Workshop on Essential Abstractions in GCC

Parallelization and Vectorization in GCC 4.5.0

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

July 2010

Outline

- An Overview of Loop Transformations in GCC
- Parallelization and vectorization based on Lambda Framework
- Parallelization based on Polytope Model
- Conclusions

The Scope of this Tutorial

- What this tutorial does not address
  - Algorithms used for parallelization and vectoriation
  - Machine level issues related to parallelization and vectoriation
- What this tutorial addresses

Basics of Discovering Parallelism using GCC

Part 1

Loop Transformations

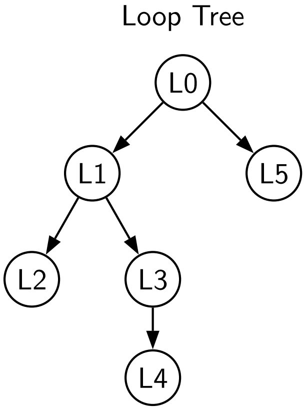
Loop Transforms in GCC

Implementation Issues

- Getting loop information (Loop discovery)
- Finding value spaces of induction variables, index expressions, and pointer accesses
- Analyzing data dependence
- Performing linear transformations

Loop Information

```
Loop0
{   Loop1
    {   Loop2
        {
        }
        Loop3
        {  Loop4
           {
           }
        }
    }
    Loop5
    {
    }
}
```

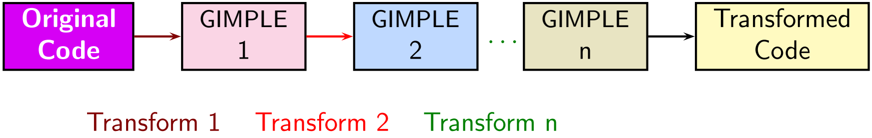


Problems with Classical Loop Nest Transforms

- Difficult to undo loop transforms - transforms are applied on the syntactic form
- Difficult to compose transformations - intermediate translation to a syntactic form after each transformation
- Ordering of transforms is fixed - as defined in file *passes.c*

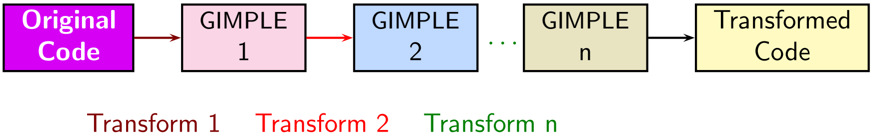
Expected Loop Nest Transforms

Classical Loop Transforms:

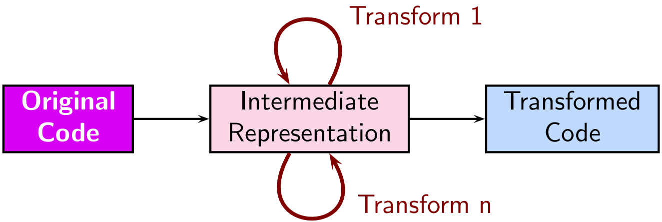


Expected Loop Nest Transforms

Classical Loop Transforms:



Expected Loop Transforms with Composition:



Loop Transformation Frameworks in GCC

- Linear Loop Transformations in GCC 4.5.0 are performed on two frameworks:
  - Lambda Framework - performs transformations of loops using non-singular matrix
  - Polyhedral Model - performs transformations of loops by representing them as a convex polyhedra
- The polyhedral model handles a wider class of programs and transformations than the unimodular framework
- Polyhedral Models generalize the classical transforms to imperfectly-nested loops with complex domains

Part 2

Parallelization and Vectorization in GCC using Lambda Framework

Representing Value Spaces of Variables and Expressions

Chain of Recurrences: 3-tuple ⟨Starting Value, modification, stride⟩

```
for (i=3; i<=15; i=i+3)
{
    for (j=11; j>=1; j=j-2)
    {
        A[i+1][2*j-1] = ...
    }
}
```

Entity	CR
Induction variable i	$\{3, +, 3\}$
Induction variable j	$\{11, +, -2\}$
Index expression i+1	$\{4, +, 3\}$
Index expression 2*j-1	$\{21, +, -4\}$

Advantages of Chain of Recurrences

CR can represent any affine expression
⇒ Accesses through pointers can also be tracked

```
int A[32], B[32];
int i, *p;
p = &B
for(i = 2; i<N; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

Advantages of Chain of Recurrences

CR can represent any affine expression

⇒ Accesses through pointers can also be tracked

```
int A[32], B[32];
int i, *p;
p = &B
for(i = 2; i<N; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

{&B,+,4bytes}

Advantages of Chain of Recurrences

CR can represent any affine expression

⇒ Accesses through pointers can also be tracked

```
int A[32], B[32];
int i, *p;
p = &B
for(i = 2; i<N; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

{&B,+,4bytes}

{&B+4bytes,+,4bytes}

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_dce_loop);
    NEXT_PASS (pass_lim);
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_tree_unswitch);
    NEXT_PASS (pass_scev_cprop);
    NEXT_PASS (pass_empty_loop);
    NEXT_PASS (pass_record_bounds);
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_linear_transform);
    NEXT_PASS (pass_graphite_transforms);
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of GIMPLE IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_dce_loop);
    NEXT_PASS (pass_lim);
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_tree_unswitch);
    NEXT_PASS (pass_scev_cprop);
    NEXT_PASS (pass_empty_loop);
    NEXT_PASS (pass_record_bounds);
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_linear_transform);
    NEXT_PASS (pass_graphite_transforms);
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of GIMPLE IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_dce_loop);
    NEXT_PASS (pass_lim);
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_tree_unswitch);
    NEXT_PASS (pass_scev_cprop);
    NEXT_PASS (pass_empty_loop);
    NEXT_PASS (pass_record_bounds);
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_linear_transform);
    NEXT_PASS (pass_graphite_transforms);
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of GIMPLE IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC: Our Focus

Data Dependence	Pass variable name	`pass_check_data_deps`
	Enabling switch	`-fcheck-data-deps`
	Dump switch	`-fdump-tree-ckdd`
	Dump file extension	`.ckdd`
Loop Distribution	Pass variable name	`pass_loop_distribution`
	Enabling switch	`-ftree-loop-distribution`
	Dump switch	`-fdump-tree-ldist`
	Dump file extension	`.ldist`
Vectorization	Pass variable name	`pass_vectorize`
	Enabling switch	`-ftree-vectorize`
	Dump switch	`-fdump-tree-vect`
	Dump file extension	`.vect`
Parallelization	Pass variable name	`pass_parallelize_loops`
	Enabling switch	`-ftree-parallelize-loops=n`
	Dump switch	`-fdump-tree-parloops`
	Dump file extension	`.parloops`

Compiling for Emitting Dumps

- Other necessary command line switches
  - `-O3 -fdump-tree-all`
    `-O3` enables `-ftree-vectorize`. Other flags must be enabled explicitly
- Processor related switches to enable transformations apart from analysis
  - `-mtune=pentium -msse4`
- Other useful options
  - Suffixing `-all` to all dump switches
  - `-S` to stop the compilation with assembly generation
  - `--verbose-asm` to see more detailed assembly dump
  - `-fno-predictive-commoning` to disable predictive commoning optimization

Example 1: Observing Data Dependence

Step 0: Compiling

```
#include <stdio.h>
int a[200];
int main()
{
   int i, n;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

```
gcc -fcheck-data-deps -fdump-tree-ckdd-all -O3 -S datadep.c
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

Program

```
#include <stdio.h>
int a[200];
int main()
{
   int i, n;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

<pre>#include <stdio.h>
int a[200];
int main()
{
   int i, n;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}</pre>
	<pre><bb 3>:
	  # i_13 = PHI <i_4(4), 0(2)>
	  i_4 = i_13 + 1;
	  D.1240_5 = a[i_4];
	  D.1241_6 = D.1240_5 + 2;
	  a[i_13] = D.1241_6;
	  if (i_4 <= 149)
	    goto <bb 4>;
	  else
	    goto <bb 5>;
	<bb 4>:
	  goto <bb 3>;</pre>

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program

```c
#include <stdio.h>
int a[200];
int main()
{
   int i, n;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

`#include <stdio.h>`
`int a[200];`
`int main()`
`{`
`   int i, n;`
`   for (i=0; i<150; i++)`
`   {`
`      a[i] = a[i+1] + 2;`
`   }`
`   return 0;`
`}`
	`<bb 3>:`
	`  # i_13 = PHI <i_4(4), 0(2)>`
	`  i_4 = i_13 + 1;`
	`  D.1240_5 = a[i_4];`
	`  D.1241_6 = D.1240_5 + 2;`
	`  a[i_13] = D.1241_6;`
	`  if (i_4 <= 149)`
	`    goto <bb 4>;`
	`  else`
	`    goto <bb 5>;`
	`<bb 4>:`
	`  goto <bb 3>;`

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(evolution_function = {0, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(scalar_evolution = {1, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                 address: 4
aligned to: 128
(chrec = {1, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                 address:  0
aligned to: 128
base_object: a[0]
(chrec = {0, +, 1}_1)
```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$ • Solve for $0 \leq x, y < 150$ $y = x + 1$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$ • Solve for $0 \leq x, y < 150$ $y = x + 1$ $\Rightarrow x - y + 1 = 0$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

- Relevant assignment is $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$x - y + 1$

Min: -148 Max: +150

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$x - y + 1$

Min: -148 Max: +150

RHS belongs to $[-148, +150]$
and dependence may exist

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View	CFG View

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \le x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$x - y + 1$

Min: -148 Max: $+150$

RHS belongs to $[-148, +150]$
and dependence may exist

CFG View

```
i_4 = i_13 + 1;
D.1240_5 = a[i_4];
D.1241_6 = D.1240_5 + 2;
a[i_13] = D.1241_6;
```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$x - y + 1$

Min: -148 Max: +150

RHS belongs to $[-148, +150]$
and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
- Chain of recurrences are

  For `a[i_4]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

$x - y + 1$

Min: -148 Max: +150

RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
- Chain of recurrences are
  For `a[i_4]`: `{1, +, 1}_1`
  For `a[i_13]`: `{0, +, 1}_1`
- Solve for $0 \leq$ `x_1` < 150
  `1 + 1*x_1` − `0 + 1*x_1` $= 0$
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x + 1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS

  $x - y + 1$

  Min: -148 Max: +150

  RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
- Chain of recurrences are

  For `a[i_4]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`
- Solve for $0 \leq$ `x_1` < 150

  `1 + 1*x_1` $-$ `0 + 1*x_1` $= 0$
- Min of LHS is -148, Max is +150
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$
\begin{aligned}
y &= x + 1 \\
\Rightarrow \quad x - y + 1 &= 0
\end{aligned}
$$

- Find min and max of LHS

  $x - y + 1$

  Min: -148 Max: +150

  RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
- Chain of recurrences are

  For `a[i_4]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`
- Solve for $0 \leq$ `x_1` < 150

  `1 + 1*x_1` $-$ `0 + 1*x_1` $= 0$
- Min of LHS is -148, Max is +150
- Dependence may exist
  ```

Example 2: Observing Vectorization and Parallelization

Step 0: Compiling with `-fno-predictive-commoning`

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=4 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4`

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

`int a[256], b[256];`
`int main()`
`{`
`    int i;`
`    for (i=0; i<256; i++)`
`    {`
`        a[i] = b[i];`
`    }`
`    return 0;`
`}`
	`<bb 3>:`
	`  # i_14 = PHI <i_6(4), 0(2)>`
	`  D.1666_5 = b[i_14];`
	`  a[i_14] = D.1666_5;`
	`  i_6 = i_14 + 1;`
	`  if (i_6 <= 255)`
	`    goto <bb 4>;`
	`  else`
	`    goto <bb 5>;`
	`<bb 4>:`
	`  goto <bb 3>;`

Example 2: Observing Vectorization and Parallelization

Step 2: Observing the final decision about vectorization

```
parvec.c:9: note: LOOP VECTORIZED.
parvec.c:6: note: vectorized 1 loops in function.
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph	Transformed control flow graph

<pre><bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;</pre>
	<pre>...
	vect_var_.31_18 = *vect_pb.25_16;
	*vect_pa.32_21 = vect_var_.31_18;
	vect_pb.25_17 = vect_pb.25_16 + 16;
	vect_pa.32_22 = vect_pa.32_21 + 16;
	ivtmp.38_24 = ivtmp.38_23 + 1;
	if (ivtmp.38_24 < 64)
	  goto <bb 4>;
	else
	  goto <bb 5>;
	...</pre>

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
     /* execute the loop body to be parallelized */
}
```

- All threads are executed in parallel

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Get the number of threads

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Get thread identity

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Perform load calculations

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Assign start iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Assign end iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Start execution of iterations of the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 4>:
  i.29_21 = (int) ivtmp.28_18;
  D.1312_23 = (*b.31_4)[i.29_21];
  (*a.32_5)[i.29_21] = D.1312_23;
  ivtmp.28_19 = ivtmp.28_18 + 1;
  if (D.1308_16 > ivtmp.28_19)
    goto <bb 4>;
  else
    goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

`<bb 3>:`
`  # i_14 = PHI <i_6(4), 0(2)>`
`  D.1666_5 = b[i_14];`
`  a[i_14] = D.1666_5;`
`  i_6 = i_14 + 1;`
`  if (i_6 <= 255)`
`    goto <bb 4>;`
`  else`
`    goto <bb 5>;`
`<bb 4>:`
`  goto <bb 3>;`
	`<bb 4>:`
	`  i.29_21 = (int) ivtmp.28_18;`
	`  D.1312_23 = (*b.31_4)[i.29_21];`
	`  (*a.32_5)[i.29_21] = D.1312_23;`
	`  ivtmp.28_19 = ivtmp.28_18 + 1;`
	`  if (D.1308_16 > ivtmp.28_19)`
	`    goto <bb 4>;`
	`  else`
	`    goto <bb 3>;`

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

<pre><bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;</pre>
	<pre><bb 4>:
	  i.29_21 = (int) ivtmp.28_18;
	  D.1312_23 = (*b.31_4)[i.29_21];
	  (*a.32_5)[i.29_21] = D.1312_23;
	  ivtmp.28_19 = ivtmp.28_18 + 1;
	  if (D.1308_16 > ivtmp.28_19)
	    goto <bb 4>;
	  else
	    goto <bb 3>;</pre>

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 4>:
  i.29_21 = (int) ivtmp.28_18;
  D.1312_23 = (*b.31_4)[i.29_21];
  (*a.32_5)[i.29_21] = D.1312_23;
  ivtmp.28_19 = ivtmp.28_18 + 1;
  if (D.1308_16 > ivtmp.28_19)
    goto <bb 4>;
  else
    goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 4>:
  i.29_21 = (int) ivtmp.28_18;
  D.1312_23 = (*b.31_4)[i.29_21];
  (*a.32_5)[i.29_21] = D.1312_23;
  ivtmp.28_19 = ivtmp.28_18 + 1;
  if (D.1308_16 > ivtmp.28_19)
    goto <bb 4>;
  else
    goto <bb 3>;
```

Example 3: Vectorization but No Parallelization

Step 0: Compiling with
`-fno-predictive-commoning -fdump-tree-vect-all -msse4`

```
int a[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = a[i+4];
    }
    return 0;
}
```

Example 3: Vectorization but No Parallelization

Step 1: Observing the final decision about vectorization

```
vecnopar.c:8: note: LOOP VECTORIZED.
vecnopar.c:5: note: vectorized 1 loops in function.
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph	Vectorized Control Flow Graph

<pre><bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
 	
<bb 4>:
  goto <bb 3>;</pre>
	<pre> a.31_11 = (vector int *) &a;
	  vect_pa.30_15 = a.31_11 + 16;
	  vect_pa.25_16 = vect_pa.30_15;
	  vect_pa.38_20 = (vector int *) &a;
	  vect_pa.33_21 = vect_pa.38_20;
	
	<bb 3>:
	  vect_var_.32_19 = *vect_pa.25_17;
	  *vect_pa.33_22 = vect_var_.32_19;
	  vect_pa.25_18 = vect_pa.25_17 + 16;
	  vect_pa.33_23 = vect_pa.33_22 + 16;
	  ivtmp.39_25 = ivtmp.39_24 + 1;
	  if (ivtmp.39_25 < 64)
	    goto <bb 4>;</pre>

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph	Vectorized Control Flow Graph

<pre><bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
 	
<bb 4>:
  goto <bb 3>;</pre>
	<pre> a.31_11 = (vector int *) &a;
	  vect_pa.30_15 = a.31_11 + 16;
	  vect_pa.25_16 = vect_pa.30_15;
	  vect_pa.38_20 = (vector int *) &a;
	  vect_pa.33_21 = vect_pa.38_20;
	
	<bb 3>:
	  vect_var_.32_19 = *vect_pa.25_17;
	  *vect_pa.33_22 = vect_var_.32_19;
	  vect_pa.25_18 = vect_pa.25_17 + 16;
	  vect_pa.33_23 = vect_pa.33_22 + 16;
	  ivtmp.39_25 = ivtmp.39_24 + 1;
	  if (ivtmp.39_25 < 64)
	    goto <bb 4>;</pre>

Example 3: Vectorization but No Parallelization

- Step 3: Observing the conclusion about dependence information

```
inner loop index: 0
loop nest: (1 )
distance_vector:   4
direction_vector:     +
```

- Step 4: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```

Example 4: No Vectorization and No Parallelization

Step 0: Compiling with `-fno-predictive-commoning`

```
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i+2] = b[i] + 5;
        b[i+3] = a[i] + 10;
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=4 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4`

Example 4: No Vectorization and No Parallelization

- Step 1: Observing the final decision about vectorization

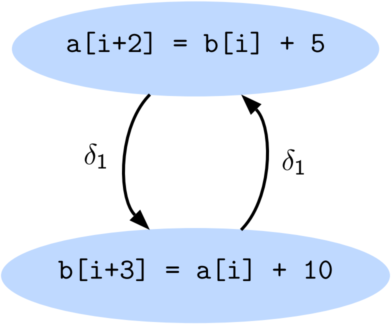
```
noparvec.c:5: note: vectorized 0 loops in function.
```

- Step 2: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```

Example 4: No Vectorization and No Parallelization

Step 3: Understanding the dependencies that prohibit vectorization and parallelization



Part 3

Parallelization in GCC using Polytope Model

Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

Polyhedral Representation

- Polytope Model is a mathematical framework for loop nest optimizations
- The loop bounds parametrized as inequalities form a convex polyhedron
- An affine scheduling function specifies the scanning order of integral points

GCC requires a rich algebraic representation that enables:

- Composition of polyhedral generalizations of classical loop transformations
- Decoupling them from the syntatic form of program

GRAPHITE

GRAPHITE is the interface for polyhedra representation of GIMPLE

goal: more high level loop optimizations

GRAPHITE

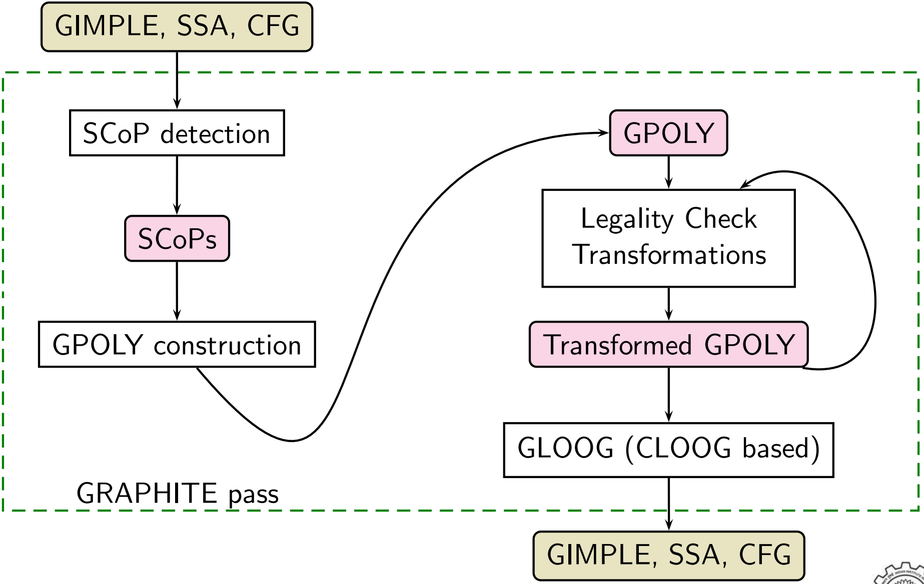
GRAPHITE is the interface for polyhedra representation of GIMPLE

goal: more high level loop optimizations

Tasks of GRAPHITE Pass:

- Extract the *polyhedral model* representation out of GIMPLE
- Perform the various optimizations and analyses on this polyhedral model representation
- Regenerate the GIMPLE three-address code that corresponds to transformations on the polyhedral model

Compilation Workflow



What Code Can be Represented?

- Structured code
- Affine loop bounds (e.g. `i < 4*n+4*j-1`)
- Constant loop strides (e.g. `i += 2`)
- Conditions containing comparisons (`<,<=,>,>=,==,!=`) between affine functions
- Affine array accesses (e.g. `A[3i+1]`)

GPOLY

GPOLY : the polytope representation in GRAPHITE, currently implemented by the Parma Polyhedra Library (PPL)

- SCoP - The optimization unit (e.g. a loop with some statements)
  scop := ([*black box*])
- Black Box - An operation (e.g. statement) where only the memory accesses are known
  black box := (*iteration domain*, *scattering matrix*, [*data reference*])
- Iteration Domain - The set of loop iterations for the black box
- Data Reference - The memory cells accessed by the black box
- Scattering Matrix - Defines the execution order of statement iterations (e.g. schedule)

Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

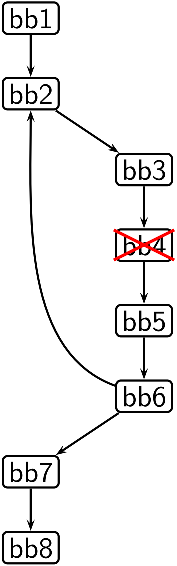
Building SCoPs

- SCoPs built on top of the CFG
- Basic blocks with side-effect statements are split
- All basic blocks belonging to a SCoP are dominated by entry, and postdominated by exit of the SCoP

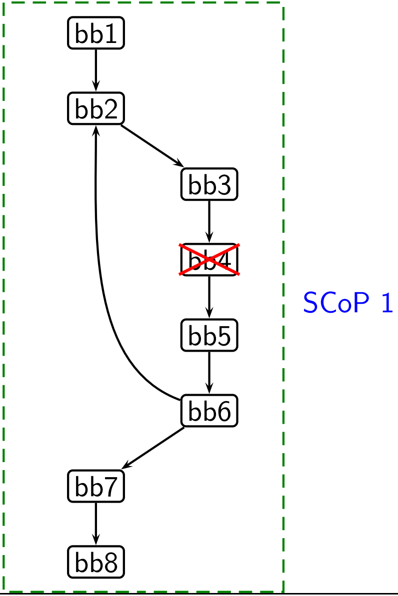
Basic blocks split for:

- smaller code chunks
- reducing number of dependences
- moving parts of code around

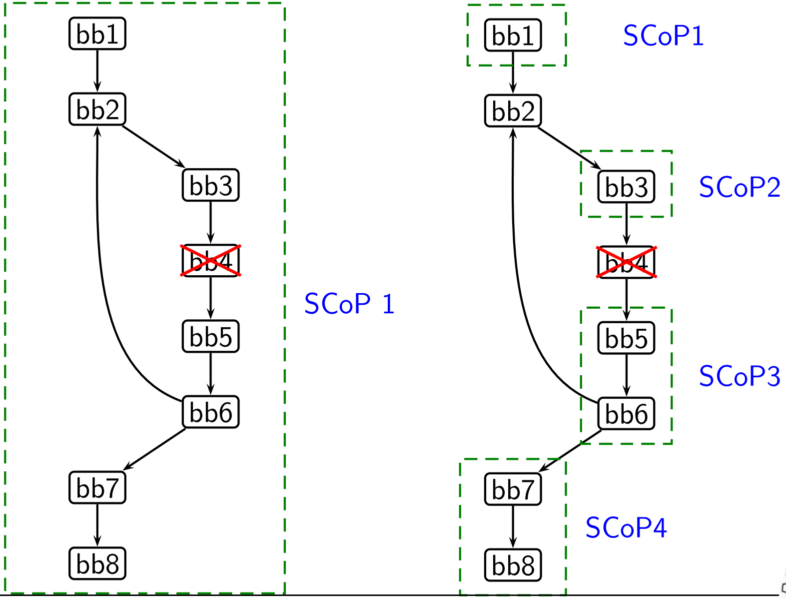
Example : Building SCoPs



Example : Building SCoPs

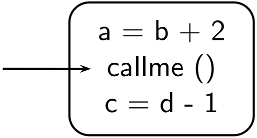


Example : Building SCoPs



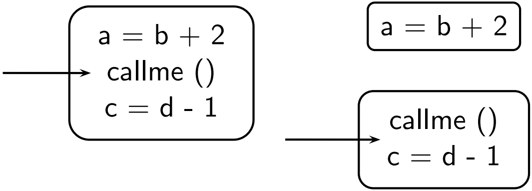
Example : Building SCoPs

Splitting basic blocks:



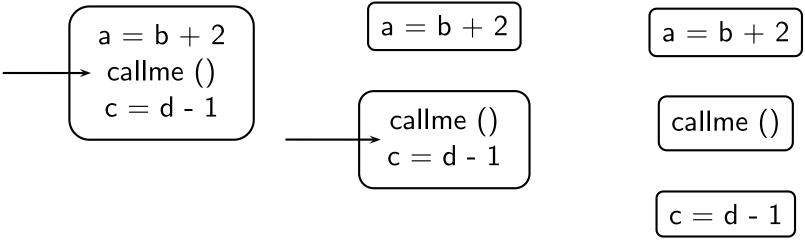
Example : Building SCoPs

Splitting basic blocks:



Example : Building SCoPs

Splitting basic blocks:



Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{(i,j) \mid 0 \leq i \leq m\text{-}1,\ 5 \leq j \leq n\text{-}1\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
       A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline & & & & \\ & & & & \\ & & & & \end{array}\right] \geq 0$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{(i,j) \mid 0 \leq i \leq m\text{-}1,\ 5 \leq j \leq n\text{-}1\}$$

```
for (i=0; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 1 & 0 & 0 & 0 & 0 \\ & & & & \\ & & & & \end{array}\right] \geq 0$$

i ≥ 0

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{(i,j) \mid 0 \leq i \leq m\text{-}1,\ 5 \leq j \leq n\text{-}1\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
        A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ & & & & \end{array}\right] \geq 0$$

$i \leq m - 1$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{(i,j) \mid 0 \leq i \leq m\text{-}1,\ 5 \leq j \leq n\text{-}1\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
        A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & -5 \end{array}\right] \geq 0$$

$j \geq 5$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)

$$\mathcal{D}^S = \{(i,j) \mid 0 \leq i \leq m-1,\ 5 \leq j \leq n-1\}$$

```
for (i=0; i<m; i++)
    for (j=5; j<n; j++)
       A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & -5 \\ 0 & -1 & 0 & 1 & -1 \end{array}\right] \geq 0$$

$j \leq n - 1$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)

$$\mathcal{F} = \{(i,a,s) \mid \mathrm{F} \times (i,a,s,g,1)^T \geq 0\}$$

```
for (i=1; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline & & & & \end{array}\right]$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)

$$\mathcal{F} = \{(i,a,s) \mid \mathrm{F} \times (i,a,s,g,1)^T \geq 0\}$$

```
for (i=1; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 2 & 0 & 0 & 0 & 0 \\ & & & & \end{array}\right]$$

2 * i

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)

$$\mathcal{F} = \{(i,a,s) \mid \mathrm{F} \times (i,a,s,g,1)^T \geq 0\}$$

```
for (i=1; i<m; i++)
   for (j=5; j<n; j++)
      A[2*i][j+1] = ...;
```

i	j	m	n	cst
2	0	0	0	0
0	1	0	0	1

j + 1

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

sequence $[s_1, s_2]$:

$\mathcal{S}[s_1] = t,$ $\qquad\qquad$ $\mathcal{S}[s_2] = t + 1$

loop $[loop_1\ s\ end_1]$: i_1 indexes $loop_1$ iterations

$\mathcal{S}[loop_1] = t,$ $\qquad\qquad$ $\mathcal{S}[s] = (t, i_1, 0)$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

```
for (i=1;i<=N;i++) {
  for (j=1;j<=i-1;j++) {
     a[i][i] -= a[i][j];
     a[j][i] += a[i][j];
  }
  a[i][i] = sqrt(a[i][i]);
}
```

Scattering Function

$$\theta_{S1}(i,j)^T = (0,i,0,j,0)^T$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

```
for (i=1;i<=N;i++) {
  for (j=1;j<=i-1;j++) {
     a[i][i] -= a[i][j];
     a[j][i] += a[i][j];
  }
  a[i][i] = sqrt(a[i][i]);
}
```

Scattering Function

$$\theta_{S2}(i,j)^T = (0,i,0,j,1)^T$$

Polyhedral Representation of a SCoP

The statements and parametric affine inequalities can be expressed by:

- Iteration Domain (bounds of enclosing loops)
- Data Reference (a list of access functions)
- Scattering Function (scheduling order)

```
for (i=1;i<=N;i++) {
  for (j=1;j<=i-1;j++) {
     a[i][i] -= a[i][j];
     a[j][i] += a[i][j];
  }
  a[i][i] = sqrt(a[i][i]);
}
```

Scattering Function

$$\theta_{S3}(i,j)^T = (0,i,1)^T$$

Analyses : Scalars, Arrays, Dependences

GRAPHITE built on top of:

- Scalar evolutions : number of iterations, access functions
- Array and pointer analysis
- Data dependence analysis (requires alias information)
- Scalar range estimations : undefined signed overflow, undefined access over statically allocated data, etc.

Dependence analysis in GRAPHITE

- Based on Violated Dependence Analysis
- Reuses the scalar evolution part to obtain the subscript bounds
- Depends heavily on may alias information
- Scalar dependences handled by converting them to zero-dimensional arrays
- Can take care of conditional and triangular loops, as the information can be safely integrated with the iteration domain
- High cost, and therefore dependence is computed only to validate a transformation

Integration of Parallelizer with GRAPHITE

Automatic parallelization integrated to GRAPHITE in GCC4.5.0

The initial analysis used for parallelizer was based on the Lambda Framework. It has been replaced with GRAPHITE based dependence analysis.

Benefits:

- More accurate dependence analysis, can detect more parallel loops
- Composition of program transformation can extract more parallelism
- Ease of incorporating a cost model

Integration of Parallelizer with GRAPHITE

Automatic parallelization integrated to GRAPHITE in GCC4.5.0

The initial analysis used for parallelizer was based on the Lambda Framework. It has been replaced with GRAPHITE based dependence analysis.

Benefits:

- More accurate dependence analysis, can detect more parallel loops
- Composition of program transformation can extract more parallelism
- Ease of incorporating a cost model

flags : `-ftree-parallelize-loops=`x, `-floop-parallelize-all`

Loop Transformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining

Loop Transformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining

Loop Interchange mostly used to improve scope of parallelization.

Loop Transformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining

Loop Interchange mostly used to improve scope of parallelization.

Original Code

```
for (i=0; i<n; i++){
   for (j=0; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

Loop Transformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining

Loop Interchange mostly used to improve scope of parallelization.

Original Code

```
for (i=0; i<n; i++){
   for (j=0; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

Outer Loop - dependence on i, can not be parallelized

Inner Loop - parallelizable, but synchronization barrier required

Total number of times synchronization executed = n

Loop Transformations in GRAPHITE

Loop transforms implemented in GRAPHITE:

- loop interchange
- loop blocking and loop stripmining

Loop Interchange mostly used to improve scope of parallelization.

Original Code

```
for (i=0; i<n; i++){
   for (j=0; j<n; j++){
       A[i][j] = A[i-1][j]
   }
}
```

After Interchange

```
for (j=0; j<n; j++){
   for (i=0; i<n; i++){
       A[i][j] = A[i-1][j]
   }
}
```

Outer Loop - parallelizable

Total number of times synchronization executed = 1

Loop Generation

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

Loop Generation

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

```
Set of constraints :

2 <= i <= n
2 <= j <= m
j <= n+2-i
m >= 2
n >= 2
```

Loop Generation

- *Chunky Loop Generator* (CLooG) is used to regenerate the loop
- It scans the integral points of the polyhedra to recreate loop bounds

```
Set of constraints :

2 <= i <= n
2 <= j <= m
j <= n+2-i
m >= 2
n >= 2
```

```
Loop generated by CLooG:

for (i=2; i<=n; i++)
   for (j=2; j<min(m,
      -i+n+2); j++) {
      S1(i,j);
   }
}
```

Part 4

Conclusions

Parallelization and Vectorization in GCC : Conclusions

- Chain of recurrences seems to be a useful generalization
- Interaction between different passes is not clear Predictive commoning and SSA seem to probihit many opportunities
- GRAPHITE dependence test is much more precise than Lambda Framework's dependence test. However, it has high complexity
- Auto-parallelization can be improved by enhancing the dependence analysis framework