IL COMPILATORE LABVIEW: CHE COSA C'È SOTTO IL COFANO

LabVIEW è un linguaggio compilato. Ciò significa che il codice G che scrivete viene tradotto in codice macchina nativo ed eseguito direttamente dal computer host

Jeffrey Phillips

La progettazione di un compilatore anche per un linguaggio di programmazione banale può diventare facilmente un problema complesso. La teoria dei compilatori è considerata una conoscenza ad elevata specializzazione anche fra gli ingegneri software professionisti. Il moderno LabVIEW è un linguaggio multiparadigmatico, che abbraccia un'ampia varietà di concetti inclusi il flusso dei dati, l'orientamento agli oggetti e la programmazione guidata da eventi. LabVIEW raggiunge inoltre piattaforme diverse, eseguendo o su più SO (Windows, Linux, Mac), più chipset (PowerPC, Intel) e anche target come dispositivi embedded e chip FPGA (Field-programmable gate array), che si differenziano dalle tradizionali architetture PC. Come potete dedurre, il compilatore di LabVIEW è un sistema sofisticato e per spiegarlo bisognerebbe andare molto al di là di un singolo articolo.

Questo particolare articolo introduce il compilatore di LabVIEW, spiega brevemente la sua evoluzione a partire dal 1986 con LabVIEW 1.0, e descrive la sua forma attuale. Inoltre, esplora le recenti innovazioni dei compilatori e sottolinea i vantaggi di queste nuove caratteristiche per l'architettura LabVIEW e per voi.

COMPILAZIONE E INTERPRETAZIONE

LabVIEW è un linguaggio compilato, cosa che può sorprendere perché, durante il tipico sviluppo G, non vi è una fase esplicita di compilazione. Al contrario, potete cambiare il vostro VI e premere semplicemente il pulsante Run per eseguirlo. Compilazione significa che il codice G che scrivete viene tradotto in codice macchina nativo ed è quindi eseguito direttamente dal computer host. Un'alternativa a questo approccio è l'interpretazione, dove i programmi sono eseguiti indirettamente da un altro programma software (chiamato interprete) anziché direttamente dal computer. Nulla nel linguaggio LabVIEW richiede che sia compilato o interpretato; infatti, la prima versione di LabVIEW usava un interprete. Nelle versioni successive, il compilatore ha sostituito l'interprete per spingere le prestazioni di esecuzione dei VI, cosa che è un comune differenziatore dei compilatori rispetto agli interpreti. Gli interpreti tendono a essere più facili da scrivere e mantenere a scapito di prestazioni di esecuzione più lente, mentre i compilatori tendono a essere più complessi da implementare ma offrono tempi di esecuzione più veloci. Uno dei benefici primari del compilatore di LabVIEW è che i miglioramenti apportati al compilatore sono visti da tutti i VI senza la necessità di cambiamenti. Infatti, uno dei focus primari della versione 2010 di LabVIEW sono state le ottimizzazioni all'interno del compilatore per velocizzare il tempo di esecuzione dei VI.

UNA PROSPETTIVA STORICA DEL COMPILATORE DI LABVIEW

Prima di saltare in una discussione approfondita degli attuali elementi interni del compilatore, conviene riassumere lo sviluppo del compilatore dalle sue prime forme, oltre 20 anni fa. Alcuni degli algoritmi che introdurremo, come propagazione dei tipi, clumping e inplaceness, sono descritti in maggiore dettaglio nella discussione sul moderno compilatore di LabVIEW. LabVIEW 1.0 è stato introdotto nel 1986. Come già detto, nella sua prima versione LabVIEW usava un interprete ed era eseguibile solo sul Motorola 68000. A quel tempo, il nuovo linguaggio LabVIEW era molto più semplice, cosa che alleviava anche i requisiti del compilatore (allora un interprete). Per esempio, non c'era polimorfismo e l'unico tipo numerico era quello in virgola mobile a precisione estesa. LabVIEW 1.1 vide l'introduzione dell'algoritmo inplaceness, o dell'"inplacer". Tale algoritmo identifica le allocazioni dei dati che potete riutilizzare durante l'esecuzione, evitando inutili copie dei dati e, di conseguenza, velocizzando spesso nettamente le prestazioni di esecuzione.

In LabVIEW 2.0, l'interprete è stato sostituito con un effettivo compilatore. Pur essendo ancora targetizzatorimanendo eseguibile ancora esclusivamente sui Motorola 68000 Motorola, LabVIEW poteva generare codice macchina nativo. Nella Version 2.0 è stato aggiunto anche l'algoritmo di propagazione dei tipi di dato che, fra le altre cose, gestisce il controllo della sintassi e la risoluzione dei tipi sul linguaggio LabVIEW in crescita. Un'altra grossa innovazione in LabVIEW 2.0 è stata

l'introduzione del clumper. L'algoritmo di clumping identifica il parallelismo nel diagramma LabVIEW e raggruppa i nodi in "clump", che possono essere eseguiti in parallelo. Gli algoritmi di propagazione dei tipi, inplaceness e clumping continuano a essere componenti importanti del moderno compilatore LabVIEW e hanno visto numerosi miglioramenti incrementali nel tempo. L'infrastruttura del nuovo compilatore in LabVIEW 2.5 ha aggiunto supporto per back end multipli, specificamente Intel x86 and Sparc. LabVIEW 2.5 ha inoltre introdotto il linker, che gestisce le dipendenze fra VI per tracciarli quando devono essere ricompilati.

Due nuovi back end, PowerPC e HP PA-RISC, sono stati aggiunti insieme al folding delle costanti in LabVIEW 3.1. LabVIEW 5.0 e 6.0 hanno ridato slancio al generatore di codice e hanno aggiunto la GenAPI, un'interfaccia comune ai molteplici back end. La GenAPI cross-compila, cosa importante per lo sviluppo real-time.

Gli sviluppatori real-time tipicamente scrivono i VI su un PC host ma li installano (e li compilano per) un target real-time. Inoltre, è stata inclusa una forma limitata di spostamento del codice loop-invariante. Infine, il sistema di esecuzione multitasking di LabVIEW è stato esteso per supportare thread multipli.

LabVIEW 8.0 si è basato sull'infrastruttura GenAPI introdotta nella Version 5.0 per aggiungere un algoritmo di allocazione dei registri. Prima dell'introduzione della GenAPI, i registri erano hardcoded nel codice generato per ogni nodo. Sono state introdotte anche forme limitate di eliminazione del codice irraggiungibile e del codice morto. LabVIEW 2009 ha apportato come novità LabVIEW a 64 bit e la Dataflow epresentation (DFIR). La DFIR è stata immediatamente usata per costruire forme più avanzate di spostamento del codice loop-invariante, folding delle costanti, eliminazione del codice morto ed eliminazione del codice irraggiungibile. Nuove caratteristiche del linguaggio introdotte in 2009, come il For Loop parallelo, sono state basate sulla DFIR.

Finalmente, in LabVIEW 2010, la DFIR offre nuove ottimizzazioni del compilatore, come la riassociazione algebrica, l'eliminazione delle sottoespressioni comuni, il loop unrolling e l'inlining dei subVI.

Questa versione include anche l'adozione di una macchina virtuale a basso livello (LLVM) nella catena del compilatore LabVIEW. La LLVM è un'infrastruttura compilatore open-source molto usata nell'industria. Con la LLVM, sono state aggiunte nuove ottimizzazioni come la pianificazione delle istruzioni, il loop unswitching, la combinazione delle istruzioni, la propagazione condizionata e un allocatore di registri più sofisticato.

L'ATTUALE PROCESSO DI COMPILAZIONE

Con una comprensione di base della storia del compilatore di LabVIEW, ora potete esplorare il processo di compilazione nel moderno LabVIEW. In primo luogo, esaminiamo una visione d'insieme ad alto livello dei vari passi di compilazione e rivisitiamo quindi ogni parte in maggiore dettaglio.

Il primo passo nella compilazione di un VI è l'algoritmo di propagazione dei tipi di dato. Questo passo complesso è responsabile del riconoscimento dei tipi implicati per i terminali che possono adattarsi al tipo di dato, nonché del rilevamento degli errori di sintassi. Tutti i possibili errori di sintassi nel linguaggio di programmazione G sono rilevati durante l'algoritmo di propagazione dei tipi. Se l'algoritmo determina che il VI è valido, la compilazione continua.

Dopo la propagazione dei tipi di dato, il VI viene convertito dal modello usato dall'editor del diagramma a blocchi nella DFIR usata dal compilatore. Dopo la conversione nella DFIR, il compilatore esegue diverse trasformazioni sul grafico DFIR per decomporlo, ottimizzarlo e prepararlo per la generazione del codice. Molte delle ottimizzazioni del compilatore – ad esempio , l'inplacer e il clumper – sono implementate come trasformazioni e vengono eseguite in questo passo.

Dopo l'ottimizzazione e la semplificazione del grafico DFIR, viene tradotto nella rappresentazione intermedia LLVM. Vengono eseguiti uUna serie di passaggi LLVM sulla rappresentazione intermedia per ottimizzarla e abbassarla ulteriormente, fino al codice macchina.

PROPAGAZIONE DEI TIPI DI DATO

Come detto in precedenza, l'algoritmo di propagazione dei tipi identifica i tipi di dato e rivela gli errori di programmazione. Questo algoritmo ha diverse responsabilità, incluse le seguenti:

Identificare i tipi di dato implicati per i terminali che possono adattarsi al tipo

Riconoscere le chiamate ai subVI e determinare la loro validità

Calcolare la direzione dei fili

Verificare i cicli nel VI

Rivelare e riportare gli errori di sintassi

Questo algoritmo viene eseguito dopo ogni cambiamento che apportate a un VI per determinare se il VI è ancora buono, quindi è un po' discutibile il fatto che questo passo faccia realmente parte della "compilazione". Tuttavia, è il passo nella catena di compilazione di LabVIEW che corrisponde più chiaramente ai passi di analisi lessicale, parsing o analisi semantica in un compilatore tradizionale.

Un semplice esempio di un terminale che si adatta al tipo di dato è la primitiva di somma in LabVIEW.

Sommando due interi, il risultato è un intero, ma se sommate due numeri a virgola mobile, il risultato è un numero a virgola mobile. Modelli simili esistono per tipi composti come array e cluster.

Vi sono altri costrutti del linguaggio come gli shift register che hanno regole di tipizzazione più complesse. Nel caso della primitiva di somma, il tipo di dato in uscita è determinato dai tipi in ingresso e si dice che il tipo si "propaga" attraverso il diagramma, da cui il nome dell'algoritmo.

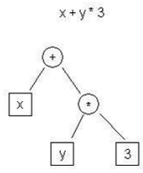
Questo esempio illustra anche la responsabilità nel controllo della sintassi dell'algoritmo di propagazione dei tipi. Supponete di cablare un intero e una stringa alla primitiva di addizione – che cosa accadrebbe? In questo caso, sommare questi due valori non ha senso, quindi l'algoritmo di propagazione dei tipi lo riporta come errore e marca il VI come "cattivo", causando la rottura della freccia di esecuzione.

RAPPRESENTAZIONI INTERMEDIE – CHE COSA E PERCHÉ

Dopo che la propagazione dei tipi decide che un VI è valido, la compilazione continua e il VI è tradotto nella DFIR. Prendiamo in considerazione le rappresentazioni intermedie (IR) in generale prima di dettagliare la DFIR.

Una IR è una rappresentazione del programma utente che è manipolata mano a mano che la compilazione progredisce attraverso le varie fasi. La nozione di IR è comune nella moderna letteratura sui compilatori e può essere applicata a qualsiasi linguaggio di programmazione.

Consideriamo qualche esempio. Oggi esiste una varietà di IR ben note. Due esempi comuni sono gli alberi di sintassi astratti (AST) e il codice a tre indirizzi.




```
t0 <- y
t1 <- 3
t2 <- t0 * t1
t3 <- x
t4 <- t3 + t2
```

La figura 1 mostra una rappresentazione AST dell'espressione

Figura 1. Esempio di IR AST

"x + y * 3", mentre la tabella 1 nella parte destra, mostra la rappresentazione basata su codice a tre indirizzi. Un'ovvia differenza fra queste due rappresentazioni è che l'AST è molto più ad alto livello. Essa si avvicina più alla rappresentazione sorgente del programma (C) che alla rappresentazione target (codice macchina). Il codice a tre indirizzi, invece, è a basso livello e assomiglia più all'assembly.

Le rappresentazioni ad alto e a basso livello hanno entrambe i loro vantaggi.

Per esempio, le analisi come l'analisi della dipendenza possono essere più facili da eseguire su una rappresentazione ad alto livello come l'AST che su una a basso livello come il codice a tre indirizzi. Altre ottimizzazioni, come l'allocazione dei registri o la schedulazione delle istruzioni, sono tipicamente eseguite su una rappresentazione a basso livello come il codice a tre indirizzi.

Poiché IR differenti hanno punti di forza e debolezza differenti, molti compilatori (incluso quello di LabVIEW) usano più IR. Nel caso di LabVIEW, la DFIR è usata come IR ad alto livello, mentre la IR LLVM è usata come IR a basso livello.

DFIR

In LabVIEW, la rappresentazione ad alto livello è la **DFIR**, che è gerarchica e basata su grafici e assomiglia allo stesso codice G. Come il G, la DFIR è composta da vari nodi, ciascuno dei quali contiene terminali. I terminali possono essere connessi ad altri terminali. Alcuni nodi, come i loop, contengono diagrammi, che possono a loro volta contenere altri nodi.

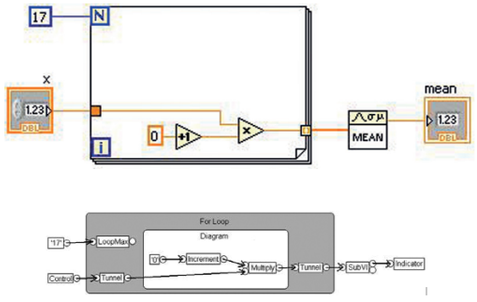


Figura 2. Codice G di LabVIEW e grafico DFIR equivalente

La figura 2 mostra un semplice VI insieme alla sua rappresentazione DFIR iniziale. Quando il grafico DFIR di un VI viene creato per la prima volta, è una traduzione diretta del codice G e i nodi nel grafico DFIR generalmente hanno una corrispondenza uno a uno con i nodi nel codice G. Mano a mano che la compilazione progredisce, i nodi DFIR possono essere spostati o divisi, oppure possono essere inseriti nuovi nodi DFIR. Uno dei vantaggi chiave della DFIR è che preserva caratteristiche come il parallelismo implicito nel vostro codice G. Il parallelismo rappresentato nel codice a tre indirizzi, al contrario, è molto più difficile da discernere.

La DFIR offre due significativi vantaggi al compilatore LabVIEW. Innanzitutto, la DFIR disaccoppia la rappresentazione del VI dell'editor da quella del compilatore. Inoltre, la DFIR agisce da hub comune per il compilatore, che ha numerosi front e back end. Consideriamo ciascuno di questi vantaggi in maggiore dettaglio.

IL GRAFICO DFIR DISACCOPPIA LA RAPPRESENTAZIONE DELL'EDITOR DA QUELLA DEL COMPILATORE

Prima dell'avvento della DFIR, LabVIEW aveva una singola rappresentazione del VI condivisa dell'editor e dal compilatore. Questo proibiva al compilatore di modificare la rappresentazione durante il processo di compilazione, cosa che, a sua volta, rendeva difficile introdurre ottimizzazioni del compilatore.

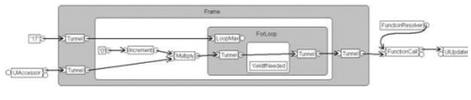


Figura 3. La DFIR fornisce un framework che permette al compilatore di ottimizzare il vostro codice

La figura 3 mostra un grafico DFIR per il VI presentato prima. Il grafico rappresenta un tempo molto successivo nel processo del compilatore, dopo che diverse trasformazioni lo hanno scomposto e ottimizzato. Come potete vedere, il grafico è parecchio diverso dal grafico precedente. Per esempio:
Le trasformazioni di scomposizione hanno rimosso i nodi controllo indicatore e SubVI e li hanno sostituiti con nuovi nodi – UIAccessor, UIUpdater, FunctionResolvere FunctionCall
Lo spostamento delle parti di codice loop-invarianti ha portato i nodi di incremento e moltiplicazione all'esterno del corpo del loop
Il clumper ha introdotto un nodo YieldIfNeeded all'interno del For Loop, provocando la condivisione dell'esecuzione del thread in esecuzione con altri elementi di lavoro in competizione
La trasformazioni sono descritte in maggiore dettaglio in un paragrafo successivo.

L'IR della DFIR serve come hub comune per i front e back end multipli del compilatore

LabVIEW funziona su numerosi target differenti, alcuni dei quali sono nettamente diversi fra loro, per esempio un PC desktop x86 e un FPGA Xilinx. Analogamente, LabVIEW presenta più modelli di calcolo all'utente. Oltre alla programmazione grafica in G, LabVIEW offre matematica testuale in MathScript, per fare un esempio. Ciò si traduce in una raccolta di front e back end, che devono tutti funzionare con il compilatore di LabVIEW. Usando la DFIR come IR comune che tutti i front end producono e tutti i back end consumano facilita il riutilizzo fra le varie combinazioni. Per esempio, un'implementazione di un'ottimizzazione a folding di costanti seguita su un grafico DFIR può essere scritta una sola volta ed applicata a target desktop, real-time, FPGA ed embedded.

SCOMPOSIZIONI DELLA DFIR

Una volta nella DFIR, il VI è sottoposto in primo luogo a una serie di trasformazioni di scomposizione. Le trasformazioni di scomposizione puntano a ridurre o normalizzare il grafico DFIR. Per esempio, la scomposizione dei tunnel d'uscita non cablati trova i tunnel d'uscita su strutture case e strutture a eventi che non sono cablati e configurati come "Use Default If Unwired". Per questi terminali, la trasformazione lascia una costante con il valore di default e la cabla al terminale, rendendo quindi il comportamento "Use Default If Unwired" esplicito nel grafico DFIR. Successivi passaggi del compilatore possono quindi trattare tutti i terminali in modo identico e assumere che tutti abbiano ingressi cablati. In questo caso, la caratteristica "Use Default If Unwired" del linguaggio è stata "compilata via" riducendo la rappresentazione a una forma più fondamentale.
Questa idea può essere applicata anche a caratteristiche più complesse del linguaggio. Per esempio, una trasformazione di scomposizione è usata per ridurre il Feedback Node in shift register su un While Loop. Un'altra scomposizione implementa il For Loop parallelo in diversi For Loop sequenziali con un po' di logica addizionale per dividere gli ingressi in parti parallelizzabili per i loop sequenziali e riunire le parti fra loro successivamente.
Anche una nuova caratteristica in LabVIEW 2010, l'inlining dei subVI, è implementata come una scomposizione DFIR. Durante questa fase della compilazione, il grafico DFIR dei subVI marcati "inline" è inserito direttamente nel grafico DFIR chiamante.
Oltre a evitare l'overhead della chiamata di un subVI, l'inlining mette in luce opportunità di ottimizzazione addizionali combinando il chiamante e il chiamato in un singolo grafico DFIR. Per esempio, consideriamo questo semplice VI che chiama il TrimWhitespace.vi da vi.lib.

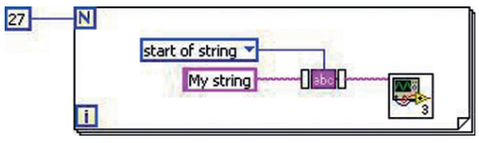


Figura 4. Esempio di un semplice VI per dimostrare le ottimizzazioni della DFIR

La TrimWhitespace.vi è definita in vi.lib nel modo seguente:

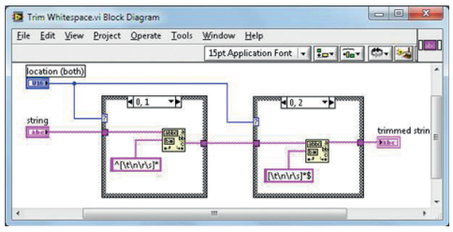


Figura 5. Diagramma a blocchi di TrimWhitespace.vi

Il subVI è inlined nel chiamante, ottenendo un grafico DFIR equivalente al seguente codice G.

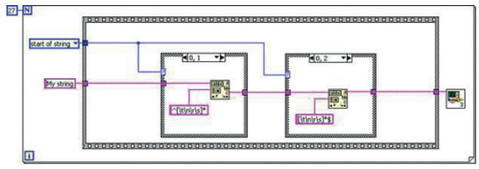


Figura 6. Codice G equivalente al grafico DFIR di TrimWhitespace.vi inlined

Ora che il diagramma del subVI è inlined nel diagramma del chiamante, l'eliminazione del codice irraggiungibile e l'eliminazione del codice morto possono semplificare il codice. La prima struttura case è sempre eseguita, mentre la seconda struttura case non è mai eseguita.

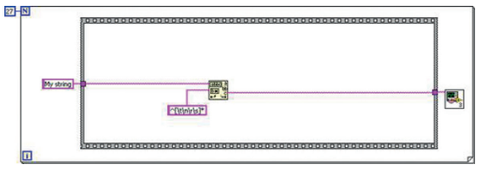


Figura 7. Le strutture case possono essere rimosse perché la logica d'ingresso è costante

Analogamente, lo spostamento del codice loop-invariante sposta la primitiva del match pattern fuori dal loop. Il grafico DFIR finale è equivalente al seguente codice G.

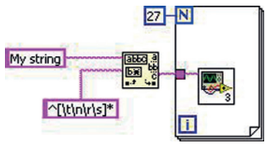


Figura 8. Codice G equivalente al grafico DFIR finale

Poiché il TrimWhitespace.vi è marcato come inline in LabVIEW 2010 per default, tutti i client di questo VI ricevono questi benefici automaticamente.

OTTIMIZZAZIONI DELLA DFIR

Dopo la profonda scomposizione del grafico DFIR, iniziano i passaggi di ottimizzazione della DFIR. Ulteriori ottimizzazioni sono eseguite in seguito durante la compilazione LLVM. Questo paragrafo copre solo una parte delle numerose ottimizzazioni. Ciascuna di queste trasformazioni è una comune ottimizzazione del compilatore, quindi dovrebbe essere facile trovare altre informazioni su una specifica ottimizzazione.

ELIMINAZIONE DEL CODICE IRRAGGIUNGIBILE

Il codice che non può mai essere eseguito è irraggiungibile. Rimuovere il codice irraggiungibile non rende direttamente il vostro tempo di esecuzione molto più veloce, ma rende il vostro codice più piccolo e accelera i tempi di compilazione perché il codice rimosso non è mai trasmesso nelle nei successivi passaggi di compilazione.

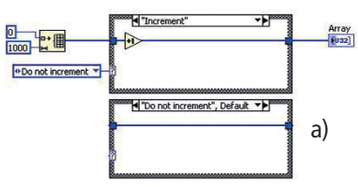
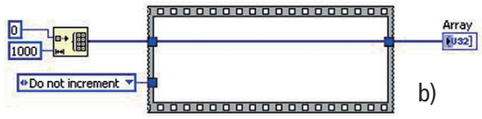


Figura 9. Codice G equivalente per la scomposizione con eliminazione del codice irraggiungibile della DFIR
a) prima dell'eliminazione del codice irraggiungibile
b) dopo l'eliminazione del codice irraggiungibile



In questo caso, il diagramma "Do not increment" della struttura case non è mai eseguito, quindi la trasformazione rimuove tale caso. Poiché la struttura case ha un solo caso rimanente, viene sostituita con una struttura in sequenza. In seguito, l'eliminazione del codice morto rimuove il frame e la costante enumerata.

SPOSTAMENTO DEL CODICE LOOP-INVARIANTE

Lo spostamento del codice loop-invariante identifica il codice entro il corpo di un loop che è possibile spostare all'esterno con sicurezza. Poiché il codice spostato viene eseguito meno volte, la velocità di esecuzione complessiva migliora.

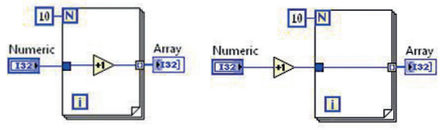


Figura 10. Codice G equivalente per la scomposizione con spostamento del codice loop-invariante della DFIR
Prima della trasformazione con spostamento del codice loop-invariante (a sinistra)
Dopo la trasformazione con spostamento del codice loop-invariante (a destra)

In questo caso, l'operazione di incremento è spostata all'esterno del loop. Il corpo del loop rimane permettendo la costruzione dell'array, ma i calcoli non devono essere ripetuti in ogni iterazione.

ELIMINAZIONE DELLE SOTTOESPRESSIONI COMUNI

L'eliminazione delle sottoespressioni comuni identifica calcoli ripetuti, esegue i calcoli una sola volta e riusa i risultati.

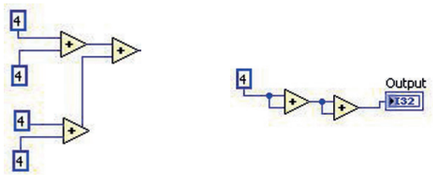


Figura 11. Codice G equivalente per la scomposizione con eliminazione delle sottoespressioni comuni della DFIR
Prima (a sinistra) e dopo (a destra)

FOLDING DELLE COSTANTI

Il folding delle costanti identifica parti di un diagramma che rimangono costanti durante l'esecuzione del codice e che quindi possono essere determinate in anticipo.

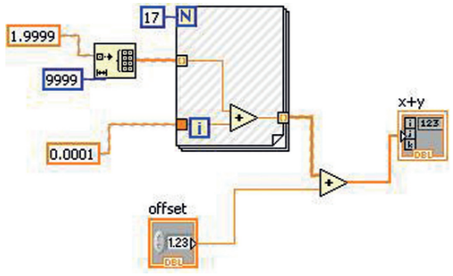


Figura 12. Il folding delle costanti può essere visualizzato nel diagramma a blocchi LabVIEW

Le marcature tratteggiate sul VI nella figura 12 indicano le parti constant-folded. In questo caso, il controllo "offset" non può essere constant folded, ma l'altro operando della primitiva di somma, incluso il For Loop, è valutato come costante.

LOOP UNROLLING

Il loop unrolling riduce l'overhead di un loop ripetendo più volte il corpo del loop nel codice generato e riducendo il numero totale delle iterazioni dello stesso fattore. Ciò riduce l'overhead del loop e mette in luce opportunità per ulteriori ottimizzazioni a scapito di un certo aumento delle dimensioni del codice.

ELIMINAZIONE DEL CODICE MORTO

Il codice morto è codice inutile. Rimuovere il codice morto velocizza il tempo di esecuzione, perché il codice rimosso non viene più eseguito.

Il codice morto è normalmente prodotto dalla manipolazione del grafico DFIR da trasformazioni che non avete scritto direttamente. Consideriamo il seguente esempio. L'eliminazione del codice irraggiungibile determina che la struttura case può essere rimossa. Ciò "crea" codice morto che la trasformazione di eliminazione del codice morto può rimuovere.

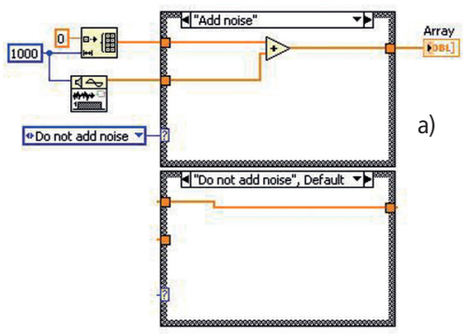
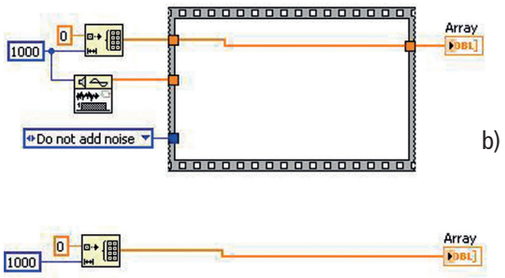


Figura 13. L'eliminazione del codice morto può ridurre la quantità di codice che il compilatore deve attraversare
a) Prima dell'eliminazione del codice irraggiungibile
b) Dopo l'eliminazione del codice morto



La maggior parte delle trasformazioni coperte in questo paragrafo hanno interrelazioni come questa. Eseguire una trasformazione può mettere in luce opportunità per eseguire altre trasformazioni.

TRASFORMAZIONI BACK-END DELLA DFIR

Dopo la scomposizione ed ottimizzazione del grafico DFIR, viene eseguita una serie di trasformazioni back-end. Tali trasformazioni valutano e annotano il grafico DFIR in preparazione alla sintesi finale del grafico DFIR nella IR LLVM IR.

CLUMPER

L'algoritmo di clumping analizza il parallelismo nel grafico DFIR e raggruppa i nodi in clump che si possono eseguire in parallelo. Questo algoritmo è strettamente legato al sistema di esecuzione run-time di LabVIEW, che usa il multitasking associato al multithreading. Ciascuno dei clump prodotti dal clumper è organizzato come task individuale nel sistema di esecuzione.

I nodi all'interno dei clump sono eseguiti in un ordine fisso, sequenziale.

Avere un ordine di esecuzione predeterminato con ogni clump permette all'inplacer di condividere le allocazioni dei dati e migliorare nettamente le prestazioni. Il clumper è anche responsabile dell'inserimento di prodotti intermedi in operazioni lunghe, come loop o I/O, in modo che tali clump siano eseguiti cooperativamente in multitask con altri clump.

Inplacer

L'inplacer analizza il grafico DFIR e identifica quando è possibile riutilizzare allocazioni dei dati e quando è necessario fare una copia.

Un filo in LabVIEW può essere un semplice scalare a 32 bit o un array di 32 MB. Fare in modo che tali dati siano riutilizzati il più possibile è critico in un linguaggio a flusso di dati come LabVIEW.

Considerate il seguente esempio (notate che il debugging del VI è disabilitato per ottenere le prestazioni e l'ingombro di memoria migliori).

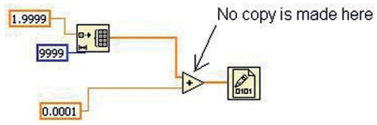


Figura 14. Semplice esempio che dimostra l'algoritmo di inplaceness

Questo VI inizializza un array, aggiunge qualche valore scalare a ogni elemento e lo scrive in un file binario. Quante copie dell'array dovrebbero esserci? LabVIEW deve creare l'array inizialmente, ma l'operazione di addizione può operare solo su quell'array, mantenendolo "al suo posto" (in place). Quindi, è necessaria una sola copia dell'array anziché un'allocazione per filo. Ciò si traduce in una significativa differenza – sia in consumo di memoria che in tempo di esecuzione – se l'array è grande. In questo VI, l'inplacer riconosce questa opportunità di operare "in-place" e configura il nodo somma in modo da trarne vantaggio.

Potete ispezionare questo comportamento nei VI che scrivete usando il tool "Show Buffer Allocations" sotto **Tools»Profile**. Il tool non mostra un'allocazione sulla primitiva addizione, indicando che non è stata fatta alcuna copia dei dati e che l'operazione di addizione avviene in place.

Ciò è possibile perché nessun altro nodo richiede l'array originale. Se modificate il VI come mostrato nella figura 15, l'inplacer deve fare una copia per la primitiva di addizione. Ciò perché la seconda primitiva Write to Binary File richiede l'array originale e deve essere eseguita dopo la prima primitiva **Write to Binary File**. Con questa modifica, il tool Show Buffer Allocations mostra un'allocazione sulla primitiva di addizione.

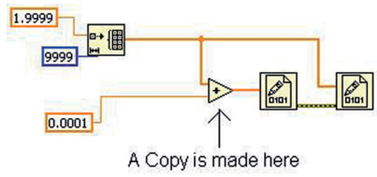


Figura 15. La diramazione del filo dell'array originale causa l'esecuzione di una copia in memoria

ALLOCATORE

Dopo che l'inplacer identifica quali nodi possono condividere locazioni di memoria con altri nodi, viene eseguito l'allocatore per creare le allocazioni che che servono al VI per la sua esecuzione. Ed è implementato analizzando ciascun nodo e terminale. I terminali che sono in-place rispetto ad altri terminali riusano le allocazioni anziché crearne di nuove.

GENERATORE DI CODICE

Il generatore di codice è il componente del compilatore che converte il grafico DFIR in istruzioni macchina eseguibili per il processore target. LabVIEW attraversa ogni nodo nel grafico DFIR nell'ordine del flusso dei dati, e ogni nodo chiama un'interfaccia nota come GenAPI, utilizzata per convertire il grafico DFIR nella forma di un linguaggio intermedio (IL) sequenziale che descrive la funzionalità di quel nodo. L'IL offre un modo indipendente dalla piattaforma per descrivere il comportamento a basso livello del nodo. Varie istruzioni nell'IL sono usate per implementare aritmetica, lettura e scrittura in memoria, eseguire confronti e salti condizionati e così via. Le istruzioni IL possono operare su memoria o su valori contenuti in registri virtuali utilizzati per memorizzare valori intermedi. Esempi di istruzioni IL includono GenAdd, GenMul, GenIf, GenLabel e GenMove. In LabVIEW 2009 e precedenti, questa forma IL era convertita direttamente in istruzioni macchina (come 80X86 e PowerPC) per la piattaforma target. LabVIEW usava un semplice allocatore di registri a singola passata per mappare i registri virtuali suoi registri fisici della macchina e ogni istruzione IL emetteva un set hard-coded di particolari istruzioni macchina per implementarlo su ogni piattaforma target supportata. Benché ciò fosse terribilmente veloce, era ad hoc, produceva poco codice e non era molto adatto all'ottimizzazione. La DFIR, essendo una rappresentazione ad alto livello, indipendente dalla piattaforma, è limitata nell'assortimento di trasformazioni del codice che può supportare. Per aggiungere supporto al il set completo di ottimizzazioni del codice in un moderno compilatore ottimizzante, LabVIEW ha recentemente adottato una tecnologia open source di terze parti chiamata LLVM.

LLVM

Low Level Virtual Machine (LLVM) è un framework compilatore open source versatile, a elevate prestazioni, inventato in origine come progetto di ricerca presso l'Università dell'Illinois. La LLVM è ora ampiamente usata in campo accademico e industriale grazie alla sua API flessibile e pulita e alle licenze senza limitazioni. In LabVIEW 2010, il generatore di codice LabVIEW è rifattorizzato per usare la LLVM al fine di generare codice macchina target. La rappresentazione IL LabVIEW esistente ha fornito un buon punto di partenza in questo sforzo, richiedendo la riscrittura di appena circa 80 istruzioni IL anziché il set molto più grande di nodi e primitive DFIR supportati da LabVIEW. Dopo avere creato il flusso di codice IL dal grafico DFIR di un VI, LabVIEW passa in rassegna ogni istruzione IL e crea una rappresentazione assembly LLVM equivalente. Invoca vari passaggi di ottimizzazione e usa quindi il framework Just-in-Time (JIT) LLVM per creare istruzioni macchina eseguibili in memoria. L'informazione di riallocazione macchina della LLVM è convertita in una rappresentazione LabVIEW, quindi quando salvate il VI su disco e lo ricaricate in un diverso indirizzo base di memoria, potete farne un patch up corretto ed eseguirlo nella nuova locazione.

Alcune delle ottimizzazioni standard del compilatore per ese-

guire le quali LabVIEW usa la LLVM includono le seguenti:

- Combinazione di istruzioni
- Salto tra thread
- Sostituzione scalare di aggregati
- Propagazione condizionale
- Eliminazione delle chiamate di coda
- Riassociazione delle espressioni
- Spostamento di codice loop-invariante
- Unswitching dei loop e splitting degli indici
- Semplificazione delle variabili di induzione
- Unrolling dei loop
- Numerazione globale dei valori
- Eliminazione delle memorie morte
- Eliminazione aggressiva del codice morto
- Propagazione sparsa condizionale delle costanti

Una spiegazione completa di tutte queste ottimizzazioni esula dallo scopo di questo articolo, ma vi sono molte informazioni sul tema su Internet e sulla maggior parte dei libri di testo sui compilatori.

Benchmark interni hanno mostrato che l'introduzione della LLVM ha prodotto un miglioramento medio del 20 percento del tempo di esecuzione dei VI. I singoli risultati dipendono dalla natura dei calcoli eseguiti dal VI; alcuni VI vedono un miglioramento maggiore di questo e alcuni non vendono alcun cambiamento nelle prestazioni. Per esempio, i VI che usano la libreria di analisi avanzata o sono comunque fortemente dipendenti da codice già implementato in C ottimizzato, vedono una piccola differenza nelle prestazioni. LabVIEW 2010 è la prima versione che usa la LLVM e vi è ancora un enorme potenziale da liberare per futuri miglioramenti.

LA DFIR E LA LLVM LAVORANO IN TANDEM

Potreste avere notato che alcune di queste ottimizzazioni, come lo spostamento di codice loop-invariante e l'eliminazione del codice morto, erano già descritte come eseguite dalla DFIR. Infatti, alcuni passaggi di ottimizzazione traggono vantaggi nell'essere eseguiti più volte e a differenti livelli del compilatore perché altre passaggi di ottimizzazione potrebbero avere trasformato il codice in modo tale da rendere disponibili altre opportunità di ottimizzazione. Il fatto è che, benché la DFIR sia un'IR ad alto e la LLVM sia un'IR a basso livello, entrambe lavorano in tandem per ottimizzare il codice LabVIEW

Note sull'autore

Jeffrey Phillips è un Product Marketing Manager di LabVIEW in National Instruments. E' laureato in ingegneria meccanica all'Università del Tennessee.