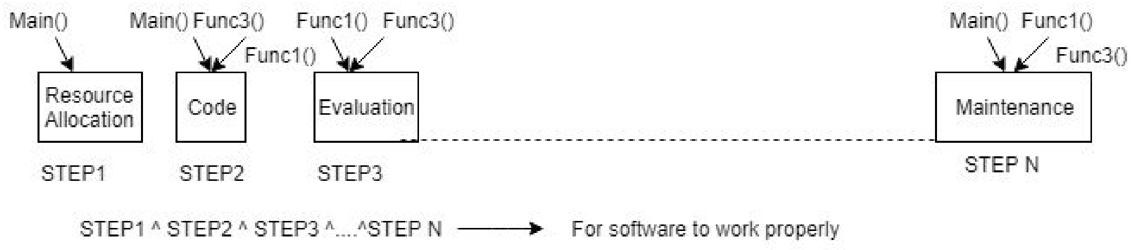
What is SAT
A problem of determining whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

Applications of SAT
(Hardware) Equivalence Checking, Asynchronous circuit synthesis (IBM), Software-Verification, Cryptanalysis. All problems can be converted into SAT. For instance Software verification involves steps.



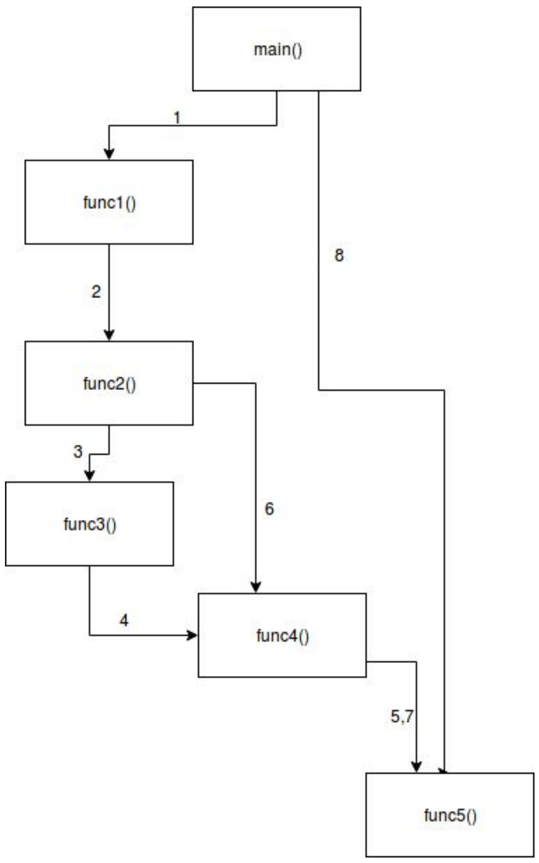
Consider every step as a clause. Every step consists of some substeps (which makes it work properly). Consider substep as a variable like functions as shown in the process of software development.

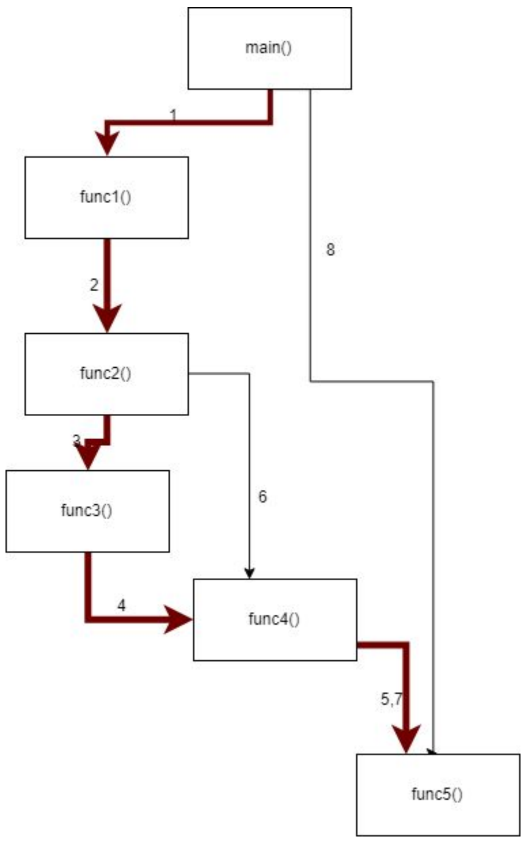
Need for SAT acceleration and Problem Statement
These applications seen above, involve millions of variables to be solved (like in circuit verification). This issue can be addressed and solved by accelerating SAT Solver either by hardware or software. Software optimizations and parallelism are being worked on, but simultaneously question arises : Why not try with hardware?

Insight of solution
There is a possibility of acceleration of SAT Solvers by a (hardware) chip. Instead of designing a hardware chip and testing it whether it really works, what we did was to emulate the results of hardware and feed these inputs to SAT solver and analyse the reduction in total execution time. This was achieved by using open source software GPROF. It analyses the flow of program and execution time at every call or instruction of a program.

GPROF Working
The information is stored by using -pg flag in gcc, which when enabled switches debugging and profiling on during the compilation of the program. This produces an executable file. This executable file when run produces a file "gmon.out" which contains the call graph information of the program. This is a binary file and is not readable by a user. So, gprof uses the symbol table and histograms and records every instance in a histogram. Histogram contains the address and the execution time of that instance. The symbol table consists of the function name and its corresponding assigned index. This information is traced by the profiler to print the call graph of the program in a readable format.
The profiler uses the approach of traversing the path from root to the last child (in that arc) in the graph which helps to figure out that what calls have been made to whom and what was the time taken inside a parent and the child separately. This information is stored in the form of arcs making parent as the start pointer. The information is then printed in such a way that there are two ways to discern the data flow : ***Flat Profile*** *: Shows the total amount of time each function takes.* ***Call Graph Profile*** *: Shows the execution time of every path possible individually.*

Technical Terms





Call Graph *Arc in callgraph*

Self time : The total execution time of the function excluding the time taken by its subroutines(children).
Child time : The time taken by the children of a function i.e. the difference between the time instant when it returns back to the caller and the time instant the callee.
Histogram : It is a data structure which consists of a low base address, high end address and the execution time which represents the callee and its corresponding caller.

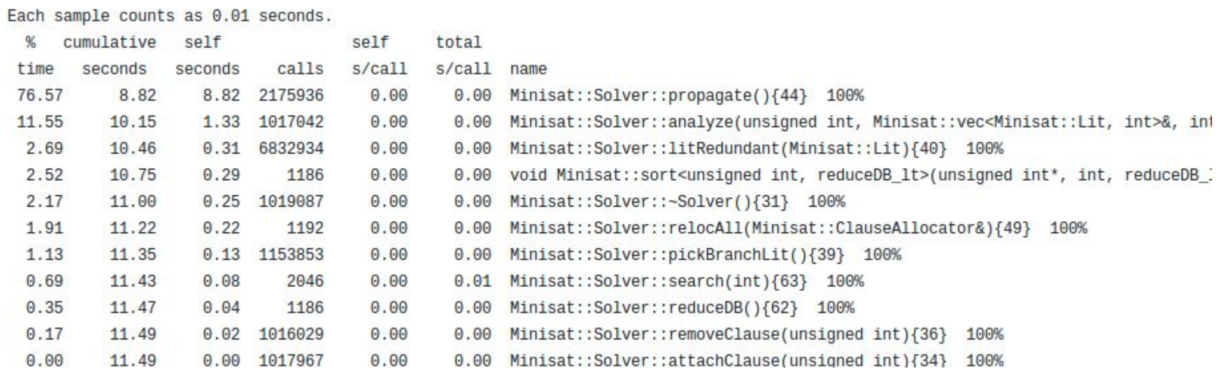
Solution
Suppose, the part of a program can be replaced by some coding paradigm which helps to reduce the execution time of that particular block of code, then we have to replace that time and see how execution pattern gets affected. This requires a change in the record of histogram corresponding to the block. Identifying the block can be done by 2 ways :
1.) Address - The address of the block can be found via printing and the corresponding block can then be changed. But there is one flaw here and that is if we run the program next time, we are not sure whether the program will load in the exact same address or not. This is what we call as platform dependency where every instance is changed.
2.) Index - The above idea seems to be a bit infeasible. So in this situation we analyze that what is not platform dependent. Address, time etc. are all machine dependent, but the call graph is not. So we assign the index to every call and then change the time for those calls(corresponding to the block we need to change). Index is assigned to every call and the respective id can be printed so that the user can identify the call by noticing its parent and respective child. The time corresponding to this id can be changed in the code. This change will be propagated and the final execution time will also get changed by the same amount.

The algorithm described below is edited in the file "hist.c" which contains the histogram data structure that is built from "gmon.out" file. The id that needs to be changed is known by printing it for every call initially and then figuring out which id corresponds to which call. A table of id and corresponding call is printed which helps to identify which id needs to be changed by looking at which call has to be modified. Ids are not machine dependent which makes it a feasible way of changing time of a specific block.

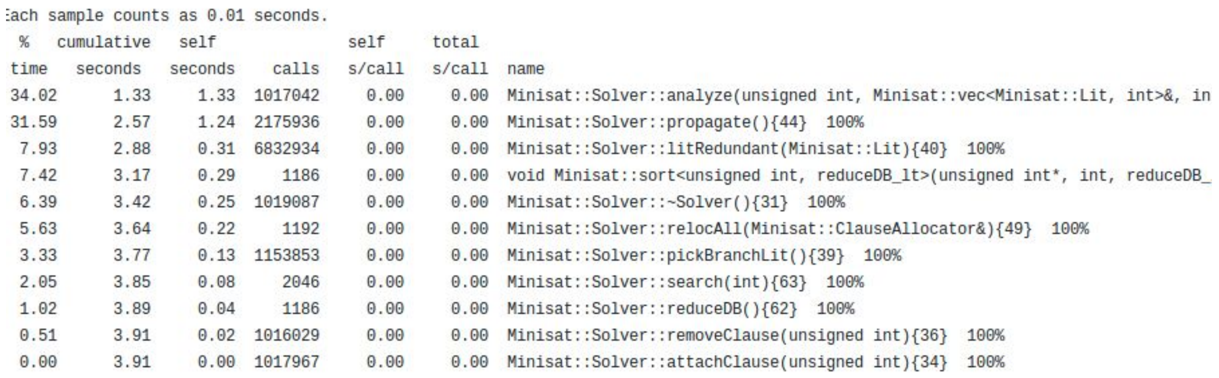
Algorithm for changing execution time of a certain part of code
Assume min to be lower limit and max as upper limit of block that needs to be replaced. c be the changed value. In case of different values for every id, it needs to be specified in the form of an array of values.

```
id = 0, total_bin = 0, count = 0
If { id>=min-1 and id <= max-1 }{
    total_bin += time
    Time = c
}
count = count+1
```

Instance of profiling minisat



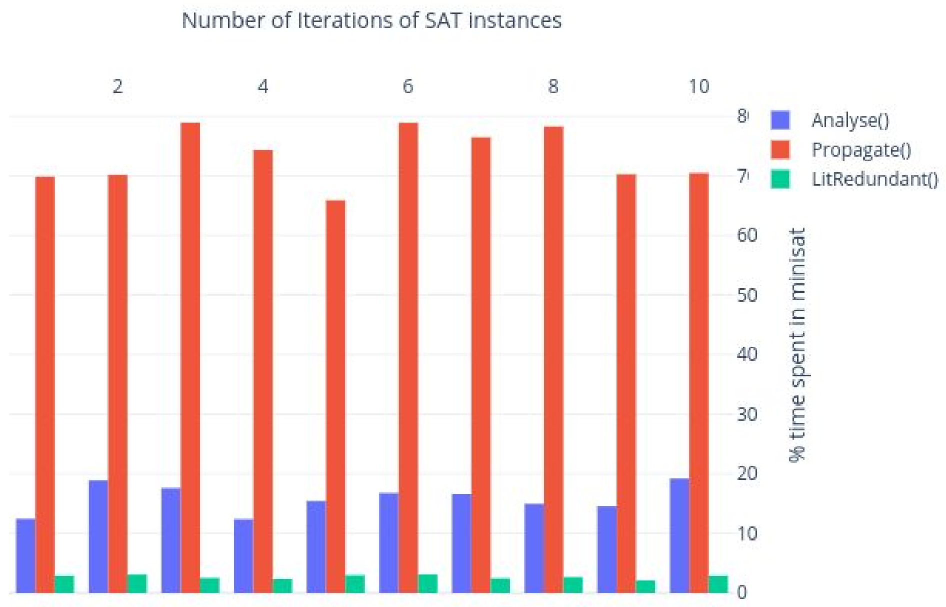
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 76.57      8.82     8.82  2175936     0.00     0.00  Minisat::Solver::propagate(){44}  100%
 11.55     10.15     1.33  1017042     0.00     0.00  Minisat::Solver::analyze(unsigned int, Minisat::vec<Minisat::Lit, int>&, in
  2.69     10.46     0.31  6832934     0.00     0.00  Minisat::Solver::litRedundant(Minisat::Lit){40}  100%
  2.52     10.75     0.29     1186     0.00     0.00  void Minisat::sort<unsigned int, reduceDB_lt>(unsigned int*, int, reduceDB_
  2.17     11.00     0.25  1019087     0.00     0.00  Minisat::Solver::~Solver(){31}  100%
  1.91     11.22     0.22     1192     0.00     0.00  Minisat::Solver::relocAll(Minisat::ClauseAllocator&){49}  100%
  1.13     11.35     0.13  1153853     0.00     0.00  Minisat::Solver::pickBranchLit(){39}  100%
  0.69     11.43     0.08     2046     0.00     0.01  Minisat::Solver::search(int){63}  100%
  0.35     11.47     0.04     1186     0.00     0.00  Minisat::Solver::reduceDB(){62}  100%
  0.17     11.49     0.02  1016029     0.00     0.00  Minisat::Solver::removeClause(unsigned int){36}  100%
  0.00     11.49     0.00  1017967     0.00     0.00  Minisat::Solver::attachClause(unsigned int){34}  100%
```

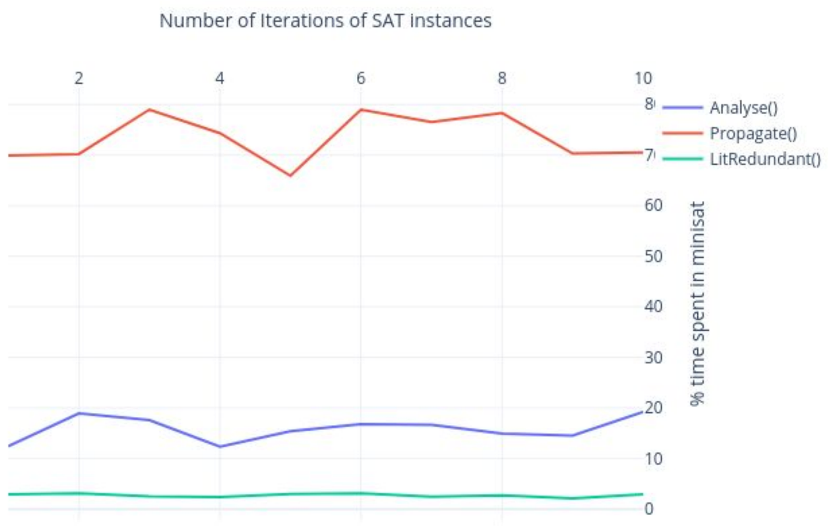


```
:ach sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
34.02      1.33     1.33  1017042     0.00     0.00  Minisat::Solver::analyze(unsigned int, Minisat::vec<Minisat::Lit, int>&, in
31.59      2.57     1.24  2175936     0.00     0.00  Minisat::Solver::propagate(){44}  100%
 7.93      2.88     0.31  6832934     0.00     0.00  Minisat::Solver::litRedundant(Minisat::Lit){40}  100%
 7.42      3.17     0.29     1186     0.00     0.00  void Minisat::sort<unsigned int, reduceDB_lt>(unsigned int*, int, reduceDB_
 6.39      3.42     0.25  1019087     0.00     0.00  Minisat::Solver::~Solver(){31}  100%
 5.63      3.64     0.22     1192     0.00     0.00  Minisat::Solver::relocAll(Minisat::ClauseAllocator&){49}  100%
 3.33      3.77     0.13  1153853     0.00     0.00  Minisat::Solver::pickBranchLit(){39}  100%
 2.05      3.85     0.08     2046     0.00     0.00  Minisat::Solver::search(int){63}  100%
 1.02      3.89     0.04     1186     0.00     0.00  Minisat::Solver::reduceDB(){62}  100%
 0.51      3.91     0.02  1016029     0.00     0.00  Minisat::Solver::removeClause(unsigned int){36}  100%
 0.00      3.91     0.00  1017967     0.00     0.00  Minisat::Solver::attachClause(unsigned int){34}  100%
```

The total time taken by minisat is 11 sec out of which propagate took around 8.8 sec which accounts for 76% of total time implying that propagate() accounts for such a huge execution time.
This analysis led us to emulate the propagate() function and replace the bin_count with a virtual value = 1, and the respective results were obtained which indicates a significant improvement of 36% in the execution time of minisat provided there is a way to parallelise propagate() function. This suggests that the hardware which can support propagate() function parallelism needs to be designed.

Which function should be parallelised?





Conclusion

Changing the time of a particular block of code could be done in several ways but considering the factor that code should be portable, reference to a call by its id turned out to be a suitable method. Adding an extra attribute to a data structure is a small change that can profile a program according to the user needs. This strategy can be used to test how much does emulated time of a block of code affects the total execution time. If its an expected number, then the hardware design for the same can be implemented on the chip instead of first designing it and then finding it to be futile. This can improve the hardware design process and also can help see whether any kind of parallelism is helping in execution as we saw in minisat.