Leistungssteigerung der Heap-Anbindung von Java-Bytecode-Mehrkernprozessoren

Martin Zabel, Andrej Olunczek, Rainer G. Spallek
{martin.zabel, rainer.spallek}@tu-dresden.de andrej@olunczek.de
Institut für Technische Informatik, Technische Universität Dresden, Dtl.

Kurzfassung

Die Java-Plattform ist vorteilhaft für die schnelle Entwicklung komplexer Software. Für die Ausführung des Java-Bytecodes auf eingebetteten Systemen eignen sich insbesondere Java-Bytecode-Mehrkernprozessoren, die zudem die auf Thread-Ebene vorhandene Parallelität eines Java-Programms ausnutzen. In dieser Arbeit werden Maßnahmen zur Entlastung der Heap-Speicherschnittstelle untersucht, um einen höheren Speed-Up zu erzielen. Der vorgestellte Objekt-Cache mit integriertem TLB erreicht dabei eine Verdopplung der Verarbeitungsleistung. Die Implementierung zeigt außerdem ein sehr gutes Nutzen-Kosten-Verhältnis.

1 Einleitung

Die Java-Plattform bietet viele Vorteile für die schnelle Entwicklung komplexer Software, darunter: durchgängige Objektorientierung, kompakter plattformunabhängiger Java-Bytecode, automatische Speicherverwaltung, Sicherheitskonzepte und eine umfangreiche Unterstützung für die Parallelisierung von Programmen auf Thread-Ebene. Für die Ausführung des Java-Bytecodes auf eingebetteten Systemen eignen sich insbesondere Java-Prozessoren, die den Bytecode als nativen Befehlssatz unterstützen. Diese bieten außerdem den Verzicht auf unterliegendes Betriebssystem, kompakte Programmspeicher und eine vereinfachte Laufzeitanalyse für Echtzeitanwendungen.

Zur Steigerung der Verarbeitungsleistung von Java-Prozessoren wurden verschiedene Konzepte für Mehrkern-Java-Bytecode-Architekturen vorgestellt, die die auf Thread-Ebene vorhandene Parallelität eines Java-Programms ausnutzen: JopCMP [PS08], jamuth [Uhr09], REALJava [TSP08] und SHAP [Zab12]. Als besonders effizient erweist sich dabei das Konzept des SHAP-Mehrkernprozessors, bei dem selbst Programme mit überdurchschnittlich vielen Speicherzugriffen einen Speed-Up von bis zu 8 erzielen. Vorteilhaft ist bei diesem Konzept, dass durch jeweils einen lokalen Stack-Speicher und einen Methoden-Cache pro Kern bereits ein Großteil der Speicherzugriffe abgefangen werden kann. Für die verbleibenden Zugriffe auf den gemeinsamen Heap-Speicher ist demnach ein zentraler Speicher ausreichend.

In dieser Arbeit sollen weitere Maßnahmen zur Entlastung der Heap-Speicherschnittstelle untersucht werden, um einen höheren Grad an Parallelverarbeitung (Speed-Up) zu erzielen. Es werden speziell Ansätze untersucht, die sich die Eigenschaften der Java-Virtual-Machine (JVM) zunutze machen. Berücksichtigt wird dabei auch das Nutzen-Kosten-Verhältnis.

2 Bisherige Arbeiten

Zwecks Beschleunigung der Heap-Speicherzugriffe wird für objektorientierte Prozessorarchitekturen im Allgemeinen der Einsatz eines Objekt-Caches vorgeschlagen (z. B. [VR00, WSW06]), die analog eines Daten-Caches Teile oder das gesamte Objekt in einer Cache-Zeile zwischenspeichern. Für eine genauere Berechnung der maximalen Ausführungszeit (engl.: worst-case execution time) schlägt Huber et. al. zudem separate Caches für verschiedene Datenbereiche vor [HPS10]: einen Standard-Daten-Cache für Daten an festen (statischen) Adressen (z. B. Klasseninformationen), nebst einem Objekt-Cache für Daten an dynamischen Adressen (z. B. Objekte).

Bei der Cache-Adressierung spielt die Art und Weise der Objektadressierung eine wesentliche Rolle. Die Java-Mehrkernprozessoren JOP und SHAP setzen eine indirekte Adressierung ein, bei

Tabelle 1: Häufigste Verursacher und Offets bei Heap-Zugriffen der JemBenchmarks

	Verursacher #1		Verursacher #2		Verursacher #3		Offset #1		Offset #2		Offset #3	
AES	*aload	31%	inv.special	15%	getstatic	11%	1	21%	14	7%	-5	5%
Bubblesort	*aload	79%	*astore	20%	io_write	<1%	1	50%	4	3%	5	3%
Kfl	getstatic	47%	inv.static	16%	*aload	13%	1	12%	10	7%	-5	5%
Lift	getfield	45%	*astore	21%	*aload	17%	1	26%	-2	17%	-5	17%
MatrixMul	*aload	83%	getfield	15%	*astore	2%	1	42%	-2	13%	-3	2%
NQueens	inv.static	40%	*aload	22%	*astore	12%	137	40%	1	17%	4	12%
Sieve	*astore	45%	getfield	33%	*aload	22%	-2	33%	1	33%	53	1%
UdpIp	*aload	37%	getstatic	14%	*astore	14%	1	30%	-3	8%	0	6%

der die Objektreferenz einer virtuellen Adresse entspricht und auf einen Eintrag in einer Objekttabelle verweist [HPS10, Zab12]. Die Einträge in der Objekttabelle enthalten die aktuelle physikalische Adresse des Objekts im Speicher und lassen sich somit zentral vom Garbage-Collector (GC) aktualisieren. Beim „jamuth"-Mehrkernprozessor hingegen enspricht die Objektreferenz direkt der physikalischen Objektadresse [UU09]. Eine Kompaktierung des Speichers nach der Objektfreigabe ist hier nicht implementiert. Beim REALJava-Prozessor können die Java-Prozessorkerne nicht direkt auf Objekte zugreifen, sondern müssen dazu eine Software-Implementierung auf einem General-Purpose-Prozessor bemühen [TSP08].

Bei der indirekten Objektadressierung liegt die Objekttabelle ebenfalls im Hauptspeicher und erfordert somit einen zusätzlichen Lesezugriff auf den von allen Kernen gemeinsam genutzten Speicher. Zwecks Vermeidung der dadurch entstehenden Latenz als auch der dafür benötigten Speicherbandbreite ist im SHAP-Mehrkernprozessor ein Translation-Lookaside-Buffer (TLB) mit 2 Zeilen für die Zwischenspeicherung der physikalischen Objektadresse integriert [Zab12]. Die Nachteile können ebenfalls mit einem virtuell adressierten Objekt-Cache vermieden werden, bei dem die Objektreferenz für die Cache-Adressierung genutzt wird [VR00]. Bei einem Cache-Hit entfällt hier die Auflösung der Objektreferenz zu einer physikalischen Adresse. In Kombination mit einem TLB, kann bei einem Cache-Miss zumindest die Adressauflösung beschleunigt werden [VR00]. Alternativ zu einer zentralen Objekttabelle ist auch der Einsatz von verteilten Tabellen in den einzelnen Ebenen der Speicherhierarchie möglich [LJW+09].

Bei verteilten Objekt-Caches ist deren Kohärenz sicherzustellen. Ein Vorteil der JVM ist dabei, dass nicht bei jedem Speicherzugriff eine Synchronisation mit dem Heap-Speicher notwendig ist, sondern nur bei Betreten eines kritischen Abschnitts (Bytecode `monitorenter`) oder Zugriff auf eine mit `volatile` gekennzeichnete Variable [LY99].

3 Analyse

Die Analyse der Heap-Zugriffe erfolgte anhand der Benchmark-Suite JemBench [SPU10] (Version 2.0), die explizit für eingebettete Java-Prozessoren vorgesehen ist. Die Auswertung von Laufzeitdaten (Trace) dieser Benchmarks anhand der SHAP-Mehrkernarchitektur zeigt, dass die meisten Objektzugriffe lesende Zugriffe auf Arrays (`*aload`) und Instanzvariablen (`getfield`) sind (Tab. 1). Dabei sind bereits die impliziten Lesezugriffe auf die Array-Länge berücksichtigt, um den Array-Index zu überprüfen. 80% der Objektzugriffe konzentrieren sich außerdem auf einen Arbeitssatz von bis zu 6 Objekten; dies entspricht zeitlicher Lokalität. Häufig zugegriffen wird außerdem nur auf die ersten benutzerdefinierten Objektvariablen (Offset -2 und 1, Offset -1 und 0 enthalten Systeminformationen); räumliche Lokalität ist daher nur bedingt vorhanden. Den größten Nutzen im Verhältnis zu benötigter Chipfläche und Verlustleistung lässt daher ein kleiner vollassoziativer Cache je Prozessorkern erwarten, der anhand einer Simulation mit den Trace-Daten weiter untersucht wurde.

Eine kostengünstige Variante wäre, nur unveränderliche (außer Objektinitialisierung) Daten zwischenzuspeichern, da somit keine Kohärenzlogik notwendig ist. Dies umfasst i. Allg. aber nur

Tabelle 2: Bedarf an FPGA-Ressourcen für $n \leq 15$ Kerne auf einem Virtex-5

Ressource	2-Zeilen-TLB	4-Zeilen-AOC	8-Zeilen-AOC
Register $\approx$	$1975 + 1447 \cdot n$	$1978 + 1501 \cdot n$	$1978 + 1580 \cdot n$
6-Input-LUTs $\approx$	$2784 + 2839 \cdot n$	$2848 + 3011 \cdot n$	$2849 + 3076 \cdot n$
18-kBit-BRAMs =	$1 + 2 \cdot n$	$1 + 2 \cdot n$	$1 + 2 \cdot n$
36-kBit-BRAMs =	$1 + 3 \cdot n$	$1 + 3 \cdot n$	$1 + 3 \cdot n$
18x18-MULs =	$2 + 3 \cdot n$	$2 + 3 \cdot n$	$2 + 3 \cdot n$
IOB FFs =	87	87	87

die Array-Länge (Offset 1) und führte nur beim BubbleSort zu einer signifikanten Einsparung von 33% der Speicherzugriffe, gefolgt von Sieve mit 20% und MatrixMul mit 17%.

Für die Zwischenspeicherung auch veränderlicher Daten bietet sich ein Write-Through-Cache an. Zur Gewährung der Kohärenz muss hier zu den o. g. Zeitpunkten nur der Cache des auslösenden Kerns invalidiert werden. Die Simulation verschiedener Cache-Größen mit der aktuell eingesetzten SRAM-Speicheranbindung (32 Bit, 1 Takt Zykluszeit durch Pipelining) bestätigt die obige Beobachtung zur Cache-Dimensionierung. Außerdem sollten nur die tatsächlich benötigten Daten anstatt der gesamten Cache-Zeile geladen werden. Der zusätzliche Aufwand eines separaten Valid-Bits pro Datenwort ist dabei gering. Eine hypothetische 64-bittige DDR-RAM-Speicheranbindung ergab keine Verbesserung: Das schnelle Befüllen größerer Cache-Zeilen wird durch die (je nach Zugriffsmuster) hohen Zykluszeiten von SDRAM relativiert.

Die Tag-Einheit des Caches kann weiterhin für den TLB wiederverwendet werden. Mit der Vergrößerung von aktuell 2 auf 4–8 Einträge können zusätzlich ein Drittel der Zugriffe auf die ebenfalls im externen Speicher abgelegte Objekttabelle eingespart werden.

4 Implementierung

Die Implementierung des kombinierten Adress- und Offset-Caches (AOC) mit Write-Through-Strategie erfolgte in VHDL für die SHAP-Mehrkernarchitektur, wobei Anzahl der Cache-Zeilen und zwischengespeicherte Offsets frei konfigurierbar sind. Entsprechend der obigen Analyse wurden wurden für die weitere Auswertung 4 bzw. 8 Cache-Zeilen mit den Offsets -2 und 1 gewählt. Jedes Wort der Cache-Zeile hat außerdem ein eigenes Valid-Bit. Für die Messung auf einem Kern wurde zusätzlich eine Variante mit 2 Cache-Zeilen synthetisiert. Bei allen Konfigurationen standen jedem Kern ein 4 KB Methoden-Cache zur Verfügung. Details zur Implementierung sind in der zugehörigen Diplomarbeit [Olu12] dokumentiert.

Die Synthese erfolgte mit dem Werkzeug Xilinx ISE 13 für das XUPV5-Development-Board mit dem Virtex-5 FPGA XC5VLX110T. Sie zeigte, dass bis zu 17 Kerne mit derselben Taktfrequenz von 80 MHz einer Einkernvariante implementiert werden können. Bei 16 und 17 Kernen musste bei der Synthese jedoch nach Geschwindigkeit statt Fläche optimiert werden. Bei 17 Kernen führte dies jedoch zu einem überdurchschnittlichem Chipflächenbedarf von 17% mehr LUTs/Kern. Bei 16 Kernen sind es immer noch 2% mehr LUTs/Kern.

Der Hardwareaufwand für $n \leq 15$ Kerne ist in Tab. 2 aufgelistet, wobei für die Approximation das Programm gnuplot verwendet wurde. Es werden für:

- 4 Cache-Zeilen: ca. 4% mehr Register und 6% mehr LUTs,
- 8 Cache-Zeilen: ca. 9% mehr Register und 8% mehr LUTs

als in der Ausgangsvariante mit nur einem 2-Zeilen-TLB benötigt. Der Bedarf an On-Chip-Speichern bleibt konstant, da für den Cache LUTs als Speicher (engl.: distributed RAM) genutzt werden. Der Bedarf an Multiplizierelementen und IOB-FFs ist unverändert.

Der Hardwareaufwand skaliert nach wie vor linear mit der Kernanzahl.

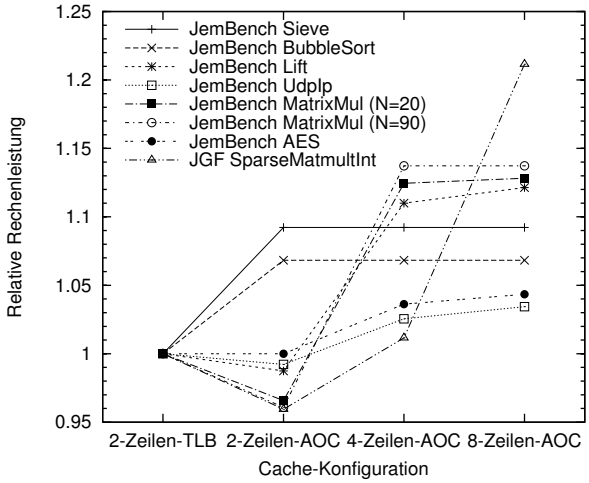

Abbildung 1: Beschleunigung auf einem Kern

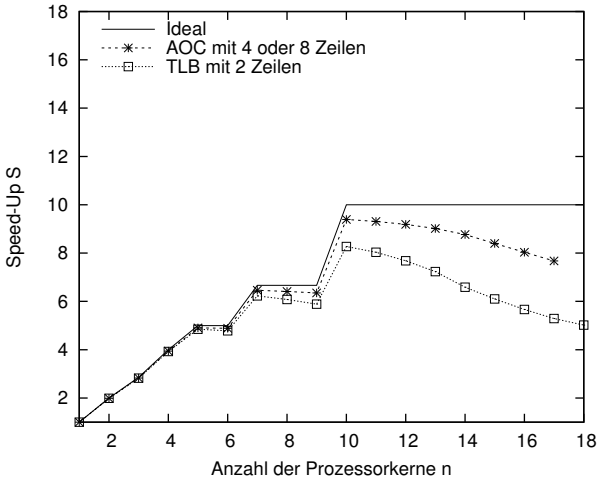

Abbildung 2: Speed-Up für MatrixMul (N=20)

5 Auswertung

In Abb. 1 ist die relative Beschleunigung auf einem Kern gegenüber der Ausgangsvariante (2-Zeilen-TLB) für 2, 4 und 8 Cache-Zeilen dargestellt. Benchmarks mit einer Veränderung <2% wurden der Übersichtlichkeit halber weggelassen; dafür ist der unten beschriebene JGF SparseMatmult aufgeführt. Wie zu sehen, tritt bei 2 Cache-Zeilen sogar teilweise eine Verlangsamung ein, da für den AOC eine zusätzlicher Taktzyklus (Latenz) benötigt wird. Erst mit 4 Cache-Zeilen trat immer eine Verbesserung von bis zu 14% ein. Von 8 Cache-Zeilen profitierte jedoch nur SparseMatmult.

Aus der JemBench-Suite eignet sich nur der MatrixMul-Benchmark für die Evaluation eines Mehrkernprozessors, da dieses Programm im Gegensatz zu NQueens einen nennenswerten Anteil von Heap-Zugriffen ausführt. Die anderen Benchmarks sind nur für 1-Kernprozessoren gedacht. Bei der Auswertung wurde eine Race-Condition in der Lastverteilung (EnumeratedExecutor) für mehrere Kerne entdeckt und auf der Projektwebseite `sf.net/projects/jembench` als Bug #3512742 dokumentiert. Eine selbst entwickelte Lösung für den Fehler wurde dort als Patch #3512739 hinterlegt und bei den folgenden Messungen verwendet.

In Abb. 2 sind die erzielbaren Speed-Ups für den MatrixMul-Benchmark in Abhängigkeit von der Anzahl der Prozessorkerne für die vorgegebene Matrixgröße von N=20 dargestellt. Als Basis für die Speed-Up-Berechnung dient die jeweilige Konfiguration mit einem Kern, also jeweils mit 2-Zeilen-TLB oder 4- bzw. 8-Zeilen-AOC. Der maximale Speed-Up kann von 8,3 auf 9,4 bei 10 Kernen gesteigert werden. Aufgrund der geringen Matrixgröße kann der Benchmark nicht von mehr als 10 Kernen profitieren, stattdessen ergibt sich der in der selben Grafik dargestellte treppenförmige Verlauf für den idealen Speed-Up. Die Verschlechterung oberhalb von 10 Kernen ergibt sich durch den auf jedem Kern laufenden Systemthread. Dieser überprüft jeweils pro Kern ob dort neue Threads zu starten sind und belegt damit Speicherbandbreite. Mit zunehmender Kern- und damit Systemthread-Anzahl wird mehr Bandbreite belegt. Mit 8 Cache-Zeilen wurden die gleichen Werte wie mit 4 Cache-Zeilen gemessen.

Für die Evaluation vieler Kerne wurde der MatrixMul-Benchmark abweichend mit N=90 ausgeführt. 90 ist durch 18 teilbar und erlaubt gerade noch eine Messung mit einem Fehler von <1%. Wie in Abb. 3 zu sehen, nähert sich der Speed-Up bei der Variante mit 2-Zeilen-TLB ab 10 Kerne langsam den Maximum von 9,5 bei 15 Kernen. Bei 4 oder 8 Cache-Zeilen steigt dagegen der Speed-Up weiter an und erreicht bei 15 Kernen ein sehr gutes Maximum von 14,0. Zusammen mit der 14%-tigen Beschleunigung auf einem Kern wird der Benchmark bei 15 Kernen und 4 Cache-Zeilen 16-mal schneller ausgeführt als bei einem Kern mit 2-Zeilen-TLB. Dies entspricht einer Steigerung von 68% gegenüber dem Maximum von 9,5 beim 2-Zeilen-TLB.

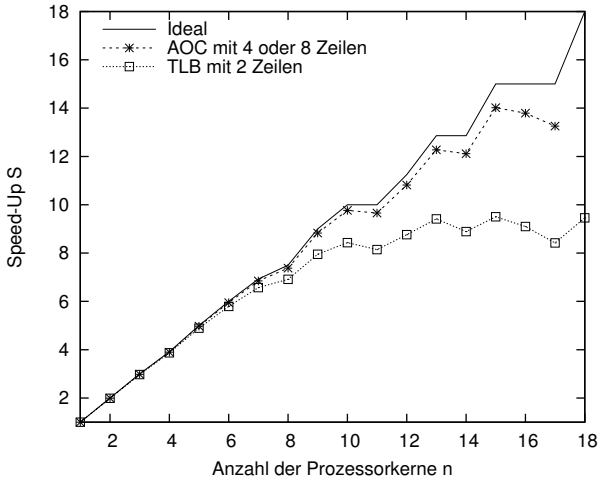

Abbildung 3: Speed-Up für MatrixMul mit N=90

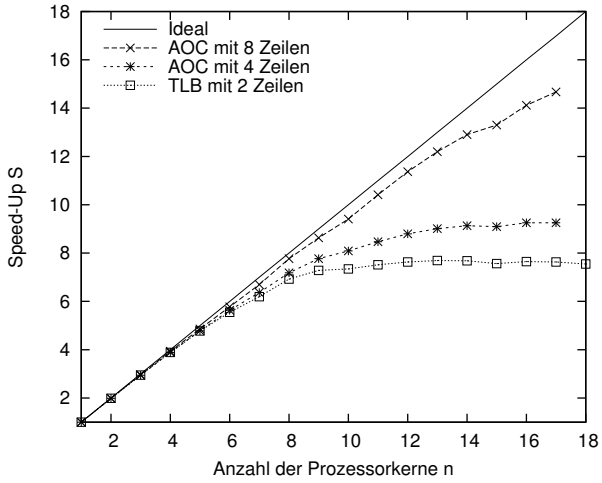

Abbildung 4: Speed-Up für SparseMatmult

Da mit dem MatrixMul-Benchmark keine Leistungssteigerung mit 8 Cache-Zeilen nachweisbar ist, wurde zusätzlich der SparseMatmult-Benchmark aus der Suite „Java Grande Forum Multithreaded Benchmarks“ [JGF] (Version 1.0) evaluiert. Der Algorithmus multipliziert eine dünn besetzte Matrix mit einem Vektor. Die komprimierte Speicherung der Matrix erfordert eine zusätzliche Indirektion und somit einen Arbeitssatz von 8 Objekten. Für eine sinnvolle Messung musste der Datentyp der Matrix- / Vektorelemente von Gleitkomma auf Ganzzahl geändert, da Gleitkommaoperationen auf SHAP aktuell nur per Software emuliert werden und damit die Ausführung überwiegend aus Arithmetik bestanden hätte. Ebenso wurde die Matrix verkleinert, da auf SHAP ein Integer-Feld maximal 16380 Elemente umfassen darf. Es wurde eine Matrix der Größe 4000×4000 verwendet, wobei nur 16000 Elemente verschieden von 0 sind. Die Messgenauigkeit ist dennoch besser als 1%.

Wie in Abb. 4 dargestellt, steigt in der Ausgangsvariante (2-Zeilen-TLB) der Speed-Up von SparseMatmult ab 8 Kernen nicht mehr nennenswert an und erreicht bei 16 Kernen das Maximum von 7,6. Unter Einsatz eines 4-Zeilen-AOC kann der Arbeitssatz der innersten Schleife abgedeckt werden und damit der Speed-Up auf maximal 9,3 bei gleicher Kernanzahl gesteigert werden. Eine wesentliche Verbesserung ergibt sich erst mit 8 Cache-Zeilen: Jetzt beträgt der maximale Speed-Up 14,7 bei 17 Kernen. Der Kurvenverlauf deutet sogar noch Steigerungspotential an, jedoch konnten, wie bereits gesagt, 18 Kerne nicht bei gleicher Taktfrequenz synthetisiert werden. Zusammen mit der 21%-tigen Beschleunigung auf einem Kern wird der Benchmark bei 15 Kernen und 8 Cache-Zeilen 16-mal schneller ausgeführt als bei einem Kern mit 2-Zeilen-TLB. Dies entspricht einer Steigerung von 112% gegenüber dem Maximum von 7,6 beim 2-Zeilen-TLB und damit einer Verdopplung der Verarbeitungsleistung. Auf 16 und 17 Kernen sind zwar noch größere Steigerungen erreichbar, die jedoch durch einen überdurchschnittlichen Hardwareaufwand erkauft werden.

Ein passender Vergleich mit anderen Architekturen war aufgrund der verschiedenen Voraussetzungen nicht möglich.

6 Zusammenfassung

In dieser Arbeit werden Maßnahmen zur Entlastung der Heap-Speicherschnittstelle untersucht, um einen höheren Speed-Up auf einem Java-Bytecode-Mehrkernprozessor mit gemeinsamen Heap-Speicher zu erzielen. Dazu werden exemplarisch anhand der Benchmark-Suite JemBench die typische Verteilung der Speicherzugriffe auf den Heap untersucht und daraus das Design eines Objekt-Caches mit integriertem TLB abgeleitet. Eine detaillierte Cache-Simulation ergab, dass 4–8 Cache-Zeilen genügen und neben der physikalischen Objektadresse nur die ersten benutzer-

definierten Objektvariablen (an den Offsets -2 und 1) häufig benötigt werden. Die prototypische Implementierung zeigt einen linearen Hardwareaufwand. Im Mehrkernprozessor können bis zu 17 Kerne mit der gleichen Taktfrequenz von 80 MHz des Einkernprozessors realisiert werden.

Die Ausführung der Mehrkern-Benchmarks demonstriert, dass der maximale Speed-Up von ehemals 8–9 auf jetzt 14 und mehr gesteigert werden kann. Zusammen mit der Beschleunigung des Einkernprozessors um 12 bis 21% kann sogar eine Verdopplung der Verarbeitungsleistung erzielt werden. An FPGA-Ressourcen werden dabei für bis zu 15 Kerne und 8 Cache-Zeilen je Kern 8% mehr LUTs und 9% mehr Register (im Vergleich zur Ausgangsvariante mit 2-Zeilen-TLB) benötigt, sodass sich ein sehr gutes Nutzen-Kosten-Verhältnis ergibt.

Literatur

[HPS10] HUBER, Benedikt ; PUFFITSCH, Wolfgang ; SCHOEBERL, Martin: WCET driven design space exploration of an object cache. In: [KV10], S. 26–35

[JGF] *Java Grande Forum Benchmark-Suite*. – `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`

[KV10] KALIBERA, Tomás (Hrsg.) ; VITEK, Jan (Hrsg.): *Proc. 8th Int'l Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'10)*. ACM, 2010

[LJW+09] LUI, Mengxia ; JI, Weixing ; WANG, Zuo ; LI, Jiaxin ; PU, Xing: High performance memory management for a multi-core architecture. In: *Proc. 9th IEEE Int'l Conf. Computer and Information Technology (CIT'09)*, IEEE Press, 2009, S. 63–68

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *The Java(TM) Virtual Machine Specification*. 2nd edition. Amsterdam : Addison-Wesley Longman, 1999. – ISBN 978–0201432947

[Olu12] OLUNCZEK, Andrej: *Leistungssteigerung der Speicherarchitektur des SHAP-Mehrkernprozessors*, Dresden, Technische Universität, Diplomarbeit, 2012

[PS08] PITTER, Christof ; SCHOEBERL, Martin: Performance evaluation of a Java chip-multiprocessor. In: *Proc. 3rd Int'l Symp. Industrial Embedded Systems (SIES'08)*, IEEE Press, 2008, S. 34–42

[SPU10] SCHOEBERL, Martin ; PREUSSER, Thomas B. ; UHRIG, Sascha: The embedded Java benchmark suite JemBench. In: [KV10], S. 120–127

[TSP08] TYYSTJÄRVI, Joonas ; SÄNTII, Tero ; PLOSILA, Juha: Instruction set enhancements for high-performance multicore execution on the REALJava platform. In: *Proc. 26th Norchip Conference*, IEEE Press, 2008, S. 190–193

[Uhr09] UHRIG, Sascha: Evaluation of different multithreaded and multicore processor configurations for SoPC. In: *Proc. 9th Int'l Workshop Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'09)*, Springer-Verlag, 2009, S. 68–77

[UU09] UHRIG, Sascha ; UNGERER, Theo: A garbage collection technique for embedded multithreaded multicore processors. In: *Proc. 22nd Int'l Conf. Architecture of Computing Systems (ARCS'09)*, Springer-Verlag, 2009 (LNCS 5455). – ISBN 978–3–642–00453–7, S. 207–218

[VR00] VIJAYKRISHNAN, N. ; RANGANATHAN, N.: Supporting object accesses in a Java processor. In: *IE Proc. Computers and Digital Techniques* 147 (2000), Nr. 6, S. 435–443. – ISSN 1350–2387

[WSW06] WRIGHT, Greg ; SEIDL, Matthew L. ; WOLCZKO, Mario: An object-aware memory architecture. In: *Sci. Comput. Program.* 62 (2006), Nr. 2, S. 145–163. – ISSN 0167–6423

[Zab12] ZABEL, Martin: *Effiziente Mehrkernarchitektur für eingebettete Java-Bytecode-Prozessoren*, Dresden, Technische Universität, Diss., 2012