Jiffy: Portierung eines JIT-Compilers auf FPGAs

Georg Acher

Lehr- und Forschungseinheit Informatik X
Rechnertechnik und Rechnerorganisation /
Parallelrechnerarchitektur (LRR-TUM)
Institut für Informatik der Technischen Universität München
Arcisstr. 21, D-80290 München, Germany
acher@in.tum.de

Zusammenfassung Java hat in der letzten Zeit immer größere Verbreitung gefunden, dies ist zu einem Teil der Maschinenunabhängigkeit der Java Virtual Machine zuzuschreiben. Allerdings ist die Geschwindigkeit von Java-Programmen ist immer noch ein gewichtiger Kritikpunkt am Java-Konzept. Während dies für state-of-the-art Systeme durch Just-In-Time-Compiler (JIT) kein echter Hinderungsgrund für den Einsatz ist, sind diese Compiler für kleinere Systeme aufwendig und benötigen zu viele Ressourcen. Im folgenden wird ein neuartiges Konzept (Jiffy) eines JIT-Compilers in HW vorgestellt, das als Mittelweg zwischen langsamer, aber ressourcenschonender Interpretierung und schneller, aber rechen- und speicheraufwendiger JIT-Compilierung in SW gelten kann.

1 Einleitung

Die effiziente Ausführung von Java-Programmen [3], die im Bytecode der Java Virtual Machine (JVM) vorliegen [8], ist immer noch einer der wichtigsten Punkte bei der Implementierung einer JVM. Üblicherweise gibt es dafür vier Möglichkeiten:

1. Interpretation des Bytecodes
2. Just-In-Time Compilation (JIT) des Bytecodes in den Maschinencode der ausführenden CPU
3. Benutzung einer speziell JVM ausführenden CPU (Java-CPU)
4. Precompilierung des Bytecodes[1]

Möglichkeit 1 ist zwar portabel, allerdings liegt die Ausführungsgeschwindigkeit um Größenordnungen unter denen von kompilierten C/C++ Code. Dennoch ist der relativ geringe Speicherplatzverbrauch des JVM-Interpreters selbst und seiner Laufzeitdatenstrukturen von Vorteil, wenn die JVM auf 'kleinen' CPUs implementiert werden soll, wie z.B. in [1] gezeigt wird. Trotzdem scheint die

[1] Auf die letzte Möglichkeit (z.B. [5]) wird nicht näher eingegangen, da sie für heterogene Systeme nur eingeschränkt nutzbar ist.

Nutzung von Java erst ab einer gewissen Prozessorleistung sinnvoll. Als Anhaltspunkt seien Prozessoren mit mindestens 32Bit und Taktfrequenzen >16MHz genannt (also im Mid-Range Controllerbereich z.B. MC68xxx und ARM).

Für den JIT-Ansatz spricht zunächst die gute Ablaufgeschwindigkeit, die die Effizienz von C/C++-Compilaten erreichen kann. Andererseits ist dies nur mit hohem Softwareaufwand und den entsprechenden Compilertechniken zu erreichen. Dies führt einerseits dazu, daß bei selten aufgerufenen Routinen die Compilierzeit wesentlich größer sein kann, als die eigentliche Ausführungszeit. Diesen Nachteil kann man z.B. mit einer dynamischen Compilierung umgehen. Die JVM-Routinen werden dabei erst dann compiliert, wenn sie häufiger benötigt werden, ansonsten werden sie 'nur' interpretiert. Sun verfolgt diesen Ansatz mit dem Hot-Spot-Konzept [10].

Ein zweiter Nachteil der Just-In-Time Compilierung, der sich besonders bei leistungsschwächeren System auswirkt, ist der hohe Speicherplatzverbrauch. Zusätzlich zum Bytecode, den ohnehin nötigen JVM-Datenstrukturen und dem erzeugten Maschinencode muß noch der (im Vergleich zum Interpreter) relativ große JIT-Compiler selbst im Speicher gehalten werden, ebenso die während der Compilierung dynamisch anfallenden Datenmengen (Datenflußgraphen etc.).

Die dritte Möglichkeit der JVM-Implementierung besteht in der Nutzung eines speziellen JVM-Prozessors (z.B. [9] oder [6]). Damit ist eine sehr effiziente Ausführung des Bytecodes erreichbar. Bislang scheint die Akzeptanz dieser Lösung allerdings nicht besonders groß zu sein. Dies mag darin liegen, daß eine neue CPU für den Hard- und Softwareentwickler stets Einarbeitungszeit in die neue Entwicklungsumgebung (mit allen ihren Fehlern) bedeutet. Alte HW-System müssen umgeändert werden, existierende Treiber für Hardwarekomponenten müssen neu geschrieben werden.

In dieser Arbeit wird eine weitere Möglichkeit zur Abarbeitung von Bytecode beschrieben, die die Ausführungsgeschwindigkeit von JIT-Systemen bietet, aber eine wesentlich schnellere Compilierung ermöglicht: Der Bytecode wird dabei von einer Spezialhardware, einem rekonfigurierbaren FPGA (Field Programmable Gate Array) in Maschinencode übersetzt. Dieser kann dann von einem Standardprozessor (z.B. x86, Alpha etc.) ausgeführt werden. Dabei wird das Augenmerk auf die sehr schnelle Compilierung bei 'guter' Ablaufgeschwindigkeit (>40-60% C/C++ Code) und nicht auf höchste Effizienz (80-100% von C/C++ Code) des Compilats gelegt.

Dieser Ansatz verspricht besonders bei Mid-Range-CPUs gute Ausführungsgeschwindigkeit bei niedrigen Speicher- und CPU-Anforderungen, dies wird in Abschnitt 2 noch näher ausgeführt. Damit ist es auch möglich, die JVM auf Systemen einzusetzen, bei denen bislang der Compiliationsvorgang das beschränkende Element für eine sinnvolle Nutzung ist. Dieses Ergebnis würde dann auch den Aufwand der Entwicklung rechtfertigen [2].

Allerdings tauchen beim Entwurf dieses Hardware-Systems eine Reihe von Anforderungen und Schwierigkeiten auf, die teilweise auf zunächst gravierende Beschränkungen des JIT-Systems führen, aber ausschließlich dazu dienen, die Flexibilität des JIT-Systems zu erhöhen. Die dazu führenden Überlegungen

werden in Abschnitt 3 dargelegt. Ein Verfahren, das diese Anforderungen erfüllt, wird in Abschnitt 4 vorgestellt. Es besteht aus einer Kombination teilweise schon bestehender Verfahren, die für besonders einfache Implementierbarkeit in HW verändert und zusammengestellt wurden. Die Vor- und Nachteile dieses Ansatzes bezüglich der Ausführungseffizienz werden in Abschnitt 5 anhand eines in SW implementierten Modells erläutert.

2 Prinzipielle Vorteile eines HW JIT-Compilers

Der hier besprochene JIT-Compiler in einem FPGA ist hauptsächlich für Mid-Range-Anforderungen in eingebetteten Systemen oder Netzwerk-PCs gedacht. Diese sind üblicherweise mit CPUs ausgestattet, die eine oder mehrere Architekturgenerationen gegenüber aktuellen Desktop/Server-CPUs zurückliegen und um den Faktor 2-20 langsamer getaktet werden. Hinzu kommen langsamere Speicheranbindung (z.B. kein/wenig Cache, schmalere Datenbusse) und weniger Ressourcen (z.B. RAM). Während an sich sehr schnelle JIT-Compiler wie z.B. CACAO [4][7] auf einer 21164-CPU mit 500MHz ca. 500000-700000 JVM-Befehle pro Sekunde übersetzen können, würde sich dies auf älteren CPUs auf (geschätzt) 5000-50000/s reduzieren. Gerade bei wenig Speicher wäre eine schnelle Übersetzung nützlich, wenn bereits compilierter Code aus Speicherplatzmangel gelöscht wurde und wieder neu compiliert werden muß.

Um dies zu erreichen, bietet sich eine JIT-Unterstützung an, die z.B. als zusätzliche Hardware in das System eingebunden werden kann. Ist diese als rekonfigurierbares FPGA realisiert, ist es auch denkbar, das FPGA nach der JIT-Compilation neu zu konfigurieren und für andere Zwecke zu benutzen (z.B. Datenerfassung). Damit erhöht sich der Zusatznutzen des FPGAs. Auf der Softwareseite ist gegenüber der Interpreterlösung kaum erhöhter Aufwand nötig, da nur kleine Wrapperfunktionen für die JIT-Zusatzfunktion erforderlich sind.

Ein weiterer Gesichtspunkt der FPGA-Nutzung wäre neben der reinen JIT-Funktionalität die Rekonfiguration des FPGAs als Garbage-Collector, Classloader und Parser (Übertragen der Informationen im class-File in die internen Datenstrukturen). Dies würde besonders bei Little-Endian- und RISC-Prozessoren zu einem stark beschleunigten Klassenladen führen, da die Daten im class-File im Big-Endian-Format und nicht auf Wortgrenzen (d.h. unaligned) abgelegt sind.

Da FPGAs in den erwähnten Mid-Range-System wesentlich weiter verbreitet sind[2], als in High-End-Systemen, ist es denkbar, daß die Nutzung des FPGAs als JIT-Compiler auch gar keinen zusätzlichen Hardware-Aufwand bedeutet.

Somit scheint der Einsatz eines FPGAs als JIT-Compiler im Prinzip sinnvoll und untersuchenswert.

3 Anforderungen an den JIT-Algorithmus für HW

Die Grundlage der JIT-Compilation von JVM-Bytecode in Maschinencode ist die Übersetzung der stackbasierten JVM-Befehle auf eine registerbasierte Archi-

[2] besonders in eingebetteten Systemen als kostengünstiger Ersatz von ASICs

tektur, wie sie alle heutigen Standard-CPUs besitzen. Als einfachste Möglichkeit bietet sich hierzu eine einfache tabellengesteuerte Übersetzung an. Diese ist relativ schnell und benötigt nur Speicherplatz für den erzeugten Maschinencode und die Tabellen. Allerdings ist der erzeugte Code umfangreich und ineffizient, da die Stackadressierung immer noch explizit vorhanden ist. Es können mehr als 50% des erzeugten Codes zur Stacknachbildung dienen.

Die Effizienz des Tabellen-Übersetzers läßt sich stark verbessern, wenn zusätzlich eine Peephole-Optimierung durchgeführt wird. Zur Eliminierung mehrerer Stackebenen sind größere Code-Ausschnitte zu betrachten, was aber zusätzlichen CPU-Aufwand bedeutet.

Andere Ansätze bringen daher im Verhältnis bessere Effizienz, sie basieren üblicherweise auf den gebräuchlichen Compilertechniken (z.B. Graph Colouring). Sehr gute Ergebnisse liefert z.B. das Übersetzungsschema von CACAO, das den JVM-Code zunächst in einen registerbasierten Zwischencode übersetzt und darauf diverse Optimierungen anwendet (einfache Registerallokation etc.). Erst anschließend wird der Zwischencode in Alpha-Code übersetzt.

Sehr viele JVM-Implementierungen (insbesondere kommerziell erhältliche) sind an einen bestimmten Prozessor (x86 bei TYA, Netscape/Symantec oder Microsoft, Alpha bei CACAO) gebunden, der Rest (z.B. kaffe) hat für JIT-Compilierung relativ schlechte Ergebnisse. Hinzu kommt, daß die eigentliche Codeerzeugung meist sehr eng mit der Zielarchitektur verzahnt ist, so daß eine Portierung auf eine andere CPU zwar möglich, aber teilweise sehr aufwendig ist. Als Beleg dafür mag bei kaffe die sehr unvollständige und ineffiziente Unterstützung anderer CPUs als der x86-Architektur gelten.

Soll nun ein JIT-Compiler in HW ablaufen, so ist die Umsetzung des Algorithmus in HW an sich schon mit viel Arbeit verbunden. Daher ist es für die Flexibilität in der Anwendung notwendig, die Übersetzung so gut wie möglich von der Ziel-CPU zu entkoppeln. Im Idealfall kann für alle Zielarchitekturen dieselbe HW ohne Änderung (z.B. Synthese in VHDL) benutzt werden, alle architekturspezifischen Daten sind z.B. in Tabellen abgelegt. Dazu ist es nötig, eine möglichst gemeinsame Basis aller zu unterstützenden Architekturen zu finden.

Wird als JIT-Ausführungsbasis eine FPGA-Architektur gewählt, kommen weitere Einschränkungen hinzu: Zwar bieten rekonfigurierbare FPGAs eine sehr weitgehende Flexibilität bezüglich ihrer internen Funktionen, allerdings geht dies auch zu Lasten der erzielbaren Taktfrequenzen bei komplexeren Entwicklungen. Obwohl inzwischen FPGAs mit 1 Million Gatterfunktionen erhältlich sind, steigen die Preise bei (zur Zeit) $>$ 20000 Gattern stark an. Damit ergibt sich, daß der JIT-Algorithmus so einfach ('hardware-freundlich') wie möglich sein muß, um möglichst hohe Taktfrequenzen und niedrige Zusatzkosten zu erreichen. Damit sind aufwendige Optimierungsverfahren (z.B. Graph-Colouring) bereits ausgeschlossen, rein lineare Verfahren wären dagegen ideal.

Speicher in den FPGAs ist meist begrenzt und feingranular, so daß größere Datenmengen ($>$ einige KB) in externen Speichern implementiert werden müssen. Damit ergibt sich für den JIT-Algorithmus die weitere Einschränkung, daß externer Speicher möglichst nur sequentiell angesprochen werden sollte, da-

mit Prefetching möglich wird. Dies hat weitere Konsequenzen, wie im folgendem Beispiel der Compilierung von Subroutinenaufrufen erläutert wird.

Die JVM verwendet einen Aufrufmechanismus, der dem von Pascal ähnlich ist: Der erste Parameter wird zuerst auf den Stack gelegt, die aufgerufene Routine entfernt beim Rücksprung alle ihre Parameter vom Stack. Im Gegensatz dazu steht die C-Aufrufkonvention, wo der erste Parameter als letztes auf den Stack gelegt wird und die Parameter vom Aufrufer entfernt werden. Beide Methoden sind gleichwertig und im Prinzip auch gleich performant in Compiliation und Ausführung. Soll jedoch der Methodenaufruf von einer JIT-HW übersetzt werden, hat die C-Version einen versteckten Nachteil: Um nach dem Aufruf die Parameter wieder zu entfernen, muß zunächst die Anzahl der Parameter der aufgerufenen Methode gelesen werden. Dies kann bei einer Anbindung des FPGAs am PCI-Bus zu Latenzzeiten von mehreren 100ns führen und damit die Übersetzung enorm verlangsamen. Die Pascal-Variante weist diesen Nachteil nicht auf, da dort die Anzahl der Parameter nur ein einziges Mal beim Start der JIT-Compilation der Methode gelesen werden muß.

Eine weitere Anforderung an den JIT-Algorithmus ist die universelle Einsetzbarkeit der JIT-HW in beliebigen virtuellen Maschinen (z.B. als JDK-Plugin oder in einer eigenen Cleanroom-Implementation). Damit (und wegen der oben angesprochenen Speicherproblematik) darf der JIT-Compiler keinen direkten Zugriff mehr auf JVM-Datenstrukturen haben. Damit wird auch Großteil der während der JIT-Phase eigentlich möglichen Resolving-Funktionen bei den komplexeren JVM-Funktionen (z.B. invoke*) erschwert bzw. unmöglich gemacht. Dies führt dazu, daß diese Funktionen wirklich erst während der Laufzeit ausgeführt werden können.

Zusammenfassend unterliegt der zu suchende JIT-Algorithmus extremen, aus Softwaresicht möglicherweise willkürlich einschränkenden Anforderungen. Ob und welche dieser Anforderungen und Beschränkungen nicht umgehbare Effizienznachteile mit sich bringen, muß noch geklärt werden.

4 Das gewählte JIT-Konzept

Das Jiffy-Konzept, was obige Anforderungen gut erfüllt, ist in Bild 1 gezeigt. Die gezeigten Abläufe sollen in der endgültigen Implementierung in einem FPGA ablaufen, das z.B. an den Systembus angeschlossen ist. Das Bild deutet auch die Ablageorte der für die Übersetzung benötigten Daten an: Daten, auf die ein kontinuierlicher Zugriff erfolgen kann (Streaming) können im System-RAM liegen, größere Tabellen müssen aus Geschwindigkeitsgründen direkt am FPGA (External FPGA RAM) angebunden sein. Kleinere Tabellen sind im FPGA (Internal FPGA RAM) abgelegt. Zwischenergebnisse können sowohl im Systemspeicher als auch im externen FPGA-Speicher liegen, letzteres vergrößert und verteuert aber auch die Kosten des nötigen Speichers.

Das Konzept besteht prinzipiell aus dem schon angesprochenen Tabellen-Übersetzer, allerdings mit einigen Optimierungen verfeinert. Der JVM-Bytecode durchläuft zunächst eine Analyse- und Optimierungsphase (letztere ist optional

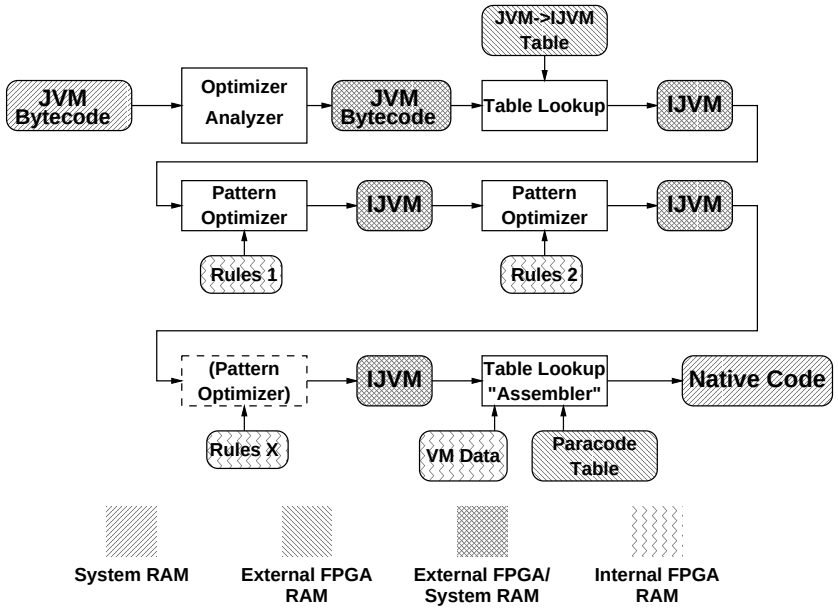

Abbildung1. Ablauf der Übersetzung

und wird später näher beschrieben). Diese bestimmt alle Sprungziele und damit die Grundblöcke, in denen optimiert werden darf. Zusätzlich ist noch eine Nutzungsstatistik der lokalen Variablen vorgesehen, dies ermöglicht später die Verlagerung dieser Variablen vom Stack in Register (noch nicht implementiert).

Der optimierte Bytecode wird anschließend tabellengestützt in einen Zwischencode (Intermediate JVM-Code, IJVM) übertragen, der alle Funktionen der JVM beherrscht, allerdings auf expliziten Stackoperationen mit push und pop basiert und zunächst nur 3 Register besitzt[3]. Die Integer bzw. Gleitkomma-Nutzung der Register ist implizit in den Befehlen enthalten. Über diesen erzeugten Zwischencode laufen mehrere Peephole-Optimierungen, die nach einem einfachen Mustererkennungs-System mit mehreren Ersetzungsregeln arbeiten. Wird ein Muster erkannt, werden die Befehle in der Pipeline (das „Peephole“) entsprechend der Regel verändert. Beispielsweise wird ein PUSH/POP-Paar durch zwei NOPs ersetzt. Ist keine Regel anwendbar, wird das Peephole weitergeschoben. Um Speicherplatz zu sparen, kann die Optimierung in-place angewendet werden, NOPs werden beim Verlassen der Pipeline nicht zurückgeschrieben. Die Implementierung dieses Systems ist in verschiedenen Effizienz-Stufen möglich, so daß eine gute Balance zwischen Geschwindigkeit der Erkennung (im Idealfall nur 1 Takt zur Ausführung aller Regeln pro Schritt) und Gatterverbrauch wählbar ist.

Die Peephole-Optimierung erzeugt Code, der im allgemeinen nur noch selten auf den Stack zugreift, der im Prinzip nach wie vor vorhanden ist. Die Parameterübergabe bei Methodenaufrufen erfolgt aber nach wie vor über den Stack, da

[3] JVM-Arrayzugriffe haben 3 Argumente

dies die einzige portable Art ist. Zusätzlich ist dabei kein Wissen über die Anzahl und Typ der Parameter nötig. Es ist aber in späteren Ausbaustufen möglich, die Parameterübergabe in Registern als zusätzliche Optimierung einzufügen.

Allerdings besitzt die Peephole-Optimierung den Nachteil, daß einige häufige JVM-Befehlskombinationen (z.B. dcmplt+ifeq, Schleifen zur Arrayinitialisierung) nicht besonders gut übersetzt werden können. Daher wird vor der Übersetzung in Zwischencode der JVM-Code optimiert. Dazu wird ebenfalls das bereits existierende Mustererkennungs-System benutzt. Dieses ersetzt einige Befehlskombinationen durch spezielle nicht-Standard-JVM-Befehle, die schneller abgearbeitet werden können (z.B. dcmplt+ifeq $\rightarrow$ if_dcmplt, erspart die aufwendige *sgn*-Funktion).

Nach der Optimierung des Zwischencodes muß dieser in nativen Code übersetzt werden. Da sich durch die Optimierungen Registernummern verändert haben und auch Parameter vom Bytecode in den Maschinencode übertragen werden müssen, kann dies nicht mehr mit einem einfachen Tabellenübersetzer geschehen. Daher sind in den Tabellen (im Bild 'Paracode' genannt) nur Schablonen der Befehle abgelegt und zusätzlich mit Informationen versehen, wo welche Register in das Binärmuster 'eingepatcht' werden müssen. Diese Bitmanipulationen sind zwar auf herkömmlichen CPUs zeitaufwendig, sind in 'echter' Hardware aber sehr einfach auszuführen. Zusätzlich kann der Assembler, der im FPGA abläuft, auch einfache Rechenoperationen ausführen. Dies wird z.B. für die Berechnung des Stackoffsets aus der Nummer einer lokalen Variable benutzt, wobei das Ergebnis in das entsprechende Displacement-Feld des Assemblerbefehls geschrieben wird.

Durch die tabellengestützte Übersetzung ist es möglich, den JIT-Compiler völlig von der Zielarchitektur zu entkoppeln. Zur Trennung des JIT-Compilers von der VM ist es aber auch nötig, daß der JIT-Compiler keine weitergehenden Resolving-Funktionen unternimmt. Es ist als einziges nur nötig, alle nicht typisierten objektorientierten-Befehle (wie ldc, get/put*, invoke*) intern mit dem Rückgabetyp zu versehen. Alle anderen Funktionen (Auflösung der Methoden etc.) geschehen erst zur Laufzeit über Stub-Routinen.

5 Erste Performance-Ergebnisse einer prototypischen SW-Implementierung

Um das beschriebene Konzept auszutesten, wurde begonnen, es zunächst vollständig in Software zu implementieren, allerdings wurde dabei streng darauf geachtet, die Abläufe so zu beschreiben, daß sie einfach in HW (d.h. in synthetisierbares VHDL) umzusetzen sind. Durch die Nähe des C-Codes zur späteren HW-Struktur ist es auch in gewissen Grenzen durch Instrumentierung des C-Codes möglich, Schätzungen der benötigten Takte der Übersetzung zu erhalten. Damit kann die zu erwartende Übersetzungsgeschwindigkeit bestimmt werden.

Als Zielplatformen wurden zwei Vertreter der CISC/RISC Architekturen gewählt:

- CISC: x86
  6 frei nutzbare 32bit Register, daher (notgedrungen) push/pop Operationen sehr gut optimiert. Neuere Prozessoren (Pentium, Pentium II/III) mit ausgefeilten internen Optimierungen wie Out-Of-Order-Execution, komplexe Branch Prediction etc.). Literale können direkt mit 32bit im Code untergebracht werden. FPU-Stack mit 8 Einträgen.
- RISC: Alpha 21164
  28 frei nutzbare 64bit Register, reagiert trotz 3-stufigem Cache sehr empfindlich auf Speicherzugriffe. Byte/Wordzugriffe (z.B. Java Boolean/Char Arrays) führen zu signifikanten Leistungseinbußen. Keine Out-Of-Order-Execution, einfache Branch Prediction. Nur 16bit Literale mit einem Befehl ladbar. FPU mit 32 Registern.

Als Grundlage der JVM wurde Suns JDK1.2 verwendet, das eine standardisierte JIT-Plugin-Schnittstelle [11] bietet. Damit ist zunächst nur die unmittelbare JIT- und Laufzeitumgebung zu entwickeln. Ein Nachteil ist doch recht ineffiziente Implementierung von Objekten (tw. mehrere Indirektionen nötig[4]) und der Native-Funktionen. Es wurden jeweils die Linux-Versionen des JDK1.2 für x86 und Alpha verwendet. Als Performance-Vergleich diente auf dem x86 von gcc/egcs erzeugter Code, der einfache JIT-Compiler TYA1.3 für JDK1.2, der von Sun entwickelte und bei JDK1.2-Linux mitgelieferte JIT-Compiler (libsunwjit.so), der JIT-Compiler von kaffe V1.0b4, der Symantec-JIT-Compiler des Netscape Communicator 4.51 (JIT-Version 210.065) und der im Microsoft Explorer 4.0 integrierte JIT-Compiler. Auf der Alpha-CPU wurden zum Vergleich gcc/egcs-Code und CACAO herangezogen.

Der gemessene Speedup wird in den folgenden Tabellen auf das Jiffy-System bezogen, da so die Vor- und Nachteile dieser JIT-Implementation im Vergleich deutlicher werden.

Da noch nicht alle Features in Jiffy (besonders die VM-Anbindung) implementiert waren, wurden zur Messung folgende Mikrobenchmarks verwendet:
sieve: Primzahlenberechnung, testet vor allem Arrayzugriffe.
sin: Aufruf von Math.sin(), testet Native-Aufrufe.
fib: Rekursive Fibonacci-Berechnung, testet Klassenmethodenaufrufe (static)
s.len: Stringlängenberechnung, testet virtuelle Methoden und Instanzvariablenzugriffe.

Im Jiffy-System waren als Optimierungen 3 Peepholeoptimierungen (Fenstergröße: max. 6 Befehle) mit 6 Ersetzungsregeln für die erste Phase und 12 Regeln für die zweite und dritte Phase eingeschaltet. Die oben erwähnten Optimierungen auf JVM-Bytecode-Ebene wurden im Jiffy-JIT nicht benutzt, da sie bei den kleinen Testprogrammen die Ergebnisse unverhältnismäßig verbessert hätten.

Die Messungen auf dem Pentium II zeigen, daß Symantec-JIT und MSIE-JIT nah an der Leistung von direkt compilierten C/C++-Code sind. Jiffy liegt dabei zwischen ca. 33-75%. Beim *sin*-Test fällt im Vergleich zu kaffe der große Overhead von JDK1.2 auf. Könnte die Native-Funktion direkt aufgerufen werden, wäre Jiffy ca. 60% schneller, daher kann der starke Rückstand in diesem Test

[4] Wurde bei Hotspot geändert.

Test	JDK/Jiffy	JDK/TYA	JDK/SunJIT	Kaffe1.0b4	NS4.51	MSIE4.0	gcc -O3
sieve	1.00	0.27	0.52	0.66	1.84	2.76	2.76
sin	1.00	0.49	0.68	2.30	(32.0)	4.32	3.01
fib	1.00	0.32	0.65	0.31	1.21	1.32	1.32
s.len	1.00	1.35	0.88	0.42	1.35	1.70	2.67

Tabelle1. Normierte Ergebnisse auf der x86-Architektur (P II)

Test	JDK/Jiffy	cacao	gcc -O3
sieve	1.00	2.21	3.70
sin	1.00	1.85	2.17
fib	1.00	1.47	2.55
s.len	1.00	3.75	3.70

Tabelle2. Normierte Ergebnisse auf der Alpha-Architektur (21164)

gegenüber dem MSIE nur zu einem Teil dem Übersetzungskonzept zugewiesen werden[5]. TYA kann Funktionen inlinen und führt ein Großteil des Resolving bereits bei der Compilation durch, daher ist es bei *s.len* etwas schneller, während es sonst (wie auch der SunJIT) wesentlich langsamer als Jiffy ist. Damit zeigt sich, daß der Performanceverlust durch das 'wirkliche' JIT-Resolving in Jiffy und der dadurch auch bei schon aufgelösten Funktionen vorhandene Overhead wesentlich schwächer ausfällt, als zunächst erwartet. Wenn auch die Ergebnisse im Vergleich zu anderen Linux-JITs sehr gut ausfallen, ist doch sichtbar, daß weitergehende Optimierungen noch starke Verbesserungen liefern würden.

Auf dem Alpha sind die Ergebnisse im Vergleich zu gcc und CACAO durchweg schlechter (ca. 30-45% von C/C++-Code), da (auch aufgrund der ineffizienten JDK-Strukturen) sehr viele Speicherzugriffe auftreten, die die Ausführung stark bremsen. Dies fällt besonders bei *sieve* (Boolean Arrays) und *s.len* (Instanzmethoden und -Variablen) auf. Allerdings sollte sich besonders das *sieve*-Ergebniss verbessern, wenn die oben beschriebene Verlagerung von lokalen Variablen in Register ausgeführt würde. Eine Optimierung von Leaf-Prozeduren wäre ebenfalls relativ einfach möglich und effizient, da im allgemeinen Fall 13 Alpha-Befehle allein für Prozeduranfang und Ende benötigt werden.

Die reine Übersetzungsgeschwindigkeit in SW beträgt auf dem P II/233MHz ca. 190000 JVM-Opcodes/s, auf dem Alpha(500MHz) ca. 300000/s. Allerdings wurde das SW-Modell keinerlei Optimierungen unterworfen, eine Steigerung durch die Verzahnung der Phasen ist möglich. Aufgrund der durch das SW-Modell gelieferten Daten kann bei einer Implementierung in HW ohne Pipelining von ca. 40-50 Takten pro JVM-Instruktion ausgegangen werden, bei moderaten 30MHz Takt also ca. 600000-750000 Instruktionen pro Sekunde. Damit ist auch für langsame Prozessoren eine sehr schnelle Compilation zu erwarten.

6 Ausblick

Die in dieser Arbeit gezeigten ersten Ergebnisse der SW-Implementierung lassen den Einsatz von FPGAs als JIT-Compiler sinnvoll erscheinen. Das einfache

[5] Der Symantec-JIT macht anscheinend Invarianten-Eliminierung.

Übersetzungskonzept erzeugt (schon in SW) sehr schnell relativ effizienten Code, wenn auch die Geschwindigkeit des erzeugten Codes (noch) nicht mit kommerziellen JIT-Compilern mithalten kann. Die Implementierung an sich ist so portabel, daß die Portierung auf eine neue CPU (d.h. die Erstellung des Assemblercodes) in wenigen Tagen abgeschlossen sein dürfte[6], ohne Änderungen an der HW vorzunehmen. Dennoch muß sich die Tauglichkeit des Systems an einer HW-Implementierung messen lassen. Daher ist die Umsetzung der besprochenen Algorithmen in ein FPGA der nächste Schritt. Besondere Beachtung muß dabei dem bezüglich Geschwindigkeit und Gatterverbrauch effizienten Entwurf der Mustererkennung gewidmet werden. Ziel ist es dabei, aus den Regeln direkt VHDL-Code zu erhalten, aus dem die FPGA-Logik generiert wird.

Weitere Forschungsarbeiten liegen in zusätzlichen Code-Optimierungen, die sich nahtlos in das Konzept einpassen lassen, und der Implementierung von VM-Funktionen in das FPGA, um das JIT-System abzurunden.

Literatur

1. H. Böhme, G. Telkamp, U. Golze. Eine JavaVM für eingebettete 8-Bit-Systeme. in A. Hegenhan, W. Rosenstiel, Tagungsband GI/ITG-Workshop Java und Eingebette Systeme, FZI Karlsruhe, September 1998
2. M. Edwards. Software Acceleration Using Coprocessors: Is it Worth the Effort? Proceedings of the 5th International Workshop on Hardware/Software Codesign, IEEE Computer Society Press, 1997.
3. D. Flanagan. Java in a Nutshell. O'Reilly & Associates, Inc., 1996
4. R. Grafl. CACAO - Ein 64bit-JavaVM-Just-In-Time-Compiler. Diplomarbeit, Technische Universität Wien, Institut für Computersprachen, 1997.
5. R. Haratsch. Spezifikation und Generierung eines Übersetzters von Java-Byte-Code nach Intel-Code. Diplomarbeit, Technische Universität München, Institut für Informatik, 1998.
6. J. Horch. A Simple Runtime System for a Hardware-Oriented Implementation of the Java Machine. in A. Hegenhan, W. Rosenstiel, Tagungsband GI/ITG-Workshop Java und Eingebette Systeme, FZI Karlsruhe, September 1998
7. A. Krall. CACAO - Eine effiziente JavaVM Implementierung. in A. Hegenhan, W. Rosenstiel, Tagungsband GI/ITG-Workshop Java und Eingebette Systeme, FZI Karlsruhe, September 1998
8. T. Lindholm, F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1996
9. Sun Microsystems, Inc. Sun Microelectronics' picoJava 1 Posts Outstanding Performance. in Press-Release Nov.18, 1996
10. Sun Microsystems, Inc. The Java Hotspot performance engine architecture. Whitepaper, http://www.javasoft.com/products/hotspot/whitepaper.html
11. F. Yellin. The Java Native Code API. Sun Microsystems, Inc., 1996

[6] Die Alpha-Umsetzung war in einem Tag abgeschlossen