

# COMPILER PROJECT DOCUMENTATION

- NAME: SAJAN. S. A
- REG NO: 2017207038
- M.E – COMPUTER SCIENCE AND ENGINEERING
- TITLE: CONSTANT PROPAGATION
- COMPILER 1: GCC
- COMPILER 2: LLVM

## **PROBLEM STATEMENT:**

**Constant propagation** is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values.

## **OPTIMISATION DEFINITION:**

**Optimization = Analysis + Transformation**

**Transformation** may make the code more efficient. But it may also change the meaning of the code. **Analysis** determines when transformations can be applied safely.

Constant propagation did not remove any instructions, but it enabled **dead-code elimination** which did remove instructions. Together, the optimizations made the code more efficient. Hence, constant propagation is known as an **enabling optimization**.

We will explore two analyses:

- 1. Liveness analysis:** Can deduce when dead-code elimination is safe. Also used in register allocation. Is a “backward analysis”.
- 2. Reaching-definitions analysis:** Can deduce when constant propagation is safe. Is a “forward analysis”.

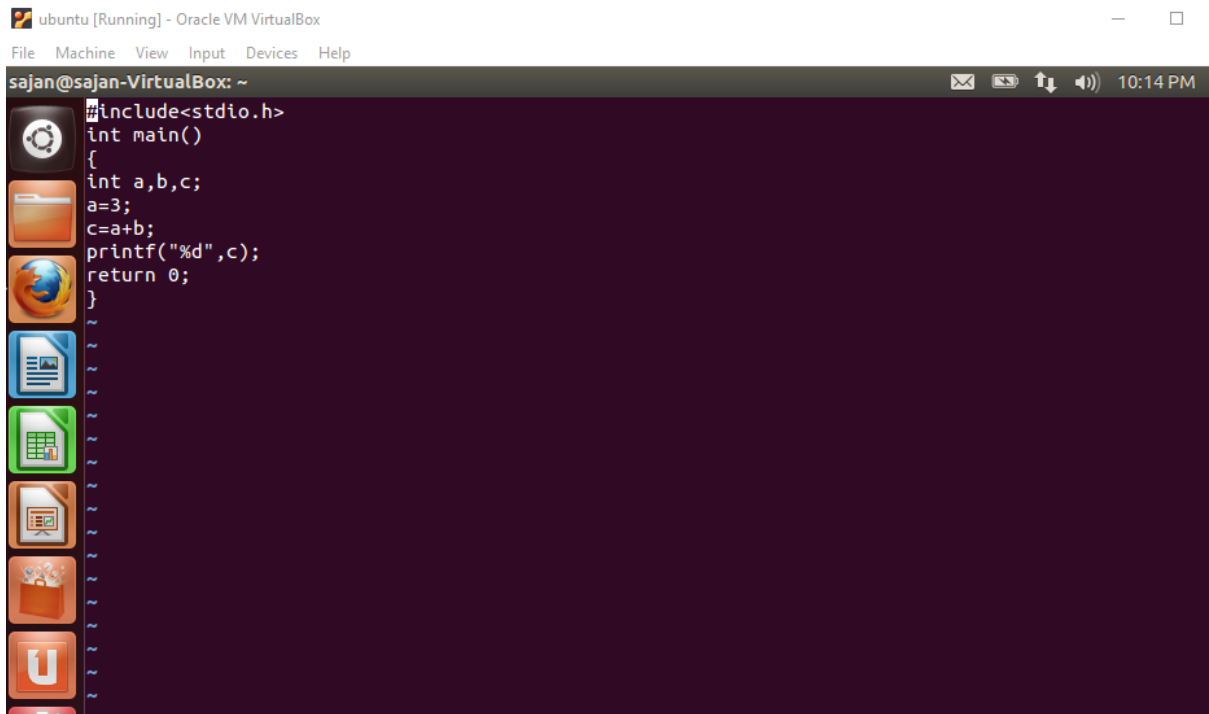
Using the results of analysis, to precisely and safely, define constant propagation.

# IMPLEMENTATION:

## 1. GCC COMPILER:

STEP 1: OPEN THE BASH TERMINAL IN LINUX OS

STEP 2: TYPE A PROGRAM IN WHICH CONSTANT PROPAGATION CAN BE IMPLEMENTED



The screenshot shows a terminal window titled 'ubuntu [Running] - Oracle VM VirtualBox'. The terminal prompt is 'sajan@sajan-VirtualBox: ~'. The user has entered the following C code:

```
#include<stdio.h>
int main()
{
    int a,b,c;
    a=3;
    c=a+b;
    printf("%d",c);
    return 0;
}
```

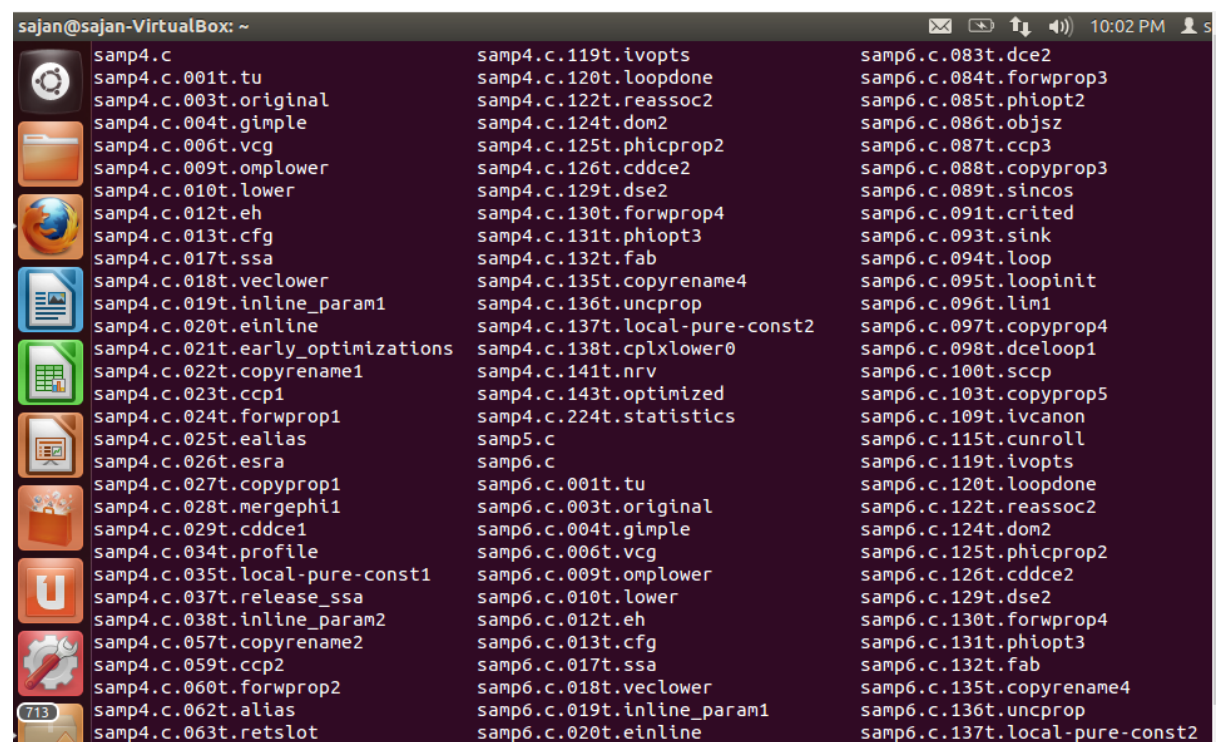
The terminal window has a dark purple background. On the left side, there is a vertical dock with several application icons: a terminal icon, a file manager icon, a web browser icon, a document icon, a spreadsheet icon, a presentation icon, a folder icon, and a 'U' icon. The top of the window shows a menu bar with 'File', 'Machine', 'View', 'Input', 'Devices', and 'Help'. The top right corner shows system icons for network, volume, and power, along with the time '10:14 PM'.

STEP 3: CHECK WHETHER THE PROGRAM EXECUTES WITHOUT ANY ERROR

STEP 4: CONVERT SOURCE CODE INTO OPTIMIZED CODE USING THE COMMAND “ gcc filename.c -O -fdump-tree-all “

```
sajan@sajan-VirtualBox:~$ gcc samp6.c -O -fdump-tree-all
```

STEP 5: VIEW ALL OPTIMIZED CODE USING “ ls “ COMMAND



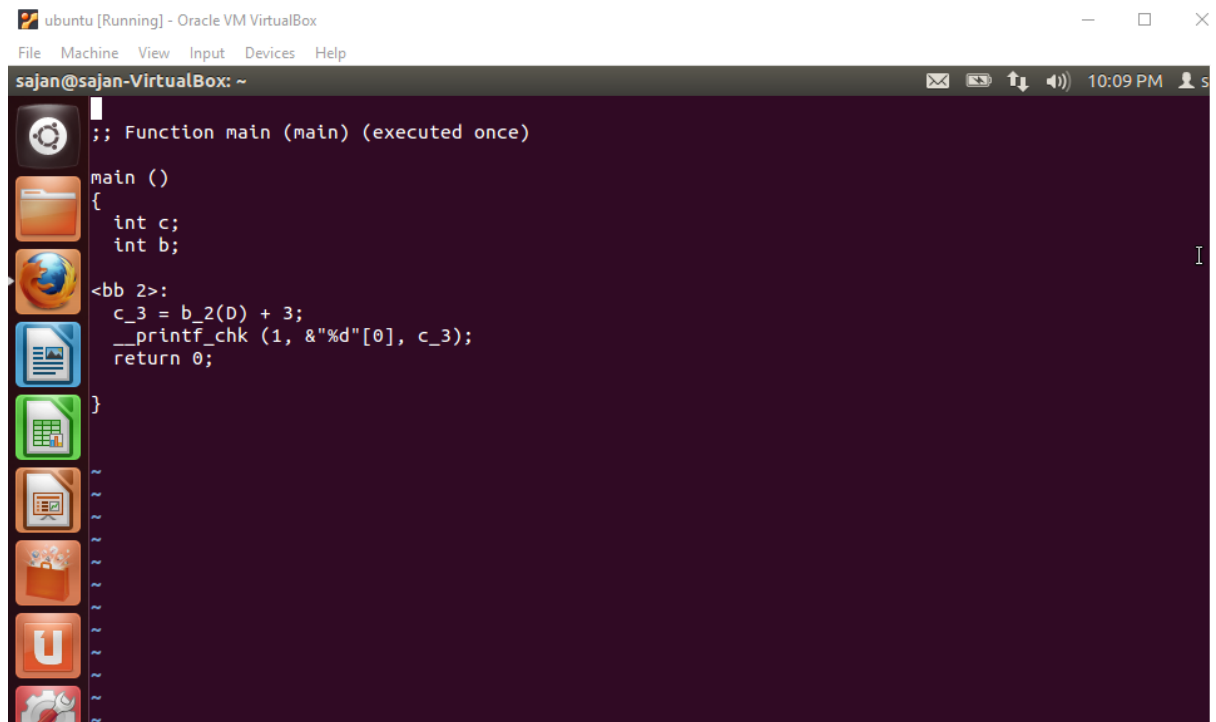
```
sajan@sajan-VirtualBox: ~  
samp4.c  
samp4.c.001t.tu  
samp4.c.003t.original  
samp4.c.004t.gimple  
samp4.c.006t.vcg  
samp4.c.009t.omplower  
samp4.c.010t.lower  
samp4.c.012t.eh  
samp4.c.013t.cfg  
samp4.c.017t.ssa  
samp4.c.018t.veclower  
samp4.c.019t.inline_param1  
samp4.c.020t.einline  
samp4.c.021t.early_optimizations  
samp4.c.022t.copyrename1  
samp4.c.023t.ccp1  
samp4.c.024t.forwprop1  
samp4.c.025t.ealias  
samp4.c.026t.esra  
samp4.c.027t.copyprop1  
samp4.c.028t.mergephi1  
samp4.c.029t.cddce1  
samp4.c.034t.profile  
samp4.c.035t.local-pure-const1  
samp4.c.037t.release_ssa  
samp4.c.038t.inline_param2  
samp4.c.057t.copyrename2  
samp4.c.059t.ccp2  
samp4.c.060t.forwprop2  
samp4.c.062t.alias  
samp4.c.063t.retslot  
samp4.c.119t.ivopts  
samp4.c.120t.loopdone  
samp4.c.122t.reassoc2  
samp4.c.124t.dom2  
samp4.c.125t.phicprop2  
samp4.c.126t.cddce2  
samp4.c.129t.dse2  
samp4.c.130t.forwprop4  
samp4.c.131t.phiopt3  
samp4.c.132t.fab  
samp4.c.135t.copyrename4  
samp4.c.136t.uncprop  
samp4.c.137t.local-pure-const2  
samp4.c.138t.cplxlower0  
samp4.c.141t.nrv  
samp4.c.143t.optimized  
samp4.c.224t.statistics  
samp5.c  
samp6.c  
samp6.c.001t.tu  
samp6.c.003t.original  
samp6.c.004t.gimple  
samp6.c.006t.vcg  
samp6.c.009t.omplower  
samp6.c.010t.lower  
samp6.c.012t.eh  
samp6.c.013t.cfg  
samp6.c.017t.ssa  
samp6.c.018t.veclower  
samp6.c.019t.inline_param1  
samp6.c.020t.einline  
samp6.c.083t.dce2  
samp6.c.084t.forwprop3  
samp6.c.085t.phiopt2  
samp6.c.086t.objsz  
samp6.c.087t.ccp3  
samp6.c.088t.copyprop3  
samp6.c.089t.sincos  
samp6.c.091t.critcd  
samp6.c.093t.sink  
samp6.c.094t.loop  
samp6.c.095t.loopinit  
samp6.c.096t.lim1  
samp6.c.097t.copyprop4  
samp6.c.098t.dceloop1  
samp6.c.100t.sccp  
samp6.c.103t.copyprop5  
samp6.c.109t.ivcanon  
samp6.c.115t.cunroll  
samp6.c.119t.ivopts  
samp6.c.120t.loopdone  
samp6.c.122t.reassoc2  
samp6.c.124t.dom2  
samp6.c.125t.phicprop2  
samp6.c.126t.cddce2  
samp6.c.129t.dse2  
samp6.c.130t.forwprop4  
samp6.c.131t.phiopt3  
samp6.c.132t.fab  
samp6.c.135t.copyrename4  
samp6.c.136t.uncprop  
samp6.c.137t.local-pure-const2
```

STEP 6: VARIOUS OPTIMIZED CODES ARE AVAILABLE WHICH CAN BE VIEWED USING “ vi required\_filename “.

## STEP 7: IN-ORDER TO VIEW CONSTANT PROPAGATION USE COMMAND

```
sajan@sajan-VirtualBox:~$ vi samp4.c.066t.copyprop2
```

## STEP 8: THE OPTIMIZED CODE CAN BE VIEWED



The screenshot shows a VirtualBox window titled 'ubuntu [Running] - Oracle VM VirtualBox'. The window contains a terminal window titled 'sajan@sajan-VirtualBox: ~'. The terminal displays the following C code, which is the optimized output of the compiler:

```
;; Function main (main) (executed once)
main ()
{
  int c;
  int b;

  <bb 2>;
  c_3 = b_2(0) + 3;
  __printf_chk (1, &"%d"[0], c_3);
  return 0;
}
```

The terminal window also shows a sidebar with various application icons on the left and a status bar at the bottom right indicating the time as 10:09 PM.

## 2. LLVM (IMPLEMENTATION USING LLVM AND CLANG)

LLVM is a compiler infrastructure designed as a set of reusable libraries with well defined interfaces. It is Open Source and a framework that comes with a lot of tools to compile and optimize code .

LLVM is fairly easy to install. For a quick overview on the process, we recommend: <http://llvm.org/releases/3.4/docs/GettingStarted.html>

### INSTALLATION STEPS OF LLVM:

```
$> svn co http://llvm.org/svn/llvm-project/llvm/tags/RELEASE_34/final llvm
$> cd llvm/tools
$> svn co http://llvm.org/svn/llvm-project/cfe/tags/RELEASE_34/final clang
$> cd ../projects/
$> svn co http://llvm.org/svn/llvm-project/compiler-rt/tags/RELEASE_34/final
compiler-rt
$> cd ../tools/clang/tools/
$> svn co http://llvm.org/svn/llvm-project/clang-tools-extra/tags/RELEASE_34/
final extra
```

### INSTALL CLANG/CLANG++:

CLANG ACTS LIKE FRONT END AND LLVM ACTS LIKE BACK END. CLANG/CLANG++ IS VERY COMPETITIVE WHEN COMPARED WITH GCC AS CLANG HAVE FASTER COMPILATION TIMES.

## VARIOUS TOOLS AVAILABLE UNDER LLVM FRAMEWORK:

```
$> cd llvm/Debug+Asserts/bin
$> ls
FileCheck          count          llvm-dis        llvm-stress
FileUpdate         diagtool      llvm-dwarfdump  llvm-symbolizer
arcmt-test         fpcmp         llvm-extract    llvm-tblgen
bugpoint           llc          llvm-link       macho-dump
c-arcmt-test       lli          llvm-lit        modularize
c-index-test       lli-child-target llvm-lto        not
clang             llvm-PerfectSf llvm-mc          obj2yaml
clang++          llvm-ar       llvm-mcmarkup   opt
llvm-as            llvm-nm       pp-trace        llvm-size
clang-check        llvm-bcanalyzer llvm-objdump     rm-cstr-calls
clang-format       llvm-c-test   llvm-ranlib     tool-template
clang-modernize    llvm-config   llvm-readobj    yaml2obj
clang-tblgen       llvm-cov      llvm-rtdyld     llvm-diff
clang-tidy
```

## COMPILATION OF C/C++ PROGRAMS:

```
$> echo "int main() {return 42;}" > test.c
$> clang test.c
$> ./a.out
$> echo $?
42
```

## COMPILING LLVM:

Once you have gotten all the files, via svn, you must compile LLVM. There are more than one way to compile it.

If you want to do it quickly, you can configure LLVM with the option `--enable-optimized` set. Otherwise, a default compilation, with debug symbols, will be performed.

```
$> cd ~/Programs/llvm # that's where I have downloaded it.

$> mkdir build

$> ../configure

$> make -j16          # Assuming you have more than 1 core.
```

## OPTIMIZATION:

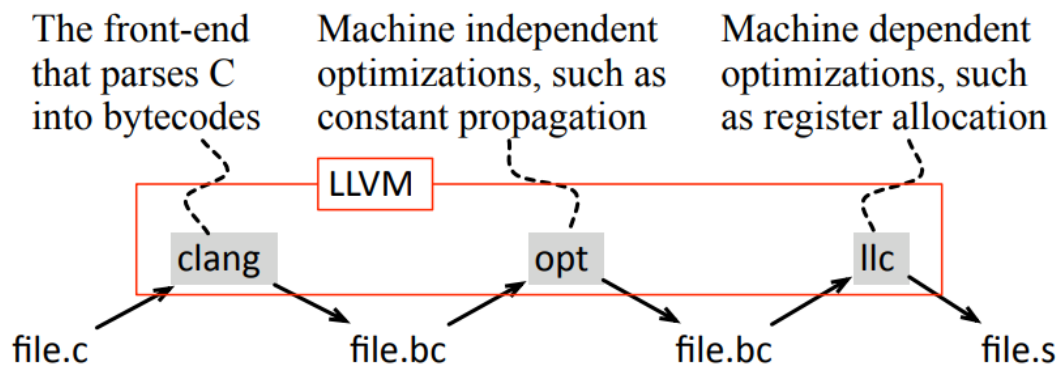
The opt tool, available in the LLVM toolbox, performs machine independent optimization.

There are many optimizations available through opt. To have an idea, type opt --help.

```
$> opt --help
Optimizations available:
  -adce           - Aggressive Dead Code Elimination
  -always-inline  - Inliner for always_inline functions
  -break-crit-edges - Break critical edges in CFG
  -codegenprepare - Optimize for code generation
  -constmerge     - Merge Duplicate Global Constants
  -constprop      - Simple constant propagation
  -correlated-propagation - Value Propagation
  -dce            - Dead Code Elimination
  -deadargelim    - Dead Argument Elimination
  -die            - Dead Instruction Elimination
  -dot-cfg        - Print CFG of function to 'dot' file
  -dse            - Dead Store Elimination
  -early-cse      - Early CSE
  -globaldce      - Dead Global Elimination
  -globalopt      - Global Variable Optimizer
  -gvn            - Global Value Numbering
  -indvars        - Induction Variable Simplification
  -instcombine    - Combine redundant instructions
  -instsimplify   - Remove redundant instructions
  -ipconstprop    - Interprocedural constant propagation
  -loop-reduce    - Loop Strength Reduction
```



## Working concept of clang and llvm:



## LEVELS OF OPTIMIZATION

Like gcc, clang supports different levels of optimizations, e.g., -**00** (default), -**01**, -**02** and -**03**

**llvm-as** is the LLVM assembler. It reads a file containing human-readable LLVM assembly language, translates it to LLVM bytecode, and writes the result into a file or to standard output.

To find out which optimization each level uses, you can try:

```
$> llvm-as < /dev/null | opt -O3 -disable-output -debug-pass=Arguments
```

## PROGRAM AND EXPLANATION OF CONSTANT PROPAGATION:

One of the most basic optimizations that **opt** performs is to map memory slots into variables. This optimization is very useful, because the clang front end maps every variable to memory.

```
$> clang -c -emit-llvm const.c -o const.bc  
  
$> opt -view-cfg const.bc
```

Code:

```
void main() {
    int c1 = 17;
    int c2 = 25;
    int c3 = c1 + c2;
    printf("Value = %d\n", c3);
}
```

After performing `opt -view-cfg const.bc`, the cfg will be,

```
%0:
%1 = alloca i32, align 4
%c1 = alloca i32, align 4
%c2 = alloca i32, align 4
%c3 = alloca i32, align 4
store i32 0, i32* %1
store i32 17, i32* %c1, align 4
store i32 25, i32* %c2, align 4
%2 = load i32* %c1, align 4
%3 = load i32* %c2, align 4
%4 = add nsw i32 %2, %3
store i32 %4, i32* %c3, align 4
%5 = load i32* %c3, align 4
%6 = call @printf(...)
%7 = load i32* %1
ret i32 %7
```

Now perform `opt-mem2reg const.bc`

```
$> opt -mem2reg const.bc > const.reg.bc

$> opt -view-cfg const.reg.bc
```

Now cfg changes as register storage is implemented,

```
%0:
%1 = add nsw i32 17, 25
%2 = call @printf(...), i32 %1
ret i32 0
```

## CONSTANT PROPAGATION:

We can fold the computation of expressions that are known at compilation time with the constprop pass.

```
$> opt -constprop const.reg.bc > const.cp.bc  
  
$> opt -view-cfg const.cp.bc
```

now cfg changes as like,

```
%0:  
%1 = call i32 @printf(4, i32 42)  
ret i32 0
```

Which is the optimized code.