PHYSICAL SHARING OF CLAUSES IN PORTFOLIO APPROACH PARALLEL SAT SOLVERS

por

Juan Luis Olate Hinrichs

Patrocinante: Leo Ferres

Propuesta de

MEMORIA DE TÍTULO

Departamento de Ingenería Informática y Ciencias de la Computación

de la

UNIVERSIDAD DE CONCEPCIÓN

Concepción, Chile

July, 2012

Table of Contents

Chapter 1

Introduction

One of the most well-known problems in computer science is the satisfiability (SAT) problem. This is because this was the first problem to be proved to be NP-complete [2], proof known as the Cook-Levin theorem[1]. One year later, in 1972, Karp proved in [4] that many common combinatorial problems could be reduced in polynomial time to instances of the SAT problem, thus drawing even more attention to SAT problems by the scientific community. Because many combinatorial problems can be reduced to SAT, it is not strange to find many practical problems with useful applications (such as circuit design and automatic theorem proving) that could be solved if there was an efficient algorithm to solve the SAT problem. Unfortunately, because of the NP-complete nature of SAT, such efficient algorithm that can solve the SAT problem in polinomial time has not been found yet, but instead, many researchers have improved the current SAT solving algorithms. Over the years, SAT solvers have shown impressive improvement, the first complete algorithm, the Davis Putnam algorithm [8], was very limited and could only handle small problems. Today, modern SAT solvers can handle instances with millions of variables, making such solvers suitable even for industrial application. In the next chapter we will point out the main features that have improved SAT solvers significantly in the past.

In the last decade parallel computing has become increasingly popular. As CPU manufacturers have found difficult and expensive to keep increasing the clock speed of processors, they have instead turn to increase the number of cores each chip has. Unfortunately, if the algorithms are not thought to be run in parallel, more cores will bring small improvements. This is the reason why there is a growing concern to parallelize algorithms so that they can take advantage of many-cores architectures of today's computers. In SAT solving it is no different. The annual SAT competition

[1] They both proved it independently.

[1], an event to determine which is the fastest SAT solver, has two main categories; sequential SAT solvers and parallel SAT solvers. In the last years parallel SAT solvers have outperformed sequential solvers in total wall clock time, so the interest in parallel solvers has grown with new designs and approaches explored for this kind of solvers. One of the most successful approaches to implement a parallel SAT solver is the *portfolio approach* with no *physical sharing* of information among cores. This approach basically runs different solvers in parallel, with each core keeping its own copy of the whole problem in memory, and wait for one of them to solve it up. No physical sharing refers to the fact that each core keeps its own copy of the problem's information, they do not access the same memory locations. This is a very simple and straight forward approach of parallelization, but we have also encountered one drawback to it: as we add more solvers to different cores of a single chip, the overall performance of the parallel solver decreases in around 20-40%. Preliminary experiments strongly suggest that this decrease in performance is caused by memory cache. Because all cores in a single chip share the same last level cache, and because each thread holds a copy of the original problem in memory, the more threads we add, the bigger the amount of data we have to handle. Since there is only one last level cache shared among all cores, the amount of total accesses from the last level cache to main memory will increase, since now there is a bigger volume of data to handle.

We plan to research the impact on cache performance of physically sharing clauses. Physical sharing obviously involves a more complex implementation, because you need to ensure data integrity when modifying the same memory locations from different threads, and also ensure that the correctness of algorithms is kept. The mechanisms to accomplish these requirements add an overhead to the solver which is not present when threads do not share information. On the other hand, as we will demonstrate in our future work, it is known that sharing information physically between threads usually increases cache performance, so there is a trade-off between both kind of approaches. The outcome of this trade-off is yet not clear and there are no serious studies about the cache performance of different parallel SAT solver implementations. So the goal of our work will be to study the impact of physically sharing clauses in

[1] www.satcompetition.org

cache performance of portfolio approach SAT solvers. We also need to stress out that it is already agreed that sharing clauses improves the overall performance of a parallel SAT solver [7], but it is not clear which sharing strategy is best for a better cache performance.

Chapter 2

Objectives

The objectives for this work are the following:

- Empirically quantify the decrease in performance, as you add threads, of portfolio parallel SAT solvers that do not share information physically.
- Prove with experimental results that this decrease is due to memory cache.
- Propose and implement different versions of a same CDCL (Conflict Driven Clause Learning, a modern type of SAT solver) parallel SAT solver to test the cache performance of different strategies of clause-sharing.
- Run standard benchmarks with the different versions of parallel SAT solvers, discuss the results and draw conclusions that could help future parallel SAT solver designers.

Chapter 3

Methodology

The following activities will be done to complete the work:

1. *Bibliographic research*: In this step we will gather and read articles and books of related SAT solving work. We are aiming to find the best performing parallel SAT solvers and get to know their internal designs.

2. *Testing of existing parallel solvers*: This work is based on experimental evidence that the best performing parallel SAT solvers suffer from considerable decrease in performance when adding more threads. We will perform various tests on some parallel SAT solvers to quantify and confirm the initial evidence.

3. *Cache tests*: After proving that there is a significant decrease in performance when adding threads to a typical portfolio approach SAT solver, we will prove through experimental results on existing solvers that the problem is memory cache.

4. *Propose a new solver*: We will implement different versions of a CDCL portfolio approach SAT solver to study the impact of different strategies of clause-sharing in cache performance.

5. *Result analysis*: We will run experiments on the different portfolio approach solver versions, analyze results and draw conclusions regarding cache performance.

6. *Report writting*: All the previous work will be written as a report to present as a graduation project.

Chapter 4

Background and Related Work

4.1 The SAT problem

Given a set of boolean variables Σ, a literal L is either a variable or the negation of a variable in Σ, and a *clause* is a disjunction of literals over distinct variables[1]. A propositional sentence is in *conjunctive normal form* (*CNF*) if it has the form $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$, where each α_i is a clause. The notation of sentences in CNF we will be using are sets. A clause $l_1 \vee l_2 \vee ... \vee l_m$, where l_i is a literal, can be expressed as the set $\{l_1, l_2, ..., l_m\}$. Furthermore, the CNF $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$ can be expressed as the set of clauses $\{\alpha_1, \alpha_2, ..., \alpha_n\}$. With these conventions, a CNF Δ is valid if Δ is the empty set: $\Delta = \emptyset$. A CNF Δ will be inconsistent if it contains the empty set: $\emptyset \in \Delta$. Given a CNF Δ, the SAT problem is answering the question: Is there an assignment of values for variables in Σ, such that Δ evaluates to true? The NP-completeness of this question lies in the combinatorial nature of the problem; to solve it one would need to try all different assignments of variables in Σ, the number of possible assignments grows exponentially as $|\Sigma|$ grows.

4.2 SAT solvers

All modern solvers today are CDCL (Conflict Driven Clause Learning) [6] SAT solvers. Given a CNF φ, a partial assignment of variables ν, Algorithm 1 outlines the general structure of a CDCL SAT solver, where x is a variable, v a truth value and β a *decision level*. We will shortly explain the main functions of this algorithm, but the details will be covered in the final work.

- UnitPropagation consists of iteratively deducting the truth value of variables. The values are deduced by logical reasoning on φ and ν.

[1] That all literals in a clause have to be over distinct variables is not standard.

Algorithm 1: Typical CDCL algorithm

```
   Input: A CNF φ and a variable assignment ν
 1 if (UnitPropagation(φ,ν)==CONFLICT) then
 2 |  return UNSAT.
 3 dl ← 0
 4 while (not AllVariablesAssigned(φ,ν)) do
 5 |  (x,v)=PickBranchingVariable(φ,ν)
 6 |  dl ← dl + 1
 7 |  ν ← ν ∪ {(x,v)}
 8 |  if (UnitPropagation(φ,ν)==CONFLICT) then
 9 |  |  β=ConflictAnalysis(φ,ν)
10 |  |  if (β < 0) then
11 |  |  |  return UNSAT
12 |  |  else
13 |  |  |  Backtrack(φ,ν,β)
14 |  |  |  dl ← β
15 return SAT
```

- PickBranchingVariable consists of selecting a variable to assign, and the respective value. Heavily relies in heuristics to pick variables.
- ConflictAnalysis consists of analyzing the most recent conflict (a conflict occurs when no variable assignment with the current ν can satisfy φ) and learning a new clause from the conflict. It returns the decision level to backtrack to.
- Backtrack undoes variable assignments as computed by ConflictAnalysis.
- AllVariablesAssigned tests whether all variables have been assigned a truth value.

4.3 Parallel SAT solvers

As mentioned before, some parallel SAT solvers have performed at the top of the last SAT competitions, but even though they all fall into the parallel solvers category, their parallel strategies and implementations vastly differ from each other. We mainly

classify parallel SAT solvers into two categories: Portfolio approach solvers and divide-and-conquer solvers.

The main idea behind portfolio approach solvers is the fact that different kinds of sequential solvers will perform differently for different kinds of SAT problems. The portfolio approach is a very straight forward strategy: They run a group of sequential solvers in parallel, each with different heuristic random values and/or different search strategies. The time they take to solve the problem will be the time of the fastest solver in the group of solvers running in parallel. Although all portfolio approach solvers share this same principle, they also have quite different kinds of implementations. We identify in this group the solvers that are pure portfolio approach, the ones that share clauses only logically, and the ones that share clauses physically and logically.

Solvers which are pure portfolio approach have the most simple design. They run completely independent solvers in parallel and wait for one of them to give an answer. Despite their simplicity, the solver ppfolio [9], a pure portfolio approach solver, was the winner of the crafted and random categories, and second place in the application category of the 2011 SAT competition of parallel solvers.

On the other hand, we have more elaborated portfolio approach solvers, which can also share clauses logically between their different solvers. One of the advantages of CDCL solvers is the fact that they can learn new lemmas as they solve a SAT problem. These new lemmas will provide additional information during the solution search, so that the solver doesn't fall into previous fruitless search paths (there are also some drawbacks to adding new lemmas, which are addressed by clause database cleanups). The idea is that different solvers running in parallel can share their learned lemmas so that they all benefit from what other solvers have learned and improve their own search. An example of these kind of solvers is ManySAT [3], which won the 2009 SAT competition in the parallel solver application category. ManySAT has its own sequential state-of-the-art SAT solver and runs different instances of it in parallel, using different VSIDS [11] heuristics (branching heuristics) and restart policies for each of it, both of which account for random factors in the solver. The difference with pure portfolio approach solvers, is that ManySAT also shares learned lemmas

between solving threads. It is called logical sharing of clauses, because the lemmas are passed as messages between threads and they never share the same physical information in memory. The advantage of logical sharing is that it is easier to implement message passing between threads, than having threads reading and modifying the same memory locations, which often requires locks that could hinder the overall solver performance. One of the best parallel performing solvers, Plingeling [1], also shares clauses logically. It is a very weak sharing though, since it only shares unit lemmas and it does so through messafe passing, using a master thread to coordinate messages between worker threads.

Portfolio approach solvers that share clauses physically have the same strategy as mentioned before, but they share clauses by allowing threads to access the same memory locations, instead of message passing. One solver in this category is SarTagnan [5], which shares clauses logically and physically.

Divide-and-conquer solvers do not try to run different solvers in parallel, they run one solving instance, but try to parallelize the search and divide it between the different threads. A common strategy to divide the search space is to use guiding paths. A guiding path is a partial assignment of variables in Σ, which restricts the search space of the SAT problem. A solver that divides its search space with guiding paths will assign threads to solve the CNF with the given partial assignment from the guiding path the thread was assigned with. Once a thread finishes searching a guiding path with no success, it can request another to keep searching. MiraXT [10] is a divide-and-conquer SAT solver which uses guiding paths. Moreover, different threads solving different guiding paths also share a common clause database, in which they store their learned lemmas. This is another example of physical clause sharing.

4.4 Memory cache

Computers today usually have three levels of memory cache and a main memory. The processor can have multiple cores in it and cores can run multiple threads in them. The difference between a core and a thread is that cores have separate copies of almost all the hardware resources. The cores can run independently unless they are using the same resource (for example the connection to the outside) at the same

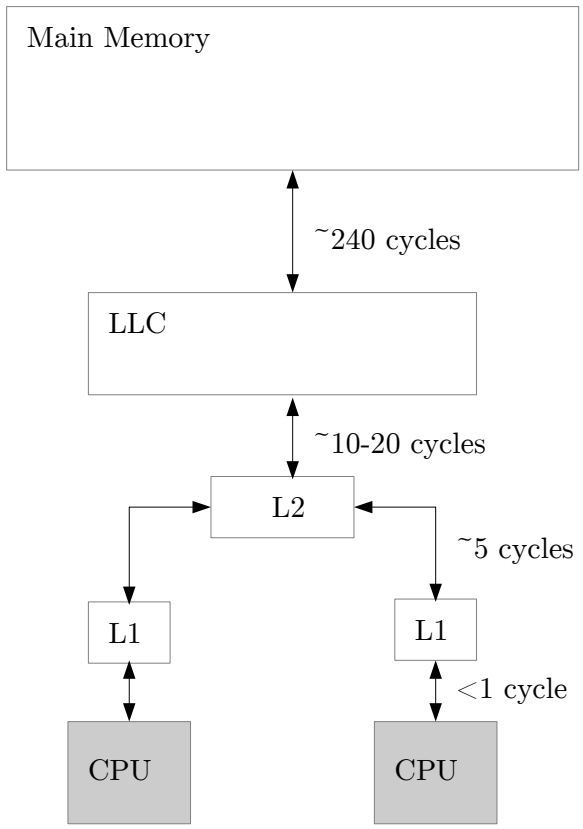

Figure 4.1: Typical memory hierarchy of modern computers.

time. Threads, on the other hand, share almost all of the processor resources. When a thread, which is running on a core, needs to fetch data, it first tries to look for it in the first level cache (the L1 cache[1]), if the data is not there, then it tries to find it in the level 2 cache (L2 cache). If the data is still not there, it then tries to fetch it from the level 3 cache (L3 cache) and if that fails too, it goes up to main memory to get the data. If it still isn't in main memory, then it has to retrieve it from the hard disk. We should notice that this hierarchy involves increasing fetch times as we go up. Getting data from the L1 cache is much faster than getting it from L2, and getting data from L2 is much faster than getting it from main memory. The problem with lower level memory is that, because of the technology and costs involved, they are much smaller. So the big picture is that at lower levels we have faster and smaller memories, and at higher levels we have massive and slower memory storages. Figure

[1]There is L1 data cache and also L1 instruction cache, we will be referring to L1 data cache

4.1 is a schematic of today's computer memory hierarchy. To get an idea of the times involved in accessing data from different memory storages, we present the following table of costs associated with hits and misses, for an Intel Pentium M:

To where	Cycles
Register	≤ 1
L1	3
L2	14
Main Memory	240

So we would like our programs to have a low number of accesses to main memory, since they are so expensive, and make a better use of the last level cache. This can usually be achieved in two ways: handling a lower amount of total data and improving our algorithms. The first one can be accomplished by sharing information between different threads of a parallel program, because if threads share information, then you do not need to duplicate it between threads and thus have a lower amount of total data to handle.

Chapter 5

Planification

Image 5.1 shows the planification of work for this project.

ITEM	Step to do	Period		Weeks	2012																							
		Start	End		December				January				March				April				May				June			
					S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
1	Bibliographic study	*01/12/11*	*20/12/11*	3																								
2	Existent SAT solvers experiments	*20/12/11*	*15/01/12*	3																								
3	Result analysis of existing solvers	*15/01/12*	*01/02/12*	2																								
4	Developing of a SAT solver	*01/03/12*	*30/04/12*	8																								
5	Result analysis of own SAT solver	*01/04/12*	*31/05/12*	8																								
6	Report elaboration	*01/12/11*	*30/06/12*	26																								

Figure 5.1: Planification of work to be done in the project.

Bibliography

[1] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1%2B2%2B3%2B6.pdf`, 2010. [Online; accessed 01-April-2012].

[2] S. A. Cook. The complexity of theorem-proving procedures. *proc. 3rd ann. ACM symp. on theory of computing*, pages 151–158, 1971.

[3] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.

[4] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[5] Stephan Kottler and Michael Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[6] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153.

[7] Ruben Martins, Vasco M. Manquinho, and Ins Lynce. An overview of parallel sat solving. *Constraints*, 17(3):304–347, 2012.

[8] Davis Putnam and Martin Hillary. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[9] O. Rousell. Description of ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf`, 2010. [Online; accessed 01-April-2012].

[10] Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: parallel sat solving with threads and message passing. *Journal of Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2009.

[11] Jaeheon Yi. The effect of vsids on sat solver performance. 2009.