Dokumententyp: Tutorium
Von NI unterstützt: Ja
Veröffentlichungsdatum: 03.08.2010

Wie funktioniert der Compiler von NI LabVIEW?

tsverzeichnis

sicht

Der Entwurf eines Compilers kann selbst für eine triviale Programmiersprache schnell zu einer komplexen Aufgabe werden. Auch unter professionellen Softwareentwicklern gehört Compiler-Theorie eher zur Kategorie Spezialwissen. NI LabVIEW ist eine Sprache, die mehrere Programmierparadigmen umfasst, u. a. das Konzept des Datenflusses, objektorientierte Entwicklung und ereignisgesteuerte Programmierung. LabVIEW kann zudem auf mehreren Plattformen, unter verschiedenen Betriebssystemen (Windows, Linux, Mac), auf unterschiedlichen Hardwarearchitekturen (PowerPC, Intel) und sogar auf Embedded-Geräten und FPGAs, die sich von traditionellen PC-Architekturen unterscheiden, eingesetzt werden. Dabei wird deutlich, dass es sich beim LabVIEW-Compiler um ein ausgeklügeltes System handelt, das den Rahmen nur einer Abhandlung bei weitem übersteigt.

In diesem Dokument wird der LabVIEW-Compiler näher vorgestellt, von den Anfängen im Jahre 1986 mit LabVIEW 1.0 bis zu seiner heutigen Form. Des Weiteren werden aktuelle Compilerinnovationen untersucht und die Vorteile dieser neuen Funktionen für die LabVIEW-Architektur und somit den Anwender hervorgehoben.

ilieren vs. Interpretieren

Bei LabVIEW handelt es sich um eine kompilierte Sprache. Dies mag überraschend sein, da es bei der Entwicklung mit der grafischen Programmiersprache G normalerweise keinen expliziten Kompilierschritt gibt. Stattdessen wird nach einer Änderung an einem VI nur auf „Ausführen“ geklickt. Kompilierung bedeutet, dass der in G geschriebene Code in nativen Maschinencode übersetzt wird, welcher dann direkt vom Host-Computer ausgeführt wird. Ein alternativer Ansatz ist die Interpretation, wobei Programme indirekt von einem anderen Softwareprogramm (Interpreter genannt) anstatt direkt vom Computer ausgeführt werden.

Die erste LabVIEW-Version verwendete einen Interpreter, der in späteren Versionen durch einen Compiler ersetzt wurde, um die Ausführungsleistung der VIs zu verbessern – ein Hauptvorteil von Compilern gegenüber Interpretern. Interpreter können in der Regel einfacher geschrieben und gewartet werden, jedoch bieten sie eine schlechtere Ausführungsleistung. Compiler sind komplexer zu implementieren, sorgen als Ausgleich aber für eine schnellere Ausführung. Einer der Hauptvorteile des LabVIEW-Compilers liegt darin, dass Verbesserungen am Compiler auf alle VIs übertragen werden, ohne dass Änderungen am VI notwendig sind. Tatsächlich war eines der Hauptanliegen von LabVIEW 2010 die Compileroptimierung, um die Ausführungszeit von VIs zu beschleunigen.

egang des LabVIEW-Compilers

Bevor der heute in LabVIEW eingesetzte Compiler im Detail besprochen wird, lohnt es sich, einen Blick auf die Entwicklung des Compilers seit den Anfängen vor mehr als 20 Jahren zu werfen. Einige der hier angesprochenen Algorithmen, wie z. B. Type Propagation, Clumping und In-Place, werden im Abschnitt zum aktuellen LabVIEW-Compiler genauer erläutert.

LabVIEW 1.0 kam im Jahr 1986 auf den Markt. Wie bereits erwähnt, verwendete LabVIEW damals einen Interpreter und war nur für den Einsatz mit dem Prozessor Motorola 68000 gedacht. Die LabVIEW-Sprache war wesentlich einfacher, wodurch auch an den Compiler (bzw. zu diesem Zeitpunkt Interpreter) geringere Anforderungen gestellt wurden. Beispielsweise gab es zu diesem Zeitpunkt keinen Polymorphismus und der einzige numerische Datentyp war das Extended-Precision-Fließkomma. In LabVIEW 1.1 wurden erstmals In-Place-Algorithmen oder "Inplacer" eingeführt. Dieser Algorithmus erkennt Datenzuweisungen, die während der Ausführung wiederverwendet werden können. Dadurch lassen sich unnötige Datenkopien vermeiden, was wiederum die Ausführungsleistung und -geschwindigkeit wesentlich verbessert.

In LabVIEW 2.0 wurde der Interpreter dann von einem echten Compiler abgelöst. LabVIEW war immer noch nur für die Ausführung mit dem Motorola 68000 konzipiert, konnte nun aber nativen Maschinencode erzeugen. Zudem wurde LabVIEW 2.0 um den Type-Propagation-Algorithmus ergänzt, der u. a. für die Syntaxüberprüfung und Typenauflösung zuständig ist. Eine weitere große Innovation von LabVIEW 2.0 war die Einführung des sogenannten Clumpers. Der Clumping-Algorithmus erkennt Parallelitäten im LabVIEW-Diagramm und fasst Knoten in "Clumps" zusammen, so dass diese parallel ausgeführt werden können. All diese Algorithmen – Type Propagation, Clumping und In-Place – sind auch heute noch wichtige Bestandteile des LabVIEW-Compilers und wurden über die Jahre zahlreichen Verbesserungen unterzogen. Die Compilerinfrastruktur von LabVIEW 2.5 bot Unterstützung für mehrere Prozessorarchitekturen, im Speziellen für Intel x86 und Sparc. LabVIEW 2.5 wurde zudem um den "Linker" ergänzt, der für eine Neukompilierung Abhängigkeiten zwischen VIs verwaltet.

LabVIEW 3.1 unterstützte zwei neue Prozessorarchitekturen, PowerPC und HP PA-RISC, und führte das Constant Folding ein. LabVIEW 5.0 und 6.0 sahen einen überarbeiteten Codegenerator sowie die GenAPI, eine gemeinsame Schnittstelle für verschiedene Backends. Die GenAPI ermöglicht eine Cross-Kompilierung, was für die Entwicklung von Echtzeit-Systemen wichtig ist. Echtzeitentwickler schreiben VIs zwar auf einem Host-PC, kompilieren diese jedoch für ein Echtzeit-Zielsystem. Eine zusätzliche Neuerung war die Einführung einer eingeschränkte Form der Loop-Invariant Code Motion (Verschieben schleifeninvarianten Codes). Zu guter Letzt wurde das multitaskingfähige LabVIEW-Ausführungssystem für die Unterstützung mehrerer Threads erweitert.

LabVIEW 8.0 baute auf der GenAPI-Infrastruktur aus Version 5.0 auf und führte einen Algorithmus für die Registerzuweisung ein. Vor der Einführung der GenAPI waren die Register für jeden Knoten fest in den erzeugten Code programmiert. Eingeschränkte Formen der Unreachable Code und Dead Code Elimination (Eliminierung von nicht erreichbarem bzw. totem Code) waren ebenso Bestandteil von LabVIEW 8.0. LabVIEW 2009 umfasste eine 64-bit-LabVIEW-Version und Dataflow Intermediate Representation (DFIR). DFIR wurde eingesetzt, um anspruchsvollere Arten von Loop-Invariant Code Motion, Constant Folding, Unreachable Code und Dead Code Elimination zu erstellen. Neue Funktionen der Sprache wie z. B. die parallele For-Schleife basieren auf dem DFIR.

In LabVIEW 2010 ermöglicht DFIR nun neue Compileroptimierungen wie Algebraic Reassociation (algebraische Reassoziation), Common Subexpression Elimination (Eliminierung gemeinsamer Unterausdrücke), Loop Unrolling (Schleifenausrollen) und SubVI-Inlining (Inlining von SubVIs). Für diese Version wurde zudem eine Low-Level Virtual Machine (LLVM) in die LabVIEW-Compilerkette integriert. LLVM ist eine weit verbreitete Open-Source-Compilerinfrastruktur, welche Optimierungen wie Instruction Scheduling (Terminierung von Anweisungen), Loop Unswitching (Schleifenausschaltung), Instruction Combining (Anweisungszusammenfassung), Conditional Propagation (bedingte Ausbreitung) sowie eine weiterentwickelte Registerzuteilung ermöglichte.

eutige Kompilierprozess

Mit diesem Grundwissen zur Entwicklung des LabVIEW-Compilers wird nun nachfolgend der Kompilierprozess in der aktuellen LabVIEW-Version näher erläutert.

Der erste Schritt in der Kompilierung eines VIs ist der Type-Propagation-Algorithmus. Dieser komplexe Schritt ist für die Handhabung implizierter Typen für VI-Anschlüsse, die an Typen angepasst werden können, und für die Erkennung von Syntaxfehlern verantwortlich. Während dieses Schritts werden alle möglichen Syntaxfehler in der G-Programmiersprache erkannt. Ist das VI laut Algorithmus gültig, wird der Kompiliervorgang fortgeführt.

Nach der Type Propagation wird das VI zuerst vom im Blockdiagramm verwendeten Modell in die vom Compiler verwendete Dataflow Intermediate Representation (DFIR) umgewandelt. Sobald dies geschehen ist, führt der Compiler auf dem DFIR-Graphen mehrere Transformationen durch, um das VI zu zerlegen, zu optimieren und für die Codeerzeugung vorzubereiten. Viele der

Compileroptimierungen, wie das In-Place und Clumping, sind Transformationen und werden in diesem Schritt ausgeführt.

Nachdem der DFIR-Graph optimiert und vereinfacht wurde, wird er in die LLVM-Zwischendarstellung übersetzt. Die Zwischendarstellung wird einer Reihe von LLVM-Durchläufen unterzogen, um sie noch weiter zu optimieren und letztendlich in Maschinencode umzuwandeln.

Propagation

Wie bereits erwähnt, löst der Type-Propagation-Algorithmus Typen auf und erkennt Programmierfehler. Tatsächlich übernimmt der Algorithmus mehrere Aufgaben, u. a.:

- Auflösung implizierter Typen für VI-Anschlüsse, die an unterschiedliche Datentypen angepasst werden können
- Auflösung von SubVI-Aufrufen und Bestimmung ihrer Gültigkeit
- Berechnung der Datenrichtung
- Prüfung auf VI-Zyklen
- Erkennung und Meldung von Syntaxfehlern

Dieser Algorithmus wird nach jeder Änderung am VI ausgeführt, um festzustellen, ob das VI noch ausführbar ist. Allerdings lässt sich streiten, ob dieser Schritt wirklich zur "Kompilierung" gehört. Jedoch entspricht dieser Schritt am deutlichsten der lexikalischen, syntaktischen und semantischen Analyse traditioneller Compiler.

Ein einfaches Beispiel für einen Anschluss, der an Typen angepasst werden kann, ist die Addier-Funktion in LabVIEW. Werden zwei Integer addiert, ist das Ergebnis ein Integer; werden jedoch zwei Fließkommazahlen addiert, ist das Ergebnis eine Fließkommazahl. Ein ähnliches Muster existiert auch für Verbundtypen wie Arrays und Cluster. Es gibt andere Sprachenkonstrukte wie Schieberegister, die komplexeren Typisierungsregeln unterliegen. Im Fall der Addier-Funktion hängt der Ausgangstyp vom Eingangstyp ab und es wird von einer "Verbreitung" (Propagation) des Typs durch das Diagramm gesprochen, daher der Name des Algorithmus.

Das oben genannte Beispiel illustriert auch die Syntaxüberprüfungsaufgabe des Algorithmus. Angenommen, ein Integer und ein String werden mit einer Addier-Funktion verdrahtet. Was sollte passieren? Da das Addieren dieser beiden Werte in diesem Fall keinen Sinn ergibt, meldet der Algorithmus diesen Fehler und markiert das VI als "inkorrekt". Dies führt dazu, dass das VI nicht ausgeführt werden kann (der Ausführpfeil wird unterbrochen dargestellt).

st eine Zwischendarstellung und wozu gibt es sie?

Nachdem der Algorithmus festgestellt hat, dass ein VI gültig ist, wird der Kompiliervorgang fortgesetzt und das VI wird in die DFIR übersetzt. Bevor die DFIR genauer beschrieben wird, sollten Zwischendarstellungen im Allgemeinen betrachtet werden.

Eine Zwischendarstellung ist eine Darstellung des Programms, das während verschiedener Phasen des Kompiliervorgangs manipuliert wird. Zwischendarstellungen werden häufig in moderner Compilerliteratur erwähnt und können für alle Programmiersprachen eingesetzt werden.

Nachfolgend einige Beispiele. Heutzutage gibt es verschiedene gängige Zwischendarstellungen. Zwei Beispiele hierfür sind abstrakte Syntaxbäume (AST) und 3-Adresscodes.

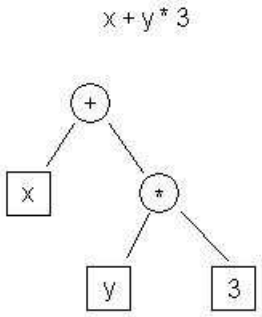


Abbildung 1: *Beispiel für einen AST*

t0 <- y

t1 <- 3

t2 <- t0 * t1

t3 <- x

t4 <- t3 + t2

Tabelle 1: *Beispiel für einen 3-Adresscode*

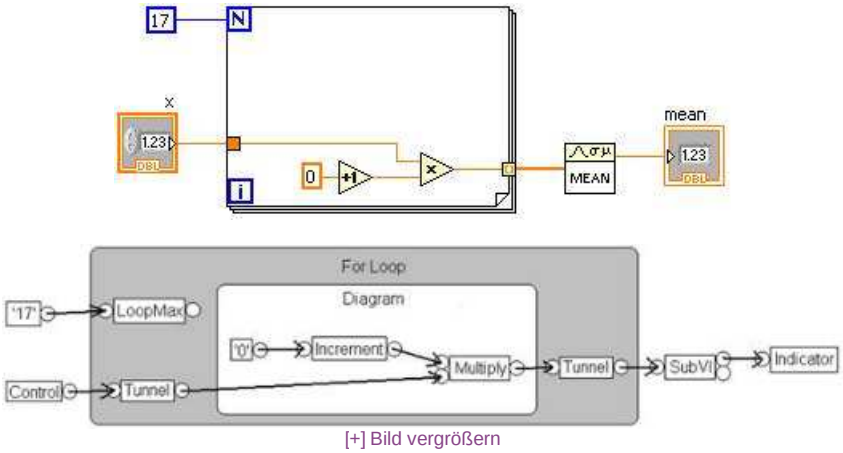
In Abbildung 1 ist der AST für “x + y * 3” zu sehen, während Tabelle 1 die 3-Adresscode-Darstellung zeigt.

Ein deutlicher Unterschied ist, dass der AST eine Higher-Level-Darstellung ist. Sie ist näher an der Ursprungsdarstellung des Programms (C) als an der Zieldarstellung (Maschinencode). Der 3-Adresscode ist im Vergleich dazu eher eine Low-Level-Darstellung und ähnelt eher Assembler-Code.

Jede dieser beiden Darstellungen hat ihre Vorteile. So lassen sich z. B. Abhängigkeitsanalysen einfacher mit einer High-Level-Darstellung wie dem AST durchführen als mit dem 3-Adresscode. Andere Compileroptimierungen wie die Registerzuordnung oder das Instruction Scheduling werden in der Regel mit einer Low-Level-Darstellung wie dem 3-Adresscode durchgeführt.

Da jede Zwischendarstellung ihre eigenen Stärken und Schwächen hat, nutzen viele Compiler (einschließlich LabVIEW) verschiedene Zwischendarstellungsoptionen. Bei LabVIEW wird die DFIR (Dataflow Intermediate Representation) als High-Level-Zwischendarstellung eingesetzt, während die LLVM (Low-Level Virtual Machine) als Low-Level-Darstellung dient.

In LabVIEW wird DFIR als High-Level-Darstellung eingesetzt. Es handelt sich dabei um eine hierarchische und graphenbasierte Darstellung, die dem eigentlichen G-Code ähnelt. Wie G-Code setzt sich auch die DFIR aus mehreren Knoten zusammen, die jeweils einen eigenen Anschluss haben. Die Anschlüsse können mit anderen Anschlüssen verbunden werden. Einige Knoten, wie z. B. Schleifen, enthalten Diagramme, die wiederum weitere Knoten enthalten können.



[+] Bild vergrößern

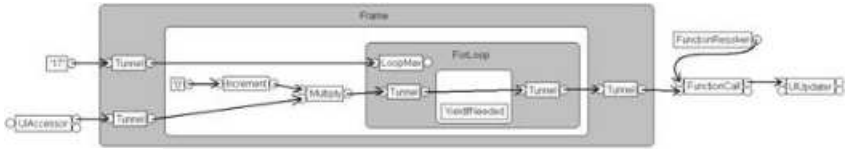
Abbildung 2: *LabVIEW-G-Code und der entsprechende DFIR-Graph*

In Abbildung 2 ist ein einfaches VI zusammen mit der anfänglichen DFIR-Darstellung zu sehen. Bei der ersten Erstellung des DFIR-Graphen für ein VI handelt es sich um eine direkte Übersetzung des G-Codes und jeder Knoten im DFIR-Graphen hat eine direkte Entsprechung im G-Code. Während des Kompiliervorgangs werden DFIR-Knoten verschoben, aufgeteilt oder neue Knoten werden integriert. Einer der Hauptvorteile des DFIR ist die Aufrechterhaltung von Eigenschaften wie Parallelitäten. Parallelität, die im 3-Adresscode dargestellt wird, ist wesentlich schwieriger zu erkennen.

DFIR bietet zwei wesentliche Vorteile für den LabVIEW-Compiler. Erstens entkoppelt die DFIR den Editor von der Compilerdarstellung des VIs. Und zweitens dient die DFIR als gemeinsame Schnittstelle für den Compiler, der über mehrere Front- und Backends verfügt. Nachfolgend werden diese Vorteile im Einzelnen näher beleuchtet.

Der DFIR-Graph entkoppelt den Editor von der Compilerdarstellung

Vor Einführung der DFIR gab es bei LabVIEW nur eine Darstellung des VIs, die sowohl vom Editor als auch vom Compiler geteilt wurde. Dies verhinderte jedoch, dass der Compiler während des Kompiliervorgangs Änderungen an der Darstellung vornehmen konnte. Aufgrund dessen war es natürlich auch schwieriger, Compilerverbesserungen einzuführen.



[+] Bild vergrößern

Abbildung 3: *DFIR bietet Rahmenbedingungen für die Codeoptimierung durch den Compiler*

Abbildung 3 zeigt einen DFIR-Graphen für das bereits vorgestellte VI. Der Graph zeigt den Kompiliervorgang zu einem wesentlich späteren Zeitpunkt, nachdem der Code durch mehrere Transformationen zerlegt und optimiert wurde. Der Graph unterscheidet sich dementsprechend von der vorhergehenden Darstellung. Folgende Schritte wurden durchgeführt:

- Zerlegungstransformationen haben die Bedien-, Anzeige- und SubVI-Knoten entfernt und durch folgende neue Knoten ersetzt: UIAccessor, UIUpdater, FunctionResolver und FunctionCall.
- Durch die Loop-Invariant Code Motion wurden die Inkrementier- und Multiplizierknoten außerhalb des Schleifenrumpfes platziert.
- Der Clumper hat einen YieldIfNeeded-Knoten in die For-Schleife platziert, wodurch der ausführende Thread die Ausführung mit anderen Arbeitsobjekten teilen muss.

Transformationen werden an späterer Stelle noch ausführlicher besprochen.

Die DFIR-Zwischendarstellung dient als gemeinsame Schnittstelle für mehrere Compiler-Front- und -Backends

LabVIEW kann mit mehreren Zielen eingesetzt werden, u. a. einem x86-Desktop-PC und einem Xilinx-FPGA, auch wenn diese sich wesentlich voneinander unterscheiden. In ähnlicher Weise stellt LabVIEW Anwendern unterschiedliche Verarbeitungsmodelle zur Verfügung. Zusätzlich zur grafischen Programmierung in G bietet LabVIEW beispielsweise mit MathScript textbasierte Mathematik. Dadurch entsteht eine Sammlung an Front- und Backends, die alle mit dem LabVIEW-Compiler kompatibel sein müssen. Der Einsatz der DFIR als gemeinsame Zwischendarstellung, die von den Frontends erzeugt und von den Backends genutzt wird, vereinfacht die Wiederverwendung bei unterschiedlichen Kombinationen. So kann z. B. die Optimierung für das Constant Folding, die auf dem DFIR-Graphen ausgeführt wird, einmal geschrieben und anschließend auf Desktop-, Echtzeit-, FPGA- und Embedded-Zielen eingesetzt werden.

Zerlegung

Befindet sich das VI im DFIR-Status wird zuerst eine Reihe an Zerlegungstransformationen durchgeführt. Diese Transformationen dienen der Reduzierung bzw. Normalisierung des DFIR-

Graphen. Die Transformation für unverdrahtete Ausgangstunnel z. B. findet in Case- und Ereignisstrukturen unverdrahtete Ausgangstunnel, die als “Standard verwenden, wenn nicht verbunden” konfiguriert sind. Bei diesen Anschlüssen legt die Transformation eine Konstante mit dem Standardwert ab und verdrahtet diese mit dem Anschluss, so dass die Eigenschaft “Standard verwenden, wenn nicht verbunden” explizit im DFIR-Graphen festgelegt ist. Bei nachfolgenden Kompilierdurchgängen werden dann alle Anschlüsse gleich und unter der Annahme, dass sie über verdrahtete Eingänge verfügen, behandelt. In diesem Fall wurde also die Eigenschaft “Standard verwenden, wenn nicht verbunden” durch die Reduzierung der Darstellung in eine elementarere Form“wegkompiliert”.

Dieses Konzept kann auch auf komplexere Sprachen angewandt werden. So kommt z. B. eine Zerlegungstransformation zum Einsatz, um den Rückkopplungsknoten in Schieberegister auf einer While-Schleife zu reduzieren. Eine weitere Transformation implementiert die parallele For-Schleife als mehrere sequenzielle For-Schleifen mit zusätzlicher Logik, die Eingänge in parallelisierbare Komponenten für die sequenziellen Schleifen zerteilt und sie anschließend wieder zusammenfügt.

Eine neue Funktion in LabVIEW 2010, das SubVI-Inlining, wird ebenfalls als DFIR-Zerlegung implementiert. Während dieser Kompilierphase wird der DFIR-Graph mit als “inline” markierten SubVIs direkt in den DFIR-Graphen des Aufrufers eingefügt. Zusätzlich zur Vermeidung des Overheads eines SubVI-Aufrufs legt das Inlining weitere Optimierungsmöglichkeiten offen, da Aufrufer und aufgerufenes Objekt in einem DFIR-Graphen zusammengefasst sind. Ein Beispiel hierfür ist dieses einfache VI, welches das TrimWhitespace.vi aus vi.lib aufruft.

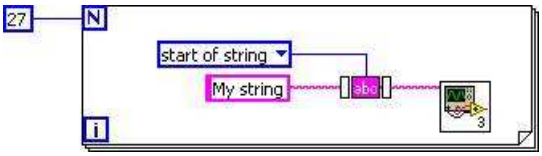
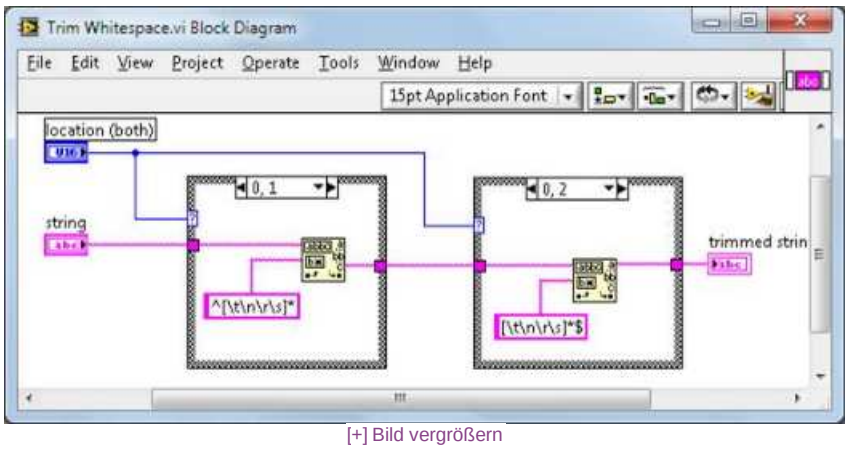


Abbildung 4: *Beispiel eines einfachen VIs zur Demonstration von DFIR-Optimierungen*

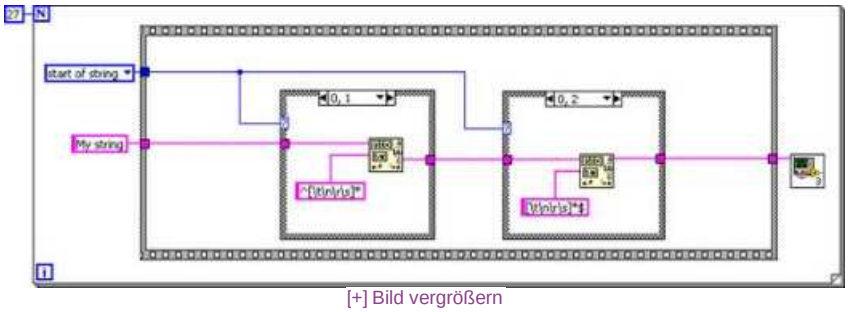
Das TrimWhitespace.vi wird in der vi.lib wie folgt definiert:



[+] Bild vergrößern

Abbildung 5: *Blockdiagramm für TrimWhitespace.vi*

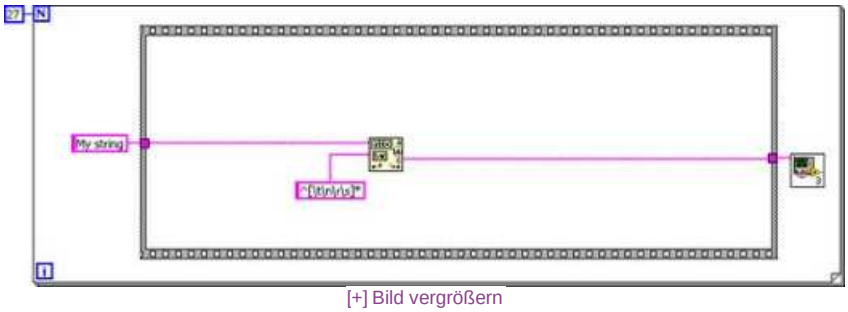
Das SubVI wird inline in den Aufrufer eingefügt und es entsteht ein DFIR-Graph, der dem folgenden G-Code entspricht.



[+] Bild vergrößern

Abbildung 6: *G-Code-Entsprechung für den DFIR-Graphen des inline gesetzten TrimWhitespace.vi*

Nachdem das SubVI-Diagramm inline in das Aufruferdiagramm gesetzt wurde, kann durch die Unreachable Code Elimination und Dead Code Elimination eine Codevereinfachung stattfinden. Die erste Case-Struktur wird immer ausgeführt, während die zweite Case-Struktur nie ausgeführt wird.



[+] Bild vergrößern

Abbildung 7: *Die Case-Strukturen können entfernt werden, da die Eingangslogik konstant ist.*

Ähnlich dessen wird durch die Loop-Invariant Code Motion die Funktion "Pattern vergleichen" aus dem Schleifenrumpf verschoben. Der finale DFIR-Graph entspricht dem folgenden G-Code:

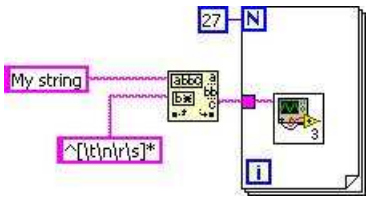


Abbildung 8: *G-Code-Entsprechung für den finalen DFIR-Graphen*

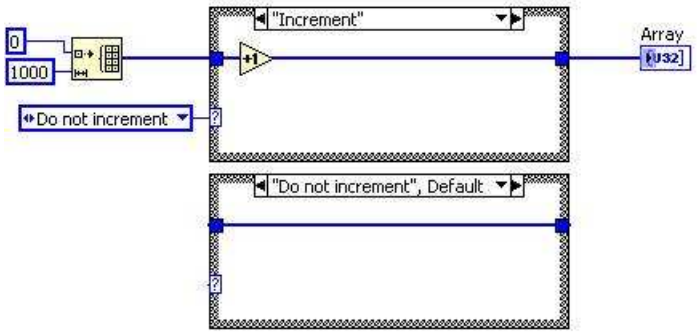
Da in LabVIEW 2010 das TrimWhitespace.vi standardmäßig als inline markiert ist, profitieren alle aufrufenden VIs automatisch von den gleichen Vorteilen.

Optimierungen

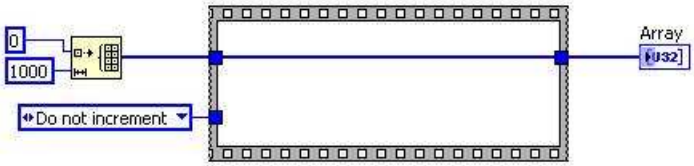
Nachdem der DFIR-Graph sorgfältig zerlegt wurde, beginnen die DFIR-Optimierungsdurchläufe. Während der späteren LLVM-Kompilierung werden noch weitere Optimierungen durchgeführt. In diesem Abschnitt wird nur ein kleiner Teil der vielen Optimierungen besprochen. Bei all diesen Transformationen handelt es sich um gängige Compileroptimierungen, so dass zusätzliche Informationen zu den einzelnen Optimierungen relativ einfach zu finden sein sollten.

Unreachable Code Elimination

Bei Programmcode, der nie ausgeführt werden kann, handelt es sich um nicht erreichbaren Code (Unreachable Code). Das Entfernen dieses Codes sorgt zwar nicht direkt für eine schnellere Ausführung, jedoch wird die Codegröße reduziert und somit auch die Kompilierzeiten verkürzt, da der entfernte Code nun nicht mehr kompiliert werden muss.



Vor der Unreachable Code Elimination



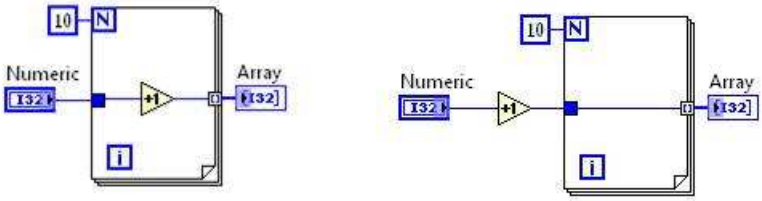
Nach der Unreachable Code Elimination

Abbildung 9: *G-Code-Entsprechung für die DFIR-Zerlegung Unreachable Code Elimination*

In diesem Beispiel wird das "Nicht inkrementieren"-Diagramm der Case-Struktur nie ausgeführt. Aufgrund dessen wird es durch die Transformation entfernt. Da die Case-Struktur nun nur noch über ein Case verfügt, wird es durch eine Sequenzstruktur ersetzt. Durch die Dead Code Elimination werden später der Rahmen und die Enum-Konstante entfernt.

Loop-Invariant Code Motion

Bei der Loop-Invariant Code Motion wird Code, der sich nicht unbedingt innerhalb des Schleifenrumpfes befinden muss, außerhalb der Schleife platziert. Da dieser Code weniger häufig ausgeführt wird, verbessert sich die Gesamtausführungsgeschwindigkeit.



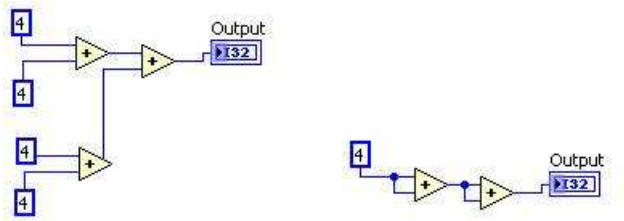
Vor der Loop-Invariant Code Motion **Nach der Loop-Invariant Code Motion**

Abbildung 10: *G-Code-Entsprechung für die DFIR-Zerlegung Loop-Invariant Code Motion*

In diesem Beispiel wurde die Inkrementieroperation außerhalb des Schleifenrumpfes platziert. Der Schleifenrumpf bleibt bestehen, damit das Array erstellt werden kann, jedoch muss die Berechnung nicht bei jeder Iteration wiederholt werden.

Common Subexpression Elimination

Die Transformation Common Subexpression Elimination sorgt dafür, dass Berechnungen nur einmal durchgeführt und die Ergebnisse wiederverwendet werden.



Vorher Nachher

Abbildung 11: *G-Code-Entsprechung für die DFIR-Zerlegung Common Subexpression Elimination*

Constant Folding

Beim Constant Folding werden die Diagrammbereiche identifiziert, die während der Ausführung konstant bleiben und somit frühzeitig bestimmt werden können.

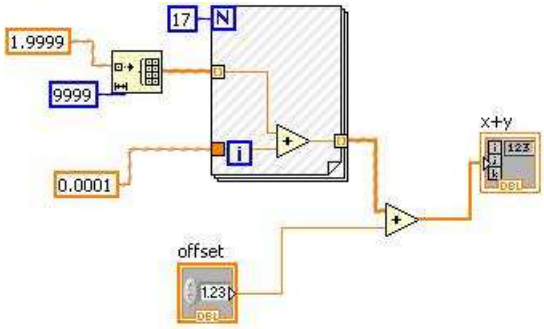


Abbildung 12: *Constant Folding kann im LabVIEW-Blockdiagram dargestellt werden*

Die Nummernzeichen im VI in Abb. 12 zeigen den Bereich an, der dem Constant Folding unterzogen wurde. In diesem Fall kann die "Offset"-Steuerung nicht dem Constant Folding unterzogen werden. Allerdings hat der Operand der Plus-Funktion, einschließlich der For-Schleife, einen konstanten Wert.

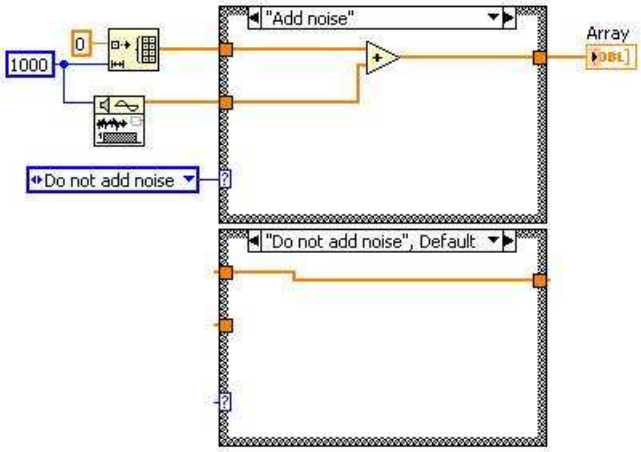
Loop Unrolling

Loop Unrolling reduziert Schleifen-Overhead, indem der Schleifenrumpf mehrmals im erzeugten Code wiederholt und die Gesamtzahl der Iterationen durch denselben Faktor reduziert wird. Dadurch bieten sich weitere Optimierungen, allerdings vergrößert sich auch der Code.

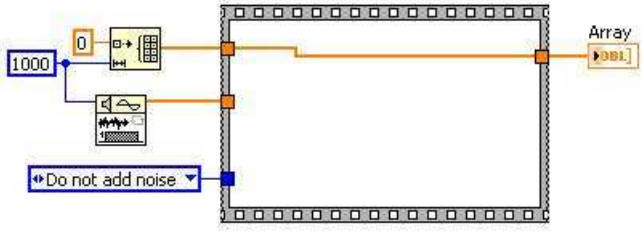
Dead Code Elimination

Bei totem Code (Dead Code) handelt es sich um überflüssigen Code. Die Entfernung dieses Codes sorgt für eine Erhöhung der Ausführungsgeschwindigkeit, da dieser Code nicht mehr länger ausgeführt wird.

Toter Code entsteht häufig durch die Veränderung des DFIR-Graphen im Rahmen von Transformationen, die nicht direkt vom Programmierer geschrieben wurden. Man betrachte das folgende Beispiel. Die Transformation Unreachable Code Elimination entscheidet, dass die Case-Struktur entfernt werden kann. Somit "entsteht" toter Code, der von der Transformation Dead Code Elimination dann endgültig entfernt wird.



Vorher



Nach der Unreachable Code Elimination

Nach der Dead Code Elimination

Abbildung 13: *Durch die Dead Code Elimination wird die Menge des zu kompilierenden Codes reduziert.*

Ein Großteil der in diesem Abschnitt besprochenen Transformationen ist miteinander verbunden, d. h. die Durchführung einer Transformation deckt Optimierungsmöglichkeiten für andere Transformationen auf.

Backend-Transformationen

Nachdem der DFIR-Graph zerlegt und optimiert wurde, wird eine Reihe von Backend-Transformationen durchgeführt. Bei diesen Transformationen wird der DFIR-Graph in Vorbereitung auf die LLVM-Zwischendarstellung evaluiert und annotiert.

Clumper

Der Clumping-Algorithmus identifiziert Parallelitäten im DFIR-Graphen und fasst Knoten in "Clumps" zusammen, so dass diese parallel ausgeführt werden können. Dieser Algorithmus ist eng mit dem LabVIEW-Ausführungssystem verbunden, das kooperatives Multitasking mit mehreren Threads nutzt. Jeder dieser "Clumps" wird als individueller Task im Ausführungssystem eingeplant. Knoten innerhalb der Clumps werden in einer festgelegten, seriellen Reihenfolge ausgeführt. Durch die festgelegte Ausführungsreihenfolge jedes Clumps kann der In-Place-Algorithmus Datenzuordnungen aufteilen, wodurch die Leistung wesentlich gesteigert wird. Der Clumper ist zudem für das Einfügen von Yields in längere Operationen wie Schleifen oder I/O verantwortlich, was für die Durchführung des kooperativen Multitaskings der Clumps notwendig ist.

In-Place

Der In-Place-Algorithmus analysiert den DFIR-Graphen und stellt fest, ob Datenzuordnungen wiederverwendet werden können oder ob Datenkopien erstellt werden müssen. Ein Draht in LabVIEW kann ein einfacher 32-bit-Skalar oder ein 32-MB-Array sein. Die Sicherstellung der häufigen Wiederverwendung von Daten ist in einer Datenflusssprache wie LabVIEW unabdingbar.

Dies wird anhand des folgenden Beispiels verdeutlicht (das VI-Debugging ist in diesem Fall deaktiviert, um eine bessere Leistung und weniger Speicherbedarf zu erreichen).

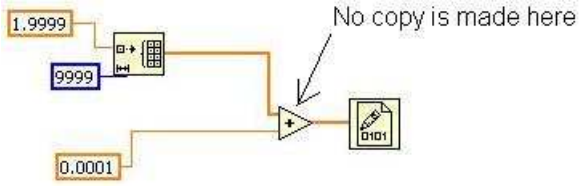


Abbildung 14: *Einfaches Beispiel zur Demonstration des In-Place-Algorithmus*

Das VI initialisiert ein Array, addiert jedem Element einen Skalarwert hinzu und schreibt es in eine Binärdatei. Wie viele Kopien des Arrays werden benötigt? LabVIEW muss das Array zunächst erstellen, jedoch wird die Addier-Operation in-place nur an diesem Array durchgeführt. Somit ist nur eine Kopie des Arrays notwendig, anstelle einer Zuweisung für jede Verdrahtung. Dies kann einen wesentlichen Unterschied bedeuten, – sowohl im Speicherbedarf als auch in der Ausführungszeit – wenn es sich um ein umfangreiches Array handelt. Bei diesem VI erkennt der Algorithmus die Möglichkeit, die Operation in-place durchzuführen und konfiguriert den Addier-Knoten entsprechend.

Dieses VI-Verhalten kann unter **Werkzeuge»Profil** "Pufferzuweisungen anzeigen..." eingesehen werden. Das Werkzeug zeigt keine Zuweisung für die Addier-Funktion an, was bedeutet, dass keine Datenkopie erstellt wurde und die Addier-Operation in-place durchgeführt wird.

Dies ist möglich, da kein anderer Knoten das Original-Array benötigt. Wird das VI wie in Abb. 15 modifiziert, muss der In-Place-Algorithmus eine Kopie für die Addier-Funktion erstellen. Dies ist notwendig, da die zweite Funktion **In Binärdatei schreiben** das Original-Array benötigt und nach der ersten Funktion **In Binärdatei schreiben** ausgeführt werden muss. Bei dieser Modifizierung zeigt das Werkzeug "Pufferzuweisungen anzeigen..." eine Zuweisung für die Addier-Funktion an.

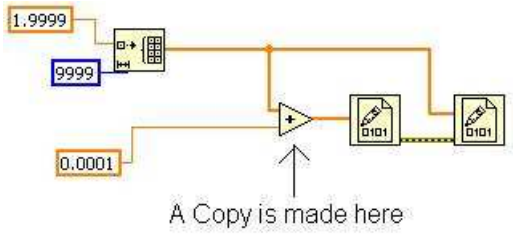


Abbildung 15: *Die Verzweigung der Original-Array-Verdrahtung macht eine Datenkopie im Speicher notwendig*

Registerzuteiler

Nachdem der In-Place-Algorithmus festgestellt hat, welche Knoten einen Speicherplatz teilen können, nimmt der Registerzuteiler die für die VI-Ausführung notwendigen Zuteilungen vor. Dies wird an jedem Knoten und Anschluss durchgeführt. Anschlüsse, die sich für andere Anschlüsse in-place befinden, verwenden diese Zuteilungen wieder, anstatt neue zu erstellen.

Codegenerator

Der Codegenerator ist der Bestandteil des Compilers, der den DFIR-Graphen in ausführbare Maschinenanweisungen für den Zielprozessor umwandelt. LabVIEW durchläuft jeden Knoten im DFIR-Graphen in der Reihenfolge des Datenflusses, wobei jeder Knoten eine Schnittstelle, genannt GenAPI, aufruft. Die GenAPI konvertiert den DFIR-Graphen in einen sequenziellen Zwischencode, welcher die Knotenfunktionen beschreibt. Der Zwischencode ist eine plattformunabhängige Darstellung für die Beschreibung des Low-Level-Knotenverhaltens. Verschiedene Anweisungen im Zwischencode werden zur Implementierung von arithmetischen Funktionen, Speicherlese- und -schreibfunktionen sowie zur Durchführung von Vergleichen und bedingten Verzweigungen genutzt. Zwischencodeanweisungen können entweder auf Werte im Arbeitsspeicher oder auf Werte im virtuellen Register angewandt werden. Beispiele für Zwischencodeanweisungen sind GenAdd, GenMul, GenIf, GenLabel und GenMove.

In LabVIEW 2009 und früheren Versionen wurde dieser Zwischencode direkt in Maschinenanweisungen (z. B. 80X86 und PowerPC) für die Zielplattform umgewandelt. LabVIEW nutzte hierzu einen einfachen One-Pass-Registerzuteiler, um virtuelle Register auf physische Maschinenregister abzubilden. Für jede Zwischencodeanweisung wurde ein fest programmierter Satz an spezifischen Maschinenanweisungen ausgegeben, um die jeweilige Anweisung auf der Zielplattform zu implementieren. Dies geschah zwar blitzschnell, jedoch handelt es sich dabei um eine Ad-hoc-Operation, die schwachen Code produziert und wenig Möglichkeiten für Optimierungen zulässt. Da die DFIR eine High-Level- und plattformunabhängige Darstellung ist, unterstützt sie nur eine begrenzte Auswahl an Codetransformationen. Um jedoch alle für moderne Compiler zur Verfügung stehenden Codeoptimierungen nutzen zu können, implementierte LabVIEW kürzlich eine Open-Source-Technologie eines Drittanbieters genannt LLVM.

Die Low-Level Virtual Machine (LLVM) ist eine vielseitige, leistungsstarke Open-Source-Compilerarchitektur, die ursprünglich als Forschungsprojekt an der University of Illinois entwickelt wurde. Sie wird nun dank ihrer flexiblen, einheitlichen API und uneingeschränkten Lizenzierung in vielen Bereichen von Industrie, Forschung und Lehre verwendet.

In LabVIEW 2010 wurde der LabVIEW-Codegenerator dahingehend überarbeitet, dass er LLVM zur Erzeugung des Zielmaschinencodes nutzt. Die bestehende LabVIEW-Zwischencodedarstellung stellte eine gute Ausgangsbasis dar, da nur ca. 80 Zwischencodeanweisungen neu geschrieben werden mussten (im Vergleich zur wesentlich größeren Anzahl an von LabVIEW unterstützten DFIR-Knoten und -Funktionen).

Am Anschluss an die Erzeugung des Zwischencode-Streams aus dem DFIR-Graphen eines VIs geht LabVIEW jede Zwischencodeanweisung durch und erstellt eine entsprechende LLVM-Assembly-Darstellung. Nach mehreren Optimierungsdurchläufen wird die Just-in-time-Kompilierung der LLVM genutzt, um im Arbeitsspeicher ausführbare Maschinenanweisungen zu erzeugen. Die Maschinenverlagerungsinformationen (Machine Relocation Information) der LLVM werden in eine

LabVIEW-Darstellung konvertiert, so dass, wenn ein VI auf der Festplatte gespeichert und an eine andere Speicheradresse geladen wird, das VI auch am neuen Ort korrekt ausgeführt wird.

Einige der von der LLVM durchgeführten Compileroptimierungen sind:

- Instruction Combining
- Jump Threading
- Scalar Replacement of Aggregates
- Conditional Propagation
- Tail Call Elimination
- Expression Reassociation
- Loop-Invariant Code Motion
- Loop Unswitching und Index Splitting
- Induction Variable Simplification
- Loop Unrolling
- Global Value Numbering
- Dead Store Elimination
- Aggressive Dead Code Elimination
- Sparse Conditional Constant Propagation

Eine vollständige Erläuterung all dieser Optimierungen würde den Rahmen dieses Dokuments sprengen. Das Internet sowie die meisten Lehrbücher zur Compilertheorie bieten jedoch eine Fülle an Informationen hierzu.

Interne Leistungsüberprüfungen haben gezeigt, dass durch die Einführung von LLVM eine durchschnittliche Steigerung von 20 Prozent bei der VI-Ausführungszeit erreicht werden kann. Einzelergebnisse hängen natürlich von der Art der durchgeführten Berechnungen ab, wobei einige VIs durchaus noch größere Leistungssteigerungen erreichen, während bei anderen VIs kein Unterschied festzustellen ist. Zum Beispiel erfahren VIs, die auf die erweiterte Analysebibliothek zugreifen oder anderweitig stark von Code abhängig sind, der bereits als optimierter C-Code implementiert ist, wenig bis keine Leistungsveränderungen. LabVIEW 2010 ist die erste Version, die LLVM einsetzt, was ein großes Potenzial für zukünftige Verbesserungen eröffnet.

und LLVM arbeiten Hand in Hand

Einige der erwähnten LLVM-Optimierungen, wie z. B. Loop-Invariant Code Motion und Dead Code Elimination, wurden bereits der DFIR zugeordnet. Allerdings kann es nützlich sein, einige Optimierungsdurchläufe mehrere Male und auf verschiedenen Compilerebenen durchzuführen, da sich dadurch noch weitere Optimierungsmöglichkeiten eröffnen können. Zusammenfassend ist zu sagen, dass die High-Level-Zwischendarstellung DFIR und die Low-Level-Zwischendarstellung LLVM Hand in Hand arbeiten, um LabVIEW-Code für die Ausführung auf dem jeweiligen Prozessor zu optimieren.

rführende Links

Webcast: Funktionen der neuen LabVIEW-Version

Testen Sie LabVIEW kostenfrei: Evaluierungsversion zum Download

LabVIEW-Softwarepakete im Vergleich

Hauptseite NI LabVIEW