Analysis of Portfolio-Style Parallel SAT Solving on Current Multi-Core Architectures*

Martin Aigner[1] and Armin Biere[2] and Christoph M. Kirsch[1] and Aina Niemetz[2] and Mathias Preiner[2]

[1] University of Salzburg
[2] Johannes Kepler University Linz

Abstract

Effectively parallelizing SAT solving is an open and important issue. The current state-of-the-art is based on parallel portfolios. This technique relies on running multiple solvers on the same instance in parallel. As soon as one instance finishes, the entire run stops. Several successful systems even use Plain Parallel Portfolio (PPP), where the individual solvers do not exchange any information. This paper contains a thorough experimental evaluation of PPP, which shows that PPP can improve wall-clock runtime. This improvement is due to the fact that memory access is still local and the memory system can hide the latency of memory access, respectively. In particular, there does not seem as much cache congestion as one might imagine. We further present some limits on the scalability of PPP and finally give one argument why PPP solvers are a good fit for todays multi-core architectures.

1 Introduction

As most state-of-the-art processors have multiple cores, there is an increasing interest in efficiently utilizing multi-core shared memory architectures for speeding up SAT solving [6, 12]. The currently dominating approach, pioneered by the authors of ManySAT [5], is based on portfolio solving. It simply runs multiple different SAT solvers on the same instance in parallel. As soon as one solver (thread) solves the instance, the overall run finishes. The portfolio approach of ManySAT, which is a variant of MiniSAT, originally used one and the same SAT solver with different search heuristics. The authors of [5] further suggested to share information, more specifically learned clauses, among the solvers. But even without sharing information, the excellent performance of ppfolio [13] in the SAT competition 2011 showed that using different heuristics or, as in ppfolio, different SAT solvers can lead to large gains in performance. Note that in the following, we consider wall-clock runtime as our primary performance metric.

We call a parallel portfolio solver like ppfolio, which synchronizes on termination but otherwise does not share any information, a *plain parallel portfolio* solver (PPP). The focus of our experimental evaluation is on running PPP solvers on multi-core machines with shared memory, as they are used in the SAT competitions. An important question for this scenario is whether memory as shared resource becomes a bottleneck. More specifically, we have measured the *slowdown* of running identical solvers on the same instance on a multi-core processor in parallel. Performance degradation witnessed by PPP solvers is due to congestion of the memory system and can be seen as an upper bound on the expected slowdown for portfolio systems in general. Analytical reasons why portfolio and PPP actually work (and often better than search

*This work was partially funded by the Austrian Science Fund (FWF) under the NFN Grants S11404-N23 and S11408-N23 (RiSE).

space partitioning) were given in [8], but potential slowdown due to shared memory resources was not taken into account. In this paper, we try to address this orthogonal issue empirically.

Earlier work on parallel SAT solving such as [17, 15] was targeted towards multiple distributed computing node clusters. The main questions discussed in the literature were how to split the search and which clauses to share. Interestingly enough, portfolio systems today, like ManySAT [5] or Plingeling [2], have almost the same software architectural model as early cluster solvers: the original formula as well as shared clauses are copied by each solver. The design decision to physically separate clauses not only simplifies solver development, but further allows synchronization overhead to be kept at a bare minimum. This is in contrast to solvers that try to parallelize on a much more fine-grained level [11, 9], and which do not seem to scale at this point, at least for more than two cores.

The drawback of this common design decision in todays portfolio solvers is that neither the original formula nor learned clauses are physically shared and thus running such a portfolio system on n cores needs n times more memory than running it on one core only. Hence, it might be expected that this common design of portfolio solvers leads to more congestion in the memory systems and accordingly should slowdown performance considerably. However, our experimental evaluation shows that most memory accesses are local, i.e., they can be satisfied by core-local caches, which keeps the slowdown low even for a large number of solvers running in parallel. These results should extend to other types of parallel solvers for shared memory systems, which also separate clauses, but use some form of search space splitting [7, 8] instead of portfolio solving. A similar but preliminary experimental evaluation can be found on pages 67-68 of [10] and [14]. We use more benchmarks and machines and also give a much more detailed analysis. In another related paper [18] the focus is on comparing cache efficiency of different data structures and of different SAT solvers, but the authors do not discuss parallel SAT solving.

2 Experimental Evaluation

In this section, we present a thorough experimental evaluation of PPP. Note that due to page constraints, we can only give a small overview of our experiments. For a more detailed evaluation (including plots for all machines), please refer to `http://fmv.jku.at/pos13/`.

Name	**Frequency**	**CPUs**	**Cores/CPU**	**VCores**
intel-i7-920-8vcores	2.67 GHz	1	4	8
intel-i7-2600-8vcores	3.40 GHz	1	4	8
amd-opteron-2350-8vcores	2.00 GHz	2	4	8
intel-xeon-e5645-24vcores	2.40 GHz	2	6	24
intel-xeon-e7-4850-80vcores	2.00 GHz	4	10	80

Table 1: This table lists the hardware details of the systems in use. Column *CPUs* denotes the number of physical sockets, column *cores/CPU* the number of cores per CPU, and column *VCores* the total number of virtual cores, i.e., the number of threads (including hyper-threads for all Intel platforms) that may run in parallel on each machine.

Name	L1d	L1i	L2	L3
intel-i7-920-8vcores	32 KB 8-way	32 KB 4-way	256 KB 8-way	8 MB 16-way
intel-i7-2600-8vcores	32 KB 8-way	32 KB 8-way	256 KB 8-way	8 MB 16-way
amd-opteron-2350-8vcores	64 KB 2-way	64 KB 2-way	512 KB 16-way	2 MB 32-way
intel-xeon-e5645-24vcores	32 KB 8-way	32 KB 4-way	256 KB 8-way	12 MB 16-way
intel-xeon-e7-4850-80vcores	32 KB 8-way	32 KB 4-way	256 KB 8-way	24 MB 24-way

Table 2: This table gives an overview of the cache size and associativity of our machines. For all platforms, level 1 (L1) and level 2 (L2) caches are per-core caches, whereas the level 3 (L3) cache is shared between all cores on one CPU. The suffixes d and i denote data caches and instruction caches, respectively.

2.1 Environment

We used five different machines to run our experiments. Table 1 gives an overview of the CPU configuration for each machine and Table 2 describes their cache setup. All machines run a 64-bit Ubuntu Linux operating system in the default NUMA policy, where available.

2.2 SAT solvers

As SAT solvers we selected several well-known SAT solvers, which have been competing in various competitions before. For the SAT solver Lingeling [2], we used one of the latest public releases, version al6. For MiniSAT [4], we used version 2.2.0, which was used in the MiniSAT hack track in the SAT Competition 2011. We further used the solvers CryptoMiniSAT 2.9.5 [16], PicoSAT 953 [3], and Glucose 2.1 [1], the official winner of the single engine track at the SAT Challenge 2012. Thus, we claim to cover state-of-the-art solvers and solvers of common interest, except that we focused on CDCL solvers. Local-search and look-ahead solvers are less suited for application instances we are interested in.

2.3 Benchmarks

Table 3 lists the benchmarks used for our evaluation selected by the following criteria. All benchmarks should be solvable by all solvers in a reasonable time w.r.t. the results of the SAT Challenge 2012. As a second criteria, the benchmarks should vary as much as possible in terms of size and origin. Further, we had to restrict the number of benchmarks due to the extensive computational resources needed for running all solvers on various hardware configurations with different numbers of cores. Note that we also expected a substantial slowdown with an increasing number of cores.

The characteristics of the selected benchmarks w.r.t. the maximum memory usage of each solver are depicted in Fig. 1 and 2. The maximum memory required by each solver for solving a benchmark w.r.t. the benchmark file size is shown in Fig. 1. The most efficient SAT solver in terms of memory usage is Lingeling, which has the lowest memory footprint of all solvers. As shown in the following, this can be beneficial in a PPP setup. Fig. 2 illustrates the runtime of each solver on the intel-i7-920-8vcores machine compared to their maximum memory usage on all benchmarks.

In general, the trend is that the runtime of the selected SAT solvers required for solving a benchmark correlates with the maximum memory needed.

Benchmark		**Size** [B]	**Time** [s]		**Memory** [MB]	
Name	Status		min	max	min	max
s69-100	u	1806	4(p)	60(g)	2(m)	18(g)
sgen4-unsat-77-1	u	1997	17(p)	338(l)	2(m)	34(g)
longmult8	u	172500	2(l)	8(c)	3(l)	17(c)
longmult12	u	274966	13(c)	58(l)	6(p)	33(c)
rbsat-v760c43649gyes1	s	514582	14(g)	455(p)	27(l)	88(c)
AProVE07-16	u	3284797	45(l)	1205(p)	23(l)	266(c)
6pipe_6_ooo.shuffled-as.sat03-413	u	10510827	19(g)	744(p)	43(l)	181(m)
traffic_pcb_unknown	u	19251005	87(g)	1151(p)	63(p)	309(c)
grid-strips-grid-y-3.055-NOTKNOWN	s	66196343	81(g)	1946(l)	174(p)	1020(m)
narain-vpn-clauses-10	u	187655748	18(m)	295(l)	624(p)	1193(c)

Table 3: This table lists all benchmarks selected for our evaluation. Column *status* denotes if the benchmark is either *satisfiable* (s) or *unsatisfiable* (u). The column *time* denotes the minimum/maximum runtime required for each benchmark, where the character at the end of the values denotes the first character of the solver that was fastest (resp. slowest). Note that we picked the minimum/maximum values from all available hardware configurations. Further note that the runtimes differ considerably depending on the system used. For a more detailed runtime overview please refer to the webpage. Column *memory* denotes the maximum memory usage of a solver for each benchmark, we again marked the solver with lowest/highest memory usage.

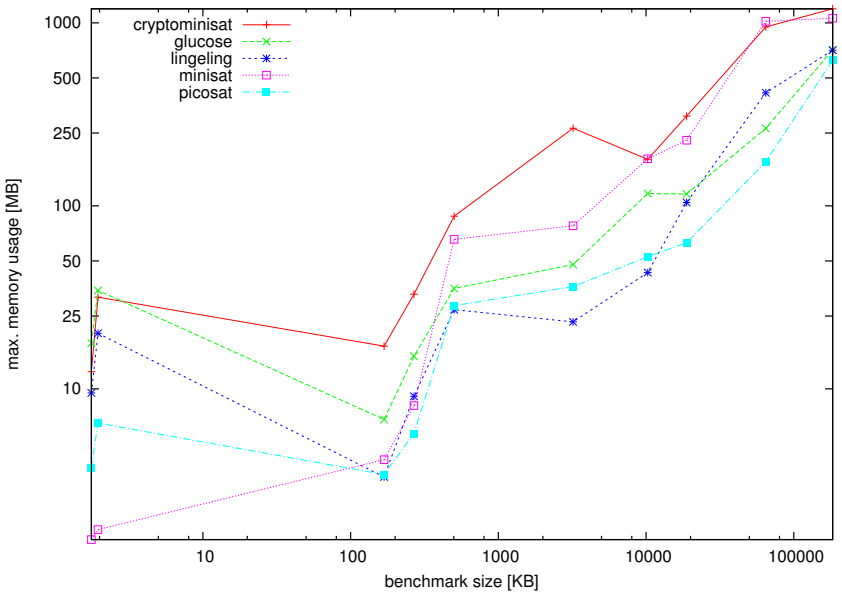

Figure 1: Relationship between benchmark file size and maximum memory usage of running one instance of each solver on the selected benchmarks on the intel-i7-920-8vcores machine.

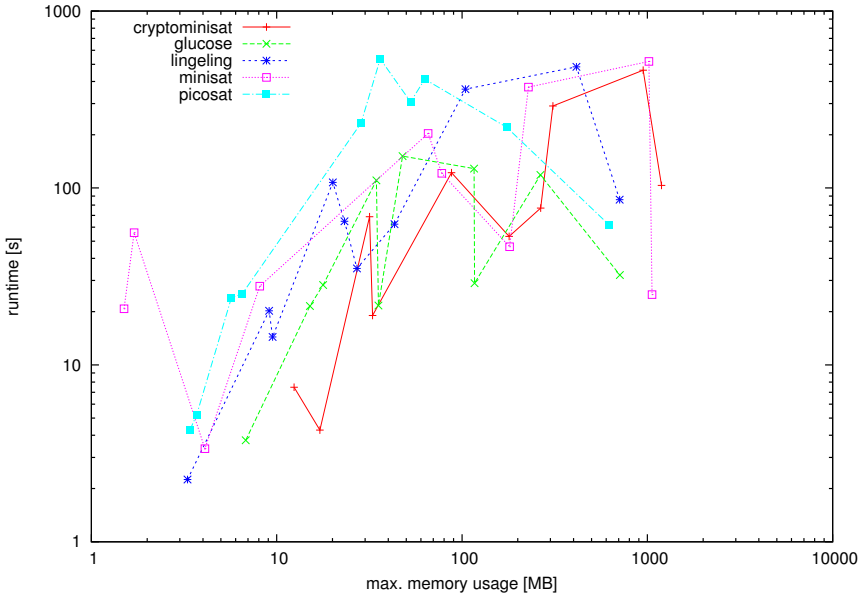

Figure 2: Correlation of a solver's runtime and maximum memory usage on the intel-i7-920-8vcores machine.

2.4 Metrics

In our experimental evaluation, we measure several performance dimensions. This section gives a brief overview of the metrics in use.

We denote the execution of one instance of a SAT solver on a given benchmark as *job*. We measure the runtime of a job in seconds (wall-clock time) and based on the runtime we define the *slowdown* for n identical jobs as the runtime of n jobs divided by the runtime of one job.

The *memory usage* of a job is measured by sampling the resident set size of the process executing the job, i.e., the number of pages that the operating system maintains in main memory for that process. For our evaluations we actually use the maximum observed memory usage given in MB.

The *working set* of a job is the memory that a job reads or writes during a given time interval. The rate at which the working set changes over time determines the spatial locality of a job. For simplicity, instead of measuring the working set directly, we measure the cache miss rate of the first level data cache (L1 cache), where the cache miss rate is the number of cache misses divided by the number of cache hits. Low cache miss rates indicate small working sets that fit into the caches. High cache miss rates may be the result of large working sets (or significant sharing of memory which we can exclude here). We employ the Linux `perf tools` to measure the hardware performance counters for L1 cache misses and cache hits.

2.5 PPP's scalability bounds

In this section we explore the scalability bounds for PPP setups. We are interested in the slowdown that occurs to independent jobs when they run in parallel on shared-memory multi-core hardware where processes compete for shared resources such as main memory and the

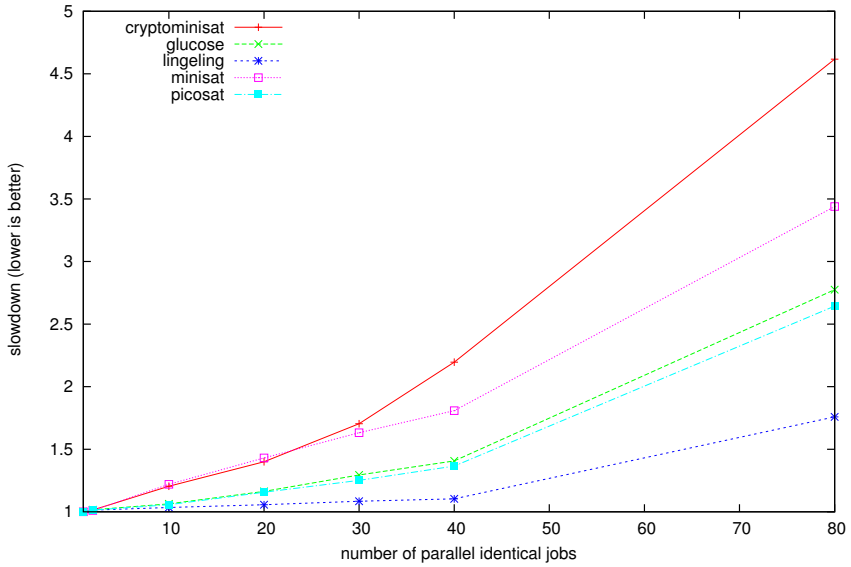

Figure 3: Slowdown of the execution for an increasing number of parallel jobs solving the 6pipe_6_ooo.shuffled-as.sat03-413 benchmark on the intel-xeon-e7-4850-80vcores machine.

memory bus. For this experiment we run an increasing number of identical but independent jobs and measure the slowdown, i.e., the time required for all instances to finish divided by the time required for one instance. Below we present the results for the three benchmarks that showed the most significant slowdown on our intel-xeon-e7-4850-80vcores machine.

Fig. 3 shows the slowdown of all solvers for the 6pipe_6_ooo.shuffled-as.sat03-413 benchmark. In this setup, Lingeling scales nearly perfectly up to 40 parallel jobs while CryptoMiniSAT and MiniSAT are slowed down by up to a factor of two. For more than 40 parallel jobs the Linux scheduler assigns jobs to hyper-threads sharing the first level cache. The impact of hyper-threading, i.e., running jobs on virtual cores is discussed in detail below.

In Fig. 4 we present the results for the traffic_pcb_unknown benchmark. MiniSAT and PicoSAT show the largest slowdown of nearly 2.5 for 40 parallel jobs. In this case however, CryptoMiniSAT which shows the largest slowdown for the 6pipe_6_ooo.shuffled-as.sat03-413 benchmark, now scales well showing a slowdown of only 1.4 for 40 parallel jobs. A similar situation is shown in Fig. 5 for the AProVE07-16 benchmark. Again, Lingeling scales well for up to 40 parallel jobs while Glucose suffers from the largest slowdown for this setup.

Fig. 6 illustrates the slowdown for the rbsat-v760c43649gyes1 benchmark on our intel-xeon-e5645-24vcores machine. Similar to the previous three benchmarks on the intel-xeon-e7-80vcores setup, all solvers scale well for up to 12 parallel jobs and slow down significantly as soon as hyper-threading comes into play. This behavior is further illustrated in Fig. 7, which shows the absolute runtime for this setup.

An interesting case is shown in Fig. 8 and 9, where we compare the absolute runtime for solving the biggest benchmark narain-vpn-clauses-10 on the amd-opteron-2350-8vcores and intel-i7-2600-8vcores machines. On the intel-i7-2600-8vcores machine we have a heavy increase of runtime (and a more drastic slowdown) compared to the amd-opteron-2350-8vcores. This is due to the fact that the Intel machine uses hyper-threading with 4 physical cores (8 virtual

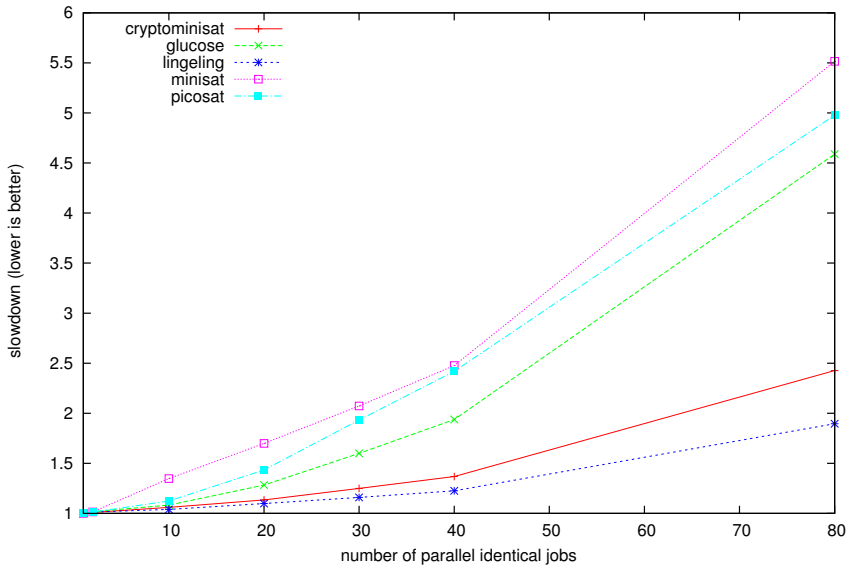

Figure 4: Slowdown of the execution for an increasing number of parallel jobs solving the traffic_pcb_unknown benchmark on the intel-xeon-e7-4850-80vcores machine.

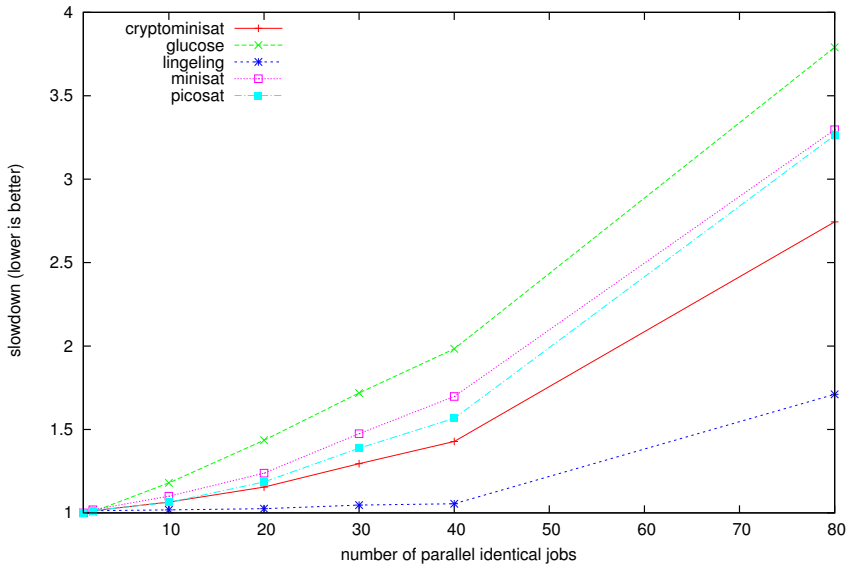

Figure 5: Slowdown of the execution for an increasing number of parallel jobs solving the AProVE07-16 benchmark on the intel-xeon-e7-4850-80vcores machine.

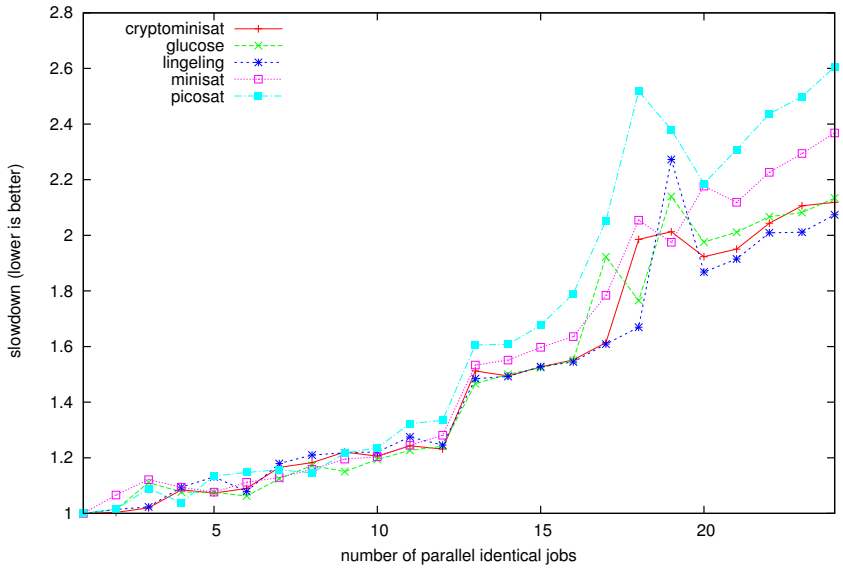

Figure 6: Slowdown of the execution for an increasing number of parallel jobs solving the rbsat-v760c43649gyes1 benchmark on the intel-xeon-e5645-24vcores machine.

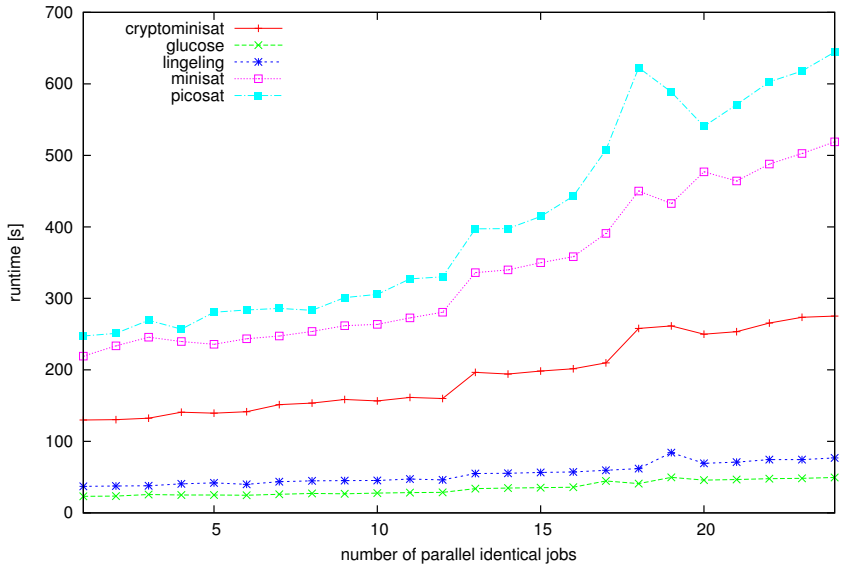

Figure 7: Absolute runtime required for an increasing number of parallel jobs solving the rbsat-v760c43649gyes1 benchmark on the intel-xeon-e5645-24vcores machine.

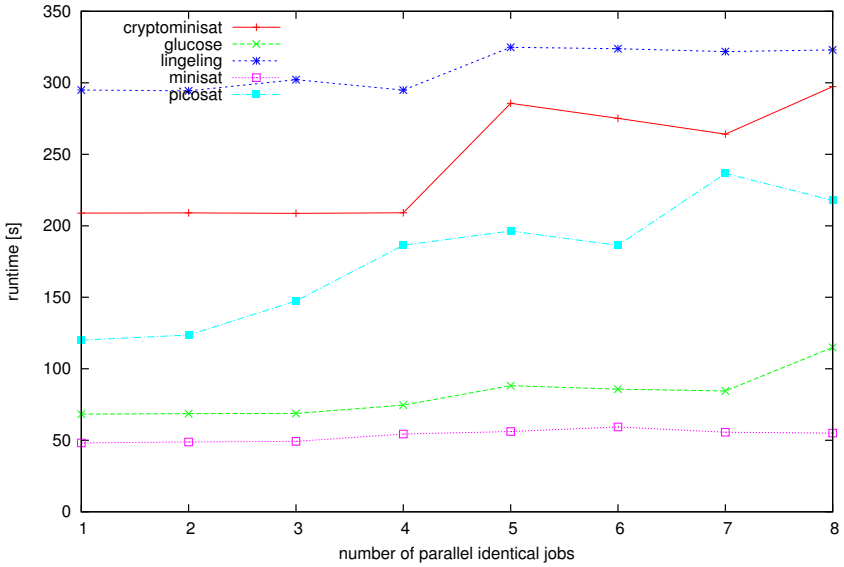

Figure 8: Absolute runtime required for an increasing number of parallel jobs solving the narain-vpn-clauses-10 benchmark on the amd-opteron-2350-8vcores machine.

cores) while the AMD machine consists of 8 physical cores, i.e., the virtual cores of the former machine have to share caches, whereas on the AMD machine every core has a dedicated cache.

We argue that the slowdown is mainly introduced by sharing resources of our multi-core platform, namely main memory and the memory bus. For as long as the working set of a process fits in the per-core local first level (L1) cache the cores of our machine can be seen as independent computers. When the working set is too large the independent parallel processes are forced to compete for shared resources thus slowing down the total execution time. To support our claim we measure the L1 miss rate, i.e., the number of L1 read misses divided by the number of L1 read hits, for a single SAT solver process. Fig. 10 shows the L1 miss rate of all solvers for the benchmarks that caused the largest slowdown as discussed above. For each benchmark, Lingeling causes the lowest L1 miss rate which explains the negligible slowdown. The other solvers cause significantly higher L1 miss rates which translate nearly linearly to the slowdown presented in Fig. 3, 4, and 5.

Another interesting topic is the expected average slowdown when using an additional physical or virtual core. For one particular combination of machine, solver and benchmark, we compute the *average core slowdown* acsd resp. the *average virtual core slowdown* avcsd as follows:

$$\mathsf{acsd} = \frac{(t_c - t_1)}{t_1 \cdot (c-1)} \text{ with } c > 1 \qquad \mathsf{avcsd} = \frac{(t_v - t_c)}{t_c \cdot (v-c)} \text{ with } v > c > 0$$

where c is the number of physical cores, v is the number of virtual cores (cf. Table 1), and t_n is the wall-clock running time for running n parallel instances. The total average (virtual) core slowdown is computed as the average of the individual average (virtual) core slowdowns over a set of runs and is shown in Table 4.

First, notice that the average slowdown per core is much larger for the two desktop processors, intel-i7-920-8vcores and intel-i7-2600-8vcores. We contribute the large slowdown for

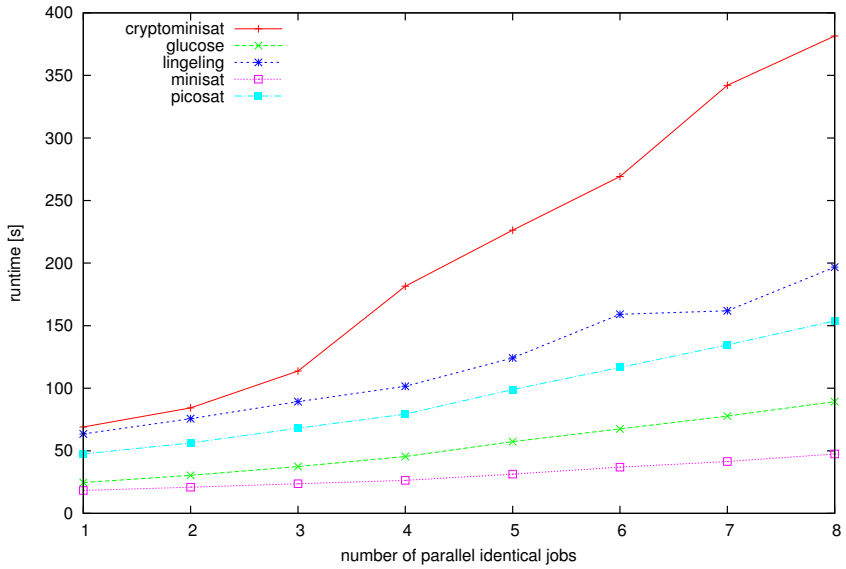

Figure 9: Absolute runtime required for an increasing number of parallel jobs solving the narain-vpn-clauses-10 benchmark on the intel-i7-2600-8vcores machine.

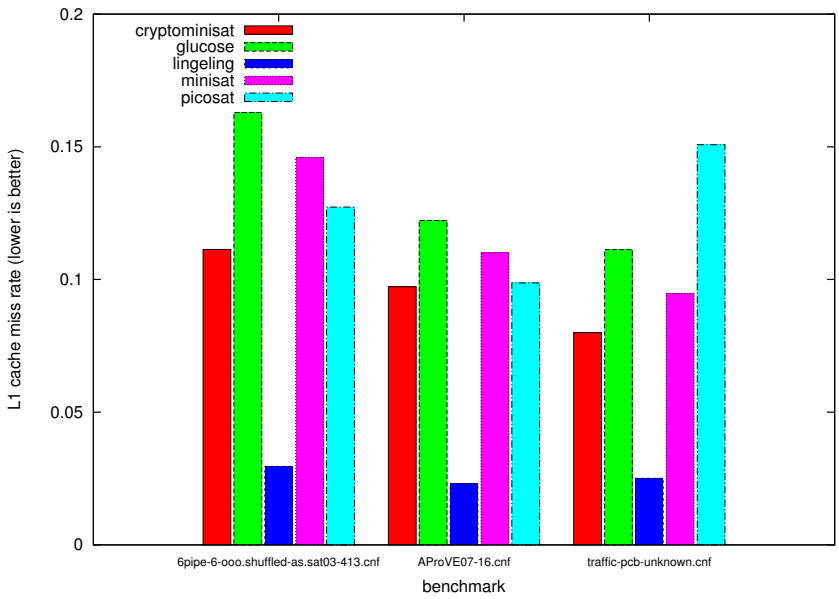

Figure 10: Level 1 (L1) cache miss rate of a single solver instance solving the benchmarks 6pip6-ooo.shuffled-as.sat03-413, AProVE07-16, and traffic-pcb-unknown.

	acsd			avcsd		
	min [%]	avg [%]	max [%]	min [%]	avg [%]	max [%]
amd-opteron-2350-8vcores	0.25	2.93	11.61	*no hyper threading*		
intel-i7-2600-8vcores	4.83	18.04	54.27	3.12	21.46	53.83
intel-i7-920-8vcores	1.95	8.92	31.20	7.14	17.49	28.60
intel-xeon-e5645-24vcores	1.52	3.84	17.02	2.86	7.16	11.78
intel-xeon-e7-80vcores	0.06	0.98	5.83	0.93	1.98	5.64
intel machines	0.06	7.94	54.27	0.93	12.02	53.83
cryptominisat	0.09	6.95	54.27	1.01	12.35	53.83
glucose	0.13	8.41	53.55	1.50	13.09	44.34
lingeling	0.08	5.34	35.23	1.25	11.31	50.41
minisat	0.06	6.44	54.27	0.93	11.74	53.83
picosat	0.12	8.06	47.16	1.19	12.24	40.81
total	0.06	6.94	54.27	0.93	12.02	53.83

Table 4: Average (virtual) slowdown per physical resp. virtual core restricted to the set of configurations described in the first column.

intel-i7-2600-8vcores to dynamic frequency scaling, which for both desktop processors was enabled. In general the server systems with many cores, and which arguably have been built to be particularly effective for multi-threaded applications have the smallest slowdown. The 80 (virtual) core system intel-xeon-e7-80vcores actually only slows-down by roughly 1%, when using an additional physical core.

Further, note that the average slowdown per virtual core is roughly twice as big as the average slowdown per physical core. This nicely fits the fact that two virtual cores share the same L1 cache, thus in essence the available L1 cache per virtual core is reduced by a factor of two compared to using only physical cores. The distribution of the slowdowns per core is rather skewed. For the largest benchmark narain-vpn-clauses-10 it ranges from 1.23% with Lingeling on intel-xeon-e7-80vcores to 54.27% with CryptoMiniSAT on intel-i7-2600-8vcores. Thus we plotted acsd and avcsd for the intel machines in Fig. 11. Between different SAT solvers the difference in slowdown per core is less than a factor of two, but Lingeling, the arguably most efficient SAT solver w.r.t. memory usage, has a small advantage in this regard if used as part of a portfolio.

3 Conclusion

This is the first detailed analysis on the expected worst-case slowdown for (plain) parallel portfolio SAT solvers running on shared memory multi-core architectures, when (A) copying instead of physically sharing clauses, and (B) ignoring synchronization overhead. In future work we plan to extend the analysis to measure slowdown w.r.t cache size. The next step is to develop strategies for dynamically measuring slowdown and using this information to control the number of workers in parallel SAT solvers.

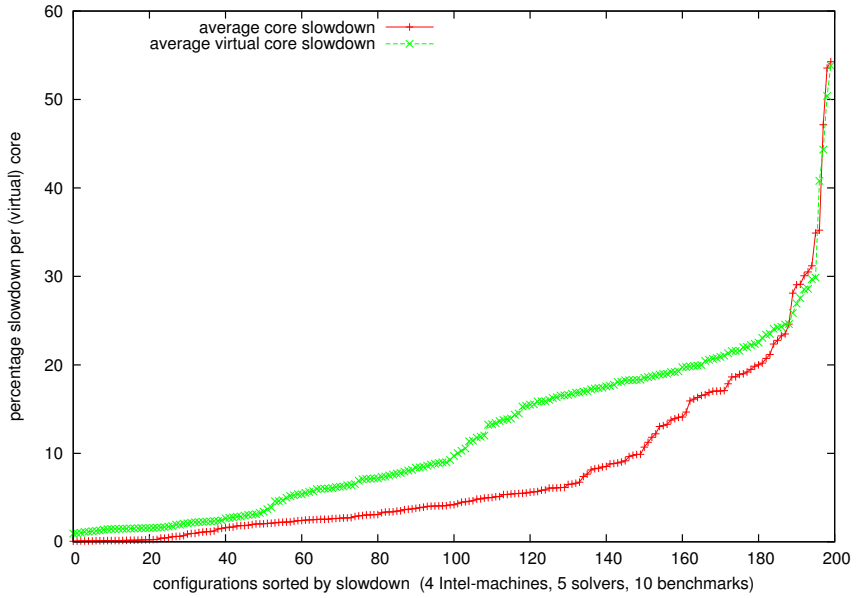

Figure 11: Distribution of both the average slowdown per physical core acsd in percent and the average slowdown of virtual cores avcsd for all benchmarks and all solvers, but only on the Intel machines with hyper-threading.

References

[1] Gille Audemard and Laurent Simon. GLUCOSE 2.1 in the SAT challenge 2012. In A. Balint, A. Belov, A. Diepold, S. Gerber, M. Järvisalo, and C. Sinz, editors, *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B, Univ. of Helsinki*, page 21, 2012.

[2] A. Biere. Lingeling and friends entering the SAT Challenge 2012. In A. Balint, A. Belov, A. Diepold, S. Gerber, M. Järvisalo, and C. Sinz, editors, *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B, Univ. of Helsinki*, pages 33–34, 2012.

[3] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[4] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. SAT'04*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[5] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[6] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. In *Proc. AAAI'12*. AAAI Press, 2012.

[7] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Proc. HVC'11*, volume 7261 of *LNCS*, pages 50–65. Springer, 2012.

[8] Antti Eero Johannes Hyvärinen and Norbert Manthey. Designing scalable parallel SAT solvers. In *Proc. SAT'12*, volume 7317 of *LNCS*, pages 214–227. Springer, 2012.

[9] Antti Eero Johannes Hyvärinen and Christoph M. Wintersteiger. Approaches for multi-core propagation in clause learning satisfiability solvers. Technical Report MSR-TR-2012-47, 2012.

[10] Daniel Le Berre. Sat4j, un moteur libre de raisonnement en logique propositionnelle, dec 2010. Habilitation à diriger des Recherches, Université d'Artois.

[11] Norbert Manthey. Parallel SAT solving - using more cores. In *Pragmatics of SAT (POS'11)*, 2011.

[12] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[13] Olivier Roussel. Description of ppfolio, 2011. SAT Competition 2011.

[14] Olivier Roussel. Behind the scene of solvers competitions: the "evaluation" experience. In Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe, editors, *Proc. of the 1st Intl. Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE'12)*, volume 873 of *CEUR Workshop Proceedings*, pages 66–77, 2012.

[15] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.

[16] M. Soos. CryptoMiniSAT. In *SAT Race solver descriptions*, 2010.

[17] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

[18] Lintao Zhang and Sharad Malik. Cache performance of SAT solvers: a case study for efficient implementation of algorithms. In *Proc. SAT'03*, volume 2919 of *LNCS*, pages 287–298. Springer, 2003.