Laufzeitanalyse für eingebettete Programme basierend auf der Programm- und Architekturklassifikation

W. Ye, R. Ernst
Technische Universität Braunschweig
Institut für Datenverarbeitungsanlagen
Hans-Sommer-Str. 66
38106 Braunschweig
ye@ida.ing.tu-bs.de

1 Einleitung

Die Programmlaufzeitvorhersage ist ein wichtiges Problem im Entwurf eingebetteter Systeme. Sie ist auch für die Performance-Optimierung unter *first-time-right*-Systembedingung und für Verifikation eines Echtzeit-Systems notwendig. In "Hard-Real-Time-Systems" ist die Programmlaufzeit ein Teil der Systemfunktion. Falsche Laufzeitvorhersage kann zum katastrophalen Ausfall führen. Formale Laufzeitverifikation ist daher von äußerstem Interesse. Sie liefert nicht die einzelne Laufzeit, sondern obere und/oder untere Zeitgrenzen für die Programmausführung. Die Gründe dafür sind die datenabhängige Ausführungszeit der Programmanweisungen und die unvollständige Programmpfadanalyse, die im Prinzip ein bekanntes unentscheidbares Problem darstellt, sowie der Einfluß von Rechnerarchitekturen. Während die datenabhängige Programmlaufzeit, die zu unvermeidbaren Unterschieden zwischen unteren und oberen Zeitgrenzen führt, existiert, führen Ungenauigkeiten der Pfadanalyse und architektonisches Modellieren zu breiteren Zeitgrenzen als notwendig. Die Laufzeitgrenzen müssen immer konservativ sein, um die korrekte minimale und maximale Laufzeit des echten Entwurfs zu sichern. Das bedeutet, daß die obere Grenze höher als die echte Laufzeit und die untere Grenze niedriger als die echte Laufzeit sein muß. Deshalb sind ungenaue Grenzen teuer, weil Systeme überdimensional werden müssen, um die geforderten Zeiten einhalten zu können.

Ein weiterer Aspekt sind die Kosten der formalen Analyse. Ein Laufzeitanalysemodell muß alle möglichen Laufzeiten von Architekturen und Programmpfade berücksichtigen, während das Simulationsmodell einen von Eingaben abhängigen Pfad ausgewählt hat. Das ist offensichtlich sehr aufwendig. Einfaches Aufzählen von Assemblerbefehlen ist nicht ausreichend für komplexere Architekturen mit Pipelining oder Cache. Das bedeutet zusätzliche Kosten für die Analyse. Um diese Kosten zu vermeiden, kann die Analyse auf der Hochsprachenebene ausgeführt werden, z.B. ausgehend von C. Man muß dabei aber die Compileroptimierungen berücksichtigen. Deswegen sind die Compileroptimierungen eine weitere Quelle für Ungenauigkeiten. Heute verlassen sich die Entwerfer immer noch auf die Simulation und das Profiling in der Laufzeitanalyse. Das ist jedoch ein riskanter Ansatz.

Im folgenden Abschnitt werden wir zuerst das Problem der Laufzeitanalyse ausführlich untersuchen und die Quellen der Ungenauigkeiten in der Laufzeitanalyse identifizieren. Wir werden dann eine Klassifikation aus orthogonaler Architektur und Programmeigenschaften entwickeln, die möglichst alle praktischen Prozessorentwürfe abdeckt. Wir werden diese Eigenschaften ausnutzen, um individuelle Analyseverfahren, die für diese spezifische Klasse sehr exakt sind, jeder Klasse zuzuweisen. Während die individuellen Analyseverfahren selbst aus der Praxis oder aus der Literatur entnommen wurden, wird dieser Artikel zeigen, für welche Klassen sie sicher geeignet sind. Ein praktischer Entwurf enthält Teile mit mehreren möglicherweise verschiedenen Eigenschaften. Das Ziel dieses Artikels ist, die individuellen Verfahren zu kombinieren, um alle möglichen Prozessorarchitekturen abzudecken. Da das neue Verfahren die besten bekannten Methoden für jede Klasse kombiniert, hat es eine höhere Genauigkeit als vorausgehende Ansätze. Dieser Ansatz wird im System SYMTA benutzt. Am Ende dieses Artikels werden Ergebnisse für eine Vielfalt von Programmen präsentiert, die die sehr hohe Analysegenauigkeit des neuen Verfahrens belegen.

2 Problem der Laufzeitanalyse

2.1 Pfadanalyse

Für Pfadanalyseverfahren in [Pa92] [MaWo95] wird ein Programm in Grundblöcke[1] aufgeteilt. Funktionsaufrufe werden als einzelne Anweisungen betrachtet. Alle Programme können in Grundblöcke aufgeteilt werden. Die Programmstruktur wird als ein gerichteter Programmflußgraph mit Grundblöcken als Knoten dargestellt. Abbildung 1 zeigt ein Beispiel für einen Teil eines FFT-Algorithmus.

Der zweite Schritt der Pfadanalyse ist, die Ausführungszeit im schlechtesten oder besten Fall für jeden Grundblock zu bestimmen. Dann wird eine Analyse für den längsten oder kürzesten Pfad auf dem Programmflußgraphen eingesetzt, um die untere und obere Laufzeitgrenze zu identifizieren.

Dieser Prozeß liefert keine ausreichende Genauigkeit. Für eine praktische Laufzeitanalyse muß man alle möglichen Pfade[2]durch ein Programm identifizieren. Aber nicht alle Pfade im Programmflußgraph sind ausführbar, weil manche Pfade unter Berücksichtigung der Eingabebedingungen nicht ausgeführt werden können. Solche Pfade werden *falsche Programmpfade* genannt. Alle anderen Pfade werden *ausführbare Programmpfade* genannt. Schließlich definieren wir *potentielle Programmpfade*, die ausführbare Pfade oder falsche Pfade sein können.

Die Identifikation der falschen Pfade ist für Programme mit Schleifen erforderlich, weil Schleifeniterationen im Flußgraph zu einer unendlichen Zahl der potentiellen Pfa-

[1]Ein Grundblock ist ein Programmteil, der nur bei der ersten Anweisung hereinkommt und nur bei der letzten Anweisung herausgeht [AhSeUl87].

[2]Ein Programmpfad (oder Trace) ist ein Pfad in dem Flußgraph, der einer möglichen Reihenfolge von Grundblöcken während der Programmausführung entspricht, d.h. der Pfad führt aus dem ersten zum letzten Grundblock des Programms.

```
n = nn << 1;
j = 1;
for (i=1; i<n; i+=2) {
  if (j > i) {
     swap(data[i],data[j]);
     swap(data[i+1],data[j+1]);
  }
  m = n >> 1;
  while ((m >= 2)&&(j > m)) {
         j  -= m;
         m >>= 1;
  }
  j += m;
}
```

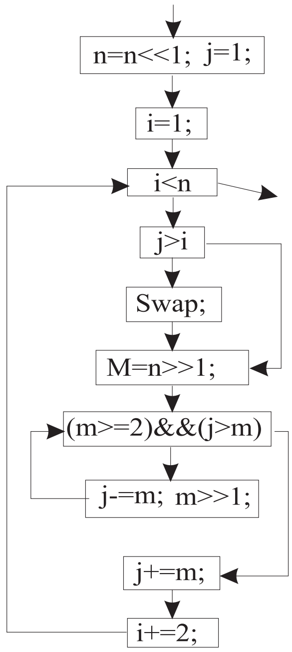

Abbildung 1: Beispiel eines Programmflußgraphen

de führen können. Die Ansätze von Mok [Mok89], Puschner und Koza [PuKo89], Park und Shaw [PaSh90] brauchen Iterationsgrenzen für alle Schleifen im Programm, die vom Benutzer durch eine Schleifeannotation spezifiziert werden müssen.

Für effektive formale Ansätze genügt die Schleifenbegrenzung allein nicht. Geschachtelte Schleifen sind oft voneinander abhängig, die Laufzeit einer Funktion ist abhängig von ihrem Aufruf und Bedingungen hängen voneinander ab. Diese Abhängigkeiten können ziemlich komplex wie das Beispiel in Abbildung 2 aus [MaWo95] sein. Hier wird Block x_5 höchstens einmal ausgeführt, was nicht von der Programmflußanalyse allein entdeckt würde. Der Benutzer muß deshalb falsche Pfade mit Annotation spezifizieren. Ein Problem dabei ist, daß die Zahl dieser falschen Pfaden sehr groß sein kann. Statt einer Aufzählung der falschen Pfade oder, umgekehrt, der ausführbaren Pfade, wird eine Sprache für die Benutzerannotation mit regulären Ausdrücken in [PaSh90] vorgestellt. Die Zahl der erforderlichen Pfadannotationen kann in der Praxis äußerst groß sein, wie schon mit kleinen Beispielen in [MaWo95] demonstriert wird.

Ein großer Fortschritt war die Einführung der indirekten Pfadaufzählung [LiMaWo95] [MaWo95]. Statt direkter Aufzählung der Pfade durch reguläre Ausdrücke liefert der Benutzer lineare Gleichungen und Ungleichungen, um falsche Programmpfade zu definieren. In Abbildung 2 kann man eine einfache Gleichung $x_5 \leq x_1$ spezifizieren. Sie bedeutet, daß jener Knoten x_5 höchstens so oft wie x_1 ausgeführt wird. Zur Auswertung dieser Gleichungen und Ungleichungen faßt [LiMaWo95] die Identifikation für die obere und untere Laufzeitgrenze zu zwei ILP-Optimierungsproblemen (integer linear programing) zusammen, eine für die kürzeste und andere für die längste Programmlaufzeit.

Alle diese Ansätze versuchen, eine einzige Analysemethode für das ganze Programm

```
/* k >= 0   */
s=k;
while (k<10) {
   if (OK)
      j++;
   else {
      j=0;
      OK=true;
   }
   k++;
}
r=j;
```

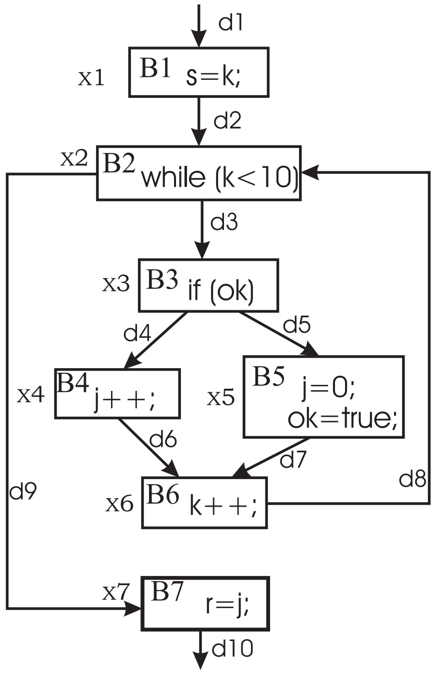

Abbildung 2: Beispielprogramm für die Pfadannotation

zu verwenden. Es ist aber auch möglich, bestimmte Programmeigenschaften auszunutzen, um die Genauigkeit zu erhöhen. Die erste Beobachtung ist, daß viele Programme in eingebetteten Systemen oder Teile solcher Programme einen einzelnen ausführbaren Programmpfad haben. Ein FIR-Filter ist ein einfaches, ein FFT ist ein komplexeres Beispiel. Mit anderen Worten: Das Programm hat nur einen Pfad, der für alle Eingabedaten ausgeführt wird, obwohl dieser Pfad über viele Schleifen, bedingte Anweisungen und selbst Funktionsaufrufe führt. Diese Programmeigenschaft wird im folgenden *single feasible path* (SFP-Eigenschaft) genannt. Die bekannten Analyseansätze in extremen Fällen liefern unterschiedliche Laufzeitgrenze für solche SFP-Programme, weil der Benutzer nicht alle Datenabhängigkeiten feststellen kann. Im besten Fall sind sie genau, aber sie erfordern einen intensiven Benutzereingriff. Im Gegensatz dazu kann eine Simulation den korrekten SFP-Pfad für beliebige Eingabedaten ohne weitere Wechselwirkung mit dem Benutzer wählen.

Offensichtlich enthalten viele praktische Systeme Nicht-SFP-Teile. Diese Eigenschaft der Segmente wird MFP-Eigenschaft (multiple feasible path) genannt. Interessanterweise zeigt eine Betrachtung eingebetteter Systeme, daß große Teile solcher Programme SFP-Eigenschaft besitzen. Deshalb lohnt es sich, die SFP-Eigenschaft auszunutzen, falls wir SFP- und MFP-Segmente isolieren können. Um unterschiedliche Methoden zu MFP- und SFP-Teilen zu verwenden, muß das Programm partitioniert werden. Die Programmpartitionierung wird später erklärt.

2.2 Datenflußanalyse

Die Datenflußanalyse ist relevant für die Programmlaufzeitanalyse, wenn die Datenzugriffszeit abhängig von Adressen ist. Die SFP-Eigenschaft überdeckt diesen Fall nicht. Aber wenn für ein bestimmtes Segment eines ausführbaren Programmpfads,

- alle Anweisungen immer auf die gleichen Daten zugreifen und der Feldindex unabhängig von den Eingabezuständen ist,

- und der Programmpfad-Teil die SFP-Eigenschaft besitzt,

dann ist die Datenzugriffsreihenfolge für diesen Teil eindeutig. Dies bezeichnen wir als *single data access sequence* (SDS-Eigenschaft). Beispiele für Programme mit SDS sind Filter oder FFT Algorithmen. Für alle anderen Fälle benutzen wir den Begriff *multiple data access sequences* (MDS-Eigenschaft).

2.3 Architektur

Eigenschaften der Architektur sind wichtig für die Laufzeit der Grundblöcke sowie des Pfadsegments. Uns interessieren Architektureigenschaften aus der Analyseperspektive. Es werden Zwei Methoden benutzt, um die Laufzeit der Grundblöcke und der Pfadsegemnts zu bestimmen.

1. Instruction-Timing-Addition (ITA)
   Die Ausführungszeiten der Befehle in einem Grundblock oder durch ein Pfadsegemnt werden zusammengezählt. Die Ausführungszeiten der Befehle wird aus einer Tabelle genommen. Der Vorteil dabei ist, daß ITA sehr kurze Berechnungszeiten erfordert.

2. Path-Segment-Simulation (PSS)
   Der Grundblock oder das Pfadsegment wird durch ein zyklengenaues Prozessormodell simuliert.

Wir unterscheiden in diesem Kontext mehrere orthogonale Architektureigenschaften:

- Datenabhängige Befehlsausführungszeiten
  Diese sind typisch für einige mikrocodierte CISC Architekturen. Beispiele sind Multiplikationsbefehle, die durch **Shift**- und **Add**- oder **Block-Move**-Befehle implementiert werden. Es kann in manchen Prozessorfamilien auch vorkommen, daß einige der Prozessoren nicht alle Befehle implementiert haben, wobei die nichtimplementierten Befehle durch Software-Funktionen mit entsprechenden datenabhängigen Ausführungszeiten emuliert werden. Die Ausführungszeiten der datenabhängigen Befehle führen zu datenabhängigen Ausführungszeiten der Grundblöcke. PSS kann hier nicht garantieren, daß die Laufzeitresultate im allgemeinen exakt sind, auch für Segmente mit SFP-Eigenschaft. ITA wäre dagegen geeignet.

- Architekturen mit Pipelining
  Pipelining macht ITA ungenau, weil es Pipelinekonflikt nicht berücksichtigt. Das Pipelining kann aber durch die Simulation genau modelliert werden. Deshalb ist

PSS genauer als ITA. Wenn die Pipeline tief genug ist, daß das Überlappen der Ausführung mehrerer Grundblöcke erlaubt ist, dann kann eine Simulation von einzelnen Grundblöcken allein nicht genügen, sondern nur eine Simulation von Pfaden. Dies ist der Fall für größere RISC-Prozessoren. Genauer gesagt muß ITA die durch Pipelining gewonnene Verarbeitungsleistungen vernachlässigen, weil es die Pipelineeffekte nur bedingt vorhersagen kann. Die Genauigkeit für PSS hängt von der Länge des simulierten Pfads ab. Um das System sicher zu gestalten, muß das Zeitverhalten am Anfang und am Ende eines Pfadsegments zum schlechtesten oder besten Fall angenommen werden. Das verursacht eine Ungenauigkeit am Anfang und am Ende des simulierten Pfadsegments. Also gilt, je länger die Pfade sind, desto präziser ist PSS.

- Superskalare Architekturen
  Superskalare Architekturen können mehrere Grundblöcke gleichzeitig ausführen. Sie verwenden dynamisches Befehlsscheduling mit Out-of-Order-Ausführung und Beendigung von Befehlen sowie spekulative Programmausführung [Joh91]. Dazu ist ITA nicht geeignet und PSS ist die korrekte Wahl. Die Genauigkeit hängt wieder von der Länge des simulierten Pfadsegments ab.

- Programmcache
  Das Verhalten von Programmcaches hängt von der Reihenfolge des Befehlsholens ab. Daher ist PSS genau für SFP-Programmsegmente (es ist angenommen, daß der Cache in der Simulation modelliert wird). ITA allein deckt die Cache-Verwendung nicht ab. Es gibt aber effiziente Erweiterungen zu ITA, die mindestens Direct-Mapped-Caches berücksichtigen können [MaWo95].

- Datencache
  Das Verhalten des Datencaches hängt von der Reihenfolge des Datenzugriffes ab. Deshalb ist PSS präzise für Pfadsegmente mit SDS. ITA hat die gleichen Probleme wie beim Programmcache.

Ein typischer Prozessor kann mehrere dieser Eigenschaften enthalten.

Der Vergleich zeigt, daß einerseits ITA allein nur für sehr einfache Architekturen geeignet ist und andererseits PSS ein Problem mit dem datenabhängigen Architekturverhalten hat. Das bedeutet, daß es keine einzige Methode gibt, die alle Fälle bedeckt.

3 Der SYMTA Ansatz

SYMTA (Symbolic hybrid Timing Analysis) ist ein hybride Ansatz, der Simulation und formale Analyse kombiniert und sich an verschiedene Architekturen und Programmeigenschaften anpaßt.

3.1 Hierarchisches Flußgraph-Clustering

Als erster Schritt wird das Eingabeprogramm durch einen hierarchischen Steuerflußgraph[3] (CFG) dargestellt. In diesem Graph entspricht jede Steuerkonstrukt, wie *if*,

[3] Wie in [MaWo95] nehmen wir an, daß strukturierte Programme keine *goto's* zwischen hierarchischen Knoten haben

case, *loop*, usw., einem Strukturknoten und die Blattknoten sind die Grundblöcke des Programms. Jedes der Steuerkonstrukte hat eine assoziierte Bedingung, die entscheidet, welcher Pfad im Konstrukt ausgeführt wird. Funktionen werden durch getrennte Graphen dargestellt und können für höhere Analysegenauigkeit in die Aufrufsstelle kopiert werden. Auf diese Weise kann eine MFP-Funktion mit einem aufrufabhängigen Steuerfluß zu einer SFP-Funktion werden.

3.2 SFP-Identifikation

Der zweite Schritt im Verfahren ist die Identifikation der SFP-Programmteile. In diesem Schritt werden wir die Knoten im Flußgraph in SFP- und MFP-Knoten partitionieren.

1. MFP setzt voraus, daß wenigstens ein Steuerkonstrukt von Eingabedaten abhängt. Wir können direkt schließen, daß jedes Pfadsegment, das keine datenabhängigen Steuerkonstrukte enthält, SFP sein muß.

2. Jetzt können wir eine einfache Induktion über die Hierarchieebene verwenden:

   - Blattknoten (Grundblöcken) sind SFP nach der Definition
   - Aus (1) $\Rightarrow$Ein hierarchischer Knoten
     - der nur SFP-Knoten enthält und
     - dessen entsprechende Bedingung unabhängig von Eingabedaten ist, muß SFP sein.

   Dies definiert eine einfache rekursive Clustering-Methode für eine Flußgraphpartitionierung. Sie zerteilt das Programm automatisch in SFP- und MFP-Pfadsegmente. Die FFT in Abbildung 1 wird ganz zu einem SFP geclustert.

   SFP-Clustering ist nicht genügend, wenn MFP-Pfadsegmente eingebettet sind. Abbildung 3a zeigt ein Beispiel. Das ist ein Algorithmus für Bubblesort. In diesem Programm sind die zwei Schleifen von Eingabedaten unabhängig, so daß sie immer die gleichen Iterationen haben, während die if-Bedingung von den Eingabedaten a[] abhängt.

   Wir erweitern den Clustering-Algorithmus, um die benachbarten SFP-Blöcke zu verschmelzen:

3. Wenn die Bedingung eines hierarchischen Knoten von den Eingabedaten abhängt, ist dieser Knoten MFP. Es werden Schnittpunkte am Anfang und am Ende des MFP-Knoten eingesetzt.

4. Wiederhole das Clustering nach (2), ignoriere den MFP-Knoten, aber berücksichtige die Schnittpunkte.

Die neuen in (4) gefundenen Cluster können auch als SFP betrachtet werden, weil es außerhalb des Segments zwischen den Schnittpunkten nur einen Pfad gibt. Für konservative Laufzeitanalyse is es daher ausreichend die MFP-Knoten separat zu analysieren und die ungünstigsten Laufzeitgrenze zwischen den Schnittpunkten anzunehmen. Bei

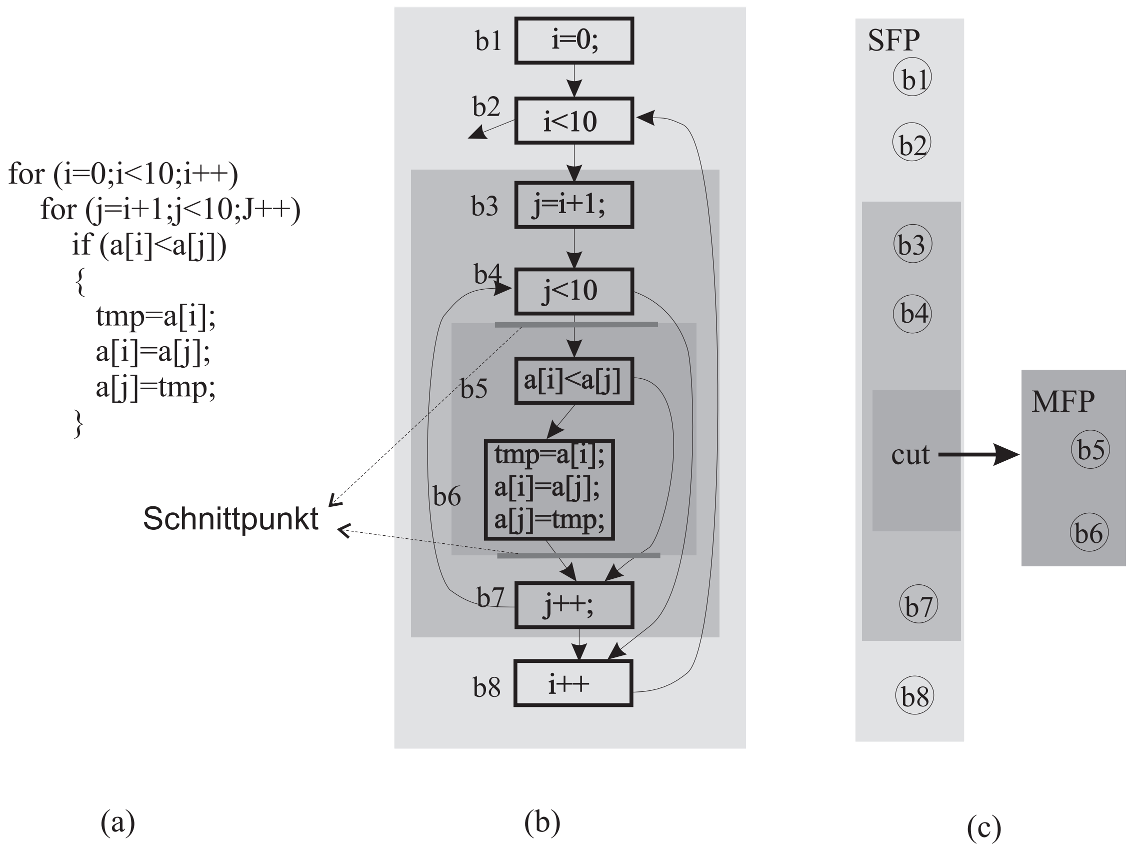

Abbildung 3: Beispielprogramm für SFP-Identifikation

diesem Vorgehen werden keine zusätzlichen falschen Pfade ausgeführt, (4) leitet die gesamte Pfadlänge mit ihren Laufzeitgrenzen ab.

Abbildung 3b zeigt das Ergebnis für das Beispiel in Abbildung 3a. Nur Block b_5, der Bedingungsbasisblock, und b_6 sind im MFP-Pfadsegment. Abbildung 3c zeigt, wie die übrigen Pfadsegmente zu SFP-Knoten geclustert werden. Dazu brauchen wir einen Algorithmus, um die Eingabedatenabhängigkeiten von den Bedingungen festzustellen. Dazu wird eine globale Datenflußanalyse [AhSeUl87] mit einer transitiven Hülle über alle Datenabhängigkeiten von Variablen im Steuerfluß benötigt. Eine globale Datenflußanalyse deckt aber typischerweise nicht Abhängigkeiten von Feldelementen und Operationen auf Daten ab. Deswegen wird die globale Datenflußanalyse durch eine symbolische Simulation ergänzt [YeEr95]. Als Nebeneffekt der symbolischen Simulation wird die SDS-Eigenschaft geprüft.

3.3 Globale Laufzeitanalyse

In der globalen Laufzeitanalyse für SFP wird jeder hierarchische Knoten zu einem einzelnen SFP-Knoten mit einer einzelnen Laufzeit verschmolzen. Alle verbleibenden hierarchischen Knoten sind MFP-Knoten und alle Blattknoten sind SFP-Knoten mit einer einzelnen gesamten Laufzeit abgesehen von datenabhängigen Befehlen und Cache-Verhalten, die noch im nächsten Abschnitt determiniert wird. Wenn die Laufzeit der Blattknoten bestimmt worden ist, verwenden wir den ILP-Ansatz, der in [MaWo95] präsentiert ist. Dies scheint der gegenwärtige mächtigste Ansatz für MFP-Segmente zu sein.

3.4 Anpassung der Laufzeit des Pfadsegments an Prozessorarchitekturen

Das nächste Problem ist die Bestimmung der Laufzeit der verbleibenden komplexen SFP-Knoten. Wie vorher gesehen, ist die Programmausführungszeit stark abhängig von den Architekturen. Bisher haben wir PSS und ITA zur Laufzeitanalyse der Grundblöcke und Segmente vorgestellt. Zum Bestimmung der Laufzeit der SFP-Knoten ist ITA für einfache Architekturen ohne Pipelining und datenabhängige Befehlslaufzeiten geeignet. Unabhängig von Pfad bekommt ITA stets die gleichen Laufzeitdaten eines Grundblocks aus einer Tabelle. Wir brauchen nun noch die Iterationszahl it_i des Grundblocks b_i, dann kann diese Iterationszahl mit der Grundblocklaufzeit t_i multipliziert werden. Hierbei ist it_i wegen der SFP-Eigenschaft eindeutig und kann deshalb durch ein Profiling der Quellencodes bestimmt werden. Das Profiling braucht keinen Simulator und kann direkt auf einer Workstation laufen.

Für den SFP des hierarchische Knotens P, der im reduzierten Flußgraph erscheint, können wir einfach zusammenfassen:

$$T = \sum_{i=0}^{N} it_i \times t_i.$$

Wenn ITA genau ist, ist es eine gute Wahl, weil es schneller als PSS ist. Ein gutes Beispiel für eine derartige Architektur ist der 8051 wegen seiner datenunabhägigen Befehlsausführungszeiten. Für datenabhängige Befehle liefert ITA eine untere und eine obere Grenze für jeden Grundblock, $t_{min}(b_i)$ und $t_{max}(b_i)$, deshalb werden eine untere und eine obere Grenze $t_{min}(P)$ oder $t_{max}(P)$ für die MFP-Analyse benutzt.

Pipeline- und Superskalar-Architekturen können exakt durch PSS behandelt werden. Beim Übergang zwischen SFP-Knoten muß der Extremfall angenommen werden, d.h. keine Verzögerung für die untere Grenze und maximale Verzögerung für die obere Grenze. Diese zusätzliche Zeit kann den SFP-Blöcken hinzugefügt oder in zusätzlichen Knoten gesammelt werden, wie z.B. in [YeEr93] vorgeschlagen.

Ein Problem sind Pipeling- und Superskalar-Architekturen mit datenabhängigen Befehlsausführungszeiten, weil weder ITA noch PSS direkt mit akzeptabler Genauigkeit anwendbar sind. Für solche Befehle nehmen wir gegenwärtig den Extremfall an. Allerdings geht es auch hier Lösungsansätze [ErYe96].

Alle anderen Kombinationen der Eigenschaften führen zu Teilproblemen der Architekturen im vorausgehenden Abschnitt. Damit ist Ansatz jetzt vollständig für Prozessorarchitektur und Programmeigenschaften.

4 Das Laufzeitanalysesystem SYMTA

Mit dem SYMTA-Ansatz wurde ein CAD-Tool entwickelt. Abbildung 4 zeigt den Flußgraph.

Der erste Schritt ist die symbolische Programmausführung und Datenflußanalyse. Die Analysenergebnisse werden für das Clustering benutzt. Der nächste Schritt ist

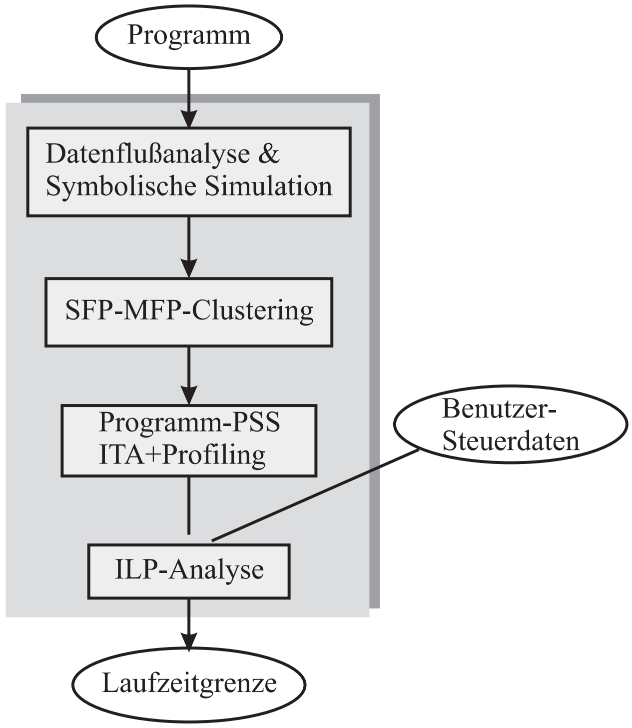

Abbildung 4: Der Flußgraph des SYMTA-Systems

entweder ein PSS des ganzen Programms oder ein Programmprofiling mit der ITA-Analyse.

PSS benutzt vom Benutzer gelieferte Eingabedaten für die Simulation, die alle Pfade im Programm abdecken sollen. Das Tool kann die nicht erfaßten Grundblöcke identifizieren und verwendet dann ITA. Das kann nur bei den im hierarchischen MFP-Knoten eingebetteten Grundblöcken vorkommen.

Wir haben zwei Prozessoren für unsere Experimente ausgewählt: einen komplexen Superscalar-Prozessor von SPARC mit 4-stufigem Pipelining und Floating-Point-Operationen und einen einfachen 8-bit 8051 Prozessor ohne Pipelining und datenabhängige Operationen, der oft in Mikrokontrollersystemen benutzt wird. PSS wird als primäre Analyse-Technik für den SPARC benutzt, während der 8051 durch ITA mit Profiling analysiert wird. Für den SPARC Prozessor benutzen wir den GNU C-Compiler und für den 8051 benutzen wir einen kommerziellen Compiler. Debugging-Informationen werden benutzt, die Grundblöcke im Assembler-Code zu identifizieren.

Die folgenden Laufzeitinformationen werden durch den PSS produziert:

- die Ausführungszeit des SFP-Knotens einschließlich der SFP-Cluster.

- die Ausführungszeit für datenabhängige Operationen im schlechtesten Fall und im besten Fall und die Ausführungsiterationszahl für Grundblöcke des Programms. Sie werden für den ILP-Solver später gebraucht.

Der letzte Schritt ist die ILP-Lösung auf dem reduzierten Programmflußgraph. Das System akzeptiert vom Benutzer gelieferte Gleichungen und Ungleichungen, um

Benchmark	Alle Knoten	Knoten in SFP		Knoten in MFP		Sourceline
3D-image	94	85	90%	9	10%	164
diesel	65	65	100%	0	0%	160
fft[EmBr91]	78	78	100%	0	0%	145
bsort	14	8	57%	6	43%	25
smooth	48	39	81%	9	19%	86
blue[Ho95]	80	53	66%	27	34%	127
check-data[Pa92]	18	0	0%	18	100%	44
whetstone[MaWo95]	12	2 122	100%	0	0%	251
line[MaWo95]	101	19	19%	82	81%	250
key3[Ho95]	100	100	100%	0	0%	151

Tabelle 1: Experimentelle Ergebnisse für das Clustering

die Genauigkeit für MFP-Segmente zu erhöhen.

Im nächsten Abschnitt werden wir einige experimentelle Ergebnisse vorstellen, die von SYMTA erzeugt werden.

5 Experimentelle Ergebnisse

Die erste Tabelle demonstriert die Cluster-Ergebnisse für eine Vielfalt von Algorithmen aus verschiedenen Quellen. Wir haben die SFP-Analyse ausgewertet, die in Tabelle 1 gezeigt wird. Die erste Spalte liefert die gesamte Zahl der Knoten in Programm-CFG. Die zweite Spalte enthält die Zahl der Knoten, die in SFP-Clusters gefunden werden. Die dritte Spalte enthält die Knoten in MFP-Segmenten. Die experimentellen Ergebnisse haben gezeigt, daß viele Teile der Programme die SFP-Eigenschaft haben und durch Simulation genau analysiert werden können.

Tabelle 2 zeigt die Laufzeitergebnisse in SYMTA. Um die Ergebnisse unseres Ansatzes auszuwerten, müssen wir die echten Grenzen der Programme wissen. Deshalb haben wir solche Programme ausgewählt, wo die Eingabedaten zum schlechtesten Fall durch einigen Aufwand klar identifiziert werden können. Die gemessenen Grenzen in der ersten Spalte werden durch die Simulation mit diesen Daten erhalten. Der zweite enthält die Laufzeitgrenzen der Beispielprogramme, die auf beiden Prozessoren durch unseren Ansatz analysiert wurden.

Schließlich haben wir unseren Ansatz, der die Simulation und die formale Analyse kombiniert, mit dem Ansatz in [LiMaWo95] verglichen. Das wird in Tabelle 3 gezeigt. Wir können die Ergebnisse für den i960 nicht generieren, weil wir für diesen Prozessor keinen geeigneten Simulator haben. Seine Eigenschaften sind aber ähnlich denen des SPARC Prozessors, den wir hier benutzt haben.

Programen	Simulierte Zeit(cycles)		Analysierte Zeit(cycles)		Analysezeit*
	BCET	WCET	BECT	WCET	(sec)
SPARC					
3D-image	34908	37848	33874	38037	0.79
diesel	62944	62994	61445	63333	0.84
fft	1498817	1499176	1494650	1499290	135.69
bsort	4423	8938	4416	8938	0.34
smooth	3635651	4846511	3570227	4881135	304.90
blue	3564938	316865761	3345041	346541760	4325.23
check-data	80	431	65	435	0.23
whetstone	2928459	3369459	2880230	3378098	298.19
line	514	1619	381	2035	0.39
8051					
fft	26421460	26421460	26419338	26488288	0.23
bsort	9347	15045	7804	18167	0.12
smooth	9737378	9737516	9737469	9737522	0.23
key3	1218229	1223314	1164883	1265227	0.39
check-data	68	559	63	588	0.17

** Die Beispielprogramme sind auf der SPARC 10 Workstation analysiert worden.*

Tabelle 2: Experimentelle Ergebnisse der Beispielprogramme in SYMTA

Programe	SYMTA-Verfahren SPARC		SYMTA-Verfahren 8051		ILP nur mit (Un)gleich. i960 [LiMaWo95]	
	WCET	Abweich.*	WCET	Abweich.	WCET	Abweich.
3D-image	38037	0.00				
diesel	63333	0.01				
fft	1499290	0.00	26488288	0.00	2.05e6	0.63
bsort	8938	0.00	18167	0.17		
smooth	4881135	0.01	9737522	0.00		
blue	346541760	0.09				
check-data	435	0.00	588	0.05	1039	1.36
whetstone	3378098	0.00			6.83e6	1.01
line	2035	0.23			4845	0.75

**Die Ungenauigkeit wird definiert wie $(T_a - T_m)/T_a$, wobei T_a analysierte Grenze ist, und T_m gemessene Grenze ist.*

Tabelle 3: Vergleich unseres Ansatzes mit anderen Ansätzen

6 Zusammenfassung

Wir haben einen Ansatz zur formalen Laufzeitanalyse und Verifikation mit engen Laufzeitgrenzen präsentiert. Der Ansatz kombiniert mehrere Analyse- und Simulation-Verfahren und nutzt dabei das jeweils beste Verfahren für die verschiedene Programm- und Architektureigenschaften aus. Datenflußanalyse und symbolische Ausführung werden benutzt, um die Analysetechnik mit möglichst hoher Analysegenauigkeit für jedes Programmsegment zu wählen. Der Ansatz wurde auf zwei sehr unterschiedliche Prozessor-Architekturen angewandt. Andere potentielle Anwendungen des Ansatzes sind Softwaretest und Laufzeitanalyse in der High-Level-Synthese.

Literatur

[AhSeUl87] A.V. Aho, R. Sethi, J.D. Ullman, *Compiler principles, Techniques and Tools,* Bell Telephone Laboratories, Inc. 1987.

[EmBr91] P.M. Embree, K. Bruce, *C Language Algorithms for Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Ho95] Ulrich Holtmann, *Synthese von Co-Processoren mit spekulativer Ausfhrung.* Ph. Thesis, Technical University of Braunschweig, 1995.

[Joh91] M. Johnson, *Superscalar Microprocessor Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[LiMaWo95] Y-T.S Li, S. Malik, A. Wolfe, *Performance Analysis of Embedded Software with Instruction Cache Modelling,* Proc. of ICCAD 1995. IEEE Society Press, pp. 380-387, 1993.

[MaWo95] S. Malik, W. Wolf, A. Wolfe, Y. S. Li, T. Yen, *Performance Analysis of Embedded Systems,* NATO ASI, Workshop on Hardware–Software Co-Design, Tremezzo, Italy, 1995.

[Mok89] A. Mok et al. *Evaluating Tight Execution Time Bounds of Programs by Annotations,* Proc. IEEE WS Real-Time Operating Systems and Software, May 89, pp. 74-80.

[Pa92] C. Y. Park, *Predicting Deterministic Execution Times of Real–Time Programs,* PhD Thesis, University of Washington, Seattle 98195, Aug. 1992.

[PaSh90] C.Y. Park, A.C. Shaw, *Experiments with a program timing tool based on source-level timing schema.* Proc 11th IEEE Real-Time system Symp., pp. 72-81,1990

[PuKo89] P. Puschner Ch. Koza, *Calculating the maximum execution time of real–time programs,* The Journal of Real–Time Systems, 1(2), 160–176, Sept. 1989.

[YeEr93] W. Ye, R. Ernst, Th. Benner, J. Henkel, *Fast Timing Analysis for Harware–Software Cosynthesis,* Proc. of ICCD 1993. IEEE Society Press, pp. 452-457, 1993

[YeEr95] W. Ye, R. Ernst, *Worst Case Timing Estimation based on Symbolic Execution,* COBRA report, Institute of Computer Engineering, Technical University Braunschweig, Oct. 1995.

[ErYe96] R. Ernst, W. Ye *Embedded Program Timing Analysis based on Program and Architecture Classification,* Tech. report COSYMA-06, Institute of Computer Engineering, Technical University Braunschweig, Oct, 1996.