Optimizing Loop Fusion

Paper: *Removing Impediments to Loop Fusion Through Code Transformations*

Group 11: Alec Korotney, Vijairam, Viroshan Narayan, Ibrahim Abouarabi

Intro to Loop Fusion

- **Loop Fusion:** method of code optimization via combining multiple loops into one
- Previous methods:
  - Objective function optimization
  - Emphasis on new data reuse or parallelism opportunities
- Issues?
  - Mismatched loops
    - Different trip counts
    - Not control flow equivalent
    - Etc.
  - Control dependencies

Intro to Loop Fusion

Benefits*

+ Fewer loop branch executions
+ Data reuse
+ Opportunities for parallelism (specific to hardware)

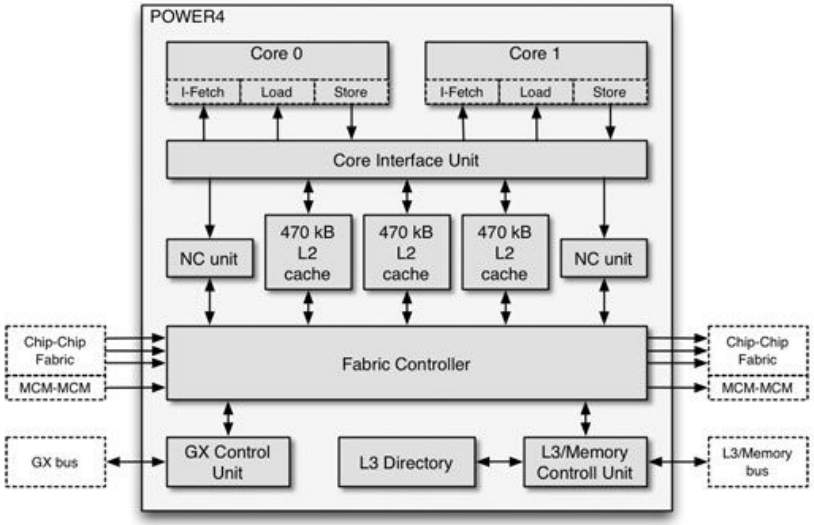
Drawbacks*

- Increased code size
- Higher register pressure
- Overutilization of hardware resources
- Complex control flow

* Dependent on nature of the loop fusion performed

What this paper implements

- Method for eliminating loop fusion **hindrances** first
- **Maximal** loop fusion
  - Greedy algorithm for selecting most profitable fusions
- Code aggregation
- Note: Specific to **IBM XL compiler suite**



Outline

- Initial code elimination
- Loop Fusion Pass
- Fusing Loops
- Results & Conclusions

Key Idea

- Eliminate hindrances early on, then fuse **aggressively**
- Delay *actual code aggregation* until **loop distribution**

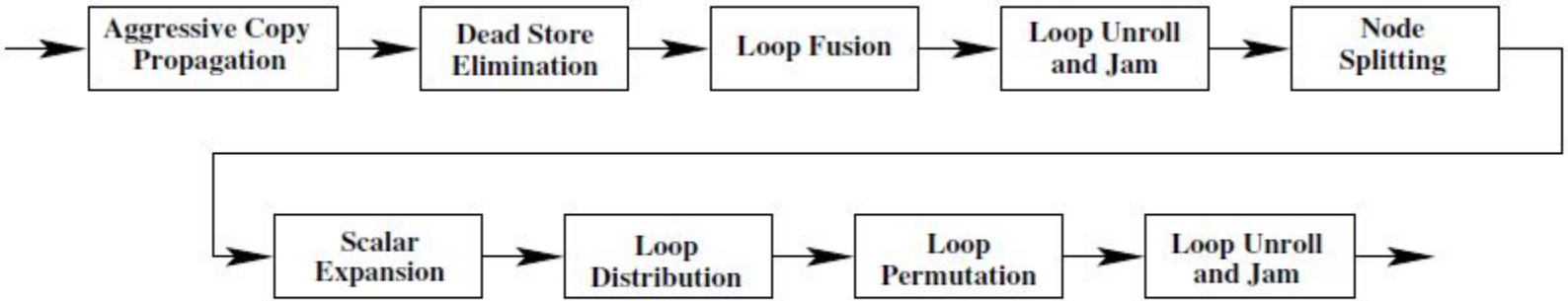


Fig. 1. Loop Optimizations

AGGRESSIVE COPY PROPAGATION AND DEAD STORE ELIMINATION

Conventional copy propagation algorithms do not move computations into a loop to prevent the enlargement of the dynamic path length of the loop. Our aggressive propagation,however, does move statements into a loop to enable the creation of perfectly nested loops.

```
for(i=0 ; i<k ; i++)
 {
   s = x*y;
   for(j=0 ; j<n ; j++)
     V[i][j] = V[i][j] + s;
 }
```

(a)

```
for(i=0 ; i<k ; i++)
  {
   s = x*y;
   for(j=0 ; j<n ; j++)
     V[i][j] = V[i][j] + x*y;
  }
```

(b)

```
for(i=0 ; i<k ; i++)
  {
   for(j=0 ; j<n ; j++)
     V[i][j] = V[i][j] + x*y;
  }
```

(c)

Fig. 2. Example of aggressive copy propagation followed by dead store elimination

Loop Distribution

loop-distribution is useful in isolating parts of the loop that can be parallelized.

Before Loop Distribution

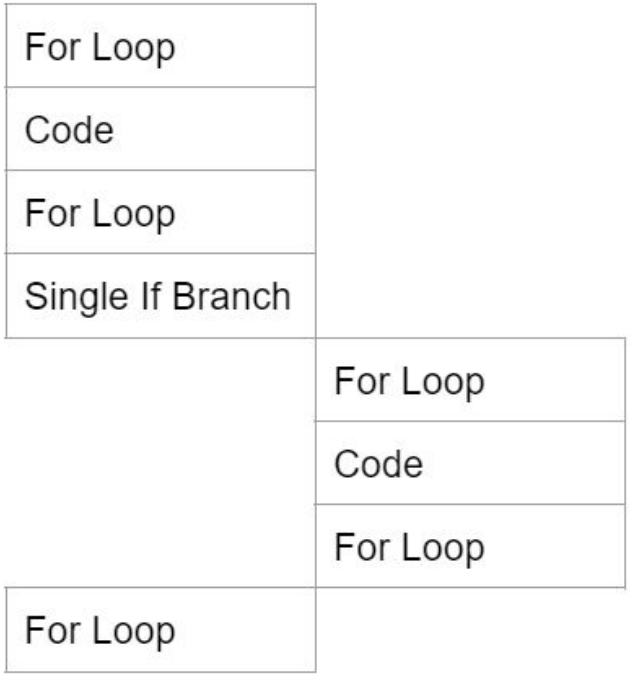
```
    for i = 1:N  {
S1      A(i) = B(i) + 1;
S2      C(i) = A(i) + C(i-1);
S3      D(i) = A(i) + X;
    }
```

After Loop Distribution

```
    for i = 1:N  {
S1      A(i) = B(i) + 1;
    }
    for i = 1:N  {
S2      C(i) = A(i) + C(i-1);
    }
    for i = 1:N  {
S3      D(i) = A(i) + X;
    }
```

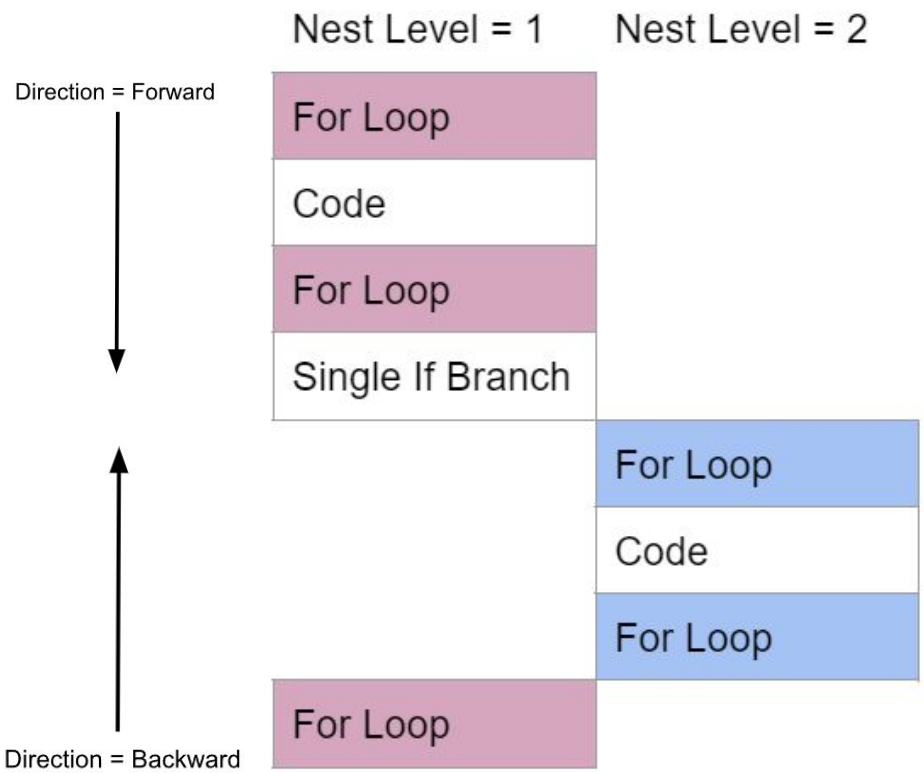
Loop Fusion Pass

```
LoopFusionPass(S_i, Direction)
1.  FusedLoops = False
2.  foreach pair of loops L_j and L_k in S_i, such that L_j
        dominates L_k, in Direction
3.      if InterveningCode(L_j, L_k) = True and
           IsInterveningCodeMovable(L_j, L_k) = False
4.          continue
5.      endif
6.      σ ← |κ(L_j) − κ(L_k)|
7.      if L_j and L_k are non-conforming and
           σ cannot be determined at compile time
8.          continue
9.      endif
10.     if DependenceDistance(L_j, L_k) < 0
11.         continue
12.     endif
13.     MoveInterveningCode(L_j, L_k, Direction)
14.     if InterveningCode(L_j, L_k) = False
15.         if L_j and L_k are non-conforming
16.             L_m ← FuseWithGuard(L_j, L_k)
17.         else
18.             L_m ← Fuse(L_j, L_k)
19.         endif
20.         S_i ← S_i ∪ L_m − {L_j, L_k}
21.         FusedLoops = True
22.     else
23.         continue
24.     endif
25. endfor
26. return FusedLoops
```



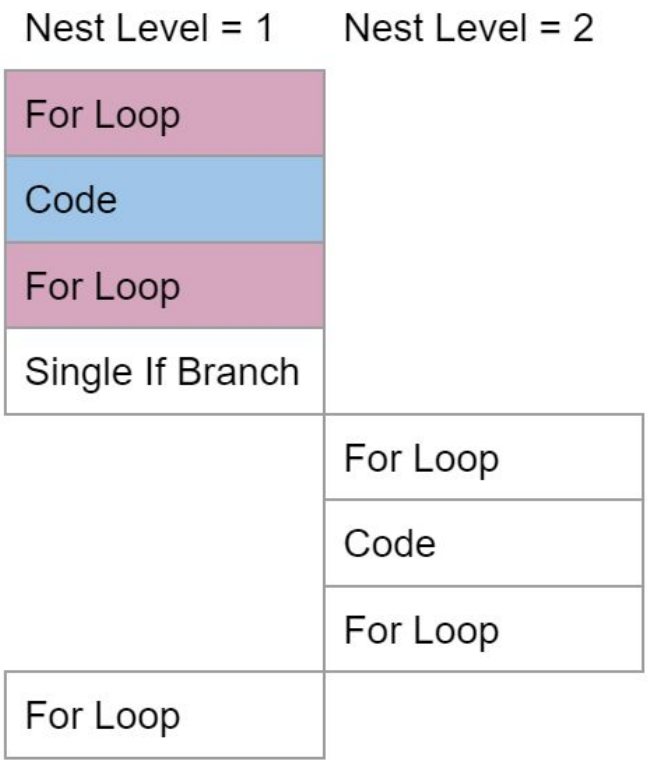
Loop Fusion Pass

```
LoopFusionPass(S_i, Direction)
1.  FusedLoops = False
2.  foreach pair of loops L_j and L_k in S_i, such that L_j
        dominates L_k, in Direction
3.      if InterveningCode(L_j, L_k) = True and
           IsInterveningCodeMovable(L_j, L_k) = False
4.          continue
5.      endif
6.      σ ← |κ(L_j) − κ(L_k)|
7.      if L_j and L_k are non-conforming and
           σ cannot be determined at compile time
8.          continue
9.      endif
10.     if DependenceDistance(L_j, L_k) < 0
11.         continue
12.     endif
13.     MoveInterveningCode(L_j, L_k, Direction)
14.     if InterveningCode(L_j, L_k) = False
15.         if L_j and L_k are non-conforming
16.             L_m ← FuseWithGuard(L_j, L_k)
17.         else
18.             L_m ← Fuse(L_j, L_k)
19.         endif
20.         S_i ← S_i ∪ L_m − {L_j, L_k}
21.         FusedLoops = True
22.     else
23.         continue
24.     endif
25. endfor
26. return FusedLoops
```



Loop Fusion Pass

```
LoopFusionPass(S_i, Direction)
1.  FusedLoops = False
2.  foreach pair of loops L_j and L_k in S_i, such that L_j
        dominates L_k, in Direction
3.      if InterveningCode(L_j, L_k) = True and
           IsInterveningCodeMovable(L_j, L_k) = False
4.          continue
5.      endif
6.      σ ← |κ(L_j) − κ(L_k)|
7.      if L_j and L_k are non-conforming and
           σ cannot be determined at compile time
8.          continue
9.      endif
10.     if DependenceDistance(L_j, L_k) < 0
11.         continue
12.     endif
13.     MoveInterveningCode(L_j, L_k, Direction)
14.     if InterveningCode(L_j, L_k) = False
15.         if L_j and L_k are non-conforming
16.             L_m ← FuseWithGuard(L_j, L_k)
17.         else
18.             L_m ← Fuse(L_j, L_k)
19.         endif
20.         S_i ← S_i ∪ L_m − {L_j, L_k}
21.         FusedLoops = True
22.     else
23.         continue
24.     endif
25. endfor
26. return FusedLoops
```



Loop Fusion Pass

```
IsInterveningCodeMovable(L_j, L_k)
1.  InterveningCodeSet ← {a_x | L_j ≺_d a_x and L_k ≺_pd a_x}
2.  if any node in InterveningCodeSet is non-movable
3.     return False
4.  Build a DDG G of InterveningCodeSet
5.  CanMoveUpSet ← ∅
6.  foreach a_y ∈ G in topological order
7.        if CanMoveUp(Predecessors(a_y)) and L_j ⊀_δ a_y
8.           CanMoveUpSet ← CanMoveUpSet ∪{a_y}
9.        endif
10. endfor
11. CanMoveDownSet ← ∅
12  foreach a_z ∈ G in reverse topological order
13.       if CanMoveDown(Successors(a_z)) and a_z ⊀_δ L_k
14.          CanMoveDownSet ← CanMoveDownSet ∪{a_z}
15.       endif
16. endfor
17. if InterveningCodeSet −
       (CanMoveUpSet ∪ CanMoveUpSet) = ∅
18.    return True
19. return False
```

- Code is split into *aggregate nodes*, which are minimum code segments that must be moved as a unit
- An aggregate node is intervening if it is dominated by the preceding loop and post dominated by the following loop
- An aggregate node is non-movable if it has side effects (ex. volatile load/store, I/O)
- Aggregate nodes can be moved up or down if there are no data dependencies between the loop and adjacent intervening code

Loop Fusion Pass

```
LoopFusionPass(S_i, Direction)
1.  FusedLoops = False
2.  foreach pair of loops L_j and L_k in S_i, such that L_j
        dominates L_k, in Direction
3.      if InterveningCode(L_j, L_k) = True and
           IsInterveningCodeMovable(L_j, L_k) = False
4.          continue
5.      endif
6.      σ ← |κ(L_j) − κ(L_k)|
7.      if L_j and L_k are non-conforming and
           σ cannot be determined at compile time
8.          continue
9.      endif
10.     if DependenceDistance(L_j, L_k) < 0
11.         continue
12.     endif
13.     MoveInterveningCode(L_j, L_k, Direction)
14.     if InterveningCode(L_j, L_k) = False
15.         if L_j and L_k are non-conforming
16.             L_m ← FuseWithGuard(L_j, L_k)
17.         else
18.             L_m ← Fuse(L_j, L_k)
19.         endif
20.         S_i ← S_i ∪ L_m − {L_j, L_k}
21.         FusedLoops = True
22.     else
23.         continue
24.     endif
25. endfor
26. return FusedLoops
```

- Attempt symbolic subtraction between the upper bounds of the normalized loops
- Stop if the difference can't be computed at compile time or there are negative distance dependencies between the loops
- Move any intervening code identified in the previous step

Loop Fusion Pass

```
LoopFusionPass(S_i, Direction)
1.  FusedLoops = False
2.  foreach pair of loops L_j and L_k in S_i, such that L_j
        dominates L_k, in Direction
3.      if InterveningCode(L_j, L_k) = True and
          IsInterveningCodeMovable(L_j, L_k) = False
4.        continue
5.      endif
6.      σ ← |κ(L_j) − κ(L_k)|
7.      if L_j and L_k are non-conforming and
          σ cannot be determined at compile time
8.        continue
9.      endif
10.     if DependenceDistance(L_j, L_k) < 0
11.       continue
12.     endif
13.     MoveInterveningCode(L_j, L_k, Direction)
14.     if InterveningCode(L_j, L_k) = False
15.       if L_j and L_k are non-conforming
16.         L_m ← FuseWithGuard(L_j, L_k)
17.       else
18.         L_m ← Fuse(L_j, L_k)
19.       endif
20.       S_i ← S_i ∪ L_m − {L_j, L_k}
21.       FusedLoops = True
22.     else
23.       continue
24.     endif
25. endfor
26. return FusedLoops
```

Fuse the loops if all requirements are met:

1. Fuse the code into a single loop
2. Update our set of loops, S_i

How do we fuse the code?

Fusing Conforming Loops

- Loops are **conforming** if they have **identical trip counts**
- Fusion is just combining the contents into a single loop

```
for (int i = 0; i < 20; i++) {
    a[i] = a[i] * 4;
}

for (int i = 0; i < 20; i++) {
    b[i] = b[i] - 20;
}
```

```
for (int i = 0; i < 20; i++) {
    a[i] = a[i] * 4;
    b[i] = b[i] - 20;
}
```

Fusing Non-Conforming Loops

- Need to be able to **statically determine difference** in trip count
- Merge the loops with a guard - **FuseWithGuard**

```
for (int i = 0; i < n; i++) {
    a[i] = a[i] * 4;
}

for (int i = 0; i < n - 2; i++)
{
    b[i] = b[i] - 20;
}
```

```
for (int i = 0; i < n; i++) {
    if (i < n-2) {
        a[i] = a[i] * 4;
        b[i] = b[i] - 20;
    }
    else {
        a[i] = a[i] * 4;
    }
}
```

Fusing Non-Conforming Loops

Guarding One Loop (Alternative)

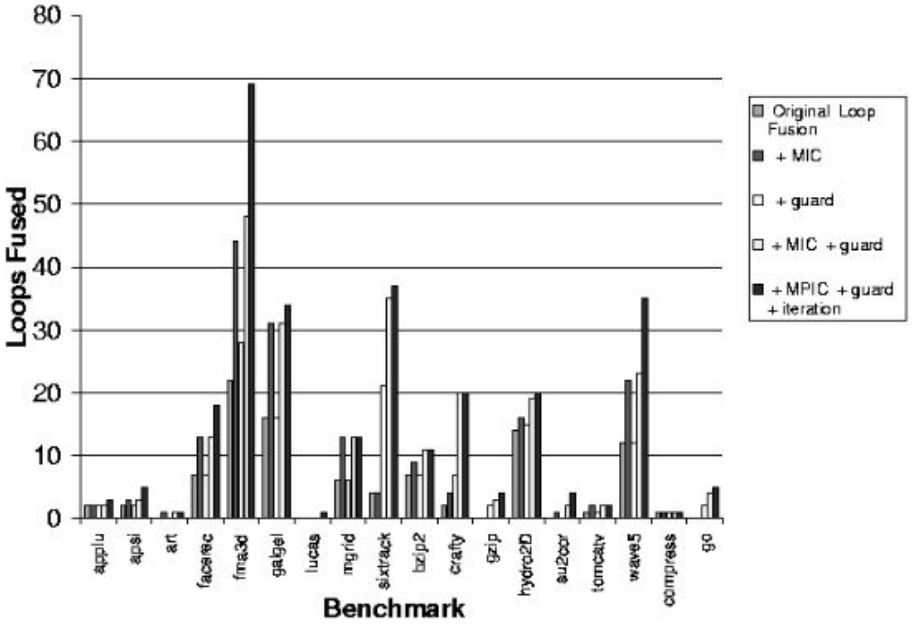
```
for (int i = 0; i < n; i++) {
    a[i] = a[i] * 4;

    if (i < n-2) {
        b[i] = b[i] - 20;
    }
}
```

Guarding Both Loops (Proposed)

```
for (int i = 0; i < n; i++) {
    if (i < n-2) {
        a[i] = a[i] * 4;
        b[i] = b[i] - 20;
    }
    else {
        a[i] = a[i] * 4;
    }
}
```

Creates **more code-growth** than simply guarding the second loop, but it lets the **fused-loop contents stay together** (better for later potential optimizations)

Results



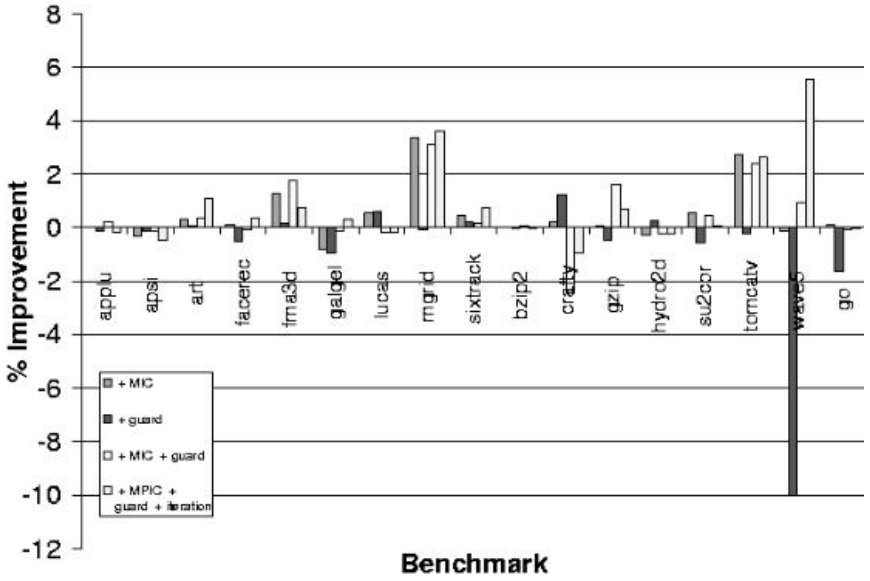
MIC - Single pass w/ moving intervening code

guard - fuse non-conforming loops

MPIC - move partial intervening code

iteration - keep doing algo until you can’t anymore

Results



MIC - Single pass w/ moving intervening code

guard - fuse non-conforming loops

MPIC - move partial intervening code

iteration - keep doing algo until you can’t anymore

Results

- Only fusing non-conforming loops (+guard) yielded little benefit (if any)
- The algorithm generally led to performance improvement, and losses were very low if it didn’t
- In some cases, the algorithm led to performance gains in the range of 1-5%!

Conclusion

Pros

- General performance increases
- Fuses more loops -> fewer overall loop branches
- Increased parallelism
- More optimization can happen in later compilation stages

Cons

- Can result in some performance loss
- May inhibit software pipelining (guard adds control flow)
- Increased code size

Q&A