15. Ottimizzazione globale

Indice

Calcolo delle Informazioni di Vivacità (Live Information)

Un valore v è considerato *vivo* in un punto p se esiste un cammino da p a un'istruzione che utilizza v lungo il quale v non viene ridefinito. I problemi relativi al flusso di dati vengono espressi come un sistema di equazioni simultanee.

Annotazioni sui Blocchi

Ogni blocco viene annotato con due insiemi, $LIVEOUT$ e $LIVEIN$, definiti come segue:

$LIVEOUT(b) = \bigcup_{s \in succ(b)} LIVEIN(s)$
$LIVEIN(b) = UEVAR(b) \cup (LIVEOUT(b) \cap \neg VARKILL(b))$
$LIVEOUT(n_f) = \emptyset$

Dove:

- $UEVAR(b)$ è l'insieme dei nomi utilizzati nel blocco b prima di essere definiti in b.
- $VARKILL(b)$ è l'insieme dei nomi definiti nel blocco b.
- $succ(b)$ rappresenta i successori del blocco b.

L'insieme $LIVEOUT$ opera sul dominio delle variabili.

Caratteristiche del Problema

L'analisi della vivacità è un problema di flusso di dati **all'indietro**. Questo significa che l'informazione fluisce dai successori ai predecessori, contrariamente al calcolo dei dominatori che si muove in avanti. Il compilatore può risolvere queste equazioni utilizzando un semplice algoritmo iterativo.

Algoritmo Iterativo con Lista di Lavoro (Worklist)

Procedura

L'algoritmo utilizza una lista di lavoro ($WorkList$), inizializzata con tutti i blocchi del grafo. Procede iterativamente come segue:

1. Rimuovere un blocco b dalla $WorkList$.
2. Calcolare $LIVEOUT(b)$ utilizzando i successori di b.
3. Calcolare $LIVEIN(b)$ basandosi su $UEVAR(b)$ e $LIVEOUT(b)$.
4. Se $LIVEIN(b)$ cambia rispetto alla precedente iterazione, aggiungere i predecessori di b alla $WorkList$.

L'algoritmo continua fino a quando la $WorkList$ non diventa vuota.

Perché Funziona?

Gli insiemi $LIVEOUT$ e $LIVEIN$ sono sottoinsiemi di 2^{Names}, l'insieme delle parti delle variabili. Gli insiemi $UEVAR$ e $VARKILL$ sono costanti per ciascun blocco b, mentre le equazioni sono monotone: ogni modifica ai set porta a un'ulteriore riduzione o espansione controllata degli insiemi.

Poiché il numero di aggiunte agli insiemi è finito, l'algoritmo raggiunge sempre un punto fisso. La velocità di convergenza dipende dall'ordine in cui i blocchi vengono rimossi dalla lista di lavoro e dai relativi insiemi ricalcolati.

Implementazione della Worklist

La $WorkList$ dovrebbe essere implementata come un insieme, in modo da evitare duplicati. Questo garantisce efficienza e previene ricalcoli inutili per blocchi già processati.

Uso delle Informazioni di Vivacità: Eliminazione degli Store Inutili

Trasformazione: Eliminazione degli Store Inutili

Un'operazione di *store* può essere eliminata quando il valore in un registro:

1. Ha già avuto la sua ultima definizione.
2. Non verrà mai più utilizzato.

In tal caso, lo *store* è considerato "morto" (ad eccezione degli utilizzi per il debugging). Il compilatore può eliminare tali operazioni, ottimizzando il codice.

Piano

Per eliminare gli *store* inutili, si procede come segue:

1. Calcolare gli insiemi $LIVEIN$ e $LIVEOUT$ per ciascun blocco del grafo.
2. Attraversare ogni blocco del programma partendo dal basso verso l'alto, calcolando l'informazione $LIVE$ localmente in modo incrementale.
3. Per ogni operazione di *store* incontrata, verificare se il target dello *store* è presente in $LIVE$.
    - Se il target non è in $LIVE$, eliminare l'operazione di *store*.
4. Se tutte le operazioni di *store* relative a una variabile locale vengono eliminate, lo spazio per la variabile può essere rimosso dal record di attivazione.

La dimensione di $LIVEOUT$ dipende dal numero di variabili, $|variables|$.

Sicurezza

Se una variabile $x \notin LIVE(s)$ in corrispondenza di uno *store* s, il valore di x non viene utilizzato lungo alcun cammino che parte da s e arriva al nodo di uscita del grafo di flusso di controllo (CFG). Questo implica che il valore di x non viene letto ed è, quindi, "morto". L'affidabilità di questa analisi dipende dalla correttezza del calcolo degli insiemi $LIVE$.

Profitto

L'eliminazione di uno *store* è vantaggiosa se il costo di non eseguire l'operazione è inferiore al costo della sua esecuzione. L'assunto è che gli *store* rappresentino un costo computazionale evitabile.

Opportunità

Un approccio lineare può essere utilizzato per cercare le operazioni di *store* blocco per blocco. Tuttavia, per migliorare l'efficienza, è possibile costruire una lista di operazioni di *store* durante il calcolo iniziale dell'insieme $UEVAR$. Questo permette di ridurre il costo computazionale del successivo passaggio di eliminazione.

Riposizionamento dei Blocchi di Codice

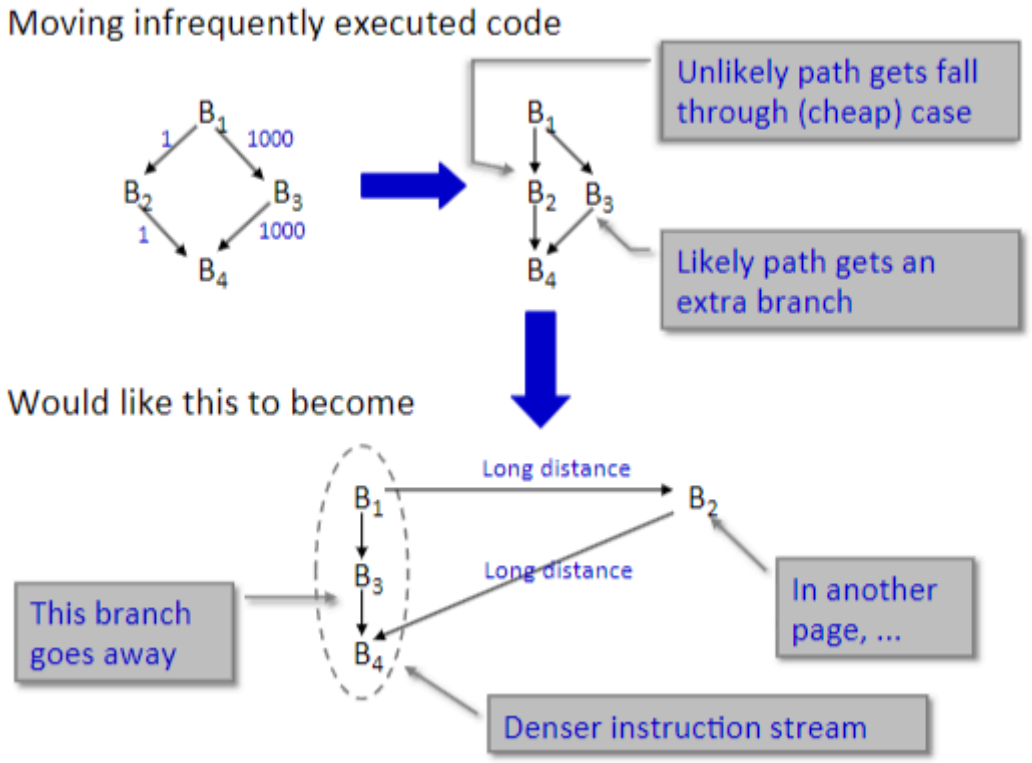
Il posizionamento dei blocchi di codice in memoria è fondamentale per l'efficienza dell'esecuzione, poiché un ordine errato può aumentare la dimensione del working set e causare mancati accessi alla TLB (Translation Lookaside Buffer) o page misses. I percorsi di tipo fall-through, che non richiedono branch, sono più efficienti rispetto ai percorsi con branch, sia in termini di costo che di località.

Per ottimizzare il posizionamento, è necessario identificare i percorsi eseguiti più frequentemente, detti hot paths, utilizzando informazioni sul profilo di esecuzione. Successivamente, i blocchi vengono riorganizzati in modo da mantenere i percorsi frequenti in memoria contigua, riducendo i branch e migliorando la località dei dati. Questa strategia rende il caso più probabile il percorso fall-through, spostando i casi meno probabili lontano dalla linea principale di esecuzione.

L'identificazione dei percorsi frequenti si basa sull'osservazione delle differenze nelle frequenze di esecuzione dei branch. Rendendo il caso più probabile il fall-through e spostando i casi improbabili fuori linea, è possibile ottimizzare l'efficienza del codice.

Il riposizionamento dei blocchi offre numerosi vantaggi. Permette di creare sequenze di codice più lunghe senza branch, aumentando il numero di operazioni eseguite per linea di cache e riducendo i cache misses. Il flusso di istruzioni diventa più denso e il codice raro, spostato lontano dalla linea principale, migliora l'uso delle pagine riducendo i page faults.

Nell'esempio illustrato nell'immagine, il codice eseguito raramente è stato inizialmente posizionato vicino ai blocchi principali, causando branch inutili e una dispersione delle istruzioni. La trasformazione sposta il codice raro lontano dalla linea principale, riducendo il numero di branch, migliorando la densità delle istruzioni e ottimizzando l'accesso alla cache e alla memoria. Questo processo rende il flusso di esecuzione più fluido e veloce, massimizzando la località e minimizzando i costi di accesso alla memoria.



Panoramica sull'Ottimizzazione del Posizionamento dei Blocchi

L'obiettivo del processo è costruire catene di percorsi frequentemente eseguiti, basandosi sui dati di profilo e organizzando il codice in modo che i percorsi "caldi" seguano ramificazioni brevi e in avanti.

Fase 1: Costruzione delle Catene di Percorsi Frequenti

Per identificare i percorsi caldi, si utilizzano i dati di profilo. Le informazioni sugli archi (edge profiles) sono preferibili a quelle sui nodi (node profiles), poiché offrono maggiore precisione. I blocchi di codice vengono combinati in catene utilizzando un semplice algoritmo greedy.

La raccolta dei dati di profilo può essere effettuata strumentando l'eseguibile, utilizzando campionamenti statistici o inferendo i conteggi degli archi dai dati di performance. Sebbene la precisione sia desiderabile, una buona approssimazione è spesso sufficiente.

L'idea è formare catene che possano essere posizionate per creare codice lineare. Si parte dalla costruzione dei percorsi caldi, considerando che architetture come PA-RISC

prevedono la maggior parte dei branch in avanti come presi, mentre quelli all'indietro come non presi. L'obiettivo è posizionare i target dei branch vicino alle loro sorgenti per facilitare i branch in avanti.

L'algoritmo di costruzione delle catene procede come segue:

1. Per ogni blocco b del grafo di controllo, creare una catena degenerata d contenente solo b e assegnare a d una priorità iniziale.
2. Ordinare gli archi del grafo di controllo (CFG) in base alla frequenza decrescente.
3. Per ogni arco $< x, y >$ (dove $x \neq y$):
    - Se x è la coda di una catena a e y è la testa di una catena b, unire b ad a.
    - Aggiornare la priorità della catena risultante in base alla frequenza e all'ordine di elaborazione.

Fase 2: Posizionamento del Codice

Una volta costruite le catene, il codice viene disposto per minimizzare i branch inutili e garantire che i branch tra catene siano preferibilmente in avanti. Il processo segue un'idea intuitiva, partendo dal nodo di ingresso (n_0) e tentando di posizionare i target dei branch più vicini alle sorgenti.

1. La catena che inizia con il nodo di ingresso del CFG è selezionata per prima. Questa catena viene aggiunta a una lista di lavoro ($WorkList$) con la sua priorità.
2. Finché la $WorkList$ non è vuota:
    - Rimuovere la catena con priorità più bassa.
    - Posizionare i blocchi della catena nell'ordine della catena stessa, aggiungendoli alla fine del codice eseguibile.
    - Per ogni blocco nella catena, considerare i suoi archi verso i blocchi non ancora posizionati. Se il target di un arco appartiene a una catena non ancora elaborata, aggiungere la catena corrispondente alla $WorkList$.

Questa procedura garantisce che gli archi più probabilmente presi rimangano come branch in avanti, mentre gli archi di probabilità inferiore vengono lasciati solo se inevitabili.

Benefici Previsti

Questo approccio consente di massimizzare le prestazioni riducendo il costo dei branch, migliorando la località dei dati e aumentando la densità del flusso di istruzioni. Inoltre, si ottimizza l'accesso alla memoria, riducendo le penalità derivanti da branch imprevisti e cache misses.

Procedure Splitting

Il *procedure splitting* consiste nello spostare il codice che ha un conteggio di esecuzione pari a zero, o molto basso, in una posizione lontana nella memoria. Questo codice, definito "fluff", raramente viene eseguito. Spostandolo, è possibile ottenere diversi vantaggi, tra cui una maggiore densità delle istruzioni nella cache delle istruzioni (I-cache) e un utilizzo più efficace di quest'ultima. Tuttavia, ciò comporta una esecuzione più lenta del codice raramente utilizzato, ma questo compromesso è accettabile data la sua bassa frequenza di esecuzione.

L'implementazione di questa trasformazione prevede la creazione di una procedura separata, senza collegamenti diretti, e con un nome inventato. A questa nuova procedura viene assegnata una priorità che consente al linker di posizionarla alla fine del codice. Il branch originale verso il codice spostato viene sostituito con un branch a profilo zero, che punta a una chiamata a profilo zero. In questo modo, il codice di collegamento si sposta anch'esso alla fine della procedura per mantenere la densità delle istruzioni. Il branch verso il codice fluff diventa un breve branch verso un lungo branch, e il blocco contenente il lungo branch viene ordinato alla fine della procedura corrente.

Questa trasformazione è sicura, poiché cambia esclusivamente la posizione del codice e non i valori che esso calcola. In assenza di bug nell'implementazione, il processo non altera il comportamento del programma.

In termini di profitto, questa tecnica aumenta il numero di branch di tipo *fall-through* e, ove possibile, incrementa il numero di branch predetti correttamente dal compilatore. Migliora anche la località del codice, rendendo il programma più efficiente.

L'opportunità di applicare questa trasformazione deriva dai dati di profilo, che mostrano gli archi ad alta frequenza nel grafo di controllo. Durante la trasformazione, vengono analizzati tutti i blocchi e gli archi del programma, portando a una complessità di $O(N + E)$, dove N è il numero di blocchi e E è il numero di archi. Molte trasformazioni nell'ottimizzazione del codice presentano una complessità simile.