Trasformazioni del codice sorgente

Trasformazioni sui cicli

- I cicli sono i migliori candidati per le ottimizzazione
  - Sono eseguiti molte volte
  - Sono porzioni di codice relativamente piccole e localizzate
- Le principali trasformazioni sui cicli sono le seguenti
  - Loop urolling
  - Induction variables
  - Loop unswitch
  - Loop fusion
  - Loop interchange
  - Loop tiling
  - Loop peeling
  - While-do / do-while
  - Zero output condition
  - Software pipelining

Loop urolling

- Trasformazione
  - Replica il codice del corpo di un ciclo per un certo numero di volte
- Obiettivo
  - Aumentare la dimensone del/dei blocchi base che costituiscono il corpo del ciclo al fine di permettere al compilatore di eseguire ottimizzazioni con maggiore efficienza
- Note
  - Unrolling completo
    - Possibile solo se i limiti di iterazione sono noti a compile time
  - Unrolling di un fattore N
    - Sempre possibile
    - Poco conveniente quando il numero minimo e medio di iterazioni di un cilco è minore di N
  - La trasformazione in sé è semplice
    - In cascata è bene ricorrere alla propagazione delle costanti e all'analisi delle variabili di induzione

Loop urolling

- Si consideri un ciclo
  - K iterazioni
  - N fattore di unrolling
- Struttura del codice traformato
  - Prologo
    - Gestisce i casi in cui K<N
  - Corpo
    - Esegue il ciclo (K/N) volte
    - Il corpo del ciclo è ottenuto replicando il copo originale N volte
    - Richiede l'espansione dello statement di incremento di un ciclo for
  - Epilogo
    - Gestisce le ultime (K mod N) iterazioni
- Alcuni compilatori
  - Dispongono di opportune direttive per ottimizzare la gestione del prologo e dell'epilogo

Loop urolling - Esempio

Codice originale

```
/* K is known to be 4 */

for( i = 1; i < K; i++ ) {
  y[i] = y[i-1] + x[i];
}
```

Codice trasformato

```
/* Full unrolling */

y[1] = y[0] + x[1];
y[2] = y[1] + x[2];
y[3] = y[2] + x[3];
y[4] = y[3] + x[4];
```

Codice originale

```
/* K is known to be greather than 2 */

for( i = 1; i < K; i++ ) {
  y[i] = y[i-1] + x[i];
}
```

Codice trasformato

```
/* Unrolling with N=2 */

for( i = 1; i < K; i += 2 ) {
  y[i]   = y[i-1] + x[i];
  y[i+1] = y[i] + x[i+1];
}
if( K % 2 == 0 ) {
   y[K-1] = y[K-2] + x[K-1];

}
```

Codice originale

```
/* K is known to be 4 */

t = 0;
for( j = 0; j < K; j++ ) {
  t = t + x[K-j-1] * (j+1);
}
```

Codice trasformato

```
/* Full unrolling */

t = 0;
j = 0;
t = t + x[4-j-1] * (j+1);
j++;
t = t + x[4-j-1] * (j+1);
j++;
t = t + x[4-j-1] * (j+1);
j++;
t = t + x[4-j-1] * (j+1);
```

Induction variables

- Trasformazione
  - Individuare variabili che sono funzioni dell'indice del ciclo
  - Sostituisce tali variabili con espressioni più semplici
- Obiettivo
  - Calcolare le variabili induttive in modo più semplice
  - Migliorare un ciclo dopo l'unrollig
- Note
  - In genere i compilatori
    - Identificano solo dipendenze lineari
    - Identificano solo dipendenze da una variabile indice e trascurano i cicli annidati
    - Considerano solamente variabili induttive scalari

Induction variables - Esempio

Codice originale

```
for( i = 0; i < K; i++ ) {
  y[i] = y[i] + x[2*i];
}
```

Codice trasformato

```
j = 0;
for( i = 0; i < K; i++ ) {
  y[i] = y[i] + x[j];
  j+=2;
}
```

Codice originale

```
for( i = 0; i < K; i++ ) {
  y[i] = y[i] + x[K-i];
}
```

Codice trasformato

```
j = K;
for( i = 0; i < K; i++ ) {
  y[i] = y[i] + x[j];
  j--;
}
```

Codice originale

```
for( r = 0; r < ROW; r++ ) {
  for( c = 0; c < COL; c++ ) {
    t = t + x[ r * COL + c ];
}
```

Codice trasformato

```
i=0
for( r = 0; r < ROW; r++ ) {
  for( c = 0; c < COL; c++ ) {
    i++;
    t = t + x[i];
}
```

Loop unswitch

- Trasformazione
  - Sposta condizioni invarianti fuori dal ciclo
- Obiettivo
  - Evitare di ripetere test inutili
  - Diminuire il numero di salti necessari
    - Minimizzare gli errori di branch prediction
- Note
  - In genere i compilatori eseguono il loop unswitch
    - A patto di poter determinare che una data operazione è invariante
  - Se vi sono chiamate di funzione coinvolte
    - Raramente il compilatore interviene

Loop unswitch - Esempio

Codice originale

```
for( i = 0; i < K; i++ ) {
  if( N > 0 ) {
    y[i] = y[i] + x[K-i];
  } else {
    y[i] = y[i] * x[i];
  }
}
```

Codice trasformato

```
if( N > 0 ) {
  for( i = 0; i < K; i++ ) {
    y[i] = y[i] + x[K-i];
  }
} else {
 for( i = 0; i < K; i++ ) {
    y[i] = y[i] * x[i];
  }
}
```

Loop fusion

- Trasformazione
  - Fonde il corpo di due cicli
- Obiettivo
  - Minimizzare gli accessi alla memoria e ridurre i cache miss
    - Quando i cicli operano su strutture dati comuni
  - Diminuire il numero di salti necessari
    - Minimizzare gli errori di branch prediction
  - Diminuire il numero di istruzioni di controllo
    - A volte le condizioni di uscita dal ciclo possono essere complesse
- Note
  - In genere i compilatori eseguono la loop fusion
    - Quando i cicli sono “vicini”
    - Quando i limiti di iterazione sono identici
    - Quando la “direzione” di iterazione è identica (incremento, decremento)

Loop fusion - Esempio

Codice originale

```
for( i = 0; i < 100; i++ )
  a[i] = 1;

for( i = 0; i < 100; i++ )
  b[i] = 2;
```

Codice trasformato

```
for( i = 0; i < 100; i++ ) {
  a[i] = 1;
  b[i] = 2;
}
```

Codice originale

```
for( i = 0; i < 100; i++ )
  a[i] = 1;

for( i = 99; i >= 0; i-- )
  b[i] = b[i-1];
```

Codice trasformato

```
for( i = 99; i >= 0; i++ ) {
  a[i] = 1;
  b[i] = b[i-1];
}
```

Codice originale

```
for( i = 0; i < 100; i++ ) {
  a[i] = a[i] * b[i];

for( i = 0; i < 100; i++ )
  c[i] = a[i] + d[i];
```

Codice trasformato

```
for( i = 0; i < 100; i++ ) {
  a[i] = a[i] * b[i];
  c[i] = a[i] + d[i];
}
```

Loop interchange

- Trasformazione
  - Scambia l'ordine di due (o più) cicli annidati
- Obiettivo
  - Ottimizzare gli accessi alla memoria per ridurre i cache miss
    - Modifica il pattern di accesso ai dati in modo da renderlo consistente con la loro organizzazione in memoria
- Note
  - In genere i compilatori non eseguono il loop interchange
    - Difficile da individuare
    - Se il corpo del ciclo non è più che semplice, la trasformazione potrebbe essere soggetta a molte condizioni e vincoli
  - Applicazione tipica
    - Accedere ad un array in row-major o column-major order a seconda del modello di memorizzazione adottato dal linguaggio
      - C: row-major
      - FORTRAN: colum-major

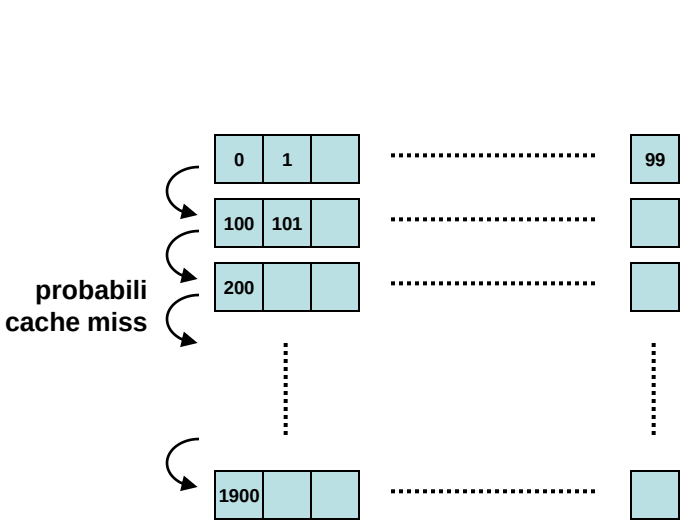
Loop interchange - Esempio

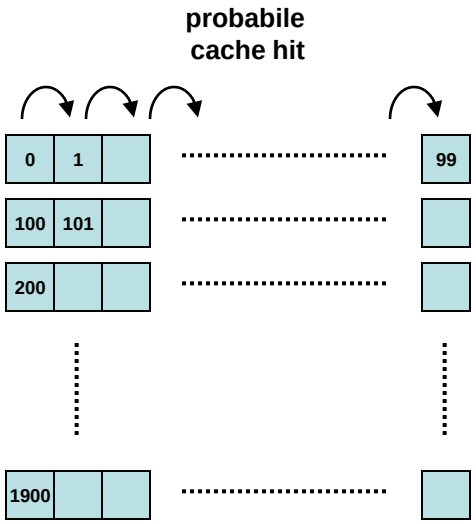
Codice originale

```
for( c = 0; c < 100; c++ )
  for( r = 0; r < 20; r++ )
    a[r][c] = r+c;
```

Codice trasformato

```
for( r = 0; r < 20; r++ )
  for( c = 0; c < 100; c++ )
    a[r][c] = r+c;
```





Loop interchange - Esempio

- In alcuni casi la trasformazione peggiora le prestazioni
  - Il pattern di accesso ad altri array riduce la località
- Nell'esempio che segue
  - Codice originale
    - L'accesso alla matrice a è in column-major order e quindi poco efficiente (nel caso pessimo si ha un miss ad ogni ciclo)
    - L'accesso ai vettori x ed y ha una forte località temporale (due load ed una store per ogni esecuzione del ciclo più esterno)
  - Codice trasformato
    - L'accesso alla matrice a è in row-major order (pochi miss)
    - L'accesso ai vettori x ed y non presenta località temporale (due load ed una store per ogni esecuzione del ciclo più interno!)

Codice originale

```
for( c = 0; c < 300; c++ )
  for( r = 0; r < 300; r++ )
    x[r] = x[r] + y[r] + a[r][c];
```

Codice trasformato

```
for( r = 0; r < 20; r++ )
  for( c = 0; c < 100; c++ )
    x[r] = x[r] + y[r] + a[r][c];
```

Loop tiling

- Trasformazione
  - Suddivide l'accesso ad una matrice in più accessi a sottoblocchi
- Obiettivo
  - Ottimizzare gli accessi alla memoria per ridurre i cache miss
    - Modifica il pattern di accesso ai dati
    - Ha lo scopo di aumentare la località spaziale dei dati
- Note
  - Alcuni compilatori eseguono casi semplificati di tiling
    - Difficle da implementare
    - Fortemente vincolato dalle dipendenze dai dati
  - È un caso particolare della più generale trasformzione che prevede una scomposizione generica dell'accesso a matrici
  - In generale il loop tiling consente ulteriori miglioramenti
    - Della località, mediante loop interchange
    - Nellàindividuaizone di espressioni costanti e variabili induttive

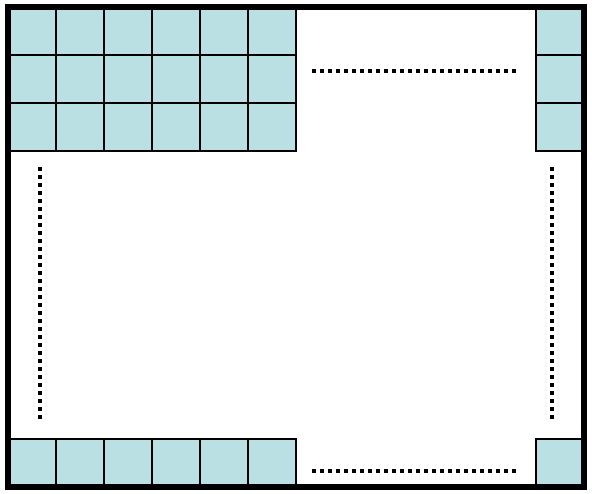
Loop tiling - Esempio

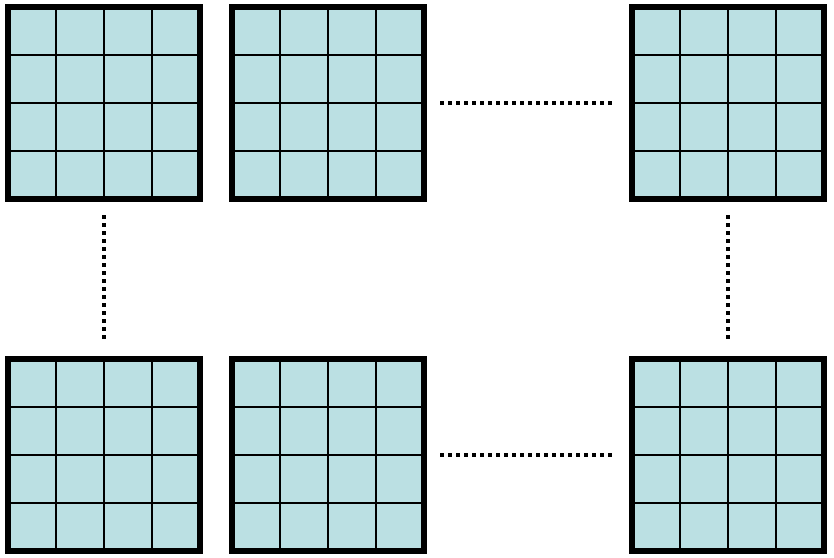
Codice originale

```
for (i = 0; i < 100; i++) {
  c[i] = 0;
  for (j = 0; j < 100; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

Codice trasformato

```
for (i = 0; i < 100; i += 4) {
  c[i] = 0;
  for (j = 0; j < 100; j += 4) {
    for (x = i; x < i + 4; x++) {
      for (y = j; y < j + 4; y++) {
        c[x] = c[x] + a[x][y] * b[y];
      }
    }
  }
}
```





Loop peeling

- Trasformazione
  - Estrae da un ciclo alcune iterazoni particolari in modo da ridurre o eliminare dipendenze dai dati e da identificare variabili induttive
    - In genere la prima e l'ultima iterazione
- Obiettivo
  - Migliorare lo sfruttamento della pipeline
    - Le dipendenze dai dati possono causare stalli
- Note
  - Alcuni compilatori eseguono casi semplificati di peeling
    - Difficle da identificare
    - Prima/ultima iterazione
  - Richiede altre trasformazioni a valle
    - Identificazione di variabili induttive
    - Propagazione delle espressioni costanti
    - Loop unswitch

Loop tiling - Esempio

Codice originale

```
k = 10
for (i = 0; i < 10; i++) {
  a[i] = b[i] + c[k];
  k = i;
}
```

Codice trasformato

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

While-do / do-while

- Trasformazione
  - Cambia un ciclo a condizione iniziale in uno a condizione finale
  - Richiede una condizione aggiuntiva iniziale
- Obiettivo
  - Minimizzare il numero di salti
  - Politiche di branch prediction e patter di traduzione possono influenzare questa trasformazione
- Note
  - In genere i compilatori eseguono questa trasformazione
    - È piuttosto semplice da realizzare

Loop tiling - Esempio

Codice originale

```
while( a > 0 ) {

  /* Loop body */
  a--;

}
```

Codice trasformato

```
if( a > 0 ) {
  do {
    /* Loop body */
    a--;
  } while( a > 0 );
}
```

- Considerando una possibile traduzione assembly
  - Nel codice originale, ad ogni iterazione si eseguono
    - Un salto condizionale (not taken)
    - Un salto incondizionato
  - Nel codice trasformato, ad ogni iterazione si esegue
    - Un salto condizionale (taken)

```
LOOP:   CMP   A, 0
        BLE   EXIT

        /* loop body translation */

        DEC   A
        BRA   LOOP
EXIT:   NOP
```

```
        CMP   A, 0
        BLE   EXIT
LOOP:   /* loop body traslation */
        DEC   A
        CMP   A, 0
        BGT   LOOP:
EXIT:   NOP
```

Zero output condition

- Trasformazione
  - Nodifica la condizione di uscita da un ciclo in modo che comporti un confronto con la costante zero
- Obiettivo
  - Sfruttare
    - Il valore lasciato nel registro di stato dalle operaizoni aritmetiche
    - Esecuzione speculativa
- Note
  - In genere i compilatori eseguono questa trasformazione
    - È piuttosto semplice da realizzare

Software pipelining

- Trasformazione
  - Replica in corpo di un ciclo più volte e riorganizza le operazioni
  - Spesso richiede un prologo ed un epilogo
- Obiettivo
  - Ridurre le dipendenze dai dati
    - Minimizzare il numero di salti
- Note
  - Alcuni compilatori eseguono questa trasformazione
    - È molto complessa da identificare e da realizzare
    - Porta notavoli benefici
  - Può essere sfruttata in modo particolarmente conveniente
    - Su architetture superscalari
    - Su architetture VLIW

Software pipelining - Esempio

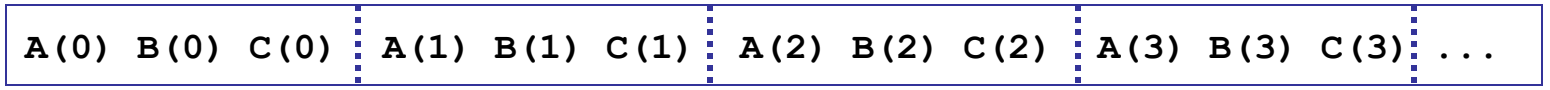
- La trasformazione è piuttosto complessa
  - Consideriamo un esempio in pseudocodice
- Si consideri il sguente codice

```
for( i=0; i < N; i++ ) {
  A(i);    // Load
  B(i);    // Compute
  C(i);    // Store
}
```

- In cui
  - A(i), B(i), C(i) sono tre operazioni sul dato i
  - Ogni operazione dipenden dalla precedente
    - B dipende da A
    - C dipende da B
  - Operazioni su dati diversi sono indipendenti
    - A(i+1) non dipende da A(i), e così per le altre operazioni

Software pipelining - Esempio

- La normale esecuzione del ciclo è la seguente



- Se ogni istruzione
  - Richiede un ciclo di clock per essere eseguita senza interlock
  - Richiede tre cicli di clock per essere eseguita con interlock
- Ogni iterazione del ciclo consta di tre istruzioni
- Ogni iteraizone richiede 1 + 3 + 3 = 7 cicli di clock
  - 1 ciclo per A
    - Non dipende dall'operaizone C al ciclo precedente)
  - 3 cicli per B e C
    - Dipendono dall'operazione precedente
- Il tempo medio per operazione è 7 / 3 = 2.33 cicli

Software pipelining - Esempio

- Riorganizzando l'esecuzione del ciclo come segue

```
A(0) A(1) A(2) B(0) B(1) B(2) C(0) C(1) C(2) | A(3) A(4) A(5) ...
```

- Ogni iterazione del ciclo costa di 9 istruzioni
  - Tre A, tre B e tre C
- Ogni iterazione richiede 1 + … + 1 = 9 cicli
  - 1 ciclo per ogni istruzione A
    - Non dipende dall'operaizone C al ciclo precedente)
  - 1 ciclo per ogni istruzione B
    - La dipendenza da A non causa stalli poiché le bolle sono sostituite dalle due operazioni A su elementi indipendenti
  - 1 ciclo per ogni istruzione C
    - Per la stessa ragione
- Il tempo medio per operazione è 9 / 9 = 1 ciclo

Software pipelining - Esempio

- Riorganizzando l'esecuzione del ciclo come mostrato
  - Si eliminano le dipendenze dai dati
  - Si riempiono le bolle della pipeline con operazioni utili

```
for( i=0; i < (N-2)/3; i=i+3 ) {
  A(i);
  A(i+1);
  A(i+2);
  B(i);
  B(i+1);
  B(i+2);
  C(i);
  C(i+1);
  C(i+2);
}
```

Trasformazioni sulle espressioni

- Migliorano l'efficienza di calcolo delle espressioni
  - Richiedono analisi di tipo data-flow
- Le principali trasformazioni sono
  - Strength reduction
  - Common subexpression elimination
  - Constant folding
  - Constant propagation
  - Copy propagation
  - Induction variables
    - Già vista nella trattazione dei cicli
  - Dead store elimination

Strength reduction

- Trasformazione
  - Sostituisce operazioni complesse con operazioni più semplici
  - Fortemente legata all'architettura target
- Obiettivo
  - Ridurre il tempo di esecuzione complessivo
    - Non necessariamente il tempo di calcolo
- Note
  - I compilatori eseguono questa trasformazione
    - A livello di generaizone del codice target
    - Si applica pressoché unicamente ad operazioni tra interi
- Esempi
  - X * 2 diviene X+X oppure X<<1
  - X * 3 diviene X+X+X oppure X<<1 + X oppure X<<2 – X
  - ...

Common subexpression elimination

- Trasformazione
  - Individua sottoespressioni comuni a più espressioni
  - Introduce variabili temporanee
- Obiettivo
  - Minimizzare il numero di istruzioni per il caloclo di espressioni
    - Le sottoespressioni comuni vengono valutate una sola volta e usate più volte
- Note
  - I compilatori eseguono questa trasformazione
    - Richiede tecniche di analisi data-flow standard

Common subexpression elimination - Esempio

- È necessario garantire che i valori delle variabili coinvolte nella sottoespressione non cambino tra i suoi diversi usi
- Si considerino i seguenti esempi
  - Nell'esempio a sinistra d ed e rimangono costanti fra i due usi
  - Nell'esempio a destra e cambia: il riuso dell'espressione costituisce un errore semantico.

```
X = b + c + 3 * ( d + e );
b++;
Y = f + b * ( d + e );
```

```
t = d + e
X = b + c + 3 * t;
b++;
Y = f + b * t;
```

```
X = b + c + 3 * ( d + e );
e++;
Y = f + b * ( d + e );
```

```
t = d + e
X = b + c + 3 * t;
e++;
Y = f + b * t; // ERROR!!
```

Constant folding

- Trasformazione
  - Calcola espressioni costanti a compile time
- Obiettivo
  - Minimizzare il numero di istruzioni a run-time per il calcolo di espressioni che coinvolgono porzioni costanti
- Note
  - I compilatori eseguono questa trasformazione
    - Molto semplice da implementare
    - Viene eseguita a valle del preprocessore
  - Cosa è considerato costante?
    - Valori numerici letterali
    - Puntatori a stringhe globali inizializzate
    - Variabili dichiarate const
    - Valori enumerati
    - Funzioni costanti (solo C++0x)

Constant propagation

- Trasformazione
  - Sostituisce i valori costanti nelle espressioni
- Obiettivo
  - Minimizzare il numero di istruzioni per il calcolo di espressioni
    - Le costanti complesse vengono calcolate una sola volta a compile time
- Note
  - I compilatori eseguono questa trasformazione

Constant propagation

- Trasformazione
  - Sostituisce le occorrenze di variabili oggetto di assegnamento diretto con il riferimento all'oggetto asseganto
- Obiettivo
  - Minimizzare il numero di operaizoni copia/trasferiento
- Note
  - I compilatori eseguono questa trasformazione

- Esempio

Codice originale

```
x = y;
z = a[i] + y;
W = b[j] * x;
```

Codice trasformato

```
z = a[i] + y;
W = b[j] * y;
```

Dead store elimination

- Trasformazione
  - Elimina un asseganmento ad una variabile non usata in seguito
- Obiettivo
  - Evitare inutili accessi in memoria
- Note
  - I compilatori eseguono questa trasformazione
  - Si tratta di un caso particolare di dead-code elimination

Trasformazioni varie

- Alcune importanti trasformazioni sono
  - Inlining
  - Tail recursion elimination
  - Conditional expression reordering
  - Function special cases

Inlining

- Trasformazione
  - Replica il codice di una funzione in ogni punto in cui è usata
- Obiettivo
  - Evitare l'overhead di chiamata
    - Costruzione e distruzione dello stack
    - Salti
- Note
  - I compilatori eseguono questa trasformazione
    - Automaticamente o sotto il controllo dell'utente
  - È indispensabile valutare il compromesso tra
    - Efficienza del codice
    - Dimensione del codice
  - Migliori candidati sono le funzioni
    - Di piccole dimensioni
    - Eseguite molto spesso
    - Chaimate da pochi punti del codice

Tail recursion elimination

- Trasformazione
  - Elimina la ricorsione in coda e la trasforma in un ciclo
- Obiettivo
  - Evitare l'overhead di chiamata
    - Costruzione e distruzione dello stack
    - Salti
  - Ridurre l'uso dello stack
- Note
  - Non viene mai eseguita dai compilatori
  - In genere
    - Comporta la ridefinizione dell'algoritmo
    - Può essere eseguita in modo standard

Tail recursion elimination - Esempio

- Una funzione è ricorsiva in coda se la chiamata a sé stessa è l'ultima operazione eseguita
- Esempio: Fattoriale non ricorsivo in coda

```
int fact( int n ) {
  if( n == 0 )
    return 1;
  return n * fact(n - 1);
}
```

- Esempio: Fattoriale ricorsivo in coda

```
int fact_tr( int n, int a ) {
  if( n == 0 )
    return a;
  return fact_tr(n - 1, n * a);
}
```

```
int fact( int n ) {
  fact_tr( n, 1 );
}
```

Tail recursion elimination - Esempio

- Ora che si ha ricorsione in coda si può procedere
  - Si eseguono le operazioni sugli argomenti della chiamata ricorsiva secondo quanto previsto dalla chiamata originale

```
int fact_tr( int n, int a ) {
  if( n == 0 )
    return a;
  a = a * n;
  n = n - 1;
  return fact_tr(n, a);
}
```

  - Si sostituisce la chiamata con un salto all'inizio della funzione

```
int fact_tr( int n, int a ) {
beginning:
  if( n == 0 )
    return a;
  a = a * n;
  n = n - 1;
  goto beginning;
}
```

Conditional expression reordering

- Trasformazione
  - Modifica l'ordine di valutaizone di una condizione logica composta
- Obiettivo
  - Sfruttare le shortcut evaluations del compilatore
- Note
  - Sii basa sulla conoscenza delle probabilità che una condizione ha di essere vera o falsa
- Esempio
  - Se C1 è vera 99 volte su 100, mentre C2 solo 50 volte su 100
    - (C1 || C2) è più efficiente di (C2 || C1)
    - (C2 && C1) è più efficiente di (C1 && C2)
  - In generale si pone più a sinistra la condizione
    - Più probabilmente vera per espressioni in OR (1 è il valore forzante dell'OR)
    - Più probabilmente false per espressioni in AND (0 è il valore forzante dell'AND)

Function special cases

- Trasformazione
  - Elimina la chaiamta ad una funzione in casi speciali
- Obiettivo
  - Evitare l'overhead di chiamata quando il risultato di una funzione è semplice e noto a priori
- Note
  - Si applica in generale a funzioni matematiche
  - Non è mai applicata dai compilatori poiché richiede la comprensione della semantica della funzione chiamata
- Esempio

Codice originale

```
#include <math.h>

x = sin( alpha );
```

Codice trasformato

```
#include <math.h>

x = (alpha == 0.0) ? 0.0 : sin( alpha );
```