Compiler Transformations

High-Level Dataflow and Loop Optimization for High-Performance Parallel Computing through the SUIF Compiler

Giovanni Agosta
11/27/00
Summary

- Dataflow Optimization
- Partial Evaluation
- Redundancy Elimination
- Loop Optimization
  - Preparation phase
  - Reordering
  - Post-Processing
Parallelism

• Task level

  *Two procedures running in parallel*

• Loop level

  *Two iterations of the same loop executed at the same time*

• Instruction level

  *Two instruction executed in parallel*
High-level Optimization

- Simplification of the program
- Removal of extraneous code
- Preparation for further optimization passes
Dataflow Optimization

- Constant Propagation & Folding
- Copy Propagation
- Common Subexpression Elimination
- Loop–Invariant Code Motion
- Loop Unswitching
Constant & Copy Propagation

- Constant Propagation replaces instances of variables whose value is known at compile time with said value.
- Constant Folding replaces expressions whose value is known at compile time with the result.
- Copy Propagation replaces copies of a variable with the original name, eliminating redundant copies.
Constant Propagation

- Original Code
  
  ```c
  int i, a[64];
  int n = 64;
  int c = 3;
  for(i=0; i<n; i++)
    a[i] = a[i] + c;
  ```

- After Constant Propagation
  
  ```c
  int i, a[64];
  for(i=0; i<64; i++)
    a[i] = a[i] + 3;
  ```
Copy Propagation

- **Original Code**
  ```c
  main(int i, int c){
    int s, t, a[64];
    t = i*4;
    s = t;
    a[t] = a[t] + c;
  }
  ```

- **After Copy Propagation**
  ```c
  main(int i, int c){
    int t, a[64];
    t = i*4;
    a[t] = a[t] + c;
  }
  ```
Common Subexpression Elimination

- Replaces multiple instances of an expression with a temporary variable holding its result
- Through common subexpression elimination, a subexpression is evaluated only once
- This optimization may have undesirable side effects, for it generates new temporary variables, which may need to be stored in memory
Common Subexpression Elimination

- Original Code
  ```c
  main(int i){
      int s, t, a[64];
      t = i*4;
      s = i*4;
      a[t] = a[t] + s;
  }
  ```

- After Common Subexpression Elimination
  ```c
  main(int i, int c){
      int t, a[64];
      t = i*4
      a[t] = a[t] + t;
  }
  ```
Partial Evaluation

• Algebraic Simplification
  • Simplifies arithmetic expressions by applying algebraic rules
  • Sample rule: \( x + 0 = x \)

• Short–Circuiting
  • Stops evaluating boolean expression when the result is known
  • It can change the result of the evaluation if the expression has side effects
  • Short–Circuiting is required by the C language
Algebraic Simplification

• Original Code
  ```c
  main(int x, int y)
  {
    x = (y*1 + 0)/1;
  }
  ```

• After Algebraic Simplification
  ```c
  main(int x, int y)
  {
    x = y;
  }
  ```
Short-Circuiting

- Original Code
  ```c
  int main(int x, int y)
  {
    if((x==1) && (y==2)) return 5;
  }
  ```

- After Short-Circuiting
  ```c
  int main(int x, int y)
  {
    if(x==1)
      if(y==2) return 5;
  }
  ```
Redundancy Elimination

- Unreachable Code Elimination
  - Removes code which may never be reached, due to previous optimizations

- Useless Code Elimination
  - Removes the defining statement of a variable which is not live immediately after its definition

- Dead Variable Elimination
  - Removes variables whose value is never used
Unreachable Code Elimination

• Original Code
  ```c
  int main(){
    int x=1;
    if(x>1) return 5;
    else return x;}
  ```

• After Constant Propagation and Unreachable Code Elimination
  ```c
  int main(){
    int x=1;
    return 1;
  }
  ```
Useless Code Elimination

- **Original Code**
  ```c
  int main()
  {
    int x=1;
    return 1;
  }
  ```

- **After Useless Code Elimination**
  ```c
  int main()
  {
    int x;
    return 1;
  }
  ```
Dead Variable Elimination

- **Original Code**
  ```c
  int main()
  {
    int x;
    return 1;
  }
  ```

- **After Useless Code Elimination**
  ```c
  int main()
  {
    return 1;
  }
  ```
Loop Preparation

- Loop Normalization
- Loop Peeling
- Forward Substitution
- Reduction Recognition
Loop Normalization

- **Original Code**
  ```c
  int main()
  {
    int i, x[10];
    for(i=1; i<10; i++)
      x[i]=i+1;
  }
  ```

- **After Loop Normalization**
  ```c
  int main()
  {
    int i, x[10];
    for(i=0; i<9; i++)
      x[i+1]=(i+1)+1;
  }
  ```
Loop Peeling

- Original Code
  ```c
  int i, x[10], y[10];
  for(i=1; i<10; i++) x[i] = i+1;
  for(i=2; i<10; i++) y[i] = i*3;
  ```

- After Loop Normalization and Fusion
  ```c
  int i, x[10], y[10];
  x[1]=2;
  for(i=2; i<10; i++){
    x[i] = i+1;
    y[i] = i*3;
  }
  ```
Forward Substitution

• Original Code
  ```c
  int i, x, y, a[10];
  y = x + 1;
  for(i=0; i<x; i++)
  a[y] = a[y] + a[i];
  ```

• After Forward Substitution
  ```c
  int i, x, y, a[10];
  for(i=0; i<x; i++)
  a[x+1] = a[x+1] + a[i];
  ```
Loop Reordering

- Loop Unrolling
- Loop Interchange
- Loop Skewing
- Loop Tiling
- Loop Coalescing
- Strip Mining
Loop Unrolling

- **Original Code**
  
  ```c
  int i, n, a[10];
  for(i=2; i<n; i++)
    a[i] = a[i] + a[i-1] * a[i+1];
  ```

- **After Loop Unrolling**
  
  ```c
  int i, a[10];
  for(i=2; i<n-1; i++){
    a[i] = a[i] + a[i-1] * a[i+1];
    a[i+1] = a[i+1] + a[i] * a[i+2];
  }
  if(n%2)
    a[n-1] = a[n-1] + a[i-2] * a[n];
  ```
Loop Interchange

- **Original Code**
  ```c
  int i, s[10], a[10][10];
  for(i=0; i<10; i++)
    for(j=0; j<10; j++)
      s[i] = s[i] + a[i,j];
  ```

- **After Loop Interchange**
  ```c
  int i, s[10], a[10][10];
  for(j=0; j<10; j++)
    for(i=0; i<10; i++)
      s[i] = s[i] + a[i,j];
  ```
Loop Skewing

- **Original Code**
  
  ```c
  int i, s[N], a[N][N];
  for(i=1; i<N-1; i++)
    for(j=1; j<N-1; j++)
      a[i,j] = (a[i-1,j] + a[i,j-1] + a[i+1,j] + a[i,j+1]) / 4;
  ```

- **Original Space**
  - Neither loop can be parallelized
  - Wavefront could be parallelized
  - Wavefront = array diagonal
Loop Skewing

- **After Loop Skewing**

  ```c
  int i, s[N], a[N][N];
  for(i=1; i<N−1; i++)
    for(j=i+1; j<i+N−1; j++)
      a[i,j−i] =
        (a[i−1,j−i] + a[i,j−1−i]
        a[i+1,j−i] + a[i,j+1−i]) / 4;
  ```

- **Skewed Space**
  - Both loop can now be parallelized
  - A Loop Interchange is needed
Strip Mining & Loop Tiling

- Useful to adjust the granularity of vector operations
- Both generate a chosen number of independent computations from a parallelizable loop
- Loop Tiling is a multidimensional extension of Strip Mining, working on nested loops
- Other specialized versions of Strip Mining exist, aiming to expose parallelism hidden in sequential loops (ex.: Cycle Shrinking)
Loop Post-Processing

- Loop Distribution
- Loop Fusion
- Loop Spreading
Loop Distribution

- Original Code
  
  ```c
  for(i=0; i<10; i++)
  {
      a[i] = a[i] + c;
      s[i] = s[i] + s[i+1] + a[i];
  }
  ```

- After Loop Distribution
  
  ```c
  for(i=0; i<10; i++)
  {
      a[i] = a[i] + c;
  }
  for(i=0; i<10; i++)
  {
      s[i] = s[i] + s[i+1] + a[i];
  }
  ```
Loop Fusion

- Loop Fusion is the inverse transformation of Loop Distribution.
- It can have the following beneficial effects:
  - reducing loop overhead
  - increasing instruction level parallelism
  - improving data locality
  - improving the load balance of parallel loops
Loop Spreading

• Original Code
  
  ```
  for(i=0; i<5; i++)
      a[i] = a[i] + c;
  for(i=0; i<10; i++)
      s[i] = s[i] + s[i+1];
  ```

• After Loop Spreading
  
  ```
  for(i=0; i<5; i++) {
      a[i] = a[i] + c;
      s[i] = s[i] + s[i+1];
  }
  for(i=5; i<10; i++)
      s[i] = s[i] + s[i+1];
  ```