

LOOP FUSION


Experience feedback on ~~hacking~~ implementation in PIPS

Mehdi AMINI – PIPS DAYS 25/10/2010

Idea : combine multiple loop nest into one

Ex :

```
for( i=0; i<N; i++) {  
    A[i] = .....;  
}  
for( i=0; i<N; i++) {  
    ... = A[i] .....;  
}
```



```
for( i=0; i<N; i++) {  
    A[i] = .....  
    .... = A[i] + .....;  
}
```

Pros :

- ✓ May improve data locality
- ✓ Reduce loop overhead
- ✓ Reduce synchronizations in case of parallel loops
- ✓ Enables array contraction
- ✓ May enable better instruction scheduling

Cons :

- ✓ May hurt data locality

What about the legality ?

« A loop-independent dependence between statement in two different loops (i.e. From S1 to S2) is *fusion preventing* if fusing the two loops causes the dependence to be carried by the combined loop in the reverse direction (from S2 to S1). » Kennedy & McKinley.

```
for( i=0; i<N; i++) {  
  A[i] = B[i] + C;  
}  
for( i=0; i<N; i++) {  
  D[i] = A[i+1] + E;  
}
```

FUSION

```
for( i=0; i<N; i++) {  
  A[i] = B[i] + C;  
  D[i] = A[i+1] + E;  
}
```

Backward carried

INVALID!

```
for( i=0; i<N; i++) {  
  B[i] = A[i] + C;  
}  
for( i=0; i<N; i++) {  
  A[i+1] = D[i] + E;  
}
```

FUSION

```
for( i=0; i<N; i++) {  
  B[i] = A[i] + C;  
  A[i+1] = D[i] + E;  
}
```

Backward carried

INVALID!

```
for( i=0; i<N; i++) {  
  A[i] = B[i] + C;  
}  
for( i=0; i<N; i++) {  
  A[i+1] = D[i] + E;  
}
```

FUSION

```
for( i=0; i<N; i++) {  
  A[i] = B[i] + C;  
  A[i+1] = D[i] + E;  
}
```

Backward carried

INVALID!

PIPS internals overview

```
for(i=low;i<sup;i+=inc)  
  S1.1  
  S1.2  
  S1.3  
}
```

Take two loops

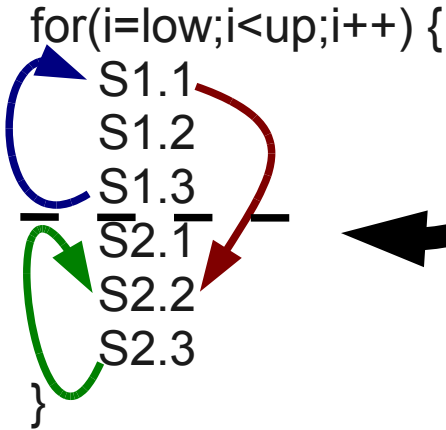
```
for(i=low;i<sup;i+=inc)  
  for(i=low;i<sup;i+=inc) Assert that headers  
  are compatible
```

```
for(i=low;i<sup;i+=inc)  
  S2.1  
  S2.2  
  S2.3  
}
```

Build a sequence like if it's fused

```
{  
  S1.1  
  S1.2  
  S1.3  
  S2.1  
  S2.2  
  S2.3  
}
```

Assert that there's no dependences from S2.X to S1.X



Build a new DG

```
for(i=low;i<sup;i+=inc)  
  S1.1  
  S1.2  
  S1.3  
  S2.1  
  S2.2  
  S2.3  
}
```

Replace the first loop body with the new Sequence and delete the second loop

Clean & free no longer used memory (optional)

PIPS internals overview

```
for(i=low;i<sup;i+=inc)  
  S1.1  
  S1.2  
  S1.3  
}
```

Take two loops

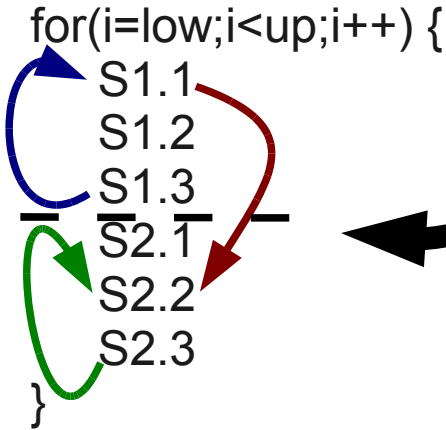
```
for(i=low;i<sup;i+=inc)  
  for(i=low;i<sup;i+=inc) Assert that headers  
    are compatible
```

```
for(i=low;i<sup;i+=inc)  
  S2.1  
  S2.2  
  S2.3  
}
```

Build a sequence like if it's fused

```
{  
  S1.1  
  S1.2  
  S1.3  
  S2.1  
  S2.2  
  S2.3  
}
```

Assert that there's no dependences from S2.X to S1.X

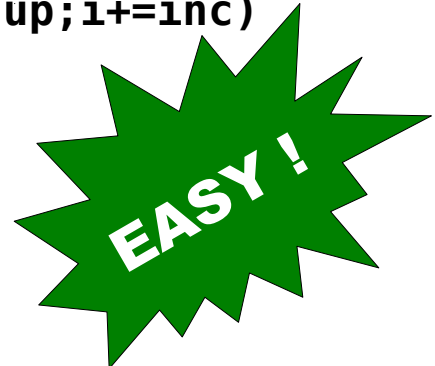


Build a new DG

Replace the first loop body with the new Sequence and delete the second loop

Clean & free no longer used memory (optional)

```
for(i=low;i<sup;i+=inc)  
  S1.1  
  S1.2  
  S1.3  
  S2.1  
  S2.2  
  S2.3  
}
```



Not in PIPS ;-)

Compatible header ?

aka “easy part”

```
for(i=low;i<sup;i+=inc)
  ↑
  ↓
for(i=low;i<sup;i+=inc)
```

```
/**
 * @brief Check that two loop statements have the same bounds
 */
static bool loops_have_same_bounds_p(loop loop1, loop loop2) {
    bool same_p = FALSE;

    range r1 = loop_range(loop1);
    range r2 = loop_range(loop2);

    same_p = range_equal_p(r1, r2);

    return same_p;
}

bool range_equal_p(range r1, range r2)
{
    return expression_equal_p(range_lower(r1), range_lower(r2))
        && expression_equal_p(range_upper(r1), range_upper(r2))
        && expression_equal_p(range_increment(r1), range_increment(r2));
}
```

Compatible header ?

aka “easy part”

```
for(i=low;i<sup;i+=inc)
  ↑ ↓
  ↑ ↓
  ↑ ↓
for(i=low;i<sup;i+=inc)
```

```
/**
 * @brief Check that two loop statements have the same bounds
 */
static bool loops_have_same_bounds_p(loop loop1, loop loop2) {
    bool same_p = FALSE;

    range r1 = loop_range(loop1);
    range r2 = loop_range(loop2);

    same_p = range_equal_p(r1, r2);

    return same_p;
}

bool range_equal_p(range r1, range r2)
{
    return expression_equal_p(range_lower(r1), range_lower(r2))
        && expression_equal_p(range_upper(r1), range_upper(r2))
        && expression_equal_p(range_increment(r1), range_increment(r2));
}
```

```
int start = 0, sup = 10, inc = 1;
```

```
for (i = start; i < sup; i += inc) {
    ...
}
start = 2; sup = 20; inc=2;
for (i = start; i < sup; i += inc) {
    ...
}
```

Hey ! But we are
doing store
dependent
comparison !

Compatible header ?

aka “easy part”

```
/**
 * @brief Check that two loop statements have the same bounds
 */
static bool loops_have_same_bounds_p(loop loop1, loop loop2) {
    bool same_p = FALSE;

    range r1 = loop_range(loop1);
    range r2 = loop_range(loop2);

    same_p = range_equal_p(r1, r2);

    return same_p;
}

bool range_equal_p(range r1, range r2)
{
    return expression_equal_p(range_lower(r1), range_lower(r2))
        && expression_equal_p(range_upper(r1), range_upper(r2))
        && expression_equal_p(range_increment(r1), range_increment(r2));
}
```

```
for(i=low;i<sup;i+=inc)
    ↓ ↓ ↓
for(i=low;i<sup;i+=inc)
```

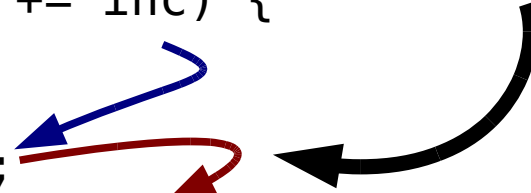
```
int start = 0, sup = 10, inc = 1;
```

```
for (i = start; i < sup; i += inc) {
    ...
}
```

```
start = 2; sup = 20; inc=2;
for (i = start; i < sup; i += inc) {
    ...
}
```

Hopefully dependences
will prevent that we try
to fuse these loops

Hey ! But we are
doing store
dependent
comparison !



Build&validate new candidate body

aka “(might be) easy part...”

```
for(i=low;i<sup;i+=inc)
```

```
S1.1 }  
S1.2 } body1  
S1.3 }
```

```
}  
for(i=low;i<sup;i+=inc)
```

```
S2.1 }  
S2.2 } body2  
S2.3 }
```

```
}
```

```
list fused = gen_concatenate(copy_statement(body1),  
                             copy_statement(body2));
```



```
{  
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

Build&validate a new candidate body

aka “(might be) easy part...”

```
for(i=low;i<sup;i+=inc)
```

```
S1.1 }  
S1.2 } body1  
S1.3 }
```

```
}  
for(i=low;i<sup;i+=inc)
```

```
S2.1 }  
S2.2 } body2  
S2.3 }
```

```
}
```

```
list fused = gen_concatenate(copy_statement(body1),  
                             copy_statement(body2));
```

```
{  
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

```
for(i=low;i<up;i++) {
```

```
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

Build
a new DG

Build&validate a new candidate body

aka "(might be) easy part..."

```
for(i=low;i<sup;i+=inc)
```

```
S1.1  
S1.2 } body1  
S1.3
```

```
for(i=low;i<sup;i+=inc)
```

```
S2.1  
S2.2 } body2  
S2.3
```

```
list fused = gen_concatenate(copy_statement(body1),  
                             copy_statement(body2));
```

```
{  
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

Assert that there's
no dependences
from S2.X to S1.X

```
for(i=low;i<up;i++) {
```

```
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

Build
a new DG

Replace the first
loop body with the new
Sequence and delete the
second loop

Clean & free no longer used
memory (optional)

```
for(i=low;i<sup;i+=inc)
```

```
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3
```

```
}
```

EASY!

Build&validate a new candidate body

aka "(might be) easy part..."

```
for(i=low;i<sup;i+=inc)
```

```
S1.1  
S1.2 } body1  
S1.3 }
```

```
for(i=low;i<sup;i+=inc)
```

```
S2.1  
S2.2 } body2  
S2.3  
}
```

Assert that there's
no dependences
from S2.X to S1.X

```
for(i=low;i<up;i++) {
```

```
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

```
list fused = gen_concatenate(copy_statement(body1),  
                              copy_statement(body2));
```

```
{  
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

Build
a new DG

Replace the first
loop body with the new
Sequence and delete the
second loop

Clean & free no longer used
memory (optional)

```
for(i=low;i<sup;i+=inc)
```

```
S1.1  
S1.2  
S1.3  
S2.1  
S2.2  
S2.3  
}
```

EASY!

Build&validate a new candidate body

aka “(might be) easy part... **Or not!**”

```
for(i=low;i<sup;i+=inc)
  S1.1
  S1.2
  S1.3
}
```

body1

```
for(i=low;i<sup;i+=inc)
  S2.1
  S2.2
  S2.3
}
```

body2

```
list fused = gen_concatenate(copy_statement(body1),
                             copy_statement(body2));
```

```
{
  S1.1
  S1.2
  S1.3
  S2.1
  S2.2
  S2.3
}
```

Assert that there's no dependences from S2.X to S1.X

```
for(i=low;i<sup;i+=inc)
  S1.1
  S1.2
  S1.3
  S2.1
  S2.2
  S2.3
}
```

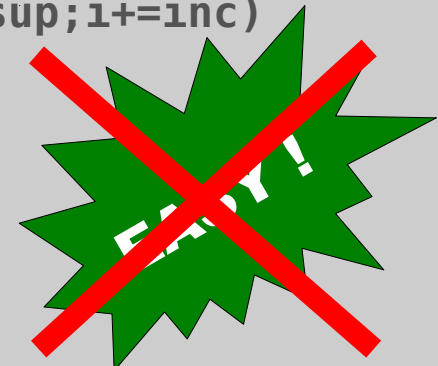
We are out of the PipsMake way!

Build a new DG

Replace the first loop body with the new Sequence and delete the second loop

Clean & free no longer used memory (optional)

```
for(i=low;i<sup;i+=inc)
  S1.1
  S1.2
  S1.3
  S2.1
  S2.2
  S2.3
}
```



Not in PIPS!

Build a DG out of PipsMake

aka “*the tricky part...*”

rice_full_dependence_graph > MODULE.dg

- < PROGRAM.entities
- < MODULE.code
- < MODULE.chains
- < MODULE.cumulated_effects

cumulated_effects > MODULE.cumulated_effects

- < PROGRAM.entities
- < MODULE.code
- < MODULE.proper_effects

atomic_chains > MODULE.chains

- < PROGRAM.entities
- < MODULE.code
- < MODULE.proper_effects

summary_effects > MODULE.summary_effects

- < PROGRAM.entities
- < MODULE.code
- < MODULE.cumulated_effects

proper_effects > MODULE.proper_effects

- < PROGRAM.entities
- < MODULE.code
- < **CALLEES**.summary_effects

Recursion

We'll *cheat* a little instead of mimic PipsMake

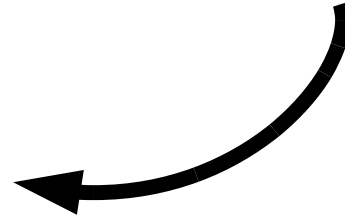
Prepare DG construction

aka “*the cheating & hacking part*”

```
list fused = gen_concatenate(copy_statement(body1),  
                             copy_statement(body2));
```



```
{  
  S1.1  
  S1.2  
  S2.1  
  S2.2  
}
```



We are working on the real original statements, **with effects attached to them !**

```
// Construct the fused body statement
```

```
statement fused_statement = make_block_statement(fused);
```

```
// Replace the loop body with the fused one
```

```
loop_body( loop1 ) = fused_statement;
```

```
// Cheat chains & dg on ordering
```

```
statement_ordering( fused_statement ) = 999999999; // FIXME : dirty  
add_ordering_of_the_statement_to_current_mapping(fused_statement);
```

```
// Cheat chains on proper_effects
```

```
store_proper_rw_effects_list(fused_statement, NIL);
```

```
// Cheat DG on enclosing_loops
```

```
set_enclosing_loops_map(loops_mapping_of_statement(sloop1));
```

Get the DG out of PIPS

aka “*should have been easy part*”

```
// Build chains on the new loop
graph chains = statement_dependence_graph(sloop1);

// Build DG on the new loop
graph dg = rdg_on_statement(sloop1);
```

If only it was so simple.... Rice dependence graph building is a kind of opaque black box, with a lot of static global variable, and `rdg_on_statement` is static and need initialization.

When hacking PIPS, axe might not be enough,
sometimes chain saw is required.

Go inside `ricedg` source code and add new entry point :

```
// have to be done before call :
// * set_ordering_to_statement
// * set_enclosing_loops_map
// * loading cumulated effects
graph compute_dg_on_statement_from_chains( statement s, graph chains )
```


LOOP FUSION

Part 2 : selection algorithm

Mehdi AMINI – PIPS DAYS 25/10/2010

```

#define N 100
void loop_fusion03( int a[N][N], int b[N][N] ) {
    int i, j;
    int k;

    /* These loop nests can be fused together, even
with the reduction on k */
    k = 0;
    for ( i = 0; i < N; i++ ) {
        for ( j = 0; j < N; j++ ) {
            a[i][j] = i + j;
        }
        k += a[i][j];
        for ( j = 0; j < N; j++ ) {
            b[i][j] += a[i][j];
        }
    }
}

```

```

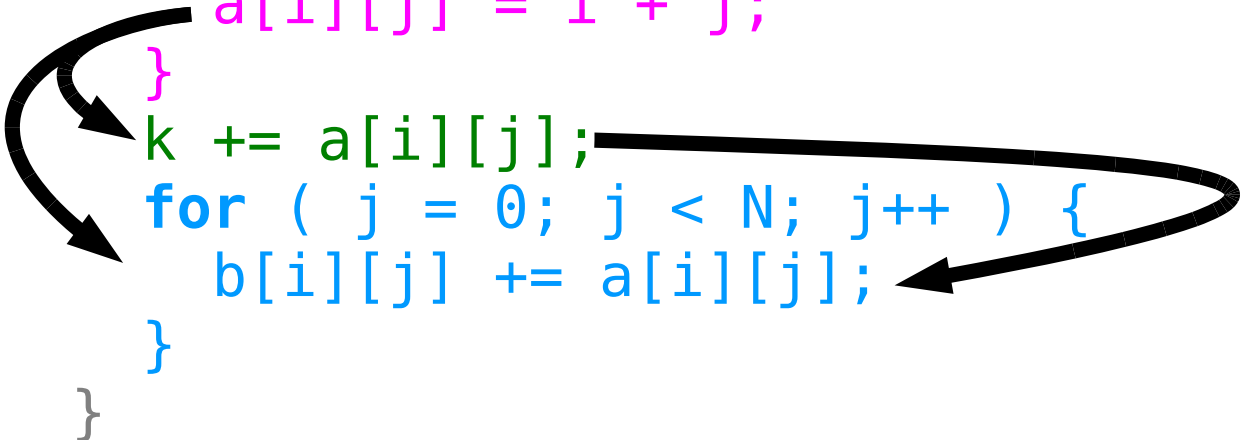
#define N 100
void loop_fusion03( int a[N][N], int b[N][N] ) {
    int i, j;
    int k;

    /* These loop nests can be fused together, even
with the reduction on k */
    k = 0;
    for ( i = 0; i < N; i++ ) {
        for ( j = 0; j < N; j++ ) {
            a[i][j] = i + j;
        }
        k += a[i][j];
        for ( j = 0; j < N; j++ ) {
            b[i][j] += a[i][j];
        }
    }
}

```

```
#define N 100
void loop_fusion03( int a[N][N], int b[N][N] ) {
    int i, j;
    int k;

    /* These loop nests can be fused together, even
with the reduction on k */
    k = 0;
    for ( i = 0; i < N; i++ ) {
        for ( j = 0; j < N; j++ ) {
            a[i][j] = i + j;
        }
        k += a[i][j];
        for ( j = 0; j < N; j++ ) {
            b[i][j] += a[i][j];
        }
    }
}
```

A diagram illustrating loop fusion. It features three black arrows. The first arrow starts at the closing brace of the innermost loop (the one containing 'a[i][j] = i + j;') and points to the reduction statement 'k += a[i][j];'. The second arrow starts at the closing brace of the middle loop (the one containing both 'a[i][j] = i + j;' and 'k += a[i][j];') and points to the reduction statement. The third arrow starts at the closing brace of the outermost loop (the one containing the two inner loops) and points to the reduction statement. This indicates that the reduction can be fused into the outer loop.

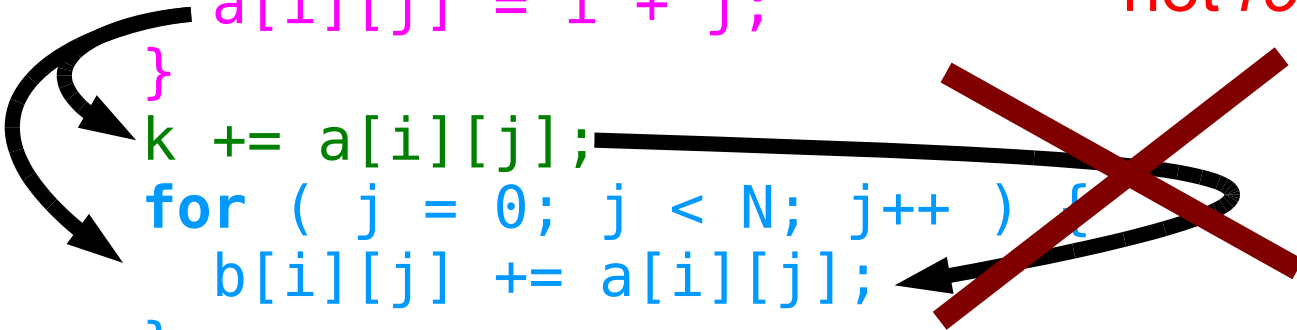
```

#define N 100
void loop_fusion03( int a[N][N], int b[N][N] ) {
    int i, j;
    int k;

    /* These loop nests can be fused together, even
with the reduction on k */
    k = 0;
    for ( i = 0; i < N; i++ ) {
        for ( j = 0; j < N; j++ ) {
            a[i][j] = i + j;
        }
        k += a[i][j];
        for ( j = 0; j < N; j++ ) {
            b[i][j] += a[i][j];
        }
    }
}

```

Read after read are
not *real* dependence !



```

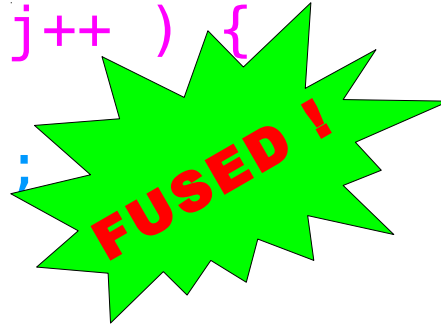
#define N 100
void loop_fusion03( int a[N][N], int b[N][N] ) {
    int i, j;
    int k;

    /* These loop nests can be fused together, even
with the reduction on k */
    k = 0;
    for ( i = 0; i < N; i++ ) {
        for ( j = 0; j < N; j++ ) {
            a[i][j] = i + j;
        }
        k += a[i][j];
        for ( j = 0; j < N; j++ ) {
            b[i][j] += a[i][j];
        }
    }
}

```

```
#define N 100
void loop_fusion03( int a[N][N], int b[N][N] ) {
    int i, j;
    int k;

    /* These loop nests can be fused together, even
with the reduction on k */
    k = 0;
    for ( i = 0; i < N; i++ ) {
        for ( j = 0; j < N; j++ ) {
            a[i][j] = i + j;
            b[i][j] += a[i][j];
        }
        k += a[i][j];
    }
}
```



Current limitations

- Fuse only inside sequence
- No fusion possible without dependence
- Headers compatible but with different loop indices are unsafe :-(
- Limited by DG accuracy, which is limited by chains & effects, which are waiting some work on *points_to* for C language
- Don't even think about running valgrind ! ;-)

Current limitations



- Fuse only inside sequence
- No fusion possible without dependence
- Headers compatible but with different loop indices are unsafe :-)
- Limited by DG accuracy, which is limited by chains & effects, which are waiting some work on *points_to* for C language
- Don't even think about running valgrind ! ;-)