Seminar
Analyse von Programmausführungszeiten

CAU Kiel, WS 2002/2003, Prof. Dr. R. v. Hanxleden

Vortrag zum Thema:

Low-Level-Analyse II: Pipelining

Vorgetragen am 9.12.2002 von Hendrik Janz

Überblick

Zwei Verfahren zur kombinierten Analyse von *Caching* und *Pipelining*

Jeweils:

- Allgemeines
- Schritte des Verfahrens
- Experimente und Ergebnisse
- Grenzen, Probleme

Verfahren von Healey et al.

Erlaubt die Bestimmung von *BCET (Best Case Execution Time)* und *WCET (Worst Case Execution Time)*

Es existiert eine graphische Benutzeroberfläche, Darstellung des Quellcodes, Maschinencodes und der Analyseergebnisse

Das Verfahren wurde im folgenden Artikel beschrieben:
Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, Marion G. Harmon; **Bounding Pipeline and Instruction Cache Performance**, *IEEE Transactions on Computers*, Vol. 48, No. 1, January 1999, pp. 53-70

Für viele Anwendungen reicht zwar die Bestimmung der WCET, dieses Verfahren ermöglicht aber auch die Bestimmung der BCET.

Die Benutzeroberfläche stellt Quellcode und Maschinencode sowie die Gruppierungen von Basisblöcken (Codesegmente deren Ausführung mit einem bestimmten Befehl beginnt und die erst am Ende mit einem (bedingten) Sprung wieder verlassen werden können) gegenüber. In jeder Darstellung kann man Bereiche selektieren und sich die zugehörigen Informationen anzeigen lassen.

Verfahren von Healey et al.
Schritte

Instruction Caching Analyse

Pipeline Path Analyse

Schleifen-Analyse

Programm-Analyse

Die verschiedenen Analysen sollten *aufeinander aufbauen* und somit *nacheinander durchführbar* sein.

Verfahren von Healey et al.
Instruction Caching Analyse

Statische Simulation

Klassifizierung in *alway/first hit/miss*

Unterschiedlich für *BCET* und *WCET*

Die hier verwendete *Statische Simulation* versucht, alle Zugriffe auf *Instruktionen* gemäß ihres Caching-Verhaltens zu kategorisieren. Das Verfahren beschäftigt sich nicht mit der Klassifizierung von Zugriffen auf *Daten* durch einen Cache.
Zunächst wird die *Kontrollfluss-Information* als Graph dargestellt. Für jede Funktion wird der Kontrollfluss zwischen den Basic Blocks beschrieben, dazu kommt ein *Instance-Graph*. Dieser beschreibt alle Möglichkeiten der Aufrufe von Funktionen, es handelt sich um einen gerichteten azyklischen Graphen, da keine Rekursionen erlaubt sind. Die Verwendung dieses Verfahrens verbietet auch indirekte Funktionsaufrufe.
Anschliessend wird mit Hilfe dieser Information für jeden Basic Block bestimmt, welche Codebereiche sich im Cache befinden können.
Daraus wird für jede Instruktion das Caching-Verhalten abgeleitet. Die Kategorien sind *alway hit*, *always miss* und *first hit* bzw. *first miss*. Die beiden letzten stehen für unterschiedliches Verhalten im ersten und den nachfolgenden Schleifendurchläufen. Ist das Verhalten nicht klar, wird es *entsprechend der gewünschten Abschätzung* (BCET oder WCET) eher als hit oder miss klassifiziert.

Verfahren von Healey et al.
Pipeline Path Analyse

Zunächst: Sammeln der Informationen zum Resourcenbedarf *separat für jeden Befehl*

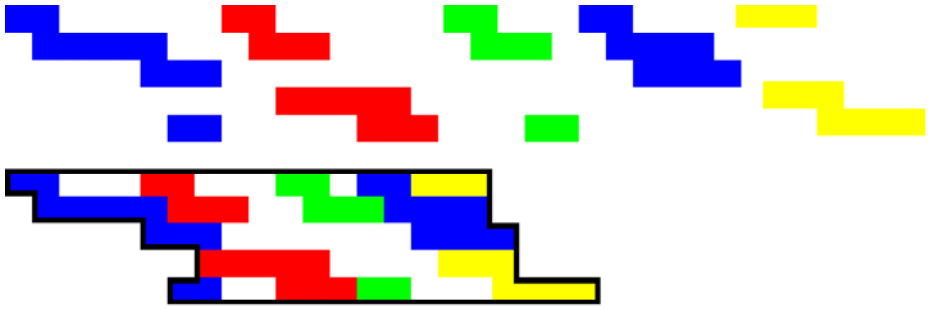
Dann: *Zusammenfassung von Basic Blocks*, neue Info hat genau die *gleiche Struktur*

Kontrollfluss bestimmt *mögliche Sequenzen* von solchen Blöcken

Für jeden Befehl wird die Information zum Resourcenbedarf zusammengetragen. Diese umfasst die Zeitpunkte, zu denen die einzelnen Stufen und Ausführungseinheiten benötigt werden. Es werden aber auch Datenabhängigkeiten berücksichtigt, die Ergebnisse aus der Caching-Analyse gehen ebenfalls ein, indem die Instruction-Fetch-Phase gegebenenfalls entsprechend verlängert wird. Neben der *Gesamtdauer* werden nur die *Zeitpunkte der ersten und letzten Benutzung jeder Resource*, vom Anfang bzw. Ende aus gesehen, benötigt.
Diese Informationen für die einzelnen Befehle werden nun kombiniert. Alle Befehle in einem Basic Block werden zusammengefasst. Die Art der Information bleibt die gleiche: Gesamtdauer und erste bzw. letzte Nutzung der jeweiligen Resourcen.

Verfahren von Healey et al.
Pipeline Path Analyse

Schema zur Kombination von Befehlen/Blöcken



Das Schema soll veranschaulichen, wie sich die Nutzung verschiedener Resourcen überlappen kann. Dabei kann eine Resource entweder eine Einheit der CPU (z.B. Befehlsdecodierer, Fließkommaeinheit, ...) aber auch ein bestimmtes Register oder ein Bussystem sein.
Der Block, der aus der Kombination der fünf Befehle entstanden ist, lässt sich mit den gleichen Mitteln beschreiben.

Verfahren von Healey et al.

Pipeline Path Analyse

SPARC Instructions

```
inst 1: faddd   %f2,%f0,%f2
inst 2: sub     %o4,%g1,%i2
inst 3: std     %f2,[ %o0+8]
```

Pipeline Diagram

		cycle															
		1	2	3	4	5	...	11	12	13	14	15	16	17	18	19	20	21	22
stage	IF	**1**	2	2	2	2	...	2	**3**										
	ID		**1**						2	**3**									
	EX									**2**	3	3	3	3	3	**3**			
	FEX			**1**	1	1	...	1	1	1	1	1	1	1	1	**1**			
	MEM										**2**						3	3	**3**
	WB											**2**							
	FWB																**1**		

Fig. 1. Example of overlapping pipeline stages with a cache miss.

STRUCTURAL HAZARD INFORMATION FOR THE INSTRUCTIONS IN FIG. 1

Stage	IF	ID	EX	FEX	MEM	WB	FWB
Beginning Inst	1	1	2	1	2	2	1
Cycles from Beg	**0**	**1**	**12**	**2**	**13**	**14**	**19**
Ending Inst	3	3	3	1	3	2	1
Cycles from End	**10**	**9**	**3**	**3**	**0**	7	**2**

DATA HAZARD INFORMATION FOR THE INSTRUCTIONS IN FIG. 1

Register	%g1	...	%o0	...	%o4	...	%i2	...	%f0	...	%f2	...
First Needed	**12**	**N/A**	**13**	**N/A**	**12**	**N/A**	**N/A**	**N/A**	**2**	**N/A**	**2**	**N/A**
Last Produced	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**9**	**N/A**	**N/A**	**N/A**	**3**	**N/A**

Das *Codestück* und das *Pipelinediagramm* in Fig. 1 zeigen drei Instruktionen für einen MicroSPARC I Prozessor. Die Zahlen im Pipelinediagramm entsprechen den Nummern der Befehle im Codestück. Sie zeigen, welcher Befehl zum jeweils oben angegebenen Zeitpunkt durch die entsprechende, auf der linken Seite angegebene Hardwareeinheit verarbeitet wird.
Der erste Befehl ist eine Floating-Point-Addition, die insgesamt 20 Zyklen benötigt. Beim Laden der zweiten Instruktion tritt ein Cache-Miss auf, der hier zu einer Verzögerung von neun Zyklen führen soll. Der dritte Befehl benötigt schliesslich das Ergebnis aus dem ersteh Befehl, daher muss die MEM-Stufe verzögert werden. Dadurch wird die Verzögerung durch den Cache-Miss in diesem Beispiel durch die Fliesskomma-Operation *vollständig überdeckt*, fällt also nicht ins Gewicht. Würde man Cache und Pipeline unabhängig voneinander betrachten, würde man davon ausgehen, dass durch den Cache-Miss eine Verzögerung von neun Zyklen entsteht und zu einer Vorhersage von 31 statt 22 Zyklen kommen. Dies zeigt die *Notwendigkeit einer kombinierten Analyse*.
Die Tabellen zeigen die *Informationen über Resourcenkonflikte* in der Form, in der sie vom vorgestellten Verfahren verwendet werden. Wichtig sind jedoch nur die fett gedruckten Zahlen.

Die oben gezeigten Diagramme und Tabellen sind aus dem zuvor angegebenen Artikel von Healey et al. entnommen.

Verfahren von Healey et al.
Schleifen-Analyse

Pfade: Kombination von Basic Blocks, reichen vom Beginn der Schleife bis zu einem möglichen Ende

Infos zu Pfaden einer Schleife werden aus Komplexitätsgründen zusammengefasst

Anzahl der Schleifendurchläufe ist wichtig. Bestimme diese automatisch oder durch Angabe durch Benutzer

In der *Schleifen-Analyse* werden *Basic Blocks* zu *Pfaden* kombiniert. Ein Pfad beginnt mit dem Schleifenkopf und endet an einer rückwärts gerichteten Kante oder dort, wo die Schleife verlassen wird.
Eigentlich müssten alle Kombinationen von Pfaden die bei mehrfachem Schleifendurchlauf genommen werden könnten, zur Bestimmung der Laufzeit herangezogen werden. Dadurch würde aber die Komplexität zu groß werden. Zur Vereinfachung werden die *Effekte der Pipeline für alle Pfade zusammengefasst*, d.h. die Informationen über erste und letzte Benutzung werden einheitlich angenommen.

Funktionen werden als Schleifen mit einmaligem Durchlauf betrachtet. Für andere Schleifen kann die Anzahl der Durchläufe teilweise automatisch bestimmt werden, ansonsten wird der Benutzer gebeten, eine Abschätzung anzugeben.

Verfahren von Healey et al.
Programm-Analyse

Konstruktion eines *Timing Analysis Tree*

Dieser wird *bottom-up* durchlaufen

Änderung der Cache-Klassifizierung in geschachtelten Schleifen

Die *Programm-Analyse* dient dem Zweck, Vorhersagen für Programmsegmente zu machen, die veschachtelte Schleifen und Funktionsaufrufe enthalten.
Zunächst wird ein *Timing Analysis Tree* konstruiert. Die Knoten stellen Funktionen oder Schleifen dar. Untergeordnete Knoten sind dementsprechend Schleifen, die in der übergeordneten Schleife aufgerufen werden. Der Wurzelknoten entspricht dem kompletten Hauptprogramm.
Zur Bestimmung der Laufzeiten wird der Baum *bottom-up* durchlaufen, es werden also zunächst für die kleinsten, tiefsten Schleifen die Ausführungszeiten bestimmt. Aus diesen Zeiten ergeben sich dann die Zeiten der übergeordneten Schleifen.
Im User-Interface kann im Prinzip die Information zu jedem dieser Knoten abgerufen werden.

Bei der Bestimmung der Laufzeiten muss teilweise die Cache-Klassifizierung angepasst werden. Ein als first miss klassifizierter Befehl in einer inneren Schleife muss häufig nur bei der ersten Ausführung der gesamten Schleife geladen werden, nicht aber, wenn diese Schleife erneut von der umgebenden Schleife aufgerufen wird.

Verfahren von Healey et al.
Experimente, Ergebnisse

Test an sechs Programmen aus unterschiedlichen Bereichen, SPARC-Architektur

Programme waren teilweise so konstruiert, dass Probleme provoziert wurden

Meist gute Ergebnisse, häufig exakte Voraussagen

Probleme bei verschachtelten Schleifen (sort) und bei Befehlen mit variabler Ausführungszeit

Kombinierte Cache-Pipeline Analyse ist besser (3%)

Die Testprogramme sind verhältnismässig einfach, entsprechen damit aber in ihrer Größe Programmbereichen, die in üblichen Echtzeitsystemen engen zeitlichen Anforderungen unterworfen sind. Es werden unterschiedliche Bereiche angesprochen: mathematisch, ganzzahlig vs. Fliesskomma, viel vs. wenig oder keine Entscheidungen, usw. Teilweise sind besondere Probleme eingebaut, wie z.B. extra Funktionsaufrufe oder die Provokation von Cache-Konflikten. Die Größe des Cache wurde auch stark eingeschränkt.
Die Vorhersagen sind im Vergleich zu primitiveren Verfahren sehr gut. Häufig werden *nahezu exakte Vorhersagen* gemacht. Bei einigen Programmen ergaben sich aber *Probleme mit geschachtelten Schleifen*, wenn die äußere Schleife die Durchläufe der inneren Schleife bestimmt. Ein anderes Problem sind Befehle, deren Ausführungszeit von den Datenwerten abhängen (z.B. Multiplikation).

Verfahren von Healey et al.
Grenzen, Probleme

Keine rekursiven Funktionsaufrufe erlaubt

Keine indirekten Aufrufe erlaubt

Daten-Cache wird nicht berücksichtigt

Noch keine Ergebnisse für andere Prozessoren

Das Verfahren erlaubt, wie schon zu Beginn erwähnt, *keine rekursiven oder indirketen Funktionsaufrufe*, da die Konstruktion des Graphen für die Cache-Analyse hierfür nicht möglich oder zu komplex ist.

Die Analyse des Caching-Verhaltens wurde *auf Befehle beschränkt*, ein *Datencache wurde nicht berücksichtigt*. Diese Möglichkeit soll aber in der weiteren Forschung der Autoren Berücksichtigung finden.

Es liegen noch keine Ergebnisse für andere Prozessoren vor, das Team arbeitet noch daran, möglichst viel konfigurierbar zu machen (*Retargetability*).

Verfahren von Ferdinand et al.

Nur Bestimmung der *WCET*

Es wurde Wert auf Modularität und Anpassbarkeit (*generic or generative approach*) gelegt

Abstract interpretation, abstract states, CRL

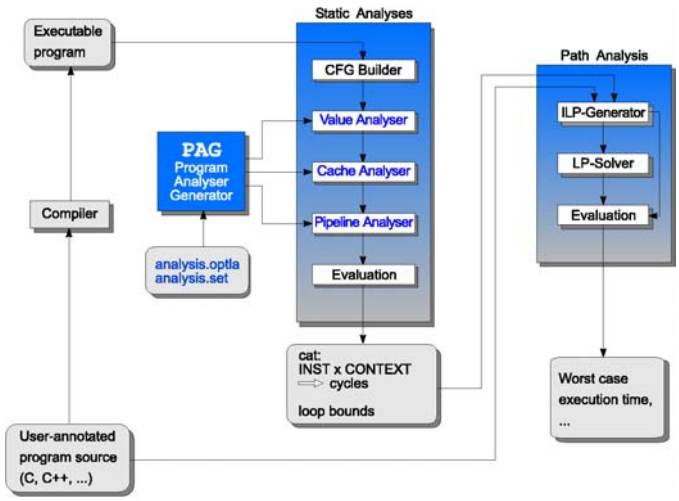
Berücksichtigt sowohl *Instruktions Cache* als auch *Daten Cache*

Das Verfahren wurde im folgenden Artikel vorgestellt:
Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, Reinhard Wilhelm; **Reliable and Precise WCET Determination for a Real-Life Processor,** *Proceedings of the Embedded Software First International Workshop (EMSOFT 2001)*, Tahoe City, CA, USA, October, 8-10, 2001. *Lecture Notes in Computer Science*, Volume 2211, Issue , pp 469-485

Bei der Beschreibung des Verfahrens wurde Wert auf *Modularität* und *universelle Verwendbarkeit* gelegt. Das Projekt wurde aber *konkret für den Motorola ColdFire* durchgeführt und umgesetzt.

Das Verfahren nutzt *Abstrakte Interpretation*, d.h. die konkreten Befehle werden durch neue Befehle ersetzt, die auf abstrakten Daten operieren. Dadurch ist es möglich, auch mit ungewissen Daten und Wertebereichen umzugehen. Die Abstract States in der Pipeline-Analyse beschreiben entsprechend mögliche Zusatände der Pipeline.
CRL (Control Flow Representation Language) wird als abstrakte Beschreibung des Kontrollflusses verwendet. Hierauf basiert die Werte-, Cache- und Pipeline-Analyse.

Verfahren von Ferdinand et al.
Schritte



14

Auch hier wünscht man, dass die Analyseschritte (*Werte-Analyse, Caching-Analyse, Pipeline-Analyse und Pfad-Analyse*) separat durchgeführt werden könnten. Es zeigt sich jedoch, dass die Caching- und die Pipeline-Analysen zusammengefasst werden müssen.

Die Abbildung ist dem zu Beginn genannten Artikel von Ferdinand et al. entnommen.

Verfahren von Ferdinand et al.
Werte-Analyse

Abstrakte Interpretation als Analysemethode

Berechnung von Adressbereichen für mögliche indirekte Speicherzugriffe

Ermöglicht in einigen Fällen auch die Erkennung nicht erreichbaren Codes

Mit Hilfe der *Abstrakten Interpretation* werden *mögliche Wertebereiche* für die Register bestimmt. Daraus ergeben sich Werte für Daten, aber insbesondere auch für Indexregister. Damit werden *Vorhersagen für Zugriffe* auf bestimmte Speicherbereiche und das damit verbundene *Caching* möglich.
Bei Verzweigungen kann man aus den möglichen Werten auch vorhersagen, dass bestimmte Pfade nicht ausgeführt werden. Leider heisst das nicht unbedingt, dass der entsprechende Code nicht in den Cache geladen werden kann. Eventuell kann durch *falsche Branch-Prediction* dieser Bereich trotzdem geladen werden.

Verfahren von Ferdinand et al.

Werte-Analyse: Abstrakte Interpretation

Operator $\sqcup$ zum Verbinden von zwei abstrakten Werten (Register oder Speicher):

$$\sqcup : D_{abs} \times D_{abs} \to D_{abs},$$
$$[l_1, u_1] \sqcup [l_2, u_2] := [min(l_1, l_2), max(u_1, u_2)]$$

Befehl *add* unter Berücksichtigung eines möglichen Overflows:

$$add: D_{abs} \times D_{abs} \to D_{abs},$$
$$[l_1, u_1] \quad [l_2, u_2] := \begin{cases} [l_1 \quad l_2, u_1 \quad u_2] & \text{falls kein Überlauf möglich} \\ undef. & \text{sonst} \end{cases}$$

16

Die abstrakten Befehle arbeiten mit *Intervallen oder Bereichsangaben* für die Werte. Die Operation zum Verbinden von Werten wird dort benötigt, wo verschiedene Wege des Kontrollflusses wieder vereinigt werden, z.B. nach einer switch- oder einer if-then-else-Anweisung. Um die Komplexität nicht unnötig zu erhöhen, wird immer ein zusammen-hängendes Intervall angegeben, selbst wenn sicher wäre, dass ein Bereich in der Mitte nicht möglich ist.
Beim Beispiel *add* wurde bei einem Überlauf *undef.* erzeugt. Es ist aber auch denkbar, hier den gesamten Wertebereich anzugeben (z.B. 0-255 für ein 8-Bit-Register). Eventuell könnte sogar ein bestimmtes Intervall angegeben werden, z.B. wenn immer ein Überlauf zustande kommt. Beim Überlauf wird häufig der Übertrag einfach abgeschnitten. Es entsteht also kein undefinierter Wert, was berücksichtigt werden kann.

Verfahren von Ferdinand et al.
Caching-Analyse

Must- und May- Analysen

Verschiedene Typen von Caches werden unterstützt

Benutzt die Ergebnisse aus der Werte-Analyse zur Klassifizierung

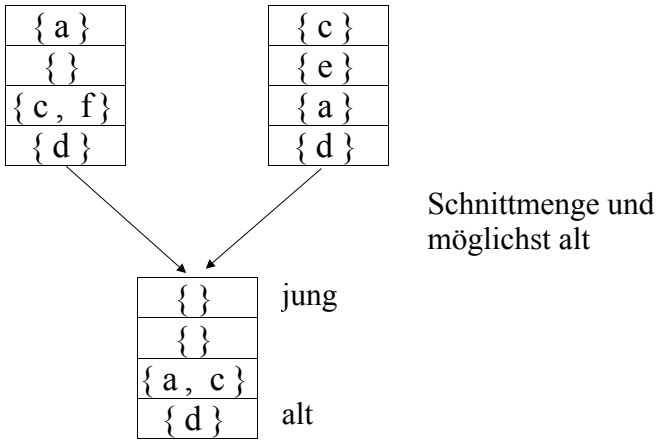
Der Ansatz beschreibt eine Unterteilung in *always hit/miss* und *unbekannt*

Schleifenrumpf wird einmal vor die Schleife kopiert, Schleife einmal weniger durchlaufen

In der *Must-Analyse* werden die Speicherblöcke bestimmt, die zu einem gegebenen Zeitpunkt *sicher im Cache* sind. Ein Zugriff ist also *immer ein hit*. Die *May-Analyse* bestimmt die Blöcke, die sich *im Cache befinden können*. Daraus können die Blöcke bestimmt werden, die sich *nicht im Cache befinden können*, die also beim Zugriff *sicher einen miss* erzeugen. Alle übrigen Blöcke werden als *unbekannt* klassifiziert. Durch das einmalige *„Auffalten“* der Schleifen unterscheidet sich diese Klassifizierung nicht grundsätzlich von der Einteilung in *first/always hit/miss*. Der erste Schleifendurchlauf, der in einem eigenen Kontext steht kann z.B. als hit und die folgenden als miss klassifiziert werden, das entspricht dann einem first hit. Die unbekannt klassifizierten Blöcke werden entsprechend der durchzuführenden Abschätzung eingeordnet, bei der hier gewünschten WCET-Analyse also als miss. Im Prinzip ist das Verfahren für beliebige Cache-Typen anwendbar, für *direct mapped* (Jedem Speicherblock ist eine bestimmte Stelle im Cache zugeordnet.) und *set-associative caches* (Ein Speicherblock kann in einer Menge von Cache-Plätzen abgelegt werden) existieren genauere Beschreibungen und Implementierungen. In der Beschreibung des weiteren Algorithmus wurde von einem *fully associative cache* (Jeder Speicherblock kann an eine beliebige Stelle im Cache abgelegt werden, also an einer noch freien Stelle oder dort wo ein lange nicht benötigter Block liegt) ausgegangen, es wurden also keine besonderen Annahmen mehr getroffen.

Verfahren von Ferdinand et al.

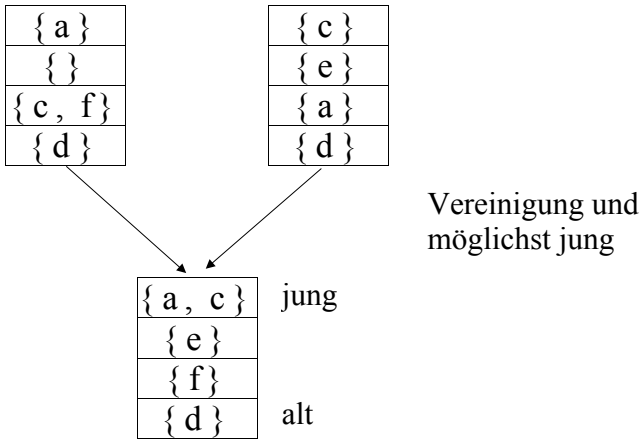
Caching-Analyse: Must-Analyse



In der *Must-Analyse* wird die Menge der Speicherbereiche bestimmt, die sich an einem gegebenen Programmpunkt auf jeden Fall im Cache befinden. Zusätzlich wird ermittelt, wie lange sich der jeweilige Speicherblock im Cache befindet. Je größer diese Menge ist, desto mehr läßt sich daraus ableiten, desto mehr Befehle lassen sich in dem entsprechenden Kontext als *always hit* klassifizieren.
Werden zwei Mengen *kombiniert* (beim Zusammenführen des Kontrollflusses), bleiben nur die Bereiche übrig, die in *beiden Ausgangsmengen enthalten* waren. Unterscheiden sich die Altersangaben eines Bereiches in den beiden Mengen, so wird die *ältere Angabe übernommen*. Es wird so sichergestellt, dass dieser Block bei einem Konflikt nicht zu lange als sicher angenommen wird. Zumindest gilt dies bei einem *fully-associative* Cache. Aber auch bei *direct-mapped Caches mit parallelen Lines* könnte dies gelten (Wie beim direct mapped cache bestimmt der höherwertige Teil der Adresse den Platz im Cache, es können aber jeweils vier Blöcke mit gleichem Index gehalten werden. Das Verhalten ist also eigentlich auch set-associative). Beim *ColdFire* kann jedoch nicht gesagt werden, welche Line betroffen ist, daher muss der Cache behandelt werden, als hätte er nur eine Line, die Information über das Alter ist dann nicht mehr wichtig.

Verfahren von Ferdinand et al.

Caching-Analyse: May-Analyse



In der *May-Analyse* wird die Menge der Speicherbereiche bestimmt, die sich an einem gegebenen Programmpunkt möglicherweise im Cache befinden. Es wird ebenfalls ermittelt, wie lange sich der jeweilige Speicherblock im Cache befindet. Je kleiner diese Menge ist, desto mehr läßt sich daraus ableiten, da die *komplementäre Menge* angibt, welche Bereiche auf keinen Fall im Speicher sind. Können sich nur wenige Speicherblöcke im Cache befinden, lassen sich viele Bereiche in dem betrachteten Kontext als *always miss* klassifizieren.
Werden zwei Mengen *kombiniert* (beim Zusammenführen des Kontrollflusses), kann jeder Bereich im Cache sein, der *in mindestens einer der beiden Ausgangsmengen enthalten* war. Unterscheiden sich die Altersangaben eines Bereiches in den beiden Mengen, so wird die *jüngere Angabe* übernommen. Es wird so sichergestellt, dass dieser Block nicht zu früh als sicher entfernt angenommen wird, wenn Blöcke wegen Überfüllung des Caches wieder entfernt werden müssen. Beim einfachen direct-mapped Cache ist diese Information - wie bei der Must-Analyse - nicht von Bedeutung.

Verfahren von Ferdinand et al.
Pipeline-Analyse

Komplexe Pipelines häufig schlecht dokumentiert

Starke Vereinfachungen, pessimistische Annahmen für WCET-Analyse

Häufig *starke gegenseitige Beeinflussung* von Cache und Pipeline, daher auch *Zusammenlegung der Analysen* denkbar

Häufig sind komplexe Pipelines vom Hersteller *schlecht dokumentiert*. Für die hier betrachtete WCET-Analyse muss daher zunächst ein *Modell* der Pipeline gebildet werden, das *konservativ* ist, also *pessimistische Annahmen* über die Überlappung der Befehle in der Pipeline macht und damit eher schlechtere Laufzeit-Prognosen liefert. Um die Zustände der Pipeline zu vereinfachen, müssen oft weitere, das Ergebnis verschlechternde Annahmen gemacht werden.
Auch für die Analyse der Pipeline wird die *Abstrakte Interpretation* verwendet. Aus den konkreten möglichen Zuständen der Pipeline werden abstrakte Zustände abgeleitet. Diese enthalten im wesentlichen die in einem Zustand benötigten Resourcen, aber auch eine Beschreibung der anstehenden Befehle sowie die Klassifikation der Speicherzugriffe. Die *Updatefunktion* beschreibt die *Veränderungen des Zustands nach einem Schritt*. An Punkten, an denen der Kontrollfluss einen gemeinsamen Punkt erreicht, müssen die abstrakten Zustände vereinigt werden, das geschieht als Bildung der Vereinigungsmenge.

Bei komplexen Konfigurationen ist der Inhalt des Caches häufig vom Verhalten der Pipeline abhängig und natürlich auch umgekehrt. Um möglichst genaue Ergebnisse zu bekommen, kann es sinnvoll sein, Cache- und Pipeline-Analyse zu verbinden. Allerdings steigt hierdurch die Komplexität des Verfahrens stark an.

Verfahren von Ferdinand et al.
Pipeline-Analyse: Pipeline des ColdFire

Fetch-Pipeline:

- Befehle aus Speicher laden, teilweise dekodieren
- *Branch Prediction* ⇒ Umlenken des Ladens bei unbedingten Sprüngen und bei Rücksprüngen

Befehlspuffer: FIFO für acht komplette Befehle

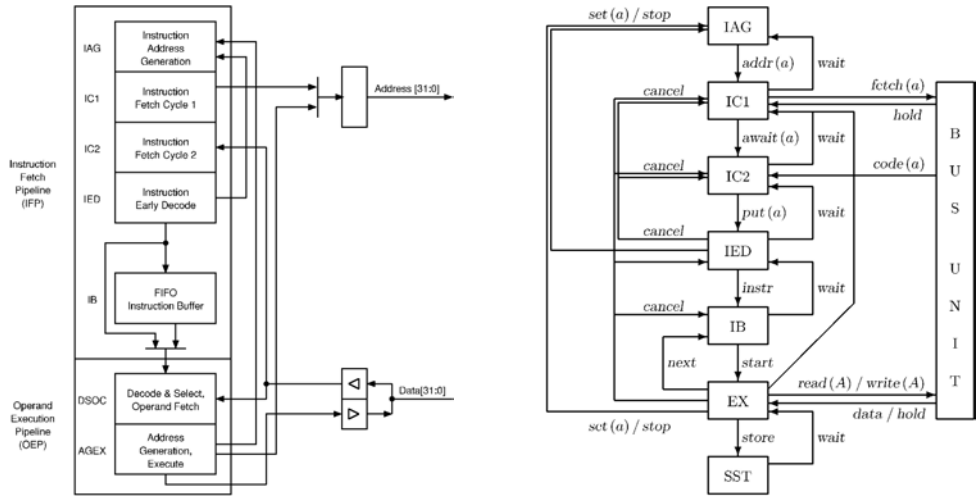
Zweistufige Execution-Pipeline:

- Befehl aus Puffer holen und dekodieren
- Adressgenerierung und Ausführung

Die *Fetch-Pipeline* lädt Befehle aus dem Speicher und decodiert sie soweit, dass Sprungbefehle erkannt werden. Anschliessend werden die Befehle an den *Befehlsbuffer*, der bis zu acht Befehle aufnehmen kann, weitergegeben. Die *Execution-Pipeline* holt dann die Befehle aus dem Puffer und führt sie aus. Die Execution-Pipeline ist zweistufig. In der ersten Stufe werden die Befehle decodiert und Registeroperanden werden geladen. In der zweiten Stufe erfolgt die Adressgenerierung, der Befehl wird ausgeführt, das Schreiben der Ergebnisse erfolgt ebenfalls hier.
Die Fetch-Pipeline führt zudem eine *Vorhersage für Sprünge* durch. Bei unbedingten Sprüngen wird das Laden sofort am Sprungziel fortgesetzt. Das Laden der Befehle, die dem Sprungbefehl folgen, wird abgebrochen. Auch bei *bedingten Sprüngen, die rückwärts gerichtet* sind, wird das Laden am Sprungziel fortgesetzt, da diese Sprünge *typisch für (mehrfach durchlaufene) Schleifen* sind und damit der Programmfluss an diesen Stellen häufiger zurück als linear weiter geht.
Es kommt so natürlich zu *falschen Vorhersagen*. Wird ein bedingter Sprung anders als vorhergesagt ausgeführt, muss der Puffer geleert und neu gefüllt werden.
Damit hat das Verhalten der Pipelines einen großen Einfluss auf das Caching. Vom Inhalt des Caches hängt aber ab, wie viele Befehle schon in den Puffer geladen werden konnten. Es besteht also eine starke gegenseitige Abhängigkeit. Daher ist eine kombinierte Analyse notwendig, das Verhalten kann dann aber präzise beschrieben werden.

Verfahren von Ferdinand et al.

Pipeline-Analyse: Pipeline des ColdFire



In dem *Modell für die kombinierte Cache- und Pipeline Analyse* haben die einzelnen Stufen der Pipeline jeweils ihre eigenen *inneren Zustände* und kommunizieren über *Signale* miteinander. Bei den Signalen wird zwischen *immediate* und *delayed* unterschieden.
Die Implementierung unterscheidet sich etwas von der tatsächlich vorhandenen Hardware. Die Umsetzung der Fetch Pipeline ist eng an der Hardware orientiert. Die Execution Pipeline ist aber anders modelliert. Die Umsetzung für das Verfahren hat nur eine statt zwei Stufen, es gibt jedoch eine zusätzliche Einheit, die Verzögerungen bei schnell aufeinander folgenden Schreibzugriffen realisiert.

Die Abbildungen sind dem zu Beginn genannten Artikel von Ferdinand et al. entnommen.

Verfahren von Ferdinand et al.
Pfad-Analyse

Hier wird der *Integer Linear Programming* Ansatz verwendet

Universeller Ansatz, wie zu Beginn gefordert

Die Pfad-Analyse wird im vorgestellten Verfahren mit Hilfe des *Integer Linear Programming* durchgeführt. Dieses Vorgehen genügt dem Anspruch der Autoren nach einem universellen Vorgehen, da in diesem Schritt von speziellen Eigenschaften der Zielarchitektur abgesehen werden kann. Das ILP wurde im Vortrag von Ingo Schiller ausführlich vorgestellt.

Verfahren von Ferdinand et al.
Experimente, Ergebnisse

Experimente basieren auf speziellem real-time Benchmark von AIRBUS, entspricht realen Anwendungen

Prozessor war ein Motorola ColdFire

Ergebnisse waren zufriedenstellend, evtl. wird das Verfahren bald zur Validierung herangezogen

Die Experimente beziehen sich auf den ColdFire, dessen Architektur hier auch vorgestellt wurde. Die verwendeten Testprogramme stammen aus einem speziellen Benchmark von AIRBUS. Es geht um die Überprüfung der Einhaltung von kritischen zeitlichen Beschränkungen aller zwölf (voneinander unabhängigen) Teilaufgaben. Diese Aufgaben entsprechen in ihrem Verhalten tatsächlich auftretenden Steuerungs- und Kontrollaufgaben.

Die Untersuchungen der Werteanalyse zeigten, dass sich bei *90% bis 95%* der Speicherzugriffe die Zugriffsadresse *exakt* bestimmen lässt. Etwa bei weiteren 5% ließ sich die Adresse auf einen Bereich von maximal 256 Bytes einschränken. Die Caching- und Pipeline-Analyse wurde in zwei verschiedenen Speicherkonfigurationen durchgeführt (alles cachebar bzw. Unterteilung in Speicherbereiche mit unterschiedlichem Caching-Verhalten).

Die Autoren waren mit den Ergebnissen zufrieden, genaue Angaben über die Güte wurden nicht gemacht. Auch Verantwortliche bei AIRBUS beurteilten die Ergebnisse positiv. Daher wird geprüft, ob dieses Verfahren als Grundlage für die zeitliche Validierung eingesetzt werden soll.

Verfahren von Ferdinand et al.
Grenzen, Probleme

Am Beispiel des ColdFire wird klar, dass Cache-Analyse sehr problematisch sein kann, besonders für BCET-Analyse

Modularität wurde teilweise wieder aufgegeben: *kombinierte Cache-Pipeline-Analyse*

Es wurden wichtige Aspekte, besonders zur Beurteilung der BCET, angesprochen, die auch für das andere Verfahren wichtig sein könnten. Das Verhalten des Caches im ColdFire für den Optimalfall ist aufgrund der vier parallelen Blöcke nur schwer einzuschätzen. Die Kombination von Cache- und Pipelineanalyse aufgrund der gegenseitigen Beeinflussung ist ebenfalls allgemein von Bedeutung.

Zusammenfassung

Die vorgestellten Verfahren dienen zur Analyse des Laufzeitverhaltens unter Berücksichtigung von *Caches und Pipelines*.

Das Verfahren von Healey et al. basiert im wesentlichen auf *Statischer Simulation* und *Kontrollflussgraphen*.

Das Verfahren von Ferdinand et al. benutzt die *Abstrakte Interpretation* für die ersten Teile der Analyse, für die Pfad-Analyse wird *ILP* verwendet.

Zusammenfassung

Während das Verfahren von Healey et al. nur den *Instruktions-Cache* berücksichtigt, werden im Verfahren von Ferdinand et al. sowohl *Instruktions-* als auch *Daten-Cache* einbezogen.

Auch sind bei Healey et al. *keine rekursiven oder indirekten Funktionsaufrufe* erlaubt.

Das Verfahren von Healey et al. erlaubt die Bestimmung von *BCET und WCET*, Ferdinand et al. bestimmen nur die *WCET*.