Eine Optimierungsbeschreibungssprache zur Verbesserung von Laufzeitabschätzungen

Andreas Monitzer

9. August 2005

Zusammenfassung

In letzter Zeit wurden Laufzeitberechnungen immer wichtiger, allerdings werden diese durch die von Übersetzern durchgeführten Optimierungen (Schleifenaufrollung, Umstrukturierung der Befehle) sehr ungenau, und beschreiben nicht mehr das tatschliche Verhalten des Programms im ausführbaren Zustand.

Eine Beschreibungssprache wird vorgestellt, die demonstriert, wie man Optimierungen compilerunabhängig darstellen kann. In weiterer Folge wird gezeigt, dass diese Sprache dazu verwendet werden kann, Laufzeitberechnungen unter Berücksichtigung von Optimierungen zu ermöglichen.

1 Einleitung

Im Bereich der eingebetteten Programmierung gelten gewisse Einschränkungen, die bei der herkömmlichen nicht vorhanden sind (im Regelfall). Beispielsweise muss das Programm in einen vergleichsweise winzigen Speicher passen, oder gewisse Geschwindigkeitsvorgaben einhalten (z.B. muss ein Prallkissen innerhalb einer gewissen Zeit auslösen, sonst wird es mehr zur Lebensgefahr als zum Lebensretter). Um dieses Bestreben erfüllen zu können, gibt es automatisierte Verfahren, die direkt beim Übersetzungsverfahren gewisse Verbesserungen vornehmen.

Die maximale Laufzeit eines Programms kann man bestimmen, indem man sämtliche Eingabemöglichkeiten und Umgebungsparameter testet, und dann immer die Laufzeit misst. Die höchste, die dabei vorkommt, ist die gesuchte (und einzig relevante bei eingebetteter Programmentwicklung). Allerdings ist diese Versuchsanordnung meist nicht möglich, daher wird dann auf eine heuristische Suche

zurückgegriffen. Diese darf den Aufwand allerdings keinesfalls unterschätzen, sollte ihn aber auch so wenig wie möglich überschätzen.

Bei der Übersetzung eines Quelltextes in ein ausführbares Maschinenprogramm gehen gewisse Details verloren, die notwendig sind, um akkurate Laufzeitabschätzungen durchzuführen. Daher wird diese bevorzugt direkt am Quelltext gemacht. Durch die oben erwähnten Optimierungsverfahren hat der Quelltext allerdings nicht mehr viel mit dem Maschinenprogramm gemein, die Abschätzung wird dadurch sehr ungenau.

Es gibt daher zwei Möglichkeiten, dieses Problem anzugehen:

1. Die Analyse am Quelltext vornehmen, aber Informationen vom Compiler über die verwendeten Optimierungen einbeziehen.
2. Die Analyse am Maschinenprogramm vornehmen, und die benötigten Informationen vom Quelltext übernehmen.

Variante 2 ist einfacher zu implementieren, da hier bereits vorhandene Werkzeuge nur erweitert werden müssen. Da dies aber einen schweren Eingriff in den Übersetzer darstellt, würde man sich an einen konkreten binden, wenn man diese Funktionalität direkt einbaut. Um dieses Problem zumindest teilweise in den Griff zu bekommen, ist eine Beschreibungssprache für Übersetzeroptimierungen geeignet, da diese relativ einfach in andere Übersetzer eingebaut werden kann. Diese Beschreibungssprache (genannt ODL) beschreibt neben der Optimierung selbst, wie die Optimierungen sich auf die Laufzeit auswirken, was direkt für die Auswertung der maximalen Laufzeit verwendet werden kann.

2 Architektur

Abbildung 1 zeigt, wie ein Werkzeug zur Laufzeitanalyse aufgebaut werden kann. Die höhere Analyse erzeugt aus dem Programm eine Art Ablaufdiagramm, welches Informationen darüber enthält, wann und wie oft ein s.g. Grundblock[1] ausgeführt wird. Der Übersetzer wurde leicht modifiziert, um die Informationen zu erhalten, die der Ko-Transformator benötigt. Der übersetzte Text wird von der niedrigeren Analyse dahingehend untersucht, wie lange ein Grundblock zur Ausführung benötigt (im günstigsten und im schlechtesten Fall), diese Information wird daraufhin zusammen mit den Daten vom Ko-Tranformator verwendet, um ein entgültiges Resultat der Laufzeitanalyse zu erhalten.

[1]Ein Grundblock ist ein Teil eines Programms, der absolut linear abgearbeitet wird, d.h. keinerlei Sprünge enthält.

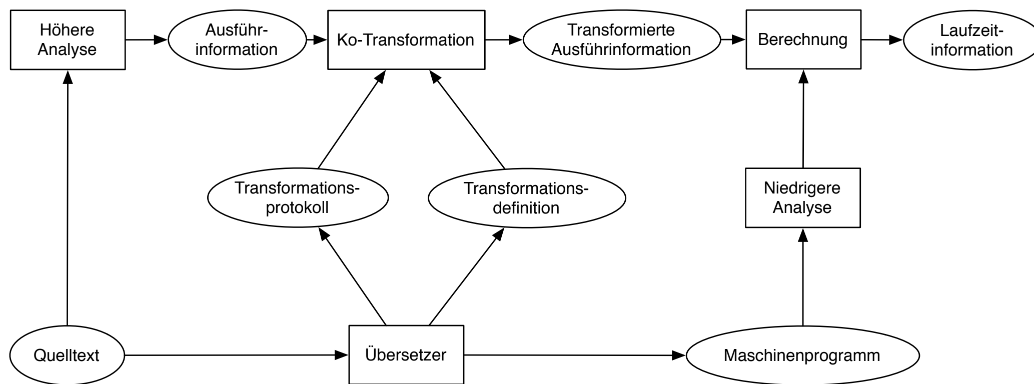

Abbildung 1: Die Komponenten der Laufzeitanalyse [1, Abb. 1]

Die Berechnungskomponente muss wissen, wo die Grundblöcke aus der höheren Analyse in der niedrigeren Analyse zu finden sind. Diese Informationen stellt der Ko-Transformator bereit. Dazu werden Optimierungen, die der Übersetzer am Quelltext durchführt, an den Daten der höheren Analyse parallel dazu durchgeführt.

Es gibt grundsätzlich drei Herangehensweisen, wie man die dazu benötigten Informationen erhalten kann:

1. Der Übersetzer gibt zusätzlich zum optimierten Programm noch eine Art Protokoll aus, das dann von einem separat laufenden Ko-Transformator gelesen und weiterverarbeitet wird.
2. Der Ko-Transformator wird in den Übersetzer integriert, wobei die Quelltextveränderungen und die Laufzeitveränderungen getrennt geführt werden (und damit auch parallel aktualisiert werden müssen).
3. Der Ko-Transformator wird in den Übersetzer integriert, die Laufzeitveränderungen werden aus den Quelltextveränderungen geschlussfolgert.

Möglichkeit 1 ist am Einfachsten zu implementieren und wurde von Jakob Engblom eingesetzt. In Zunkunft sollte an Variante 2 gearbeitet werden, Variante 3 benötigt noch viel Forschungszeit [2, Seite 23].

Sobald der Ko-Transformator weiß, welche Optimierungen durch den Übersetzer durchgeführt wurden, muss er daraus folgern können, welche Informationen er den Ausführinformationen beifügen muss, die die höhere Analyse bereits gewonnen hat. Dazu ist die Optimierungsbeschreibungssprache (ODL) von Jakob Engblom et al. geeignet.

3 ODL

Die Optimierungsbeschreibungssprache ist nur dazu ausgelegt, befehlsübergreifende Optimierungen innerhalb von Funktionen zu beschreiben, andere wie zB Konstantenfaltung (z.B. $3*4 \rightarrow 12$), Befehlskombination ((a++)++→a+=2) oder Funktionsintegration (eine Funktion wird in eine andere kopiert, um die Kosten des Funktionsaufrufs selber zu eliminieren) werden nicht berücksichtigt.

Wie bereits erwähnt, wird das Programm in ein Auflaufdiagramm umgewandelt. Dabei entsteht ein Graph, der genau einen Ein- und einen Ausgangsknoten hat. Einzelne Knoten in diesem Graphen können unter Umständen zusammengefasst werden zu einem größeren Knoten (beispielsweise die Ausführungszweige einer Bedingungsanweisung), dieser Knoten enthält dann selbst wieder ein Auflaufdiagramm, das aus Knoten besteht. Um die Eindeutigkeit zu gewährleisten, wird jedem Knoten ein eindeutiger Name verliehen.

Eine Beschreibung einer Optimierung besteht aus 3 Abschnitten: Dem Eingangsmuster, dem Ausgangsmuster und der Transformation.

Das Eingangsmuster ist der Teil, dem der Quelltext entsprechen muss, damit diese Optimierung durchgeführt werden kann. Es definiert gleichzeitig auch Namen für Programmteile, die in den anderen Abschnitten verwendet werden.

Hier ist ein Beispiel dafür ([2, Seite 32]):

```
inpattern
  fragment loop(S,LoopBegin,LoopEnd)
    cnode(loopbody,LoopBegin,LoopEnd,LoopBodyData);
  endfragment

  fragment repr(S,Loop,Loop)
    node(Loop,LoopData);
  endfragment
```

Das Ausgangsmuster beschreibt das Endergebnis. Hierbei muss neu eingeführten Knotenpunkten im Auflaufdiagramm des Programmteils ein neuer Name gegeben werden (dazu dient „`NewNames`“), wenn dies nicht notwendig ist, kann `id` verwendet werden, um die Beibehaltung zu signalisieren.

Das Muster zum oberen Eingangsmuster ist ([2, Seite 33]):

```
outpattern
```

```
fragment loop(looppeeled,loopmain)
  cnode(looppeeled,loopbody,NewNames,NewOut,[loopmain]);
  cnode(loopmain,loopbody,id,LoopBodyOut,[(loopmain)]);
end fragment

fragment repr(Loop,Loop)
  node(Loop,LoopDataOut,[]);
end fragment
```

Hierbei wird das Programm dahingehend verändert, dass von einer Schleife die erste Iteration herausgenommen wird und vor die Schleife selber gestellt wird („loop peel" [3, Kapitel 1.1]). Die runden Klammern um „loopmain" bedeuten, dass hier die Schleife erhalten bleiben soll.

Als letzter Schritt muss noch die Transformationsinformation bereitgestellt werden ([2, Seite 33]):

```
transfers
  %% update the loop iteration count
  LoopDataOut.iterations = sub(LoopData.iterations,1);
  LoopDataOut.scope = LoopData.scope;
  LoopDataOut.execcount = LoopData.execcount;

  %% update the execcounts of the loop body
  LoopBodyOut.scope = LoopBodyData.scope;
  LoopBodyOut.iterations = LoopBodyData.iterations;
  LoopBodyOut.execcount = forall( X1:LoopBodyData |
    capsub(X1.execcount,LoopData.iterations,1));

  %% update the peeled code (only exec once)
  NewOut.scope = forall( X2:LoopBodyData | copy(LoopData.scope));
  NewOut.iterations = forall( X3:LoopBodyData | copy(1));
  NewOut.execcount = forall( X4:LoopBodyData |
    fraction(X4.execcount,LoopData.iterations,LoopData.execcount));
```

Die neue Schleife hat einen Schleifendurchgang weniger (`LoopDataOut.iterations`), und die herausgenommene erste Iteration hat genau einen Schleifendurchgang (`NewOut.iterations`).

Alle verwendeten Funktionen (`sub`, `fraction`, etc) sind in einer externen Erlang-Datei definiert. Eine genaue Beschreibung der Programmiersprache kann unter [2, Kapitel 5.2.3] gefunden werden.

3.1 Ein komplettes Beispiel

Das aus [2] übernommene Beispiel zeigt die konkrete Anwendung der Ko-Transformation in Erlang. Für das bessere Verständnis sollte Abbildung 1 herangezogen werden.

Abbildung 2 zeigt das Programm, so wie es die höhere Analyse verlässt. Es gibt 3 Funktionen, die gleichzeitig auch Stammknoten im Baum darstellen, in denen die Grundblöcke (`bb` von *Basic Block*) aufgelistet sind. Der erste Parameter von `bb` gibt den Namen des Blocks an, der zweite die Liste der Nachfolgeblöcke und der dritte enthält die Daten (hier die Anzahl der Ausführungen, der Schleifenbereich und die Anzahl der Iterationen).

```
odlprograminstance Test1

odlinstanceversion 1

trace
  include "loops.trace";

callgraph

  node(c1,[],cesar)
    bb(block1,[block2],"[1,cesar,1]")
    bb(block2,[block3],"[10,loop1,1]")
    bb(block3,[block2,block4],"[10,loop1,1]")
    bb(block4,[block5],"[1,cesar,1]")
    bb(block5,[block6,block8],"[6,loop3,1]")
    bb(block6,[block7],"[3,loop3,1]")
    bb(block8,[block9,block10],"[3,loop3,1]")
    bb(block9,[block11],"[2,loop3,1]")
    bb(block10,[block11],"[2,loop3,1]")
    bb(block11,[block7],"[3,loop3,1]")
    bb(block7,[block5,block12],"[6,loop3,1]")
    bb(block12,[block13],"[20,loop2,1]")
    bb(block13,[block14,block5],"[20,loop2,1]")
    bb(block14,[],"[1,cesar,1]")

    %% loop representatives
    bb(cesar,[],"[1,main,1]")
    bb(loop1,[],"[1,cesar,10]")
    bb(loop2,[],"[1,cesar,20]")
    bb(loop3,[],"[20,loop2,6]")

  node(i1,[],iffy)
    bb(if1,[if2],"[1,iffy,1]")
    bb(if2,[if3,if4],"[100,loopbig,1]")
    bb(if3,[if5],"[100,loopbig,1]")
    bb(if5,[if7],"[100,loopbig,1]")
    bb(if7,[if9,if10],"[16,loop2,1]")
    bb(if9,[if11],"[12,loop2,1]")
    bb(if10,[if11],"[4,loop2,1]")
    bb(if11,[if12,if7],"[16,loop2,1]")
    bb(if12,[if13],"[100,loopbig,1]")
    bb(if13,[if2,if14],"[100,loopbig,1]")
    bb(if14,[],"[1,iffy,1]")
    bb(if4,[if6],"[0,iffy,1]")
    bb(if6,[if8],"[0,iffy,1]")
    bb(if8,[if12],"[0,iffy,1]")

    %% loop representatives
    bb(iffy,[],"[1,main,1]")
    bb(loop2,[],"[100,loopbig,16]")
    bb(loopbig,[],"[1,iffy,100]")

  node(h1,[],hamlet)
    bb(h1,[h2],"[1,hamlet,1]")
    bb(h2,[h2,h3],"[7,loopb2,1]")
    bb(h3,[h3,h4],"[10,loopb3,1]")
    bb(h4,[h4,h5],"[10,loopb4,1]")
    bb(h5,[h6],"[10,loopb56,1]")
    bb(h6,[h5,h7],"[10,loopb56,1]")
    bb(h7,[h8],"[1,hamlet,1]")
    bb(h8,[h9,h13],"[30,loopu,1]")
    bb(h9,[h10,h11],"[30,loopu,1]")
    bb(h10,[h12],"[20,loopu,1]")
    bb(h11,[h12],"[19,loopu,1]")
    bb(h12,[h15],"[30,loopu,1]")
    bb(h13,[h14],"[30,loopu,1]")
    bb(h14,[h15],"[30,loopu,1]")
    bb(h15,[h16,h17],"[30,loopu,1]")
    bb(h16,[h18],"[25,loopu,1]")
    bb(h17,[h18],"[20,loopu,1]")
    bb(h18,[h19,h8],"[30,loopu,1]")
    bb(h19,[],"[1,hamlet,1]")

    %% loop representatives
    bb(loopb2,[],"[1,hamlet,7]")
    bb(loopb3,[],"[1,hamlet,10]")
    bb(loopb4,[],"[1,hamlet,10]")
    bb(loopb56,[],"[1,hamlet,10]")
    bb(loopu,[],"[1,hamlet,30]")

end odlprograminstance
```

Abbildung 2: `loops.prog`

Abbildung 3 zeigt das Transformationsprotokoll. Die einzelnen vorgenommenen Optimierungen werden aufgelistet, wobei die Funktionen hier als Parameter dienen.

In Abbildung 4 ist die Transformationsdefinition zu finden. Diese beschreibt, wie der Übersetzer das Programm verändert. Hier ist nur der Kopf zu sehen, die genauen Transformationsdefinitionen würden den Rahmen sprengen und sind in [2], Anhang B zu finden. `fix.erl` (Abbildung 5) definiert die hier verwendeten Operationen.

```
odltraceprogram Test1
odltraceversion 0

transformations
  include "loops.trans" ;

trace

  IfOpt(iffy,if2,if3,if11,if4,if8,if12);

  DeadCode(cesar,block13);
    %% create preheader to enable loop collapsing
  CreatePreheader(cesar,block5,block7,phloop3,loop3);
  LoopCollapse(cesar,phloop3,phloop3,block12,block12,loop2,block5,block7,loop3,loop57);
  CreatePreheader(cesar,block5,block7,phloop57,loop57);

    %% peel the nest in cesar
  LoopPeel(cesar,block5,block7,loop57,
    %% NOTE: quoted list is a name translation table
    "[{block5,new5},{block6,new6},{block7,new7},
    {block8,new8},{block9,new9},{block10,new10},{block11,new11}]");

    %% rebuilding hamlet
  LoopDistribution(hamlet,h5,h5,h6,h6,loopb56,loopb5,loopb6);
  LoopFusion(hamlet,h4,h4,h5,h5,loopb4,loopb5,loopb45);
  LoopFusion(hamlet,h3,h3,h4,h5,loopb3,loopb45,loopb345);
  BlockMerge(hamlet,h4,h5,h45);
  LoopUnswitching(hamlet,h8,h8,h9,h12,h13,h14,h15,h18,loopu,
    newif,newjoin,"[{h8,new8}]",
    "[{h15,new15},{h16,new16},{h17,new17},{h18,new18}]",
    loopa,loopb);
  TwoPieceIf(hamlet,h9,h10,h10,h11,h11,h12,h9_lo);

    %% interchanging loops!
  LoopInterchange(iffy,loopbig,if2,if5,if12,if13,loop2,if7,if11);
end odltraceprogram
```

Abbildung 3: `loops.trace`

```
odltransprogram Loops
odltransversion 0

bbformats
  default = ( execcount, scope, iterations ) ;

functions
  include "fix.erl" ;

transformations
  %% see elsewhere

end odltransprogram
```

Abbildung 4: `loops.trans`

```
%%% File : fix.erl
%%% Author : Jakob Engblom <jakob@Zapata.DoCS.UU.SE>
%%% Purpose : Test function module for cotransformer
%%% Created : 17 Jul 1997 by Jakob Engblom <jakob@Zapata.DoCS.UU.SE>

-module(fix).
-author('jakob@Zapata.DoCS.UU.SE').
-export([
  copy/1,
  mul/2,
  sub/2,
  add/2,
  cap/2,
  capsub/3,
  fraction/3
]).

copy(X) ->
  X.

mul(X,Y) ->
  X*Y.

sub(X,Y) ->
  X-Y.

add(X,Y) ->
  X+Y.

cap(X,Cap) when (X>Cap) ->
  Cap;
cap(X,_) ->
  X.

capsub(X,Cap,Sub) ->
  %% simple workaround for lack of composition in funcalls
  cap(X,Cap-Sub).

fraction(OldExec,OldIter,NewIter) ->
  OldExec * NewIter / OldIter.
```

Abbildung 5: fix.erl

Abbildung 6 zeigt die Ausgabe der Ko-Transformation, die dann der Berechnung zugeführt wird.

4 Berechnung der Laufzeit

Die schlussendliche Analyse passiert von unten nach oben: Um die Laufzeit einer Funktion ermitteln zu können, muss man zuerst wissen, wie lange alle Grundblöcke zur Ausführung benötigen (diese können wiederum Funktionsaufrufe enthalten, die zuerst analysiert werden müssen).

4.1 Plattformspezifische Überlegungen

Es gibt viele Faktoren, die zusammen eine Plattform ergeben:

- Der Prozessor (wie lange ein Befehl zur Ausführung braucht, Zwischenspeichers, etc., siehe auch Kapitel 4.2)

```
odlprograminstance Converted
odlinstanceversion 1

trace
  include "";

callgraph

  node(c1,[],cesar)
    bb(block1,[block2],"[1,cesar,1]")
    bb(block10,[block11],"[40,loop57,1]")
    bb(block11,[block7],"[60,loop57,1]")
    bb(block14,[],"[1,cesar,1]")
    bb(block2,[block3],"[10,loop1,1]")
    bb(block3,[block2,block4],"[10,loop1,1]")
    bb(block4,[phloop57],"[1,cesar,1]")
    bb(block5,[block6,block8],"[119,loop57,1]")
    bb(block6,[block7],"[60,loop57,1]")
    bb(block7,[block14,block5],"[119,loop57,1]")
    bb(block8,[block10,block9],"[60,loop57,1]")
    bb(block9,[block11],"[40,loop57,1]")
    bb(cesar,[],"[1,main,1]")
    bb(loop1,[],"[1,cesar,10]")
    bb(loop57,[],"[1,cesar,119]")
    bb(new10,[new11],"[0.333333,cesar,1]")
    bb(new11,[new7],"[0.500000,cesar,1]")
    bb(new5,[new6,new8],"[1.00000,cesar,1]")
    bb(new6,[new7],"[0.500000,cesar,1]")
    bb(new7,[block5],"[1.00000,cesar,1]")
    bb(new8,[new10,new9],"[0.500000,cesar,1]")
    bb(new9,[new11],"[0.333333,cesar,1]")
    bb(phloop57,[new5],"[1,cesar,1]")

  node(h1,[],hamlet)
    bb(h1,[h2],"[1,hamlet,1]")
    bb(h10,[h12],"[20,loopa,1]")
    bb(h11,[h12],"[19,loopa,1]")
    bb(h12,[h15],"[30,loopa,1]")
    bb(h13,[h14],"[30,loopb,1]")
    bb(h14,[new15],"[30,loopb,1]")
    bb(h15,[h16,h17],"[30,loopa,1]")
    bb(h16,[h18],"[25,loopa,1]")
    bb(h17,[h18],"[20,loopa,1]")
    bb(h18,[h8,newjoin],"[30,loopa,1]")
    bb(h19,[],"[1,hamlet,1]")
    bb(h2,[h2,h3],"[7,loopb2,1]")
    bb(h3,[h45],"[10,loopb345,1]")
    bb(h45,[h3,h6],"[10,loopb345,1]")
    bb(h6,[h6,h7],"[10,loopb6,1]")
    bb(h7,[newif],"[1,hamlet,1]")
    bb(h8,[h9],"[30,loopa,1]")
    bb(h9,[h10,h11],"[30,loopa,1]")
    bb(loopa,[loopb],"[1,hamlet,30]")
    bb(loopb,[],"[1,hamlet,30]")
    bb(loopb2,[],"[1,hamlet,7]")
    bb(loopb345,[loopb345],"[1,hamlet,10]")
    bb(loopb6,[],"[1,hamlet,10]")
    bb(new15,[new16,new17],"[30,loopb,1]")
    bb(new16,[new18],"[25,loopb,1]")
    bb(new17,[new18],"[20,loopb,1]")
    bb(new18,[new8,newjoin],"[30,loopb,1]")
    bb(new8,[h13],"[30,loopb,1]")
    bb(newif,[h8,new8],"[1,hamlet,30]")
    bb(newjoin,[h19],"[1,hamlet,30]")

  node(i1,[],iffy)
    bb(if1,[if2],"[1,iffy,1]")
    bb(if10,[if11],"[25.0000,loop2,1]")
    bb(if11,[if12,if7],"[100.000,loop2,1]")
    bb(if12,[if13],"[16.0000,loopbig,1]")
    bb(if13,[if14],"[16.0000,loopbig,1]")
    bb(if14,[],"[1,iffy,1]")
    bb(if2,[if3],"[16.0000,loopbig,1]")
    bb(if3,[if5],"[16.0000,loopbig,1]")
    bb(if5,[if7],"[16.0000,loopbig,1]")
    bb(if7,[if10,if9],"[100.000,loop2,1]")
    bb(if9,[if11],"[75.0000,loop2,1]")
    bb(iffy,[],"[1,main,1]")
    bb(loop2,[],"[16.0000,loopbig,100]")
    bb(loopbig,[],"[1,iffy,16]")

end odlprograminstance
```

Abbildung 6: `loops.prog.coxed`

- Die Speicherarchitektur. Beispielsweise haben viele eingebettete Systeme eine Trennung von Daten- und Programmspeicher, mit unterschiedlichen Zugriffszeiten.
- Ein/Ausgabe: Diverse Bussysteme haben ein völlig unterschiedliches Zeitverhalten.
- Das Betriebssystem beeinflusst die Laufzeit durch Unterbrechungen, Mehrprozesssysteme können durch Kontextwechsel auch unvorhergesehene Verzögerungen erzeugen (beispielsweise durch Zwischenspeicher-Zugriffsverfehlungen).
- Verschiedene Bibliotheken (z.B. `stdc`-Bibliothekimplementationen) haben verschieden schnelle Implementierungen der gleichen Funktionen. Hier kann man beispielsweise das Laufzeitverhalten schon vorberechnen und mit der Bibliothek mitliefern.

Unterbrechungen stellen ein besonderes Problem dar, da diese jederzeit im Programmablauf passieren können. Speziell im Bereich von harten Echtzeitsystemen werden diese allerdings genau wegen dieser Problematik nicht eingesetzt (bzw. bei kritischen Bereichen deaktiviert), daher is es die Auffassung des Autors, dass man diese bei der Berechnung der Laufzeit nicht zu berücksichtigen braucht.

4.2 Mikroarchitekturbezogene Modifikationen der Laufzeitanalyse

Bei einfach gebauten Prozessoren ist die Laufzeitanalyse ein aufaddieren der Laufzeiten der einzelnen Befehle des zu analysierenden Programms. Um die Verarbeitungsgeschwindigkeit zu steigern, wurden allerdings einige Mechanismen in moderne Prozessoren eingebaut, die den Vorgang verkomplizieren.

Hier drei Beispiele dazu:

- Viele Prozessoren können mehrere Operationen gleichzeitig ausführen (wenn diese bestimmten Einschränkungen entsprechen, z.B. wenn eine davon eine ganzzahlige und die andere eine Fließkommaberechnung ist). Da diese Operationszusammenfassung auch über Grundblöcke hinweg passieren kann, darf die niedrigere Analyse nicht nur für jeden Block speichern, wie lange die Ausführungszeit ist, sondern sie muss die Auswirkungen auf die Berechnungseinheiten des Prozessors beschreiben (das „Pipeline-Verhalten“).
- Ein weiterer Mechanismus ist der Zwischenspeicher, der einen wiederholenden Speicherzugriff beschleunigt. Zusammen mit heuristischen Algorithmen, die im Vorhinein Daten in den Zwischenspeicher laden, auf den Verdacht hinauf, dass sie eventuell später gebraucht werden könnten, kann dies

eine vollständig akkurate Laufzeitanalyse schwer bis komplett unmöglich machen. Hier muss ein Zwischenspeicher-Simulator eingesetzt werden [4]. Mehr zu diesem Thema kann man bei [16] finden.

- Eine Verzweigungsvorhersage-Funktion bewirkt, dass die Verarbeitungszeit zwischen zwei Grundblöcken nicht konstant ist, und muss auch einberechnet werden.

4.3 Methoden zur Laufzeitanalyse

Es gibt viele Ansätze, um die Laufzeit eines Programms zu berechnen. Manche davon limitieren allerdings das, was ein Programm machen darf (beispielsweise sind Schleifen in CHaRy [5] nicht erlaubt).

- Der naïve Ansatz ist, einfach sämtliche möglichen Pfade im Programm zu analysieren. Das Problem ist, dass dieser Ansatz bei den meisten Programmen zu einem exponentiellen Anstieg der komplexität führt, beispielsweise eine Bedingungsoperation in einer Schleife führt zum Ansteigen der Pfadzahl um 2^n.
- Mit der Einschränkung, dass ein Programm keine Sprungbefehle enthält, kann man einen abstrakten Syntaxbaum aufbauen, und diesen zur Zeitanalyse verwenden. Dieser Ansatz wurde vom MARS-Projekt der TU Wien verfolgt.
- Chapman [6] verwendet reguläre Ausdrücke, um Ausführpfade darzustellen. Am Schluss entsteht eine Formel, in die eingesetzt werden kann, um ein Laufzeitergebnis zu erhalten. Das Problem mit diesem Ansatz ist, dass er die in Kapitel 4.2 angesprochenen Analyseprobleme nicht berücksichtigen kann.
- Die implizite Pfadenumeration [7] entwickelt aus dem Programm ein Berechnungssystem aus linearen Einschränkungen, mit dem man mittels Maximieren oder Minimieren die Laufzeiten berechnen kann. Dieser Ansatz hat den Vorteil, dass keinerlei Einschränkungen im Bezug auf das Programm gemacht werden muss, aber trotzdem für übliche Programme ein sehr überschaubares System entwickelt wird. Dieser Ansatz lässt sich auch in die in Kapitel 2 vorgestellte Architektur leicht integrieren, und wird daher in [1] auch eingesetzt.

4.4 Verwandte Probleme

Das Bestreben, mehr Informationen ber ein Programm als den reinen Assembertext beizubehalten, ist nicht alleine der Laufzeitanalyse vorbehalten.

Beispielsweise muss ein Übersetzer für bestimmte Optimierungen, die z.B. Wissen über Feldabhängigkeiten benötigen, ebenfalls Ko-Transformationen einsetzen [8].

Ein weiteres großes Feld ist die Fehlersuche zur Laufzeit von optimierten Programmen. Hier muss ebenfalls bekannt sein, wo eine bestimme Variable zu finden ist, und welchem Quelltextabschnitt die aktuelle Assemblerzeile zugeordnet werden kann (für Haltepunkte beispielsweise). Diese Informationen sind oftmals sehr schwer zu erhalten (In [9] wird dieses Problem näher erläutert).

Allerdings werden bei der Fehlersuche nur die *statischen* Informationen benötigt, was noch nicht ausreichend für eine Laufzeitanalyse ist.

5 Benutzerschnittstelle

Eine Benutzerschnittstelle zu einem Laufzeitanalysewerkzeug muss dem Benutzer folgende Dinge bereitstellen:

- Der Benutzer muss Informationen über das Programm bereitstellen können, beispielsweise Beschränkungen der Eingangswerte. Dies kann auch direkt im Quelltext erfolgen.
- Eine Einstellungsmöglichkeit der Beschränkung der zu analysierenden Programmteile muss vornehmbar sein. Hier kommt wieder das in Kapitel 4.4 erwähnte Problem zum tragen, dass in optimierten Programmen einer Assemblerzeile oft keine bestimmte Quelltextzeile mehr zugeordnet werden kann.
- Darstellung des Endergebnisses. Im Idealfall würde man hier einfach dem Quelltext Zeile für Zeile die Laufzeit gegenüberstellen, was aber wie erwähnt im Allgemeinen nicht möglich ist. Eine Zeile Quelltext kann beispielsweise auf mehrere vervielfältigt werden, mit unterschiedlichen Laufzeiten. Wenn man allerdings einen Text bereitstellt, der dem ausgeführten Assemblerprogramm ähnlich ist, kann der Benutzer dessen Beziehung zum ursprünglichen Programm nicht mehr herstellen.

Eine Analyse eines weniger optimierten Programms wäre sinnlos, da dieses völlig andere Laufzeiten hat als das Endprodukt. Weitere Möglichkeiten, die Messer-

gebnisse zu quantifizieren sind:

- Grundblöcke: Dieser Ansatz ist auch nicht zweckmäßig, da diese Grundblöcke sich durch die Optimierung ändern können, und nicht sehr intuitiv sind (beispielsweise im Bezug auf das `?:`-Konstrukt in C).
- Funktionen: Moderne Optimierungen verschieben wie in Kapitel 3 schon erwähnt oft Funktionskörper direkt in die aufrufenden Funktionen, daher funktioniert dies im Allgemeinen auch nicht.
- Programm: Ein Ergebnis für das komplette Programm auf einmal könnte erzeugt werden. Dies ist leider nicht sehr detailliert[2], und birgt das Problem, dass Programme im eingebetteten Bereich üblicherweise eine Endlosschleife sind, d.h. die Laufzeitanalyse des Ganzen ist immer ident und wertlos. Wenn man das Programm dahingehend modifizieren würde, dass es terminiert, wäre die Analyse aber wieder nicht direkt am Endprodukt und damit auch nicht zielführend.

In [10] wird aufgezeigt, dass eigentlich korrekte Optimierungen des Übersetzers das Verhalten eines Programms beeinflussen können. Dies kann beispielsweise durch nicht abgefangene Feldüberläufe passieren, wo die nachfolgenden Speicherbereiche verändert werden. Um dieses Problem dem Programmentwickler darzustellen, sollte er visuell über die durchgeführten Optimierungen informiert werden können. Dieser Ansatz würde auch der Laufzeitanalyse zugute kommen.

6 Nicht gelöste Probleme

Mit der in diesem Artikel beschriebenen Technik lassen sich viele Optimierungen ausdrücken („loop peeling“, Entfernung von nicht aufgerufenen Programmteilen, etc.), andere allerdings nicht. Dies ist dadurch gegeben, dass die verwendete Programmiersprache ODL nur innerhalb von Funktionen operiert, daher ist unter anderem das Kopieren einer Funktion in eine andere nicht darstellbar, außerdem können Veränderungen innerhalb von Operationen nicht dargestellt werden.

Der Autor ist überzeugt, dass durch eine Erweiterung der Sprache die Ausdrücksstärke enorm gesteigert werden könnte.

[2] Das Ziel der Laufzeitanalyse ist ja, mögliche Flaschenhälse identifizieren zu können, was mit diesem Ansatz nicht mehr einfach möglich ist.

7 Verwandte Werke

[11] beschreibt die aktuelle Entwicklung bei der in dieser Arbeit vorgestellten Technik der Ko-Transformation.

[12] zeigt die Mikrorchitekturprobleme in Bezug auf RISC-Prozessoren.

[13] beschreibt die Berechnung der maximalen Ausfhrungszeit, und zeigt, wie man mittels neu eingeführter Sprachkonstrukte die dynamischen Laufzeitinformationen in ein Programm einbetten kann.

[14] zeigt eine Methode, wie automatisch aus einem C-Programm die benötigten dynamischen Laufzeitinformationen (z.B. wie oft eine Schleife ausgeführt wird) gewinnen kann, ohne Hilfe des Benutzers.

[15] präsentiert die Möglichkeit, einen evolutionären Algorithmus zu verwenden, um eine genauere Laufzeitanalyse durchführen zu können.

[16] untersucht die Laufzeitanalyse in Bezug auf objektorientierte Sprachen und mehrprozessfähige Betriebssysteme.

[17] untersucht Zwischenspeicher-Zugriffsverfehlungen und Prozessorleitungsstillstand genauer.

[18] geht einen völlig anderen Weg und stellt ein neues Programmierparadigma vor, das speziell auf eine einfache Laufzeitanalyse ausgerichtet ist.

[19] geht ebenfalls auf diese Materie ein und stellt die neuesten Entwicklungen vor.

[20] stell eine Methode vor, wie automatisch Testdaten generiert werden können, um die Laufzeitanalyse durchführen zu können. Hierbei soll eine genaue Kenntnis der Mikroarchitektur nicht mehr erforderlich sein.

[21] diskutiert, warum die aktuellen Entwicklungen der Laufzeitanalyse in der Wirtschaft keine Verwendung finden, und was dagegen getan werden könnte.

8 Zusammenfassung

Die Problematik der Laufzeitanalyse unter Berücksichtigung von Übersetzern ist dargestellt worden. Eine Technik namens Ko-Transformation ist vorgestellt wor-

den, die es ermöglicht, trotzdem noch realistische Vorhersagen zu treffen. Eine Optimierungsbeschreibungssprache namens ODL ist vorgestellt worden, die es ermöglicht, übersetzerunabhängig Optimierungen und deren Ko-Transformationen zu beschreiben. Die Problematik der Darstellung der Ergebnisse dem Benutzer gegenüber ist gezeigt worden.

Literatur

[1] Jakob Engblom, Andreas Ermedahl, and Peter Altenbernd. Facilitating Worst-Case Execution Time Analysis for Optimized Code. In Proc. 10th Euromicro Real-Time Workshop, Berlin, Deutschland, Juni 1998.

[2] Jakob Engblom. Worst-case execution time analysis for optimized code. Masters thesis, Department of Computer Systems, Uppsala University, Sept. 1997. DoCS MSc Thesis 97/94.

[3] L. Song, K.M. Kavi and R. Cytron. An unfolding-based loop optimization technique. Proceeding of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03), Springer Verlag Lecture Notes on Computer Science (LNCS), Wien, 24.-26. September 2003.

[4] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In Proceedings of the IEEE Real-Time Systems Symposium, Dezember 1995.

[5] Peter Altenbernd. CHaRy: The C-Lab Hard Real-Time System to Support Mechatronical Design. In Proceedings IEEE International Symposium and Workshop on Systems Engineering of Computer Based Systems. IEEE Computer Society Press, 1997.

[6] Roderick Chapman. Worst-case timing analysis via nding longest paths in SPARK Ada basic-path graphs. Technical Report YCS-94-246, Department of Computer Science, York University, Oktober 1994.

[7] Peter Puschner and Anton Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[8] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. Proceedings of the IEEE, 83(12):1625-1995, Dezember 1995.

[9] Richard Gerber. Languages and tools for real-time systems: Problems, solutions and opportunities. Technical Report UMD CS-TR-3362, UMIACS-TR-94-117, Department of Computer Science, University of Maryland, October 1994.

[10] Max Copperman. Debugging optimized code without being misled. Technical Report UCSC-CRL-92-01, University of California, Santa Cruz, Mai 1992.

[11] Raimund Kirner, Peter Puschner. Transformation of Meta-Information by Abstract Co-Interpretation. 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03). September 2003.

[12] Sung-Soo Lim, Young H. Bae, Gyu T. Jang, Byung-Do Rhee, Sang L. Min, Chang Y. Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. Software Engineering, 21(7):593-604, 1995.

[13] Peter Puschner, Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. Journal of Real-Time Systems, Volume 1, Number 2, pp. 159-176, September 1989.

[14] Andreas Ermedahl, Jan Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. 1997.

[15] Gro, H.-G.: Measuring Evolutionary Testability of Real-Time Software. PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, Juni 2000.

[16] Matteo Corti, Roberto Brega, Thomas Gross. Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems, Proceedings of the ACM SIGPLAN 2000 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2000), Juni 2000.

[17] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, Marion G. Harmon. Bounding Pipeline and Instruction Cache Performance. IEEE Transactions on Computers archive Volume 48 , Issue 1. Jänner 1999.

[18] Janosch Fauster, Raimund Kirner, Peter Puschner. Intelligent Editor for Writing Worst-Case-Execution-Time-Oriented Programs. 3rd International Conference on Embedded Software (EMSOFT 2003). Oktober 2003.

[19] Jan Gustafsson, Björn Lisper, Raimund Kirner, Peter Puschner. Code Analysis for Temporal Predictability. Journal of Real-Time Systems. 2005.

[20] Raimund Kirner, Peter Puschner, Ingomar Wenzel. Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. 4th Euromicro Workshop on WCET Analysis. 2004.

[21] Raimund Kirner, Peter Puschner. Discussion of Misconceptions about Worst-Case Execution-Time Analysis. 3rd Euromicro International Workshop on WCET Analysis. Juli 2003.