Loop Quasi-Invariant Chunk Motion

by peeling with statement composition

Thomas Rubiano

From an idea of L. Kristiansen
supervised by J. Y. Moyen
in collaboration with T. Seiller
funded by the Elica project

ICC implementation in Compilers

ICC techniques in a loop optimization

Thomas Rubiano

From an idea of L. Kristiansen
supervised by J. Y. Moyen
in collaboration with T. Seiller
funded by the Elica project

Welcome to the real world !

- This work shows that we can do something in real ~~world~~ languages
- Using data flow analysis seen in ICC papers (“size-change graphs” and “*mwp*-bounds”)

Motivations

Loop-Invariant

```
int x=rand()%100;
while(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
```

Loop-Invariant

```
int x=rand()%100;
while(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
```

```
int x=rand()%100;
if(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
while(i<100){
   use(y);
   i=i+1;
}
```

Loop-Invariant

```
int x=rand()%100;
while(i<100){
    y=x+x; //invariant
    use(y);
    i=i+1;
}
```

```
int x=rand()%100;
if(i<100){
    y=x+x; //invariant
    use(y);
    i=i+1;
}
while(i<100){
    use(y);
    i=i+1;
}
```

- Obviously already in compilers : called “*Loop Invariant Code Motion*” (**32111** instructions hoisted over **3808** loops in *vim*. . .)

Loop-Quasi-Invariants

```
while(i<100){
   z=y*y; //quasi-invariant
   use(z);
   y=x+x; //invariant
   use(y);
   i=i+1;
}
```

Loop-Quasi-Invariants

```
while(i<100){
    z=y*y; //quasi-invariant
    use(z);
    y=x+x; //invariant
    use(y);
    i=i+1;
}
```

```
if(i<100){
  z=y*y;
  use(z);
  y=x+x;
  use(y);
  i=i+1;
}
if(i<100){
  z=y*y;
  use(z);
  use(y);
  i=i+1;
}
while(i<100){
  use(z);
  use(y);
  i=i+1;
}
```

Loop-Quasi-Invariants

```
while(i<100){
    z=y*y; //quasi-invariant
    use(z);
    y=x+x; //invariant
    use(y);
    i=i+1;
}
```

```
if(i<100){
  z=y*y;
  use(z);
  y=x+x;
  use(y);
  i=i+1;
}
if(i<100){
  z=y*y;
  use(z);
  use(y);
  i=i+1;
}
while(i<100){
  use(z);
  use(y);
  i=i+1;
}
```

- Peeling is removing instructions out of the loop while unrolling it
- Done but not done. . .

  . . . regarding to invariants (only loop size and trip count) !

Loop-Quasi-Invariant Chunks

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

Loop-Quasi-Invariant Chunks

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

Loop-Quasi-Invariant Chunks

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

Loop-Quasi-Invariant Chunks

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

Loop-Quasi-Invariant Chunks

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

- Definitely new !

Introduction

A WHILE-language

(Variables)	X	$::=$	$X_1 \mid X_2 \mid X_3 \mid \ldots \mid X_n$
(Expression)	exp	$::=$	$X \mid \texttt{op}\,(exp, \ldots, exp)$
(Command)	com	$::=$	$X{=}exp \mid com\,;com \mid \texttt{skip} \mid$
			$\texttt{while}\ exp\ \texttt{do}\ com \mid$
			$\texttt{if}\ exp\ \texttt{then}\ com\ \texttt{else}\ com \mid$

A WHILE-language

(Variables)	X	::=	$X_1 \mid X_2 \mid X_3 \mid \ldots \mid X_n$
(Expression)	exp	::=	$X \mid \texttt{op}(exp, \ldots, exp)$
(Command)	com	::=	$X{=}exp \mid com\,;com \mid \texttt{skip} \mid$
			$\texttt{while}\ exp\ \texttt{do}\ com \mid$
			$\texttt{if}\ exp\ \texttt{then}\ com\ \texttt{else}\ com \mid$
			$\texttt{use}(X_1, \ldots, X_n)$

Quasi-Invariants

- A quasi-invariant is a variable with a value which does not change after a certain number of loop execution
- A degree of invariance is the number of time we need to iterate the loop until the variable is stable

```
while(i<100){
    z=y*y; // 2
    use(z);
    y=x+x; // 1
    use(y);
    i=i+1;
}
```

Theory : Data Flow Graph

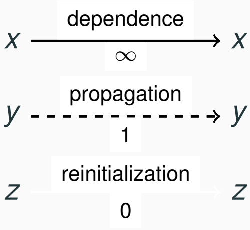
Definition (Data Flow Graph)

A Data Flow Graph represents dependencies between variables as a bipartite graph as below.

$$\mathtt{C} := [x = x + 1;$$

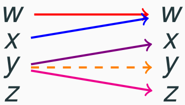
$$y = y;$$

$$z = 0;]$$



Matrix algebra

A *Data Flow Graph* for a command $\mathtt{C}$ is a $n \times n$ matrix over the semi-ring $\{0, 1, \infty\}$.

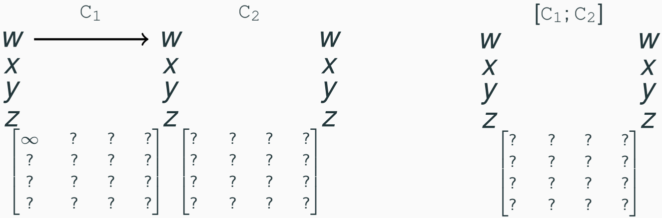


$$\rightarrow \quad \begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \\ 0 & \infty & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We write $+$ and $\times$ the two operations (*max*, $\times$).

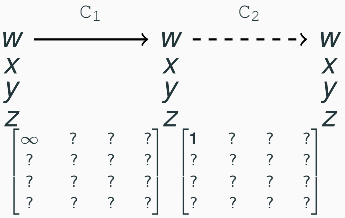
Multipath and Composition (*à la "size-change Termination"*)

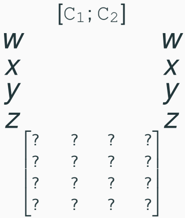
Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.



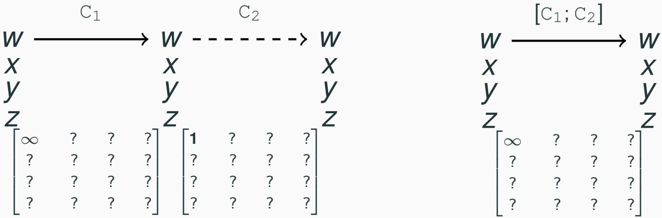
Multipath and Composition (*à la “size-change Termination”*)

Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \dots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\dots\mathrm{M}(\mathtt{C}_n)$.



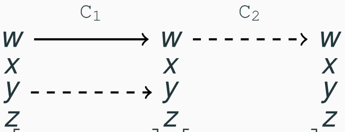


Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.

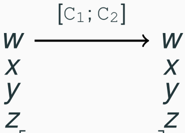


Multipath and Composition (*à la "size-change Termination"*)

Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.



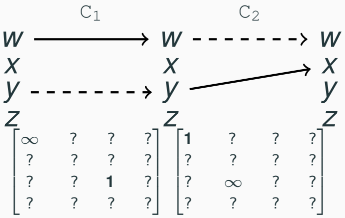
$$\begin{bmatrix} \infty & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & \mathbf{1} & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} \mathbf{1} & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

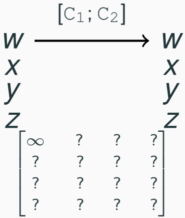


$$\begin{bmatrix} \infty & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

Multipath and Composition (*à la "size-change Termination"*)

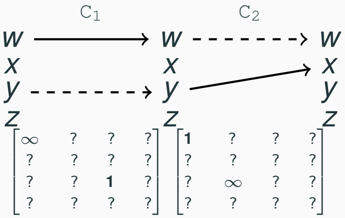
Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.

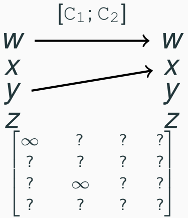




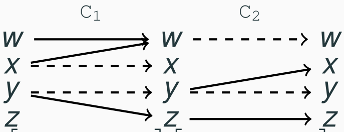
Multipath and Composition (*à la "size-change Termination"*)

Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.

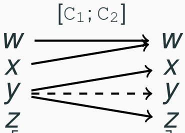




Let $\mathtt{C}$ be a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \dots; \mathtt{C}_n]$. Then $\mathrm{M}(\mathtt{C})$ is defined as the matrix product $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\dots\mathrm{M}(\mathtt{C}_n)$.



$$\begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 1 & 0 & 0 \\ 0 & 0 & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \infty & 1 & 0 \\ 0 & 0 & 0 & \infty \end{bmatrix}$$

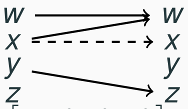


$$\begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \\ 0 & \infty & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Condition

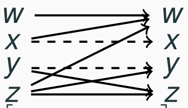
Let `C` be a command of the form `if E then C`$_1$ `else skip;`
Then $\mathrm{M}(\texttt{C}) = \mathrm{M}(\texttt{C}_1) + \mathrm{Id} + (E^t O)$

`E` `O` `C`$_1$



$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ \infty \end{bmatrix} \quad \begin{bmatrix} \infty \\ 0 \\ \infty \\ \infty \end{bmatrix} \quad \begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 1 & 0 & 0 \\ 0 & 0 & 0 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

[`if E then C`$_1$ `else skip`]



$$\begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 1 & 0 & 0 \\ 0 & 0 & 1 & \infty \\ \infty & 0 & \infty & \infty \end{bmatrix}$$

Compute the max DFG regarding to the different possibilities.
$\mathrm{Var}(\texttt{E})$ indirectly influence the dependencies.

Loop `while` (à la *“mwp-polynomials”*)

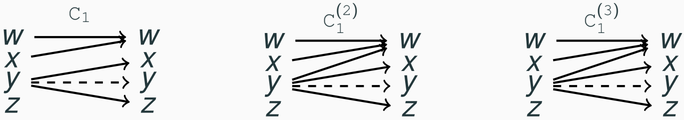
Let `C` be a command such as : `C` := `while E do` $\texttt{C}_1$;

- Number of iteration unknown :

  `skip` $\qquad \texttt{C}_1 \qquad \texttt{C}_1;\texttt{C}_1 \qquad \texttt{C}_1;\texttt{C}_1;\texttt{C}_1 \qquad$ etc. . .

- Compute the max :

$$\mathrm{M}(\texttt{C}_1^*)$$



Trivially converges by motonicity

Loop `while` (à la *“mwp-polynomials”*)

Let $\texttt{C}$ be a command such as : $\texttt{C} := \texttt{while}\ \texttt{E}\ \texttt{do}\ \texttt{C}_1;$

As for the `if` statement, a condition correction is needed.

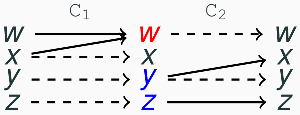
$$\mathrm{M}(\texttt{C}) = \mathrm{M}(\texttt{C}_1^*)^{[\texttt{E}]}.$$

Theory : Independence

Independence of chunks

Definition (independence)

If $\mathrm{Out}(\texttt{C}_1) \cap \mathrm{In}(\texttt{C}_2) = \emptyset$ *then* $\texttt{C}_2$ *is **independent** from* $\texttt{C}_1$*. This is denoted* $\texttt{C}_1 \prec \texttt{C}_2$*.*



Definition (self-independence)

If $\texttt{C}_1$ *is independent from itself, we say* $\texttt{C}_1$ *is **self-independent***

Lemma (Optimization for `while`)

If $\texttt{C}_1$ *is self-independent and* $\mathrm{Var}(\texttt{E}) \cap \mathrm{Out}(\texttt{C}_1) = \emptyset$ *:*

$[\![\texttt{while E do } [\texttt{C}_1]]\!] \equiv [\![\texttt{if E then } [\texttt{C}_1]; \texttt{While E do } [\texttt{skip}]]\!]$

Moving Independent Chunks

Definition (Mutual Independence)

If $\mathtt{C}_2 \prec \mathtt{C}_1$ *and* $\mathtt{C}_1 \prec \mathtt{C}_2$*, we say that* $\mathtt{C}_2$ *and* $\mathtt{C}_1$ *are* ***mutually independents****, and write* $\mathtt{C}_1 \asymp \mathtt{C}_2$*.*

Lemma (Swapping commands)

If $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then :*

$$[\![\mathtt{C}_1; \mathtt{C}_2]\!] \equiv [\![\mathtt{C}_2; \mathtt{C}_1]\!]$$

Lemma (Hoisting mutual independent commands)

If $\mathtt{C}_1$ *is self-independent (i.e.* $\mathtt{C}_1 \asymp \mathtt{C}_1$*),* $\mathrm{Var}(\mathtt{E}) \cap \mathrm{Out}(\mathtt{C}_1) = \emptyset$*, and if* $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then :*

$$[\![\mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_1; \mathtt{C}_2]]\!] \equiv [\![\mathtt{if}\ \mathtt{E}\ \mathtt{then}\ [\mathtt{C}_1]; \mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_2]]\!]$$

Moving Independent Chunks

Definition (Mutual Independence)

If $\mathtt{C}_2 \prec \mathtt{C}_1$ *and* $\mathtt{C}_1 \prec \mathtt{C}_2$*, we say that* $\mathtt{C}_2$ *and* $\mathtt{C}_1$ *are* ***mutually independents****, and write* $\mathtt{C}_1 \asymp \mathtt{C}_2$*.*

Lemma (Swapping commands)

If $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then :*

$$[\![\mathtt{C}_1;\mathtt{C}_2]\!] \equiv [\![\mathtt{C}_2;\mathtt{C}_1]\!]$$

Lemma (Hoisting mutual independent commands)

If $\mathtt{C}_1$ *is self-independent (i.e.* $\mathtt{C}_1 \asymp \mathtt{C}_1$*),* $\mathrm{Var}(\mathtt{E}) \cap \mathrm{Out}(\mathtt{C}_1) = \emptyset$*, and if* $\mathtt{C}_1 \prec \mathtt{C}_2$*, then :*

$$[\![\mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_1;\mathtt{C}_2]]\!] \equiv [\![\mathtt{if}\ \mathtt{E}\ \mathtt{then}\ [\mathtt{C}_1;\mathtt{C}_2];\mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_2]]\!]$$

Computation of the invariance degree

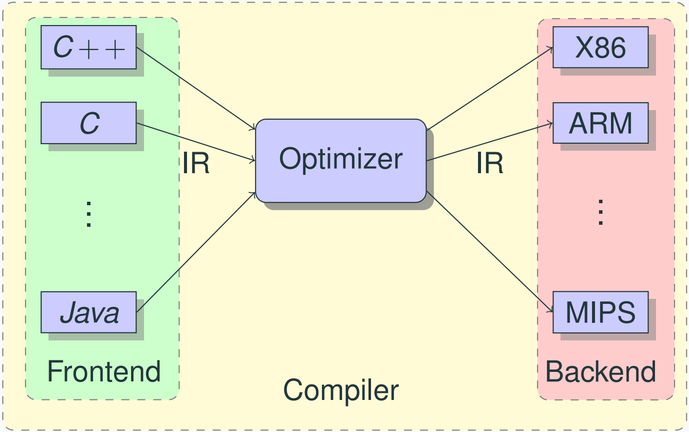
Statically easy using the DFGs and the dominance graph (order of the instructions).

Let suppose we have computed the graph of dependencies for all commands.

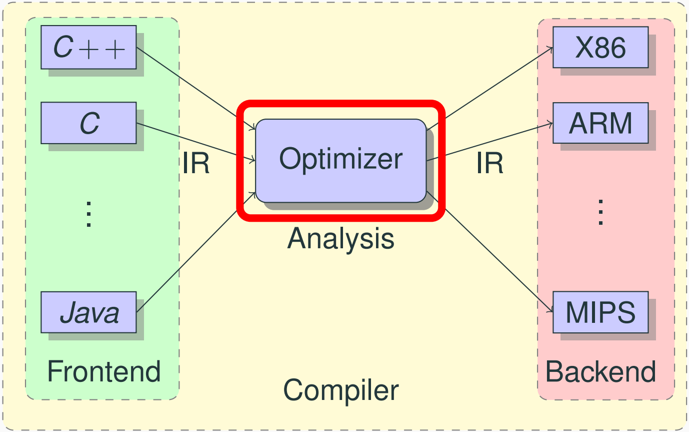
Compute recursively (depth-first order) the degrees of the depended commands and take the maximum.

Compiler

Architecture



Architecture



Order

Order is given as argument to the **pass manager** :

```
$ clang -O3 test.c
```

Order

Order is given as argument to the **pass manager** :

```
$ clang -O3 test.c
```

```
Pass Arguments: -targetlibinfo -no-aa -tbaa -scoped-noalias -assumption-tracker
     -basicaa -notti -verify-di -ipsccp -globalopt -deadargelim -domtree
     -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -inline
     -functionattrs -argpromotion -sroa -domtree -early-cse -lazy-value-info
     -jump-threading -correlated-propagation -simplifycfg -domtree -instcombine
     -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa
     -loop-rotate -licm -loop-unswitch -instcombine -scalar-evolution
     -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -function_tti
     -loop-unroll -memdep -mldst-motion -domtree -memdep -gvn -memdep -memcpyopt
     -sccp -domtree -instcombine -lazy-value-info -jump-threading
     -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -domtree
     -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob
     -block-freq -scalar-evolution -loop-vectorize -instcombine -scalar-evolution
     -slp-vectorizer -simplifycfg -domtree -instcombine -loops -loop-simplify
     -lcssa -scalar-evolution -function_tti -loop-unroll
     -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge
     -verify -verify-di
```

Order

Order is given as argument to the **pass manager** :

```
$ clang -O3 test.c
```

```
Pass Arguments: -targetlibinfo -no-aa -tbaa -scoped-noalias -assumption-tracker
     -basicaa -notti -verify-di -ipsccp -globalopt -deadargelim -domtree
     -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -inline
     -functionattrs -argpromotion -sroa -domtree -early-cse -lazy-value-info
     -jump-threading -correlated-propagation -simplifycfg -domtree -instcombine
     -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa
     -loop-rotate -licm -loop-unswitch -instcombine -scalar-evolution
     -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -function_tti
     -loop-unroll -memdep -mldst-motion -domtree -memdep -gvn -memdep -memcpyopt
     -sccp -domtree -instcombine -lazy-value-info -jump-threading
     -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -domtree
     -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob
     -block-freq -scalar-evolution -loop-vectorize -instcombine -scalar-evolution
     -slp-vectorizer -simplifycfg -domtree -instcombine -loops -loop-simplify
     -lcssa -scalar-evolution -function_tti -loop-unroll
     -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge
     -verify -verify-di
```

LLVM Intermediate Representation

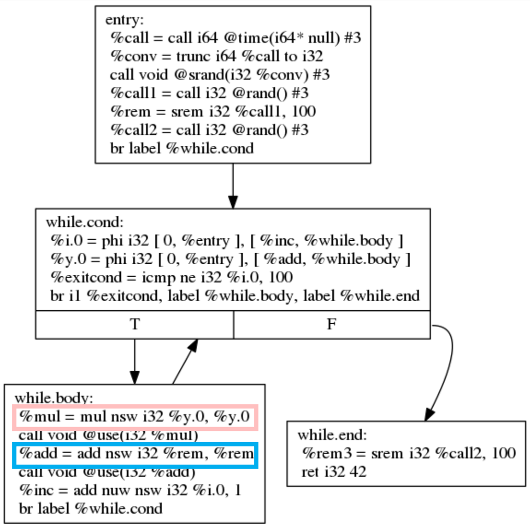
- LLVM-IR is a **Typed Assembly Language** (TAL) and a **Static Single Assignment** (SSA) based representation. This provides :
- An IR is **source-language-independent**, then optimizations and analysis should work on every languages (properly translated to this IR).

LLVM Intermediate Representation

```
define i32 @main() #0 {
entry:
  %call = call i64 @time(i64* null) #3
  %conv = trunc i64 %call to i32
  call void @srand(i32 %conv) #3
  %call1 = call i32 @rand() #3
  %rem = srem i32 %call1, 100
  %call2 = call i32 @rand() #3
  br label %while.cond

while.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %while.body ]
  %y.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
  %exitcond = icmp ne i32 %i.0, 100
  br i1 %exitcond, label %while.body, label %while.end

while.body:
  %mul = mul nsw i32 %y.0, %y.0
  call void @use(i32 %mul)
  %add = add nsw i32 %rem, %rem
  call void @use(i32 %add)
  %inc = add nuw nsw i32 %i.0, 1
  br label %while.cond

while.end:
  %rem3 = srem i32 %call2, 100
  ret i32 42
}
```

```
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   while(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i++;
   }
   return 42;
}
```

LLVM Intermediate Representation in Control Flow Graph



CFG for 'main' function

```
int main(){
    int i=0,y=0;
    srand(time(NULL));
    int x=rand()%100;
    int x2=rand()%100;
    int z;
    while(i<100){
        z=y*y;
        use(z);
        y=x+x;
        use(y);
        i++;
    }
    return 42;
}
```

Implementation

A prototype on LLVM tool chain

We implemented a pass in LLVM :

- Currently around 3000 lines of `C++`, and counting...
- tested on several relevant examples
- generates statistics while compiling...

Analysis : Degree on each Instruction

```
...
while.body:
  %mul = mul nsw i32 %y.0, %y.0    ← 2
  call void @use(i32 %mul)         ← ∞
  %add = add nsw i32 %rem, %rem    ← 1
  call void @use(i32 %add)         ← ∞
  %inc = add nuw nsw i32 %i.0, 1   ← ∞
  br label %while.cond             ← ∞
...
```

```
...
  while(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i++;
  }
...
```

Analysis : Degree on each Instruction

```
...
while.body:
  %mul = mul nsw i32 %y.0, %y.0      ← 2
  call void @use(i32 %mul)           ← ∞
  %add = add nsw i32 %rem, %rem      ← 1
  call void @use(i32 %add)           ← ∞
  %inc = add nuw nsw i32 %i.0, 1     ← ∞
  br label %while.cond               ← ∞
...
```

```
...
  while(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i++;
  }
...
```

Consider all `call` as anchors

Same example

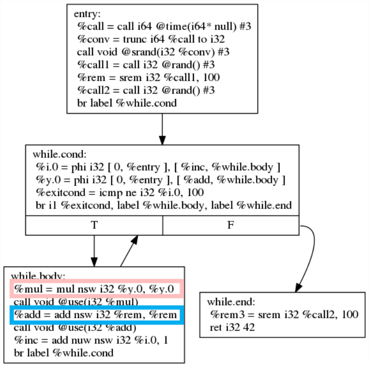
```
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   while(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i++;
   }
   return 42;
}
```

Same example

```
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   while(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i++;
   }
   return 42;
}
```

```
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   if(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i=i+1;
   }
   if(i<100){
      z=y*y;
      use(z);
      use(y);
      i=i+1;
   }
   while(i<100){
      use(z);
      use(y);
      i=i+1;
   }
   return 42;
}
```

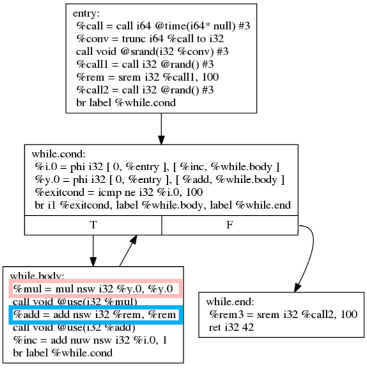
Same example



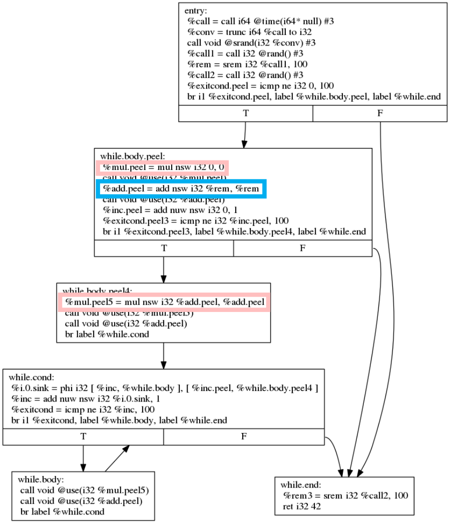
CFG for 'main' function

```
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   if(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i=i+1;
   }
   if(i<100){
      z=y*y;
      use(z);
      use(y);
      i=i+1;
   }
   while(i<100){
      use(z);
      use(y);
      i=i+1;
   }
   return 42;
}
```

Same example



CFG for 'main' function



CFG for 'main' function

Last “relevant” example

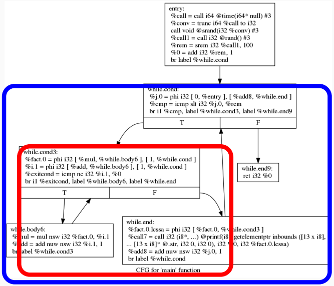
```
while(j<100){
    fact=1;
    I=1;
    while(I<=n){
        fact=fact*I;
        I=I+1;
    }
    use(fact);
    j=j+1;
}
```

Last “relevant” example

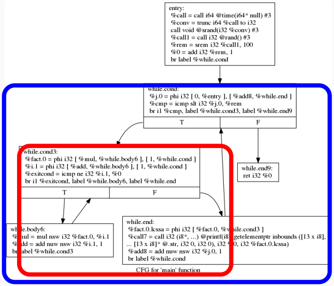
```
while(j<100){
   fact=1;
   I=1;
   while(I<=n){
      fact=fact*I;
      I=I+1;
   }
   use(fact);
   j=j+1;
}
```

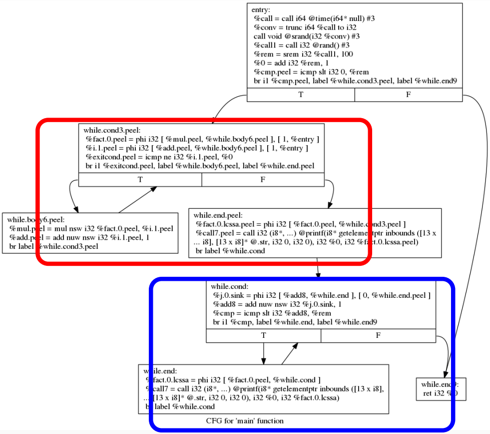
```
if(j<100){
  fact=1;
  I=1;
  while(I<=n){
    fact=fact*I;
    I=I+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

Last “relevant” example




```
if(j<100){
  fact=1;
  I=1;
  while(I<=n){
    fact=fact*I;
    I=I+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```


entry:
%call = call i64 @time(i64* null) #3
%conv = trunc i64 %call to i32
call void @srand(i32 %conv) #3
%call1 = call i32 @rand() #3
%rem = srem i32 %call1, 100
%0 = add i32 %rem, 1
br label %while.cond
while.cond:
%j.0 = phi i32 [0, %entry], [%add8, %while.end]
%cmp = icmp slt i32 %j.0, %rem
br i1 %cmp, label %while.cond3, label %while.end9
T
F
while.cond3:
%fact.0 = phi i32 [%mul, %while.body6], [1, %while.cond]
%i.1 = phi i32 [%add, %while.body6], [1, %while.cond]
%exitcond = icmp ne i32 %i.1, %0
br i1 %exitcond, label %while.body6, label %while.end
T
F
while.end9:
ret i32 %0
while.end:
%fact.0.lcssa = phi i32 [%fact.0, %while.cond3]
%call7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8],
... [13 x i8]* @.str, i32 0, i32 0), i32 %0, i32 %fact.0.lcssa)
%add8 = add nuw nsw i32 %j.0, 1
br label %while.cond
CFG for 'main' function


entry:
%call = call i64 @time(i64* null) #3
%conv = trunc i64 %call to i32
call void @srand(i32 %conv) #3
%call1 = call i32 @rand() #3
%rem = srem i32 %call1, 100
%0 = add i32 %rem, 1
%cmp.peel = icmp slt i32 0, %rem
br i1 %cmp.peel, label %while.cond3.peel, label %while.end9
T
F
while.cond3.peel:
%fact.0.peel = phi i32 [%mul.peel, %while.body6.peel], [1, %entry]
%i.1.peel = phi i32 [%add.peel, %while.body6.peel], [1, %entry]
%exitcond.peel = icmp ne i32 %i.1.peel, %0
br i1 %exitcond.peel, label %while.body6.peel, label %while.end.peel
T
F
%add.peel = add nuw nsw i32 %i.1.peel, 1
while.end.peel:
%fact.0.lcssa.peel = phi i32 [%fact.0.peel, %while.cond3.peel]
while.cond:
%j.0.sink = phi i32 [%add8, %while.end], [0, %while.end.peel]
%add8 = add nuw nsw i32 %j.0.sink, 1
%cmp = icmp slt i32 %add8, %rem
br i1 %cmp, label %while.end, label %while.end9
T
F
label %while.cond
CFG for 'main' function

LLVM statistics

```
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

 2 globalopt        - Number of globals deleted
 5 globalopt        - Number of globals marked unnamed_addr
 1 indvars          - Number of congruent IVs eliminated
 2 indvars          - Number of loop exit tests replaced
 2 indvars          - Number of exit values replaced
10 instcombine      - Number of insts combined
 1 instsimplify     - Number of redundant instructions removed
 8 lcssa            - Number of live out of a loop variables
 1 licm             - Number of instructions hoisted out of loop
 2 loop-rotate      - Number of loops rotated
11 loop-simplify    - Number of pre-header or exit blocks inserted
 2 loop-unswitch    - Total number of instructions analyzed
 1 loop-vectorize   - Number of loops analyzed for vectorization
 2 lqicm            - Number of time runOnLoop is performed...
 3 lqicm            - Number of instructions with deg != -1
 1 lqicm            - Number of innerBlocks with deg != -1
 4 mem2reg          - Number of PHI nodes inserted
 2 mem2reg          - Number of alloca's promoted with a single store
 9 simplifycfg      - Number of blocks simplified
```

LLVM statistics

```
--- vim -O1 EarlyOpt 7m5,276s
3808 number of loops
20465 number of instructions hoisted by LICM
2266 number of loops with several exit blocks
125 number of loops not well formed
2391 sum of loop not analyzed
1417 sum of loop analyzed by LQICM
2476 number of quasi-invariants detected
335 number of quasi-invariants Chunk detected
23 Number of chunks with deg >= 2

--- emacs -O1 EarlyOpt 15m52,556s
3161 number of loops
16415 number of instructions hoisted by LICM
1775 number of loops with several exit blocks
150 number of loops not well formed
1925 sum of loop not analyzed
1236 sum of loop analyzed by LQICM
2197 number of quasi-invariants detected
311 number of quasi-invariants Chunk detected
35 Number of chunks with deg >= 2
```

Difficulties...

- Apprehend LLVM tools and data structure...
- Consider all strange cases we can have in LLVM-IR
- More and more features have to be implemented to be able to compete with Loop Invariant Code Motion...

Conclusion and further work

We implemented the first skeleton of a huge project
Still a lot of work to do :

- Compile a lot of programs to have more stats
- Make the pass more flexible to take into account more cases
- Start benchmarks with the transformation and compete with LICM !
- Push the compilation community to contribute ?
- mwp-polynomials is an analysis not so far...

Conclusion and further work

We implemented the first skeleton of a huge project
Still a lot of work to do :

- Compile a lot of programs to have more stats
- Make the pass more flexible to take into account more cases
- Start benchmarks with the transformation and compete with LICM !
- Push the compilation community to contribute ?
- mwp-polynomials is an analysis not so far. . .
- Helping you to implement your analysis ?