Seminar

Analyse von Programmausführungszeiten

CAU Kiel, WS 2002/03 Prof. Dr. R. v. Hanxleden

Vortrag zum Thema:

# Hochsprachenspezifische WCET Analyse (Java)

Vorgetragen am 20.01.2003 von Holger Labenda

# Überblick

- Einleitung
- Übertragbare WCET Analyse per Java Byte Code
- High Level Analyse
- Plattformabhängige Analyse(VM Timing Model)
- Zusammenfassung

# Einleitung

- Was ist eine WCET Analyse?
- Aufteilung der Analyse in einen Hardwareunabhängigen und Hardwareabhängigen Teil
- Anforderungen an eine portierbare Analyse per JBC

## Was ist eine WCET Analyse?

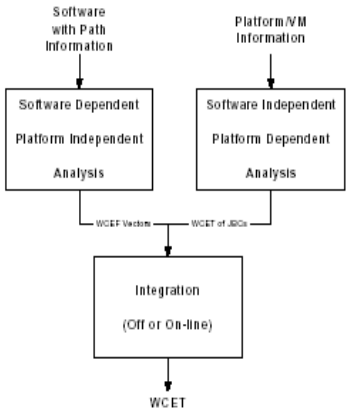
- Erstellen eines Graphen der Basis Blöcke.
- Die Timing Informationen der Basis Blöcke ergeben sich durch summieren der WCET der Instruktionen
- Diese Informationen können dann benutzt werden um ein Timingschema zu erstellen (eine Menge von Regeln um den Kontrollfluss-Graphen auszuwerten)

Man könnte auch einfach die Zeit messen, die ein Programm benötigt um ausgeführt zu werden, allerdings wäre dies höchst unsicher, da es oftmals unmöglich ist zu sagen, welche Eingabewerte wirklich zu einer WCET führen.

# Was ist eine WCET-Analyse (fortgesetzt)

- Sei WCET(S) die WCET eines Codesegements S und wir nehmen an wir haben die WCET der Basis Blöcke, dann erhält man durch folgende Regeln die gesamte WCET:
- WCET(S1;S2) := WCET(S1) + WCET(S2)
- WCET(if E then S1; else S2) := WCET(E) + max(WCET(S1),WCET(S2)
  - E ist der bedingte Ausdruck
- WCET(for (E) S;) := (n+1)WCET(E) + nWCET(S)
  - E ist der Schleifenausdruck und n die maximale Anzahl der Schleifendurchläufe

# Aufteilung der Analyse




Da Portabilität das Hauptziel dieser Analysemethode ist, ist eine Auftrennung der Analyse in eine Softwareabhängige und eine Hardwareabhängige Analyse sinnvoll. Auch bei einer portierbaren Analyse muss natürlich ein Teil der Analyse sich mit der Hardware beschäftigen auf der das Programm letztendlich ausgeführt wird. In unserem Fall nimmt sie die Form eines VMTM(Virtual Machine Timing Model) an, welches in der einfachsten Form eine Beschreibung der WCET der einzelnen JBC Instruktionen und der nativen Methoden ist.

## Anforderungen an eine portierbare Analyse per JBC

- Notwendige semantische Informationen müssen vom Sourcecode in den JBC übertragen werden
- Manipulation des JBC so, dass eine Analyse vorgenommen werden kann
- Hardwarespezifikationen der Zielplattform müssen ebenfalls übertragen werden.

Der JBC ist der Code, der von dem jeweiligen (Java)Compiler erzeugt wird. Er wird in das java-class file abgelegt. In dem Classfile gehen aber auch weitere Informationen ein, wie z.B. die Definitionen von Konstanten und Methoden. Die sogenannte "Virtual Machine" ist die Software, die dann die eigentliche Ausführung des JBC vornimmt.
Die JVM selbst ist stackbasiert und benutzt anstelle von Registern und komplexen Adressierungsmodi nur einfache Adressierungsarten. All Operationen werden durch einen Operatorstack ausgeführt. Mehr Informationen zur JVM und zu den Classfiles später.

## Portable WCET Analyse per Java Byte Code

- Predictability (Abschätzbarkeit)
- Virtual Machine Timing Model

Um die WCET Analyse des JBC durchzuführen sind einige Informationen über die Struktur des Codes erforderlich. Diese Informationen, wie z.B. Schleifenkonstrukte, ist im Allgemeine auf sourcecode-ebene verfügbar. Herkömmliche(nicht portierbare) Werkzeuge verlassen sich auf diese Informationen. Um aber eine portierbare Analyse durchführen zu können, darf man sich aber nicht auf diese Informationen aus dem sourcecode verlassen und ebenfalls nicht auf die Fähigkeit das ein compiler diese Informationen zur Verfügung stellt. Diese Informationen müssen direkt aus dem JCF(Java Class File) kommen.

## Predictability

- Auf Quellcode Ebene:
  - Es wird vorrausgesetzt, dass die Programme selber abschätzbar sind, es also z.B. keine unbeschränkten Schleifen gibt.
- Auf Library Ebene:
  - Der Code, der aus den Standardbibliotheken kommt muss ebenfalls abschätzbar sein.
- Compiler Ebene:
  - Der vom Compiler erzeugte Code muss auch abschätzbar sein.

Dies sind typische Anforderungen, die eigentlich an jede WCET Analyse gestellt werden.

## Predictability (fortgesetzt)

- Java Byte Code Instruction Set:
  - Jede der Instruktionen muss abschätzbar sein. Die meisten der Instruktionen erfüllen diese Anforderung und die Execution Time ist gut abschätzbar.
- Java VM Implementation:
  - Der von der VM generierte Code muss ebenfalls vorhersehbar sein. Die WCET wird sehr stark von der Art der Implementation der VM beeinflusst.
  - Es gibt drei Hauptarten der Implementation einer JVM:
    - Interpreted: Die JVM interpretiert jede JBC und führt sie dann aus.
    - Just in Time(JIT): Das erste Mal, wenn eine Methode aufgerufen wird wird sie compiliert und jeder folgende Aufruf profitiert davon
    - Ahead of Time(AOT): Der gesamte Code wird vor dem Ausführen in object code umgewandelt. In diesem Fall ist die VM ein Compiler.

Nicht nur der generierte Code muss vorhersehrbar sein, auch das Verhalten der VM, also Memory Management, garbage collection usw.
Verzögerungen, die sich zB. durch Interaktion mit anderen Tasks ergeben gehören nicht in die WCET Analyse. Allerdings muss man die Kosten der VM für die interne queue in das Timing Model mit einbezogen werden.
Von den Implementationen ist eigentlich nur die dritte, also die AOT für die WCET Analyse geeignet. Die Interpreted ist nicht peformant genug und ist nicht gut für Echtzeitanwendungen geeignet, wäre aber an sich für eine Timing Analyse brauchbar.
Die JIT ist nicht für eine Timing Analyse geeignet, da man die Ausführungszeit, die benötigt wird um den code zu kompilieren mit einrechnen müsste, und die ist kaum vorhersehbar und sehr pessimistisch.
Die AOT ist sehr gut für Echtzeitanwendungen geeignet, da sie die beste Performance hat und ebenfalls gut prognostizierbar ist.

## High Level Analyse des JBC

- Analyse des Control Flow
- Analyse des Data Flow
- Source Code Kommentierung

Die Analyse des JBC setzt sich aus den drei obigen Bereichen zusammen. In diesem Kontext wird gezeigt, wie man Kommentare die in den (Java)Quelltext eingefügt wurden in den JBC integriert und dort zur weiteren Analyse verwenden kann. Die rigide Struktur des JBC erlaubt es eine eine vollständige Control Flow und Data Flow Analyse durchzuführen. Auch wenn der JBC lowlevel ist, unterscheidet er sich aufgrund eben dieser festen Struktur von anderen lowlevel code(maschinencode) Formaten.

## Control Flow

- JBC Instruktionen die den Kontrollfluss beinflussen sind wohl definiert:
  - unconditional control flow
  - case
  - methodenaufrufe
  - exception
- Es gibt keine Mechanismen, die es erlauben den Kontrollfluss zu verändern ausser den obigen.

Es gibt keine indirekte Adressierung (z.B. Adresse in eine Variable laden und dann dort hinspringen) alle Sprünge erfolgen an bekannte Adressen. Das Bestimmen der Grenzen der Basis Blöcke des Code und des zugehörigen Graphen gestaltet sich daher unkompliziert.

## Data Flow

- Datenfluss Informationen und -Abhängigkeiten können nur schwierig direkt aus dem JBC extrahiert werden.
- Daher ist eine Übersetzung in ein generisches “three adress code” Format notwendig.
- Durch Schleifenextraktion auf JBC Level entstehen keine Probleme durch unterschiedliche Schleifenkonstrukte auf Hochsprachenebene.
- Statische Schleifengrenzen, die durch Konstanten im Code ausgedrückt werden, können leicht idenfiziert und extrahiert werden.

Leider wurde nicht angegeben wie dieses Adressformat aussieht. Die inhärenten Restriktionen, die ursprünglich aus Sicherheitsgründen eingeführt wurden machen eine Analyse des JBC möglich. Mit traditionellen Compilertechniken können Schleifen entdeckt, Induktionsvariablen extrahiert und Datenabhängigkeiten festgestellt werden.

## Quellcode Kommentierung

- Komplexere Konstruktionen benötigen Ünterstützung durch Kommentare im Quellcode
- Kommentare sollen durch Funktionsaufrufe aus einer vorher definierten Klasse (WCETAn) generiert werden.
- Der Compiler muss diese Aufrufe in den JBC übernehmen. Wichtig ist hier, dass die Aufrufe nicht an eine andere Stelle verschoben oder gar entfernt werden.

Die Funktionsaufrufe in dem JBC können dann von einem entsprechenden Tool idenfiziert und analysiert werden.
Die meisten Compiler bieten Einstellungen, die garantieren, dass unsere Aufrufe nicht verschoben oder “wegoptimiert” werden.

## Quellcode Kommentierung (fortgesetzt)

- Folgende Informationen im Code benötigten eine Kommentierung:
  - Code Block Grenzen
  - Schleifentyp: begrenzt, unendlich
  - Maximale Anzahl an Schleifendurchläufen
  - Maximale Anzahl an Durchläufen bei verschachtelten Schleifen
- Im allgemeinen brauchen wir Mechanismen um:
  - Tags zu erstellen: Mit Hilfe von eindeutigen Tags werden Codeblöcke identifiziert.
  - Codeblöcke zu benennen
  - Eigenschaften zu bestimmen: tote pfade identifizieren, modifizierte Funktionsaufrufe, Anfang und Ende von zu analysierenden Codesektionen markieren etc.

Wichtig ist, dass der Compiler nicht die Namen der Aufrufe ändert und diese auch nicht verschiebt, damit sie von dem WCETAn Tool korrekt identifiziert und analysiert werden können.

## Quellcode Kommentierung (fortgesetzt)

- Die WCETAn Klasse:

```
public class WCETAn {
   // WCET tags
   static class Mode  {}; // method modes
   static class Label {}; // section of code

   // Naming tags
   static void Define_Mode   (Mode  m) {};
   static void Use_Mode      (Mode  m) {};
   static void Identify_Code (Label l) {};

   // Assertions
   static void Loopcount  (int   n         ) {};
   static void Loopcount  (int   n, Mode  m) {};
   static void Dead_Path  (Mode  m, Label l) {};
   static void Begin_WCET (Label l)          {};
   static void End_WCET   (Label l)          {};
   ...
};
```

Durch statische Methoden und Klassen stellen wir sicher, dass der von unterschiedlichen Compilern generierte Code gleich ist. Tags korrespondieren mit statischen Klassen, Assertions mit statischen Funktionen.
Der Compiler übersetzt die statischen Klassen entweder als statische Variablen oder Felder.

# Beispiel

- Java Quellcode eines Programms, dass die n-te Potenz einer Zahl berechnet:

```
public  class use_pow {
 // declare modes for function pow
  static WCETAn.Mode Power_A = new WCETAn.Mode();

  static float pow (...)
  {  // label for then block
     WCETAn.Label Then1 = new WCETAn.Label();

     // tag Power_A  as a mode of Pow
     WCETAn.Define_Mode(Power_A);

     if (Some_condition)
     {      WCETAn.Identify_Code(Then1);
            Some_Code;
     }
     else
     {      Some_Other_Code;
     }
     // Loop iterations depend on the mode.
     WCETAn.Loopcount(10);
     WCETAn.Loopcount(2,Power_A);
     for (i=1; i<= some_number; i++)
     { loop_body; }
     // in mode A, then1 is never taken
     WCETAn.Dead_Path(Then1,Power_A);
  }
  ...
  // calls to foo can be simple or moded:
  public void foo (){
     float f,g; int i;
     static WCETAn.Label WCET_foo = new WCETAn.Label();

     WCETAn.Begin_WCET(WCET_foo);
       ...
       // this is a general call to pow
       f=pow(g,i); // use worst-case
       ...
       // this is constrained call to pow
       WCETAn.Use_Mode(use_pow.Power_A);
       g=pow(f,2); // results in tighter WCET
       ...
     WCETAn.End_WCET(WCET_foo);
}
```

An diesem Beispiel sieht man alle drei Arten der Kommentierung die hier vorgestellt wurden.
Eine Funktion deklariert neue Instanzen der *Mode* und *Label* Klasse, die mit den entsprechenden Elementen im Code assoziert sind, die *Define_Mode* und *Identify_Code* aufrufen.Der zu analysierende Codeabschnitt ist durch *Beginn_WCET* und *End_WCET* gekennzeichnet.
In dem obigen Beispiel soll der grösste Exponent 10 sein, also wird *Loopcount(10)* gesetzt. Es gibt einen speziellen Modus in dem das Programm arbeitet, wenn der Exponent 2 ist. In diesem Fall ist die maximale Schleifeniteration 2. Weiterhin wird die if-bedingung immer falsch sein in diesem Fall, um das zu repräsentieren wird hier *Dead_Path(Then1,Power_A)* gesetzt.

## Beispiel(fortgesetzt)

- Java Byte Code des compilierten Beispiels(Auszug):

```
 0 new #2 <Class WCETAn$Label>
 3 dup
 4 invokespecial #6 <Method WCETAn$Label.<init>()V>
 7 astore 7
...
52 aload 7
54 invokestatic #9 <Method WCETAn.
      Identify_Code(LWCETAn$Label;)V>
```

Kompletter JBC zu lang um ihn vernünftig auf Folien darzustellen. Komplettes Beispiel kann aber dem Ausdruck beiliegen.

## WCEF Analyse

- In dem hardwareunabhängigen Teil der Analyse kann die WCET nicht festgestellt werden. Hier wird die WCEF festgestellt und die Informationen darüber in das Java Class File übertragen.
- WCEF Vector:
  - Der WCEF Vektor eines Codesegment S wird durch $F(S)=\langle F_{p1}(S), F_{p2}(S),..., F_{pn}(S)\rangle$ mit $p_i \in B$ denotiert. B ist die Menge der JBC und $F_p(S)=(a,b)$ mit a ist die maximale Anzahl, die die JBC Anweisung p in dem Segment S ausgeführt wird und b ist die maximale Anzahl an internen Iterationen.

Portable WCET wird dadurch erreicht, dass man an erster Stelle feststellt, wie die maximale Ausführungsfrequenz der einzelnen JBC Instruktionen in den BBs ist und ein Timing Schema für die WCEF Vektoren erstellt. Dadurch kann man diese Informationen mit dem JCF verbreiten und die entgültige Bestimmung der WCET zurückstellen, bis sämtliche Details über die WCET der JCB Instruktionen bekannt sind.

# WCEF Timing Schema

- Es muss auch die Anzahl der Aufrufe der Methoden festgestellt werden. Insbesondere solche, die keinen JBC haben(native Methoden)
  - Wir denotieren sie einfach mit M(S) = {($name_1$,$f_1$),($name_2$,$f_2$),...}
- Berechnung der WCEF Vektoren
  - Sei im Folgenden B die Menge der JBC Instruktionen, F(S) ein Vektor der Ausführungshäufigkeit der Instruktionen in B für ein Codesegment S und M(S) der WCEF Aufruf.

## WCEF Timing Schema

- Sequentiell:
  - Für zwei sequentielle Codesegmente S1;S2 mit den Vektoren F(S1) und F(S2) ist die WCEF der Sequenz S=S1;S2:

    (1) F(S) = F(S1) + F(S2)

  und für die WCEF Aufrufe M(S2),M(S2)

    (2) M(S) = M(S1) + M(S2)
- Iteration:
  - Für eine Schleife S mit dem Schleifenkopf H, Körper B und einer maximalen Anzahl von Durchläufen n ist die WCEF:

    (3) F(S) = (n +1)F(H) + nF(B)

  und

    (4) M(S) = nM(B)

(1) + (2):
Die Addition der Vektoren hat die übliche Bedeutung(Addition der Elemente). Und die Addition der der Paare (a,b) + (c,d) = (a+c,b+d) die erwartete Bedeutung. Im schlimmsten Fall wird die Instruktion a+c mal ausgeführt und iteriert c+d mal.
Auch die Addition der WCEF Aufrufe Mengen bedeutet einfach die Vereinigung beider Mengen.
(3) + (4):
Die Addition der Vektoren und die Multiplikation eines Vektor mit einem Paar (a,b) haben die übliche Bedeutung und es ist $nM(B) = \{(name_i, nf_i)\}$ für alle $(name_i, f_i)$ aus M(B).

# WCEF Timing Schema

- Conditional:
  - Für zwei alternative Zweige S1 und S2 eines bedingten Ausdrucks S ist die WCEF von S:

    (5) $F_p(S) = \max\{F_p(S1), F_p(S2)\}$ für alle p aus B

  und

    (6) $M(S) = \max\{M(S1), M(S2)\}$

(5) + (6) $\max\{(a,b),(c,d)\} = \max\{a,c\}, \max\{b,d\}$.
$\max\{M(A),M(B)\}$ ist die Menge der Namen und Ausführungsanzahl, die in A und Bvorhanden sind mit der maximalen Ausführunganzahl aus A oder B.

## WCEF Timing Schema

- Pipeline Effekte
  - Damit Pipelineeffekte berücksichtigt werden muss die relative Ausführungshäufigkeit eines Paars von Instruktionen mit übergeben werden.

  $F_{(p,q)}(S)$ ist die WCEF der JBC p die vor q ausgeführt werden.

Nicht alle Instruktionspaare sind signifikant und es muss nur die Untermenge von signifikanten Paaren in die entgültige WCEF übergeben werden.

## Plattformabhängige Analyse

- Die Plattformabhängige Analyse wird in der Form eines VMTM(Virtual Machine Timing Model) gemacht.
  - In der einfachsten Form ist das eine Beschreibung der WCET der JBCs und zusätzliche Informationen wie die WCET der nativen Methoden.
  - Formal denotiert T(p) = (x,y) die WCET einer JBC p. T(p) ist von der Form x + yn mit x ist die Zeit, die zur Ausführung benötigt wird und y die Zeit die für jeder der internen Interationen benötigt wird.
  - Die "gain time" aufgrund von Pipelineeffekten für zwei aufeinanderfolgende Instruktionen p1;p2 wird als ?(p1,p2) geschrieben.

Der "gain factor" ist die minimale Reduktion der Ausführungszeit, die wir erwarten können. Es ist unmöglich den "gain factor" für alle Instruktionssequenzen zu berücksichtigen. Der Einfachheit halber definieren wir ? für Instruktionen mit konstanter Ausführungszeit, also wo y=0 ist.
Zusammenfassend ist das Timing Modell einer VM eine Liste mit der WCET der nativen Methoden und ein Paar von Tabellen T und ? die die WCET jeder JBC Instruktion und den "gain factor" auflisten.

# Zusammenführen der Analysen

- Der letzte Schritt ist die Bestimmung der WCET aus den WCEF Vektoren und den WCEF Aufrufen.
  - Diese Berechnung sollte in die VM integriert werden, deswegen darf sie nur sehr einfach gestaltet sein und sollte wenig Resourcen verbrauchen.
  - Gegeben sei ein Vektor $F_S$ und eine VM Timing Tabelle T, dann ist die WCET in S:

$$WCET_S = \sum_{\forall p \in B} (F_p(S)a \cdot T(p)x + F_p(S)b \cdot T(p)y) - \sum_{\forall (p,q) \in S^2} F_{(p,q)(S)} \cdot \lambda(p,q) + \sum_{\forall n \in M(B)name} M(B)f \cdot N(n)$$

Die obige Formel ist einfach nur eine Linearkombination aus den WCEF Vektoren, Namen und dem VMTM.
N(n) ist die WCET einer durch “name” gegebenen Methode.

# Übertragen der WCEF Informationen

- Die WCEF Informationen werden in dem JCF als zusätzliche code attribute verbreitet.
- Die Vektoren die im ersten Schritt der Analyse erstellt wurden der wcef_info Tabelle hinzugefügt.

**Table 1. Format of a *wcef_info* Table**

| *Type* | *Name* | *Count* |
|---|---|---|
| u2 | attribute_name_index | 1 |
| u4 | attribute_length | 1 |
| u4 | wcef | 512 |
| u2 | pair_table_length | 1 |
| pair_info | pair_table | pair_table_length |
| u2 | call_table_length | 1 |
| call_info | call_table | call_table_length |

**Table 2. Format of a *pair_info* Table**

| *Type* | *Name* | *Count* |
|---|---|---|
| u2 | JBCindex1 | 1 |
| u2 | JBCindex2 | 1 |
| u4 | wcef | 1 |

**Table 3. Format of a *call_info* Table**

| *Type* | *Name* | *Count* |
|---|---|---|
| u2 | attribute_name_index | 1 |
| u4 | wcef | 1 |

Das Format der Tabelle folgt dem Format für attribute_info Tabellen von CFs. attribute_name_index ist der Index des Attributs Wcet in dem Konstantenpool und attribute_length die Länge der Tabelle – 6 Bytes. Die folgenden 256*2 wcef Felder enthalten die Ausführungsfrequenzen und internen Iterationen des byte codes der Codesektion der Methode, wobei der i-te wcef Eintrag die WCEF des byte code i enthält. Obwohl nicht alle JBCs genutzt werden, resultiert die indizierte Version in einer kompakteren Tabelle. Danach werden die Paare von JBCs definiert. Dies ist ein Tripel aus dem Index der beiden JBCs und der execution frequency (Tabelle 2). call_table_length enthält die Länge der Tabelle, welche die Methoden auflistet, welche von der jetzigen aufgerufen werden (siehe Tabelle 3). Das attribute_name_index der call_info Tabelle zeigt auf den Namen der aufgerufenen Methode und wcef speichert die worst-case Anzahl an Aufrufen dieser Methode aus der jetzigen.

## Unterstützung für WCET Analysen in der JVM

- Jede JVM speichert ihre eigene Timing Tabelle
- Zusätzlich wird die wcet_info Tabelle genutzt, um die berechneten WCETs zu speichern.
- Diese Unterstützung wird als spezielles WCET Class File implementiert.
  - Identifikation, ob die VM eine portable WCET Unterstützung hat.
  - Untersuchung der aktuellen WCET Code Attribute die für eine Klasse verfügbar sind.
  - Berechnen der WCET eines Bereich.

Mit Hilfe der VM Timing Table und der Methoden Timing Table ist es relativ einfach die WCET zu berechnen. Um die WCET einer Methode zu erhalten, liest die VM die wcet_info Tabelle der Methode aus und berechnet die WCET für alle bytecodes. Danach addiert die VM die WCETs aller Aufrufe anderer Methoden zu den schon berechneten WCETs, d.h. für jede Methode in der call_info Tabelle durchsucht sie die method-Tabelle auf die zugehörige WCET. Wenn die WCET vorhanden ist, dann wird sie mit der Anzahl der Aufrufe multipliziert und zu dem bis jetzt berechneten Wert addiert. Falls nicht, dann wird die WCET durch einen rekursiven Aufruf des WCET analyzer berechnet und in der method-Tabelle gespeichert.

# Evaluation

- High-Level Analyse

  Es soll der folgende Code analysiert werden:

```
public static void BubbleSort(int a[])
{ int i, j, t;
  int size = 10;
  WCETAn.LoopCount(9);
  for (i=size-1; i>=1; i--)
  { WCETAn.LoopCount(9);
     for (j=1; j<=i; j++)
     { if (a[j-1] > a[j])
       {t      = a[j-1];
        a[j-1] = a[j];
        a[j]   = t;
} } } }
```

Es wurde hier einfache (pessimistische) Schleifengrenzen angenommen. Eine akkurate Analyse würde ergeben, dass die Schleife tatsächlich nur $\frac{10 * 9}{2}$ mal durchlaufen würden und nicht $9^2$ mal.

# Evaluation

- Der Graph der BBs und die WCEF Vektoren der BubbleSort Funktion

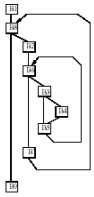



Table 4. WCEF vectors of the Basic Blocks and of the collapsed BubbleSort function

| | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | BSort |
|---|---|---|---|---|---|---|---|---|---|
| ALOAD_0 | 0 | 0 | 2 | 4 | 0 | 0 | 0 | 0 | 486 |
| BIPUSH | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| GO_TO | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 10 |
| IALOAD | 0 | 0 | 2 | 3 | 0 | 0 | 0 | 0 | 405 |
| IASTORE | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 162 |
| ICONST_0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 81 |
| ICONST_1 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 171 |
| IF_ICMPLE | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 163 |
| IFGT | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 10 |
| IINC | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 90 |
| ILOAD_1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 92 |
| ISTORE_1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| ISUB | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 243 |

- Die Auswertung des Beispiels resultiert in:
  - B1 + 10B8 + 9(B2 + 10B6 + 9(B3 + B4 + B5) + B7) + B9

Die beiden verschachtelten Schleifen können leicht identifiziert werden: B6, B3, B4, B5 und B8, B3, B4,B5, B2, B7

# Evalutation

- VMTM Beispiele:

**Table 5. VM timing models of the Kaffe and Komodo virtual machines**

| | Kaffe | Komodo |
|---|---|---|
| ALOAD_0 | 67 | 6 |
| BIPUSH | 68 | 6 |
| GO_TO | 63 | 9 |
| IALOAD | 67 | 14 |
| IASTORE | 65 | 15 |
| ICONST_0 | 60 | 6 |
| ICONST_1 | 60 | 6 |
| IF_ICMPLE | 77 | 9 |
| IFGT | 79 | 9 |
| IINC | 61 | 6 |
| ILOAD_1 | 76 | 6 |
| ISTORE_1 | 68 | 6 |
| ISUB | 69 | 6 |

Um die Ausführungszeiten der Kaffe VM zu bekommen, wurden die “caffeine benchmarks” benutzt und “cycle” genaue Messungen der Ausführungzeit erhalten. In diesen Timings war bereits der Overhead und die “gain time” enthalten. Also wird für diese VM ?(p,q) = 0 angenommen.
Komodo ist eine “stand alone Java on Chip VM”. Ihre Haupteigenschaft ist, dass sie eine 5 stage pipeline hat und fast alle JBC haben einen “gain factor” von 5. Somit ist die potentielle Ausführung einer Instruktion in einem Zyklus möglich. In diesem Fall haben fast alle Instruktionspaare einen ?(p,q) = 5, für einige Instruktionen, inklusive Kontrollflußinstruktionen, ist er nur 3.

# Evaluation

- Low-Level Analyse
  - Der letzte Teil der Analyse besteht darin die beiden Datenstrukturen zusammenzufassen um die entgültige WCET zu erhalten.
    - Kaffe VM : 128703 Zyklen (257?s bei 500MHz)
    - Komodo VM : 16737 Zyklen (523?s bei 32MHz)

Leider war nicht vermerkt, ob und wie stark das Ergebniss von der “echten” WCET abweicht.

# Referenzen

- Portable Worst-Case Execution Time Analysis Using Java Byte Code

  (Guillem Bernat, Alan Burns and Andy Wellings, University of York)

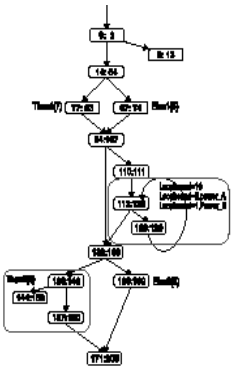
- Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis

  (I. Bate, G. Bernat, University of York and P. Puschner, Technische Universität Wien)

# Beispiel: Quelltext

```
//------------------------------------------------------------------
----
// JAVELIN PROJECT:     Worst Case Execution Time Annotations
class
//------------------------------------------------------------------
-----
// (c) Guillem Bernat. University of York. 1999-2000.   v. 1.0
// http://www.cs.york.ac.uk/~bernat/javelin
//
//      Example of usage of the WCETAn class file for
annotations.
//
//
//
//
   public  class use_pow {

      static WCETAn.Mode Power_A = new WCETAn.Mode();
      static WCETAn.Mode Power_B = new WCETAn.Mode();

      static float pow (float base, int exponent) {

         boolean exchange ;
         float    aux, result;
         int      abs_exponent;
         int      i;

        //Annotations

         WCETAn.Label Then1 = new WCETAn.Label();
         WCETAn.Label Then2 = new WCETAn.Label();
         WCETAn.Label Else1 = new WCETAn.Label();
         WCETAn.Label Else2 = new WCETAn.Label();


         WCETAn.Define_Mode(Power_A);
         WCETAn.Define_Mode(Power_B);

         if (exponent < 0)
         {
            WCETAn.Define_Path(Then1);
            exchange = true;
            abs_exponent = - exponent;
         }
         else
         {
            WCETAn.Define_Path(Else1);
            exchange = false;
            abs_exponent = exponent;
         }
         ;
aux = (float)1.0;

         WCETAn.Loopcount(10);
         WCETAn.Loopcount(2,Power_A);
         WCETAn.Loopcount(1,Power_B);
         for (i=1; i<= abs_exponent; i++)
         {
            aux = aux * base;

         }
         if (exchange )
         {
            WCETAn.Define_Path(Then2);
            result = (float)1.0 / aux;
         }
         else
         {
            WCETAn.Define_Path(Else2);
            result = aux;}

         WCETAn.Dead_Path(Power_A,Then1);
         WCETAn.Dead_Path(Power_A,Else2);
         WCETAn.Dead_Path(Power_B,Then2);
         WCETAn.Dead_Path(Power_B,Else1);

         return (aux);


      }

   public static void main (String[] args){

         float f,g;
         int  i;

         WCETAn.Scope S = new WCETAn.Scope();

         WCETAn.Begin_WCET(S);
         f=(float)1.0;
         g=f;
         i=5;
        //...
         pow(g,i);
        //...
         WCETAn.Uses_Mode(use_pow.Power_B); // next call uses this mode.
         f=pow(f,2);
         WCETAn.End_WCET(S);
        //...
      }

   }
```

# Beispiel: Graph



# Beispiel: JBC

```
Method float POW(float, int)
------------------------------------------------
   0 getstatic #28 <Field use_pow_.POW__elab_bool Z>
   3 ifne 14
------------------------------------------------
   6 new #30 <Class java.lang.ClassCircularityError>

   9 dup
  10 invokespecial #32 <Method java.lang.
        ClassCircularityError.<init>()V>
  13 athrow
  14 new #34 <Class wcetan.Label>
  17 dup
  18 invokespecial #36 <Method wcetan.Label.<init>()V>
  21 astore 5
  23 new #34 <Class wcetan.Label>
  26 dup
  27 invokespecial #36 <Method wcetan.Label.<init>()V>
  30 astore 6
  32 new #34 <Class wcetan.Label>
  35 dup
  36 invokespecial #36 <Method wcetan.Label.<init>()V>
  39 astore 7
  41 new #34 <Class wcetan.Label>
  44 dup
  45 invokespecial #36 <Method wcetan.Label.<init>()V>
  48 astore 8
  50 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
  53 invokestatic #42 <Method WCETAn.
       Define_Mode(Lwcetan/Mode;)V>
  56 getstatic #19 <Field use_pow.
       Power_B Lwcetan/Mode;>
  59 invokestatic #42 <Method WCETAn.
       Define_Mode(Lwcetan/Mode;)V>
  62 iload_1
  63 iconst_0
  64 if_icmplt 77
------------------------------------------------

  67 aload 5
  69 invokestatic #46 <Method WCETAn.
       Identify_Code(Lwcetan/Label;)V>
  72 iconst_0
  73 istore_2
  74 goto 84
------------------------------------------------
  77 aload 7
  79 invokestatic #46 <Method WCETAn.
       Identify_Code(Lwcetan/Label;)V>
  82 iconst_1
  83 istore_2
------------------------------------------------
  84 fconst_1
  85 fstore_3
  86 bipush 10
  88 invokestatic #50 <Method WCETAn.Loopcount(I)V>
  91 iconst_2
  92 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
  95 invokestatic #53 <Method WCETAn.
       Loopcount(ILwcetan/Mode;)V>
  98 iconst_3
  99 getstatic #19 <Field use_pow.Power_B Lwcetan/Mode;>
 102 invokestatic #53 <Method WCETAn.
       Loopcount(ILwcetan/Mode;)V>
 105 iload_1
 106 iconst_0
 107 if_icmple 132
------------------------------------------------
 110 iconst_1
 111 istore_9
------------------------------------------------
```

# Beispiel: JBC

```
113 fload_3
114 fload_0
115 fmul
116 fstore_3
117 iload 9
119 iload_1
120 if_icmpeq 132
---------------------------------------------------
123 iload 9
125 iconst_1
126 iadd
127 istore 9
129 goto 113
---------------------------------------------------
132 iload_2
133 ifeq 163
---------------------------------------------------
136 aload 6
138 invokestatic #46 <Method WCETAn.
      Identify_Code(Lwcetan/Label;)V>
141 fconst_1
142 fload_3
143 dup
144 fconst_0
145 fcmpl
146 ifne 157
---------------------------------------------------
149 new #55 <Class java.lang.ArithmeticException>
152 dup
153 invokespecial #57 <Method java.lang.
      ArithmeticException.<init>()V>
156 athrow
---------------------------------------------------
157 fdiv
158 fstore 4
160 goto 171
---------------------------------------------------
163 aload 8
165 invokestatic #46 <Method WCETAn.
      Identify_Code(Lwcetan/Label;)V>
168 fload_3
169 fstore 4
---------------------------------------------------
```

```
171 aload 7
173 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
176 invokestatic #61 <Method WCETAn.
      Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
179 aload 6
181 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
184 invokestatic #61 <Method WCETAn.
      Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
187 aload 7
189 getstatic #19 <Field use_pow.Power_B Lwcetan/Mode;>
192 invokestatic #61 <Method WCETAn.
      Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
195 aload 6
197 getstatic #19 <Field use_pow.Power_B Lwcetan/Mode;>
200 invokestatic #61 <Method WCETAn.
      Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
203 fload 4
205 freturn }
```