Hoe hoog springt Jazelle?

Hard Java in ARM-processoren

De objectgeoriënteerde programmeertaal Java is al jarenlang de grote belofte in embedded toepassingen maar er zijn nog altijd beperkingen in prestatie en samenwerkbaarheid. In 2001 heeft ARM aangekondigd een hardware-implementatie van Java te integreren in haar ontwerpen en de eerste microprocessoren met dit Jazelle zullen waarschijnlijk dit jaar op de markt komen. De ondersteuning door ARM zou wel eens de grote doorbraak kunnen betekenen voor Java in elektronische apparaten. Hoe hoog springt Jazelle?

HANS VAN THIEL

Oorspronkelijk is Java door Sun Microsystems ontwikkeld als platformonafhankelijke programmeertaal voor consumentenelektronica, maar er zijn inmiddels drie hoofdvarianten: J2EE (Enterprise Edition) voor bedrijfsservers, J2SE (Standard) voor de desktop en J2ME (Micro) voor mobiele apparaten en consumentenelektronica. Deze Javavarianten zijn vrijgegeven en worden onderhouden door de JCP-organisatie (Java Community Process) waarbinnen de rol van Sun geleidelijk afneemt. Er is nog een vierde versie, Java Card, voor smart cards, die onder een commerciële licentie van Sun valt. Voor de programmeur zitten de belangrijkste verschillen in de ondersteuning met diverse 'packages', standaardbibliotheken van klassen.

Objectgeoriënteerd

Java is objectgeoriënteerd, zoals te zien is in listing 1a en 1b. Hier heeft de klasse Offset twee functies (methoden in de Javaterminologie) om een getal met de constante 5 te verlagen of te verhogen, maar alleen als er een vlag is gesteld. De klasse Step past die functionaliteit toe op een lokale variabele en gebruikt een eigen vlag om de richting van de verandering te sturen.

Java is een streng getypeerde taal met talloze ingebouwde kwalificaties die ook tijdens compilatie en daarna worden geverifieerd. De syntaxis voor variabelen, assignment en control is voor een groot deel rechtstreeks van C overgenomen, zoals de listing ook illustreert. Er zijn geen globale variabelen in Java - de eenheid van compilatie is de 'public class' die wordt omgezet in een 'class file'. Dit bestand bevat behalve Java machinecode ('bytecode') ook informatie over typen en de kwalificaties die bepalen wat wel en niet kan met velden, methoden en de klasse als geheel.

De structuur van een 'class file' en de bytecode-instructieset zijn dwingend vastgelegd in de JVM-specificatie (Java Virtual Machine). Die zijn dus platformonafhankelijk en het is de taak van de JVM om de class files te vinden, te controleren, te laden, te linken en om te zetten naar werkende code voor een bepaalde processor en besturingssysteem. De JVM-specificatie schrijft verder niet voor hoe dit gebeurt, alleen het class file formaat en de semantiek van de bytecode moeten behouden blijven.

Vanwege dat laatste ligt het echter voor de hand een bytecode-representatie te gebruiken als tussenvorm, die dan in een laatste stap wordt getransformeerd naar hardware en besturingssysteem. Listing 2 geeft die bytecode-representatie voor de Step-applicatie in J2SE en de Java Hotspot Client VM van Sun. Deze JVM voor de desktop is, inclusief de omzetting naar Pentiumcode, geheel in software uitgevoerd.

De Java bytecode-representatie staat echter, zoals in listing 2 is te zien, voor een groot deel heel dicht bij assemblercode en talloze operaties zouden dan ook in hardware kunnen worden geïmplementeerd.

Zo heeft aJile Systems Java-processoren ontworpen die alle bytecode direct kunnen uitvoeren en die zelfs een microgeprogrammeerde thread-manager bevatten (threads zijn de in objecten uitgevoerde Javaprocessen). Voor deze aJile-processoren is zelfs geen RTOS vereist, maar het nadeel is natuurlijk dat ze alleen voor Java geschikt zijn. Er zijn ook Java co-processoren ontwikkeld die op een soortgelijke manier werken als floating-point co-processoren, maar dan voor bytecode-operaties. Deze oplossing is kostbaar en vereist aanpassingen aan het ontwerp. Een belangrijk bezwaar lijkt ook dat deze hardwareoplossingen door start-ups zijn ontwikkeld en van het begin af aan een markt moesten - en nog moeten - vinden. Verder heeft van de gevestigde namen alleen Xilinx een Java hardware-implementatie,

Elke seconde worden er tien ARM-chips verkocht

```
public class Offset {
     final int offset= 5;
     private boolean enable= false;
     void setEnable( boolean flag ){enable = flag;}
     int up ( int value){
          if( enable ){return (value + offset);}
          else {return (value);}
     }
     int down ( int value){
          if( enable ){return (value - offset);}
          else {return (value);}
     }
}
```

Listing 1a. Met een Offset-object kan een integer waarde met 5 worden verhoogd of verlaagd als de enable-vlag aan staat.

maar dit LavaCore is uiteraard beperkt tot FPGA-ontwerpen.
Dan verschijnt in 2001 Advanced Risc Machines (ARM) op het toneel met de aankondiging dat het voor al haar processorontwerpen een Java bytecode-implementatie gaat leveren. Dit Jazelle bestaat slechts uit 12K poorten, is volledig geïntegreerd met de ARM-architectuur en wordt ondersteund met eigen software, de Java Technology Enabling Kit (JTEK).
IP-ontwerper ARM is met haar ingebedde 32-bit RISC-processoren niet bepaald een start-up - het heeft wereldwijd een marktaandeel van 75 procent en meer dan vijftig halfgeleiderfabrikanten hebben een ARM-licentie. Er zijn meer dan een miljard ARM-chips geproduceerd en elke seconde worden er zo'n tien verkocht. Het kan dus haast niet anders of Jazelle zal een invloed hebben op het gebruik van Java in de sector waar ARM zich op richt, ingebedde systemen en consumentenelektronica. Dat is J2ME en de twee onderliggende configuraties CDC en CDLC (Connected - Limited - Device Configuration) en hun diverse profielen. Dus wat kan de Jazelle technologie hieraan bijdragen?

Instructieset architectuur

Elke Java bytecode-operatie bestaat uit een opcode van één byte, gevolgd door nul of meer operand-bytes. In assembler-mnemonics (zoals listing 2) wordt het type aangegeven door de eerste letter. Er zijn int-, long-, float-, double-, byte-, char-, short- en reference-operaties die behalve de referentie, die met 'a' begint, de voor de hand liggende voorvoegsels hebben. Voor 'boolean' is geen eigen bytecodetype ingeruimd.
De instructies werken op een 32-bit

```
public class Step {
 static boolean up = true;
 public static void main(String[] args) {
  int control = 90;
  Offset change= new Offset();
  change.setEnable(true);
  control = (up ? change.up(control): change.down( control ));
 }
}
```

Listing 1b. De klasse Step gebruikt een instantie van Offset om de lokale control-variabele met 5 te verhogen of verlagen, afhankelijk van up.

brede stack en gebruiken dan ook slechts één adres. Voor operaties met kleine constanten bestaan aparte instructies. Zo laadt 'iconst_1' het naar 32-bit vergrote integer 1 op de stack. Elke Java-thread heeft behalve een eigen PC ook een eigen stack. Binnen een thread behoort de besturing op ieder moment aan één klasse of object en één methode.
Lokale variabelen binnen een methode worden niet met geheugenadressen aangeduid maar met indices in een array of een frame. Zo laadt 'istore_1' de top van de stack in de variabele met index 1 (in listing 2 de 'control'-variabele uit de main method van de Step klasse) en verwijdert meteen die waarde uit de stack.
Naast een stack voor elke thread bestaan er een 'method area' en een 'heap' in het geheugen die beide door alle threads worden gebruikt. De 'method area' is te vergelijken met de 'text area" van een Unix-proces en bevat alle bytecode en alle velden die tijdens executie permanent aanwezig zijn. Omdat Java objectgeoriënteerd is bestaat dit vaste geheugengedeelte uit structuren die aan de klassen zijn toegewezen. Een van die structuren is de 'constant pool' die een weergave is van de 'constant_pool' tabel uit de corresponderende 'class file'. In assembler worden referenties hieraan voorafgegaan door een '#'. Zo refereren de opcodes: 'invokespecial' (voor de default Offset constructor), 'invokevirtual' (de standaard aanroep van een methode) , 'putstatic' (initialisatie van een klasse variabele) en 'getstatic' (op de stack zetten van een klasse variabele) allemaal aan de constant pool van de Step klasse.
De 'heap' is het dynamisch gedeelte van het geheugen en wordt gebruikt voor objecten en arrays.
De 'new' opcode maakt geheugen vrij op de heap voor een object dat vervolgens met 'invokespecial' wordt geconstrueerd.

Garbage collection

Alle Java-threads werken op dezelfde heap maar gebruiken een lokale kopie - synchronisatie valt onder verantwoordelijkheid van de applicatie of de JVM-software. De bekende 'garbage collection' van Java werkt op de heap en hoort ook bij de JVM.
De JVM-specificatie laat echter veel open over thread-besturing en de interactie van de garbagecollector met applicaties. In de praktijk is deze onduidelijkheid een groot probleem gebleken voor Java in embedded systemen, of, beter gezegd, voor de standaard toepassing en samenwerkbaarheid in zulke systemen. Als Jazelle een de facto standaard wordt - en gezien de status van ARM is dat heel goed mogelijk - zou het in dat opzicht een duidelijke verbetering zijn.

ARM levert van oudsher in haar 32-bit RISC-processoren een vereenvoudigde 16-bit compressie van de normale 32-bit instructies. Deze 'Thumb'-extensie kan de codedichtheid bijna halveren en is ook bedoeld voor 16-bit datapaden. Beide soorten code kunnen goed

```
Compiled from Step.java
public class Step extends java.lang.Object {
    static boolean up;
    public Step();
 /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]);
 /* Stack=2, Locals=3, Args_size=1 */
    static {};
 /* Stack=1, Locals=0, Args_size=0 */
}

Method Step()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 bipush 90
   2 istore_1
   3 new #2 <Class Offset>
   6 dup
   7 invokespecial #3 <Method Offset()>
  10 astore_2
  11 aload_2
  12 iconst_1
  13 invokevirtual #4 <Method void setEnable(boolean)>
  16 getstatic #5 <Field boolean up>
  19 ifeq 30
  22 aload_2
  23 iload_1
  24 invokevirtual #6 <Method int up(int)>
  27 goto 35
  30 aload_2
  31 iload_1
  32 invokevirtual #7 <Method int down(int)>
  35 istore_1
  36 return

Method static {}
   0 iconst_1
   1 putstatic #5 <Field boolean up>
   4 return
```

Listing 2. Gecompliceerde bytecode van de klasse Step die gebruik maakt van de functionaliteit van een Offset-object.

samen worden gebruikt omdat de overgang van 16-bit Thumb naar normale 32-bit toestand door een enkele 'Branch and Exchange'-instructie (BX) wordt uitgevoerd.
Hetzelfde principe wordt toegepast voor Jazelle - een nieuwe ARM-instructie 'BXJ Rm' schakelt de processor over van de normale toestand naar een Javatoestand. Deze 32-bit instructie test eerst conditiecodebits 28 ... 31, bewaart de normale PC, initialiseert de processor voor Java-state, springt naar een door bits 0 ... 3 (Rm) gespecificeerd adres en begint dan Java-bytecodes uit te voeren. Bytecodes worden in twee stappen opgehaald en gedecodeerd. De processormodus wordt aangegeven door het CPSR-register dat daarvoor een J-bit heeft gekregen. De Javatoestand is hierdoor geïntegreerd met het normale ARM interrupt- en exceptiemodel dat door besturingssystemen wordt gebruikt. De omschakeling tussen Java en normale modus kan binnen enkele klokcyclussen worden uitgevoerd.

Tabel 1 geeft aan hoe de Java bytecode-architectuur door ARM-registers wordt geïmplementeerd.
De bovenste drie elementen van de Java-stack, die het meest worden gebruikt, zijn gecached in registers R0 ... R3. Opmerkelijk is verder dat registers R9 ... R11 voor de JVM zijn gereserveerd.

Acht keer sneller

Zoals gezegd is bytecode voor een groot deel een soort assembler, maar niet helemaal.
De Jazelle-hardware executeert dan ook niet meer dan 140 Java-instructies direct - 94 worden geëmuleerd door normale ARM-instructies. Alle codes, inclusief de emulaties, zijn her-opstartbaar en kunnen dus worden geïnterrupteerd. ARM claimt dat Jazelle normaal gesproken acht keer sneller is dan de beste software-byte-code-optimalisaties. Bovendien levert de uitvoering in hardware een aanzienlijke stroombesparing op.
Jazelle vereist nog steeds een JVM, zoals de Sun J2ME referentie-implementaties CVM en KVM voor respectievelijk CDC- en CLDC-configuraties. Uiteraard zijn voor Jazelle dan aanpassingen nodig.

Voor de softwareondersteuning werkt ARM vooral samen met Aplix, Symbian en SavaJe. De hardware is in licentie bij Motorola, Toshiba, Qualcomm, LSI Logic, Sanyo, Mitsubishi en Texas Instruments. ARM heeft een overeenkomst met Sun om Jazelle te integreren met Java Card en voor systeemverificatie wordt samengewerkt met Mentor Graphics.
Het succes van Jazelle zal uiteindelijk afhangen van de licentienemers en vervolgens van de toepassing in producten. De eerste producten worden in 2004 verwacht. De komende jaren moet dan blijken of deze ARM-technologie de norm zal worden in de versnipperde embedded markt. ■

Tabel 1. Het gebruik van de ARM-registers in Jazelle State (Bron: ARM)

R0 ... R3	Java expressie stack cache
R4	Lokale variabele 0 (this pointer)
R5	Pointer naar tabel van SW handlers
R6	Java stack pointer
R7	Java variabelen pointer
R8	Java constant pool pointer
R9 ... R11	Gereserveerd voor JVM (niet voor hardware)
R12, R14	Scratch gebruik / Java return adres
R13	Machine stack power
R15	Java PC (program counter)