | 192 | 6.89 | 5.70 | 17.24 | 3.4e+11 | 1e+10 | 3.04 | 5.77 | 0.21 |
| 3 | CaDiCaL | 189 | 191 | 5.62 | 5.00 | 11.04 | 1.7e+11 | 5.6e+09 | 3.22 | 3.55 | 0.13 |
| 4 | Glucose | 189 | 191 | 4.50 | 3.66 | 18.72 | 2.6e+11 | 6.3e+09 | 2.44 | 6.61 | 0.20 |
| 5 | CryptoM. | 183 | 185 | 6.54 | 5.75 | 12.06 | 2.5e+11 | 8.2e+09 | 3.21 | 4.51 | 0.16 |
| 6 | Lingeling | 177 | 178 | 6.29 | 5.97 | 5.08 | 5.6e+10 | 6.3e+08 | 1.12 | 1.04 | 0.01 |
| 7 | MiniSat | 169 | 173 | 6.99 | 5.96 | 14.75 | 2.8e+11 | 3.3e+09 | 1.20 | 4.56 | 0.06 |

Table 2: Overview on the speed-up between solving when using THP for SAT solvers. We distinguish by non-THP and THP by $\cdot_n$ and $\cdot_{thp}$, respectively. # counts the number of solved instances. $t$ contains the runtime in hours for the mutually solved instances, $s$ the saved runtime in %, i.e., $s = (t_n - t_{thp})/t_n \cdot 100$ factor. TLB represents the TLB load misses on the mutually solved instances. $r_{tlb}$ summarizes the % of TLB misses over the original TLB misses, i.e., $r_{tlb} = \text{TLB}_{thp}/\text{TLB}_n \cdot 100$. $\text{tlb}[s]$ contains the TLB misses per second in relation to the physical bus speed of the CPU for 4 threads sharing the memory (9.6 GT/s /4).

## 5 Experimental Evaluation

We conducted a series of experiments using standard benchmark sets for various reasoning tools to analyze the effect of THP beyond pure SAT. All benchmark sets and our results are publicly available.

*Benchmarked Solvers and Instances* In our experimental work, we present results for recent versions of publicly available SAT solvers: CaDiCaL, CryptoMiniSat 5, Glucose 4.2.1 [3], Lingeling [9], MapleLCMDistChronoBTDL (Winner2019) [31], MergeSAT [35], and MiniSat [19]. We selected the recommended benchmark for tool tuning [24] and compare MiniSat and MergeSAT from the above set of SAT tools again on a second hardware. For answer set programming (ASP), we used clasp [29] and a robust benchmark set, which was developed for solver optimization and provides a large variety of instances, with adapted instance hardness, and free of duplicates [24]. From software model checking (SWMC), we use CBMC [17], which uses a single call to a SAT solver. As SWMC benchmark, we use the benchmark provided when introducing the LLBMC tool [38]. As another group, we collected tools, which use incremental SAT solvers as a backend. For hardware model checking (HWMC), we use the bounded model checker aigbmc [11], with an unrolling limit of 100, and use the benchmark of the deep bound track of the HWMC Competition of 2017 [10]. For optimization, we use the MaxSAT solver Open-WBO [37], which uses the SAT solver Glucose as a backend. As MaxSAT benchmark, we picked the weighted partial maxsat formulas from 2014, to make sure the incremental interface is actually used. Finally, we consider muser-2 [6], which computes a minimal unsatisfiable subformula (MUS) from a CNF formula, and use the group MUS benchmark from the MUS competition 2011[6].

[6] MUS benchmarks are available at cril.univ-artois.fr/SAT11

### 5.1 Evaluating Boolean Satisfiability (SAT) Solvers

For plain SAT solvers we carried out the following extensive study.

**Measure, Setup, and Resource Enforcements.** Our results were gathered on a cluster of RHEL 7.7 Linux machines with kernel 3.10.0-1062 with activated Meltdown and Spectre mitigations[7]. We evaluated the solvers on machines with two sockets equipped with Intel Xeon E5-2680v3 CPUs of 12 physical cores each at 2.50GHz base frequency. We forced the performance governors to 2.5Ghz [22] and disabled hyper-threading. To follow the hardware structure of the system, we manually grouped the solver resources as follows: on each socket the first two cores where restricted to controlling threads and processor cores 2,3,4,5,8 (virtual cores 2-6) and 9,10,11,12,13 (virtual cores 7-11) were assigned each to one running solver. The machines are equipped with 64GB main memory of which 60.5GB are freely available to programs. We compare wall clock time and number of timeouts. However, we avoid IO access on the CPU solvers whenever possible, i.e., we load instances into the RAM before we start solving. We run at most 4 solvers on one node, set a timeout of 900 seconds, and limited available RAM to 8GB per instance and solver. We follow standard guidelines for benchmarking [32].

**SAT Results.** Table 2 gives an overview of the number of solved instances for each solver with and without THP. Note that we report in this table only for columns $\#_n$ and $\#_{thp}$ on individually solved instances. To obtain comparability for all other columns, we present data for instances that have been solved by both configurations. The results show that a solver with activated THP solves overall more instances than without THP. When considering runtime, the configurations that employ THP solve the considered instances faster. Runtime improvements range from a smaller improvement for Lingeling, which was about 5% faster (which translates to 0.32h), up to more than 17% (i.e. one hour) faster for MergeSAT, MiniSat, and Winner2019. In terms of factor of saved runtime hours, we can see that the solvers that employ THP are up to almost 19% faster. As these values are gathered over the whole benchmark, these values are averages. For higher run-times, the speed-up is typically higher than for instances with a smaller runtime. The number of TLB misses that we observed reduce up to 2 orders, namely, $r_{tlb} = \mathrm{TLB}_{thp}/\mathrm{TLB}_n$ goes down to 1% for Lingeling and MiniSat. When we consider the number of solved instances, the value improves for all presented solvers when using THP, between 1 and 4 instances. As the presented technique is a linear speed-up, we do not expect a major change in number of solved instances for a specific timeout, due to the heavy-tailed runtime behavior of SAT solvers. Three solvers show a slightly different pattern. For Lingeling, we reduced the TLB misses to 1.1% and obtained only a speed-up of 5.4%. For CaDiCaL, we reduced the TLB misses to 3.22% and obtained a speed-up of only 12.4%. For CryptoMiniSat, we reduced the TLB misses to 3.21% and obtained a

[7] Note that we initially also ran experiments with an earlier kernel 3.10.0-693. There, non-THP runtime results were comparable, but improvements were slightly smaller.

speed-up of 13.7%. The last two columns illustrate the number of TLB misses per second (on average) in relation to the physical bus speed for 4 threads. Solvers with a higher initial value have a higher potential to benefit, which is reflected in our results.

**Discussion and Summary.** The results show that a solver with activated THP solves overall more instances than without THP. Throughout all solvers, we reduced the overall running time by 5% up to 19%. Our approach makes all solvers faster without spending a significant amount of time on optimizing the solver itself. Throughout our experiments, the number of TLB misses goes down significantly for all considered solvers. For Lingeling and MiniSat, the number of TLB misses even reduces to 1% of the original number of misses. Since unit propagation is a major source of memory accesses and a major cause of a high number of TLB misses and responsible of a large part of the solving time, the reduced TLB misses also yield a speed-up in the overall runtime. The observed results above confirm our hypothesis that improving on the number of TLB misses improves the runtime. We observe that three solvers have a somewhat different pattern, namely, CaDiCaL, CryptoMiniSat, Lingeling. While we obtain a speed-up, it is smaller than the order of the reduction of the TLB misses. If we compare the numbers to TLB misses per second and relate this to the physical bus speed that is shared by 4 threads – the value tlb[$s$] in Table 2, it suggests that the solvers CaDiCaL, CryptoMiniSat, Lingeling, and MiniSat are more optimized than the other solvers, where MiniSat also only maintains one watch list per literal, whereas the other solvers have a separate list for binary clauses, resulting in more memory fragmentation.

### 5.2 Evaluating Other Reasoners.

We believe that the THP approach does not only boost SAT solvers, but other reasoners as well. On that account, we run additional experiments on tools that are either based on SAT solvers or implemented closely to the CDNL algorithm. To broaden the applicability, we also consider tools that use SAT solvers via their incremental interface [19]. As the SAT calls in these tools are shorter, we expect that the benefit of using THP is smaller.

**Hypothesis 2** *When using incremental SAT solvers inside a reasoner, the benefit of THP is smaller.*

*Measure, Setup, and Resource Enforcements.* To support Hypothesis 2, we run a second analysis. To make sure the above results are not CPU and OS dependent, we used a second environment of the same architecture and repeat the results for MiniSat and MergeSAT. The computer has an Intel Core i5-2520M CPU running at 2.50GHz, with an Ubuntu 16.04 and Linux 4.15, using 5GB as memory limit and the 900 seconds as timeout per instance.

| Category | Tool | $t_n$ | $t_{thp}$ | $s$[%] |
|---|---|---|---|---|
| SAT | MiniSat | 8.17 | 7.03 | 13.99 |
| SAT | MergeSAT | 7.94 | 6.90 | 13.13 |
| ASP | clasp | 3.66 | 3.29 | 10.18 |
| MaxSAT | open-wbo | 1.19 | 1.09 | 8.49 |
| MUS | muser2 | 4.18 | 3.97 | 5.16 |
| HWMC | aigbmc | 0.89 | 0.86 | 4.11 |
| SWMC | cbmc | 0.23 | 0.22 | 2.76 |

Table 3: Overview on the runtime of various reasoners with(out) THP evaluated on their respective competition benchmarks. $t_n$ and $t_{thp}$ represent the overall (PAR1) runtime of the reasoner in hours and $s$ represents the saved runtime in %.

**Results.** Table 3 states the results for the considered tools and benchmarks. To measure the speed-up, we show only instances that have been solved by both variants. When using THP, we can usually solve a few more instances. First, we can see a similar improvement like in the previous setting where we used the same architecture, but different hardware. The improvement when using THP for tools with a single call is similarly high as presented above, i.e., SAT as well as ASP show improvements above 10%. Only cbmc from SWMC is an outlier, which might be related to its memory usage and the time it spends in other algorithms. For tools that use incremental SAT solvers as a backend, the improvements range from 4% from HWMC to 8% in MaxSAT. The low speed-up can be explained with their memory usage: over each benchmark, cbmc's memory usage is rather low, i.e., the median memory footprint is 8.8MB. For all the other tools and categories, the memory footprint is higher, e.g., the median for ASP is 28.8MB and for SAT 123.3MB. The tools with incremental SAT backends also consume more memory than cbmc: MaxSAT 21.3MB, HWMC 164MB, and MUS 298.1MB. As expected, tools with a higher memory footprint result in a higher speed-up due to transparent huge pages.

## 6 Conclusion and Future Work

*Summary.* Although reasoners solve NP-hard problems, they are used across the research community to solve many tasks in artificial intelligence. Reasoners are also employed in industry to verify properties, generate tests, or run similar tasks. In this paper, we introduced a simple and transparent approach to effectively speed up memory access of reasoners by reducing the number of TLB misses, which in turn allows to significantly improve their runtime. Our approach is based on a modification of glibc, which is the C standard library of the GNU Project and widely used in Linux for C and C++ programs. A user of a reasoner can benefit from our improvement simply by recompiling his favorite reasoner and enabling the feature by an environment flag when invoking it. Our experiments confirmed that an application can save up to 25% runtime on certain instances

and on average more than 10%. Since the tools based on SAT solvers are often also used for long running jobs, for example in systems biology or verification, we can save a significant amount of runtime, and hence operational cost. In that way, the number of solved instances might not always be the right measure to evaluate a reasoner, but the overall runtime should be taken into account as well.

*Other Applications.* We believe that our approach can also be very beneficial for other tools in automated reasoning, because there are many memory intensive applications that have simply not been tuned to reduce the number of TLB misses by using THP. One such domain might be graph algorithms, which can have random memory access patterns if the underlying data structure is updated often.

*Potential Benefits for CP Solvers.* Our current work is tailored to algorithms that heavily use unit propagation and two watched literal data structures. While we expect similar behavior for algorithms that employ similar features and memory usage behavior, many CP solvers implement propagators that are heavily computation intensive and less memory dependent. So far, we have no evidence that the proposed techniques improve such CP solvers. Still, it could be helpful for some solvers that use lazy clause generation. This, however, requires more detailed experimental evaluations on a variety of benchmarks.

*Industrial Effects.* We believe that our approach can be very beneficial for industry in settings where AR tools are used for continuous builds, testing, and integration. In particular, we expect that the technique proves useful when there are monetary demands introduced by using computation platforms, which allow for easy scalability, but bill resources by allocation time, such as Amazon Web Services, Microsoft Azure, and Google Cloud [4, 5].

*Future Work.* In the future, we are interested in the influence of THP on various other domains and in increasing the amount of tested benchmarks and reasoning tools. We hope that this opens up both theoretical and practical research on more general algorithm engineering techniques.

# Bibliography


[1] Arcangeli, A.: Transparent hugepage support. In: KVM forum. vol. 9 (2010)

[2] Arnold, R.S., Brown, M., Eggert, P., Jelinek, J., Kuvyrkov, M., Myers, J., O'Donell, C., Oliva, A., Schwab, A.: The GNU C library (glibc). https://www.gnu.org/software/libc/ (2019)

[3] Audemard, G., Simon, L.: Glucose in the SAT Race 2019. In: Heule, M.J., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions. Department of Computer Science Report Series, vol. B-2019-1, pp. 19–20. University of Helsinki (2019)

[4] AWS: Astera labs uses AWS to accelerate chip development. https://aws.amazon.com/solutions/case-studies/astera-labs/?did=cr_card&trk=cr_card (2020)

[5] AWS: Aws customer success story: Zalando. https://aws.amazon.com/solutions/case-studies/zalando/ (2020)

[6] Belov, A., Marques-Silva, J.: Muser2: An efficient MUS extractor. J. on Satisfiability, Boolean Modeling and Computation **8**(3/4), 123–128 (2012)

[7] Biere, A.: Lingeling essentials, a tutorial on design and implementation aspects of the the SAT solver Lingeling. In: Berre, D.L. (ed.) POS-14. Fifth Pragmatics of SAT workshop. EPiC Series in Computing, vol. 27, p. 88. EasyChair (2014). https://doi.org/10.29007/jhd7, https://easychair.org/publications/paper/xJs

[8] Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2016-1, pp. 44–45. University of Helsinki (2016)

[9] Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2017-1, pp. 14–15. University of Helsinki (2017)

[10] Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Stewart, D., Weissenbacher, G. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2017, Vienna, Austria, October 02-06, 2017. p. 9. IEEE (2017)

[11] Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)

[12] Biere, A., Heule, M.: The effect of scrambling CNFs. In: Berre, D.L., Järvisalo, M. (eds.) Proceedings of Pragmatics of SAT 2015 and 2018. EPiC Series in Computing, vol. 59, pp. 111–126. EasyChair (2019)

[13] Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam, Netherlands (Feb 2009)
[14] Bjørner, N.: SMT in verification, modeling, and testing at microsoft. In: Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing. p. 3. HVC'12, Springer Verlag, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-39611-3_3
[15] Bornebusch, F., Wille, R., Drechsler, R.: Towards lightweight satisfiability solvers for self-verification. In: Proceedings of the 7th International Symposium on Embedded Computing and System Design (ISED'17). pp. 1–5 (Dec 2017). https://doi.org/10.1109/ISED.2017.8303924
[16] Chu, G., Harwood, A., Stuckey, P.: Cache conscious data structures for Boolean satisfiability solvers. J. on Satisfiability, Boolean Modeling and Computation **6**, 99–120 (02 2009). https://doi.org/10.3233/SAT190064
[17] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
[18] D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(7), 1165–1178 (2008)
[19] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03). pp. 502–518. Springer Verlag (2003)
[20] Gomes, C., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Smolka, G. (ed.) Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97). Lecture Notes in Computer Science, vol. 1330, pp. 121–135. Springer Verlag, Linz, Austria (1997). https://doi.org/10.1007/BFb0017434
[21] Gupta, A., Ganai, M.K., Wang, C.: Sat-based verification methods and applications in hardware verification. In: Bernardo, M., Cimatti, A. (eds.) Formal Methods for Hardware Verification. pp. 108–143. Springer Verlag, Berlin, Heidelberg (2006)
[22] Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., Geyer, R.: An energy efficiency feature survey of the intel haswell processor. In: Lalande, J.F., Moh, T. (eds.) Proceedings of the 17th International Conference on High Performance Computing & Simulation (HPCS'19) (2019)
[23] Hölldobler, S., Manthey, N., Saptawijaya, A.: Improving resource-unaware SAT solvers. In: Fermüller, C.G., Voronkov, A. (eds.) Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'16). Lecture Notes in Computer Science, vol. 6397, pp. 519–534. Springer Verlag, Dakar, Senegal (2010), https://doi.org/10.1007/978-3-642-16242-8_26
[24] Hoos, H.H., Kaufmann, B., Schaub, T., Schneider, M.: Robust benchmark set selection for boolean constraint solvers. In: Proceedings of the 7th Inter-

national Conference on Learning and Intelligent Optimization (LION'13). Lecture Notes in Computer Science, vol. 7997, pp. 138–152. Springer Verlag, Catania, Italy (Jan 2013), revised Selected Papers

[25] Hykes, S., et al.: Docker ce. https://github.com/docker/docker-ce (2019)

[26] Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual (2019), order Number: 325462-069US

[27] Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning, Lecture Notes in Computer Science, vol. 7364, pp. 355–370. Springer Verlag (2012), https://doi.org/10.1007/978-3-642-31365-3_28

[28] Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern SAT solvers. In: Sakallah, K.A., Simon, L. (eds.) Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT'11), Lecture Notes in Computer Science, vol. 6695, pp. 343–356. Springer Verlag, Ann Arbor, MI, USA (June 2011), https://doi.org/10.1007/978-3-642-21581-0_27

[29] Kaufmann, B., Gebser, M., Kaminski, R., Schaub, T.: clasp – a conflict-driven nogood learning answer set solver. http://www.cs.uni-potsdam.de/clasp/ (2015)

[30] Kleine Büning, H., Lettman, T.: Propositional logic: deduction and algorithms. Cambridge University Press, Cambridge, New York, NY, USA (1999)

[31] Kochemazov, S., Zaikin, O., Kondratiev, V., Semenov, A.: MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019. In: Heule, M.J., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions. Department of Computer Science Report Series, vol. B-2019-1, pp. 24–24. University of Helsinki (2019)

[32] van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C., Heiser, G.: Benchmarking crimes: An emerging threat in systems security. CoRR **abs/1801.02381** (2018), http://arxiv.org/abs/1801.02381

[33] Kwon, Y., Yu, H., Peter, S., Rossbach, C.J., Witchel, E.: Coordinated and efficient huge page management with ingens. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16). pp. 705—721. USENIX Association, Savannah, GA, USA (2016)

[34] Luo, M., Li, C.M., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17. pp. 703–711 (2017). https://doi.org/10.24963/ijcai.2017/98

[35] Manthey, N.: MergeSat. In: Heule, M.J., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions. Department of Computer Science Report Series, vol. B-2019-1, pp. 29–30. University of Helsinki, Helsinki, Finland (2019)

[36] Marques-Silva, J., Sakallah, K.: GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (May 1999), https://doi.org/10.1109/12.769433

[37] Martins, R., Manquinho, V., Lynce, I.: Open-wbo: A modular maxsat solver,. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing – SAT 2014. pp. 438–445. Springer International Publishing, Cham (2014)
[38] Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software: Theories, Tools, Experiments. pp. 146–161. Springer Verlag (2012)
[39] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Rabaey, J. (ed.) Proceedings of the 38th Annual Design Automation Conference (DAC'01). pp. 530–535. Assoc. Comput. Mach., New York, Las Vegas, Nevada, USA (2001), https://doi.org/10.1145/378239.379017
[40] Navarro, J., Iyer, S., Druschel, P., Cox, A.: Practical, transparent operating system support for superpages. SIGOPS Oper. Syst. Rev. **36**(SI), 89–104 (Dec 2003), https://doi.org/10.1145/844128.844138, this paper describes Super Page implementation in FreeBSD. It also has performance nubmers, but really ancient ones. They roughly match the SAT solver performance improvements, though.
[41] Panwar, A., Prasad, A., Gopinath, K.: Making huge pages actually useful. In: Bianchini, R., Sarkar, V. (eds.) Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18). pp. 679–692. Assoc. Comput. Mach., New York, Williamsburg, VA, USA (Mar 2018), https://doi.org/10.1145/3173162.3173203
[42] Park, S., Kim, M., Yeom, H.Y.: GCMA: Guaranteed contiguous memory allocator. IEEE Transactions on Computers **68**(3), 390–401 (Mar 2019), https://doi.org/10.1109/TC.2018.2869169
[43] Soos, M.: Cryptominisat 5.7.1. https://github.com/msoos/cryptominisat (2020)
[44] Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR'14). Lecture Notes in Computer Science, vol. 8562, pp. 367–373. Springer Verlag, Vienna, Austria (Jul 2014), https://doi.org/10.1007/978-3-319-08587-6_28, held as Part of the Vienna Summer of Logic, VSL 2014.
[45] Torvalds, L.: kernel.org: Transparent hugepage support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt (May 2017)
[46] Voronkov, A.: Avatar: The architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) Proceedings of the 26th International Conference on Computer Aided Verification CAV'14. Lecture Notes in Computer Science, vol. 8559, pp. 696–710. Springer Verlag (2014), held as Part of the Vienna Summer of Logic (VSL).
[47] Wikichip, C.: Skylake (client) – Microarchitectures – Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client) (2020)