Verbesserte Performanz und Programmierbarkeit in Netzwerksystemen

Florian Birnthaler
Betreuer: Benedikt Elser
Seminar Future Internet SS2009
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München
Email: birnthal@in.tum.de

***Kurzfassung*— Es wird eine Prozessorarchitektur für Netzwerkgeräte beschrieben, bei der die für Speicherbausteine zur Verfügung stehende Fläche auf der Platine beliebig an Hardware-Threads oder Caches zugewiesen werden kann. Dadurch kann sich ein solcher Netzwerkprozessor selbstständig an die Programme, die auf ihm laufen, anpassen und so eine optimale Performanz erzielen.**

***Schlüsselworte*— Memory Bottleneck, Prozessorarchitektur, Multithreading, Caching, Netzwerkverkehr**

I. Einleitung

Das so genannte 'Memory Bottleneck' ist ein signifikantes Problem für Paketverarbeitungsplatformen, da der Netzwerkverkehr über das Internet in den letzten Jahren stark zugenommen hat [3] und Anwendungen zur Paketverarbeitung – wie Virenscanner und Software zur Netzwerküberwachung und Intrusion Detection – immer umfangreicher werden.

Die typischen Ansätze, dieses Problem zu umgehen, sind Hardware-Multithreading, Caching, Algorithmen, die die Anzahl der Speicherzugriffe auf den Sekundärspeicher minimieren und kleinere 'Hardware-Hacks'. Außerdem wird für bestimmte Anwendungen oft maßgeschneiderte Hardware verwendet.

Eine Alternative hierzu ist eine Prozessorarchitektur, welche die Anzahl der verwendeten Threads und den Cache den jeweiligen Ansprüchen entsprechend anpassen kann. Diese Architektur, bei der, neben hoher Performanz, ein weiteres wichtiges Ziel eine einfache Programmierung ist, wird in dieser Arbeit näher vorgestellt.

Nach einer kurzen Erklärung zum Problem des 'Memory Bottleneck', wird dabei zunächst auf die Vorteile, die eine solche Architektur bieten würde, eingegangen. Danach wird der Aufbau eines solchen Prozessors vorgestellt, um daraufhin anhand des von ihm verwendeten Algorithmus deutlich zu machen, wie sich dieser an die Programme und den spezifischen Paketstrom anpassen kann. Schließlich wird die Performanz mit üblichen Netzwerkprozessoren, wie dem Intel IXP2800 [5], verglichen und ein Fazit gezogen.

II. Grundlagen – Das 'Memory Bottleneck'-Problem

Heutige Mikroprozessoren arbeiten mit sehr hohen Taktraten, die die des Hauptspeichers bei weitem übertreffen. [4, S.292f] Das bedeutet, dass der Prozessor auf Daten, die er vom Hauptspeicher anfordert, viele Taktzyklen lang warten müsste. Wenn, aufgrund der Struktur des gerade abzuarbeitenden Programms, während dieser Wartezeit keine anderen Befehle abgearbeitet werden können, wird sehr viel Rechenkraft verschwendet.

Um dieses 'Memory Bottleneck' zu umgehen, wird eine Speicherhierarchie eingeführt, die nahe am Prozessor einen kleinen, schnellen Speicher (Register und Level 2 Cache) vorsieht und weiter nach außen immer größeren, jedoch auch langsameren Speicher (Level 3 Cache, Hauptspeicher, Festplatte). Nun wird versucht, Daten, die in den nächsten Taktzyklen vom Prozessor benötigt werden, in einem Speicher möglichst nahe am Prozessor bereitzuhalten, damit dieser bei deren Anforderung nicht lange auf sie warten muss. Dies geschieht dadurch, dass Teilbereiche der größeren Speicher mit Hilfe einer Caching-Strategie (z.B. direct mapped cache [4, S.295ff] in die kleineren Speicher kopiert werden. Benötigt nun der Prozessor Daten, die in diesem Moment auch in einem prozessornahmen Cache gespeichert sind ('cache hit'), so muss er diese nicht, wie bei einem 'cache miss', vom langsamen, größeren Speicher anfordern und darauf warten, sondern kann gleich die Daten aus dem Cache verwenden. Dadurch entsteht ein enormer Performanzgewinn.

III. Warum ein sich anpassenden Prozessor?

Auf Netzwerkprozessoren werden viele verschiedene Programme ausgeführt, die jeweils unterschiedliche Eigenschaften bezüglich Parallelismus und Daten-Lokalität haben. Während es für manche Anwendungen vorteilhaft ist, möglichst viele Threads gleichzeitig arbeiten zu lassen, laufen andere Programme performanter, wenn stattdessen der Datencache größer ist. Es ergibt sich also für jede Anwendung ein optimales Verhältnis zwischen Größe des Datencaches und Anzahl der Threads, wobei wir davon ausgehen, dass auf dem Chip nur begrenzt Platz ist, um diese Elemente unterzubringen. Diese Verhältnisse schwanken zum Teil sehr stark. [1]

Programme, die man zur ersten Gruppe zählen kann sind beispielsweise *cast* (verwendet zur Verschlüsselung) und *md5* (bildet Prüfsummen über Pakete). Da diese Anwendungen das ganze Paket verarbeiten profitieren sie nicht von Caches. Die

Latenzzeit, die benötigt wird, um die Daten zum Prozessor zu bringen, kann also nicht verringert werden. Allerdings können mehrere Threads jeweils ein anderes Paket bearbeiten und so die Abarbeitung beschleunigen.

Zur zweiten Gruppe zählt man Programme wie *stream* oder *portscan*. Hier ist es oft wegen Locks auf bestimmte Datenbereiche nicht gut möglich, parallel zu arbeiten, da sich die Threads gegenseitig blockieren. Hier ist also ein größerer Cache besser.

Die meisten der getesteten Programme liegen zwischen diesen beiden Extremen, jedoch unterscheiden sich auch diese bei den Anforderungen für Cache oder Threads merkbar.

Das Problem ist, dass das Verhältnis von Threads und Cachespeicher bei den eingesetzten Netzwerkprozessoren unveränderbar ist, da diese Verteilung zur Entwicklungszeit fest bestimmt wurde. Für Entwickler von Programmen wie oben, die den Netzverkehr verarbeiten, entsteht dadurch das Problem, dass die Programme die Hardware nicht optimal ausnutzen. Deswegen stecken sie viel Zeit in die Anpassung des Programms an die Eigenschaften der Hardware durch Veränderung der verwendeten Algorithmen und kleinere 'Hardware-Hacks'. Dadurch können die Programme aber auch schwerer wart- und erweiterbar werden.

Wenn sich allerdings das Verhältnis von Threads zu Cachespeicher automatisch an das Programm anpassen kann, so muss man diesen zusätzlichen Aufwand nicht mehr betreiben. Anwendungen und Protokolle können dadurch viel schneller entwickelt werden. Vor allem aber wird die Programmierung von Software, die die Hardware optimal ausnutzt viel einfacher.

IV. Aufbau des Prozessors

Möglich ist ein solcher Prozessor, der Threads bzw. Cache mehr oder weniger Platz auf der Platine zuweisen kann, dadurch, dass Caching und Multithreading die gleiche Ressource verwenden, und zwar schnellen Speicher, der nahe am Prozessor liegt.

Während Caching diesen Speicher verwendet, um Daten bereitzustellen, die voraussichtlich als nächstes benötigt werden, speichert Multithreading in ihm Register, die für den jeweiligen Thread wichtig sind.

Bei dem hier besprochenen Prozessor kann dieser Speicher beliebig auf Threads und Caches verteilt werden. Damit dessen Performanz später besser verglichen werden kann, orientiert sich die Gesamtgröße des Chips an dem des Intel IXP2800. Auf dieser Fläche können neun Prozessorkerne verteilt werden, von denen jeder den in Abb. 1 gezeigten schematischen Aufbau hat.

Dort sieht man auf der linken Seite den groben Aufbau, der aus einem Multiport-Register-Cache (MRC) und einem primären Datencache, der sowohl Thread-spezifische Register als auch normale Cache-Daten aufnehmen kann, besteht. Dieser MRC ist notwendig, da eine möglichst geringe Latenzzeit, eine hohe Bandbreite und Kapazität nicht gleichzeitig erreichbar sind. Deswegen besteht der Speicher aus einem kleinen, schnellen Speicher mit hoher Bandbreite (dem MRC) und einem großen, langsameren Speicher mit geringerer Bandbreite.

Die Herausforderung beim Entwickeln des Prozessors besteht nun darin, zu erreichen, dass der schnellere Speicher nach außen hin so wirkt als hätte er die Kapazität des langsameren Speichers. Traditionelle Lösungsansätze für solche Speicherhierarchien sind Register Caching (beispielsweise mittels 'last recently used') und Double Buffering. Bei Double Buffering hat man zwei Datenpuffer: Auf einem arbeitet der gerade aktive Thread, während der andere mit den Daten des nächsten Threads gefüllt wird; bei Threadwechsel werden dann einfach beide Puffer gewechselt und dem Thread stehen sofort alle benötigten Daten zur Verfügung.

Um allerdings alle relevanten Register eines Threads im Voraus vollständig zu laden zu können, würde man einen MRC benötigen, der auf dem Chip zu viel Fläche einnehmen würde. Deswegen wird eine Kombination aus beiden Ansätzen zusammen mit einem System zur Vorhersage der als nächstes benötigten Daten verwendet, was im rechten Teil von Abb. 1 dargestellt ist:

Der Prozessor verwendet zwei Registerspeicher (in der Abbildung wird nur einer (S-file) gezeigt), um Daten bzw. Threadregister zu cachen. Während einer der beiden Speicher verwendet wird, wird der andere mit Daten für den nächsten Thread gefüllt. Welche Daten das sind wird vom 'Predictive Prefetcher' vorhergesagt. Dies geschieht mit Hilfe des Programmzählers des Threads, der als nächster dran ist. Denn zu diesem Programmzähler wird jedes mal wenn der Thread zurück in den Data Cache geschrieben wird ein Bitmuster mit abgespeichert, das bestimmt, welche Register bei der nächsten Aktivierung des Threads im Voraus geladen werden sollen. Dieses Bitmuster wird durch zusätzliche Kontrollbits (V für 'valid', R für 'read', M für 'modified') in den Registerspeichern bestimmt: Wenn für ein Register das R-bit gesetzt ist (d.h. es wurde vom Thread daraus gelesen), so wird es vorgeladen.

Die miss rate, die bei den Vorhersagen erzielt wird ist bei allen getesteten Anwendungen kleiner als 2 Prozent – meist sogar unter 1 Prozent. [1] Die Vorhersagen sind also sehr zuverlässig. Die Anzahl der richtig gecachten Register weicht meist nur um zwei bis drei (von insgesamt bis zu 18 Registern) von der einer optimalen Auswahl ab.

V. Algorithmus zur Anpassung des Prozessors

Der Algorithmus zur Anpassung des Cache-Thread-Verhältinsses läuft während dem Betrieb, benötigt aber dennoch so wenig Ressourcen, dass er das System nicht belastet. Deswegen wird er auch später bei den Performaztests vernachlässigt.

Der Algorithmus basiert auf drei Beobachtungen:

1) Von einer Festlegung des Thread-Cache-Verhältnisses zur Entwicklungszeit ist abzuraten, da man so nicht auf Änderungen im Traffic reagieren kann und Wartung und Erweiterungen von Systemen, die darauf beruhen, zu umständlich sind.

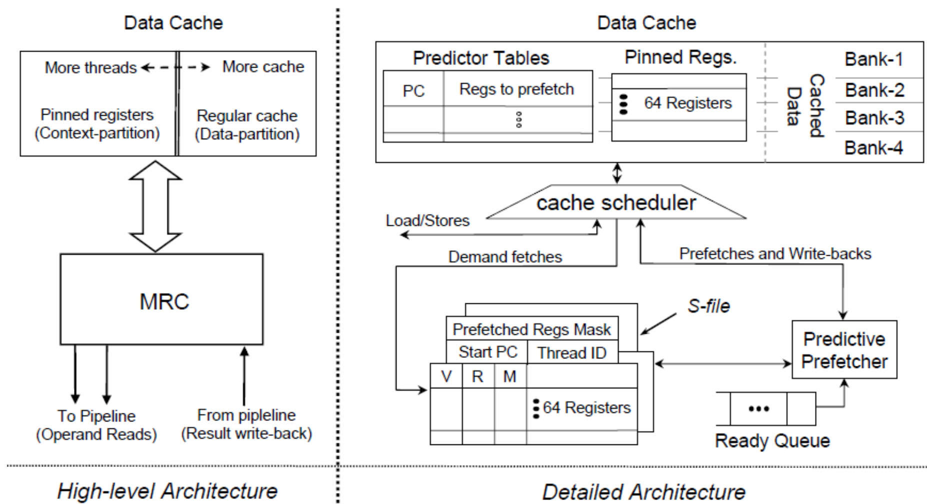

Abbildung 1. Architektur des Prozessors (Quelle: [1])

2) Wenn die Anzahl der Threads erhöht wird steigt die Performanz zunächst, fällt aber ab einem bestimmten Punkt wieder ab. Das liegt zum einen daran, dass die Cache-Kapazität im selben Maße kleiner wird, wie die Speicherzellen für die Threads zunehmen, und deswegen mehr cache misses auftreten. Ein weiterer Grund dafür ist, dass bei steigender Anzahl von Threads diese auch stärker um benötigte Seiten aus dem Speicher konkurrieren, was die Bandbreite des Caches negativ beeinflusst. Außerdem werden durch die höheren miss Raten auch öfter die Threads gewechselt. Denn bei jedem cache miss wird der Thread gewechselt, weil der gerade aktive Thread lange auf die angeforderten Daten warten muss. Dadurch gehen jedesmal zwei Taktzyklen (Dauer eines Threadwechsels) verloren.
3) Die Verringerung der Threadanzahl – und damit die Terminierung eines Threads – verringert die Performanz für eine kurze Zeit (einige hundert Pakete lang). Das liegt an folgendem: Wenn ein Thread entfernt wird, müssen alle Speicherzeilen, die von diesem Thread für die Vorhersage der als nächstes benötigten Speicherzellen verwendet werden, freigegeben werden. Allerdings können diese nicht ermittelt werden, weshalb diese 'Vorhersagespeicherzellen' von allen Threads freigegeben werden müssen. Deswegen funktionieren die nächsten Vorhersagen nicht und die anderen Threads müssen die Speicherzellen für ihre Vorhersagen nochmal neu reservieren. Dies führt zu einer Erhöhung der Bearbeitungszeit von Paketen um 6%, die aber nach den erwähnten wenigen hundert Paketen wieder verschwunden ist.

Aus diesen Beobachtungen folgt, dass die Anpassung zur Laufzeit stattfinden muss. Um die optimale Anzahl von Threads zu finden wird eine einfache lineare Suche verwendet und um seltener Threads terminieren zu müssen, wird bei der Suche nach unten immer um zwei Threads herunter geschaltet (anstatt wie bei der Suche nach oben um einen Thread).

Der Algorithmus ist in Abb. 2 im Pseudocode dargestellt.

Notation:
N_{max}: Max number of threads
N_c: Current number of threads
$\lambda[n]$: Throughput with n threads

1: **for** $n = N_c + 1$ to N_{max} **do**
2: $\lambda[n] = trialRun(n)$
3: **if** $(\lambda[n] < \lambda[n-1])$ **then**
4: break
5: **end if**
6: **end for**
7: **if** $((n-1) > N_c)$ **then**
8: return $(n-1)$ `/* increasing works */`
9: **end if**
10: `/* increasing does not work */`
11: **for** $n = N_c - 2$ down-to 1 with step (-2) **do**
12: $\lambda[n] = trialRun(n)$
13: **if** $(\lambda[n] < \lambda[n+2])$ **then**
14: break
15: **end if**
16: **end for**
17: `/*also try the last thread skipped*/`
18: $\lambda[n+1] = trialRun(n+1)$
19: return $(\lambda[n+1] > \lambda[n+2])?(n+1) : (n+2);$

Abbildung 2. Algorithmus zur Anpassung des Thread:Cache-Verhältnisses (Quelle: [1])

Hier wird nach und nach immer ein Thread zusätzlich hinzugefügt und der Durchsatz dieser Konfiguration mit der vorherigen verglichen. Dies geschieht so lange, bis keine Durchsatzsteigerung durch einen zusätzlichen Thread mehr erreicht werden kann.

Wenn allerdings nicht einmal das Hinzufügen eines einzigen Threads zur momentanen Anzahl von Threads einen Gewinn darstellt, dann werden nach und nach zwei Threads abgebaut, wodurch ein Performanzgewinn zu erwarten ist. Auch dies

wird so lange gemacht, bis es keine Durchsatzsteigerung mehr gibt.

Als Resultat bekommt man die ideale Anzahl an Threads.

Nachdem dieses erste Ergebnis gefunden ist, wird nicht mehr der ganze Algorithmus ausgeführt. Stattdessen werden nur noch kleine Anpassungen anhand von Veränderungen in der Balance, die während der Laufzeit verfolgt wird, gemacht. Diese Anpassungen finden alle 100.000 Pakete statt und erfordern vernachlässigbar geringe Prozessortätigkeit.

VI. Performanzvergleich mit anderen Prozessoren

Die Anwendungen, für die die Performanz des Prozessors getestet wird, implementieren folgende Funktionen:

- Integritätsprüfung ankommender Pakete
- Klassifizieren der Pakete
- Paketverarbeitung
- Scheduling von ausgehenden Paketen

Tabelle I zeigt eine Übersicht dieser Programme. Eine genaue Beschreibung zu diesen findet man in [2, S.19ff].

Da ein aussagekräftiger Performanztest wegen der enormen Anzahl an verschiedenen zu testenden Konfigurationen (mehrere Hunderttausend verschiedene Konfigurationen) nicht realisierbar ist, wurde der Simulator Simplescalar [6] verwendet, um repräsentative Programm-Traces zu erzeugen.

Für die Tests wurden Paket-Traces aus verschiedenen Stellen im Internet verwendet, die verschiedene Randbedingungen für die Programme bieten. So kommen beispielsweise die Pakete vom ANL-Trace (Verbindung zwischen Argonne National Lab und ihrem ISP) vom Rand eines Netzwerkes (des ANL-Netzwerkes) und haben damit ähnliche Header, was eine gute Voraussetzung für die Verwendung von Caches ist.

Auf der anderen Seite stammen die Pakete von MRA (aus [7]) aus einem Netzwerk-Zentrum und haben damit unterschiedlichste Header, wodurch hier mit Multithreading die besseren Ergebnisse erzielt werden können.

Im Vergleich zum IXP2800 [5], einem aktuellen und leistungsfähigen Netzwerkprozessor erreicht der vorgestellte Prozessor in 26% der Anwendungen einen um über 300% höheren Durchsatz. Verglichen mit einem Netzwerkprozessor, der zwar ein festes - jedoch optimales - Thread-Cache-Verhältnis hat, hat der sich anpassende Prozessor in über einem Drittel der Anwendungen einen um 60% höheren Durchsatz. Diese Resultate werden in Abb. 3 gezeigt. Ein grauer Balken gibt hier an, welcher Anteil der getesteten Programme einen bestimmten Durchsatz erreichen (beispielsweise erreichen 40% der Programme einen Durchsatz der 90-110% von dem des IXP2800 entspricht). Die obere Kurve gibt an, welcher Anteil der Programme *mindestens* einen bestimmten Durchsatz erreichen.

Zuletzt wird noch ein Vergleich (vergl. Abb. 4) zu einem hypothetischen Prozessor hergestellt, der nicht nur das Verhältnis von Cache zu Thread auf dem Chip verändern kann, sondern auch die Anzahl der Rechenkerne. Er würde also bei einer gegebenen Chipfläche ein für die Anwendung optimaler Prozessor sein. In 63% der Anwendungen ist der entwickelte

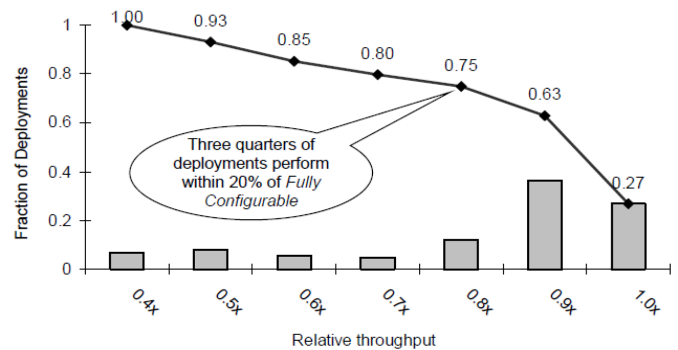

Abbildung 4. Performanz im Vergleich zum hypothetischen, voll konfigurierbaren Prozessor (Quelle: [1])

Prozessor nur um maximal 10% langsamer als der Hypothetische und in 80% der Anwendungen um maximal 30%. In 27% der Fälle erreicht er sogar den gleichen Durchsatz wie der hypothetische Prozessor.

VII. Fazit

Wie die Vergleiche mit den anderen Netzwerkprozessoren zeigen, stellt der hier vorgestellte Prozessor nicht nur eine Alternative zu ihnen dar. Vielmehr weist er in eine Richtung, in die sich zukünftige Netzwerkprozessoren orientieren sollten, um den stets steigenden Anforderung des Internetverkehrs gerecht zu werden. Dadurch, dass sich ein solcher Prozessor selbstständig an die Anforderungen der auf ihm laufenden Programme anpassen kann, muss bei der Programmierung dieser Programme nicht mehr auf die Eigenheiten der Zielhardware geachtet werden. Dadurch wird eine einfachere Programmierbarkeit erreicht.

Literatur

[1] J. Mudigonda, H. M. Vin, S. W. Keckler, *“Reconciling Performance and Programmability in Networking Systems”*
[2] J. Mudigonda, *“Addressing the Memory Bottleneck in Packet Processing Systems”*, PhD Thesis, University of Texas, 2005
[3] W. Riggert, *“Netzwerktechnologien”*, Carl Hanser Verlag, 2003
[4] A. Tanenbaum, *“Structured Computer Organization”*, Fifth Edition, Pearson, 2006
[5] *Intel IXP2800 Network Processor*, ftp://ftp5.chinaitlab.com/whitebook/Edge_Core_Applications.pdf
[6] *SimpleScalar*, http://www.simplescalar.com/
[7] *NLANR Network Traces*, http://pma.nlanr.net/Traces/

Functionality	Application	Source	Notes
Integrity verification	checksum	Free BSD	Protects headers in TCP, UDP and IP packets (RFC-1071)
	md5	R.S.A Inc.	MessageDigest 5 (MD5). Mostly used to protect the payload (RFC-1321)
Classification	classify	UT-Austin	Hashes the five-tuple: ⟨srcIP,dstIP,srcPort,dstPort,protocol⟩
Route Lookup (Longest prefix match)	patricia	Free BSD	Patricia tree. Can handle non-contiguous masks. Used in many end-systems
	bitmap	UT-Austin	Employs bitmaps to compress trie nodes. Used in many commercial routers
	bsol	UT-Austin	Binary search using hash tables. Has the best known avg. comp. complexity
	ixp	IXA SDK 3.0	Designed for IXP series of NPs. Two tries are searched simultaneously
Metering (Prioritize packets)	srtcm	IXA SDK 3.0	Enforces a *single* mean rate and a peak burst. (RFC-2697)
	trtcm	IXA SDK 3.0	Enforces *two* independent rates: mean and peak. (RFC-2698)
	tswtcm	UT-Austin	Enforces mean and peak rates over sliding windows. (RFC-2859)
Header processing	stream	Snort 2.0	TCP receive-side processing. Reassembles byte streams out of packets
	portscan	Snort 2.0	Detects portscan attack if too many ports are accessed too quickly
Payload processing	cast	SSLeay Lib	Encryption scheme. Used in Virtual Private Networks (VPNs) (RFC-2612)
	vscan	Snort 2.0	Pattern matcher. Scans packet payload for virus signatures
Scheduler	drr	UT-Austin	Deficit Round Robin. Found in many commercial routers

Tabelle I
GETESTETE PROGRAMME (QUELLE: [1])

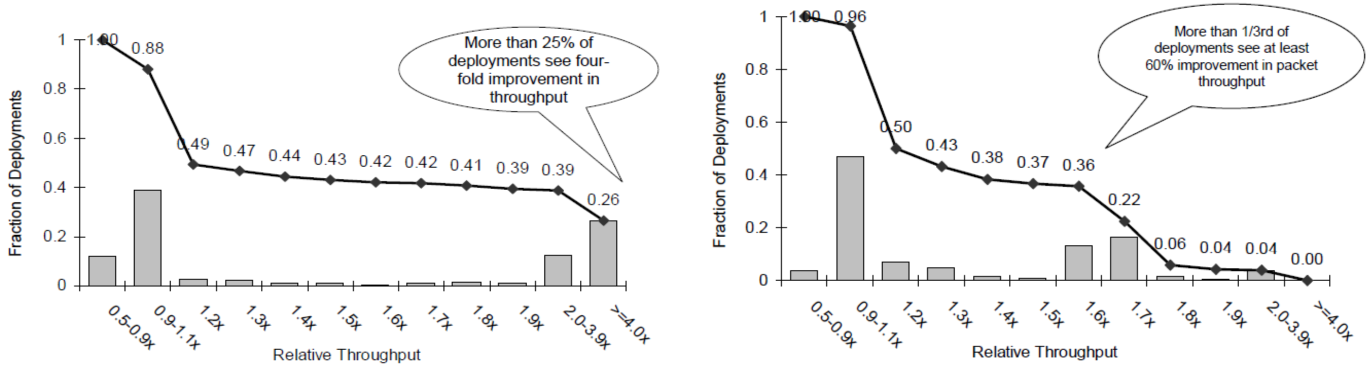

Abbildung 3. Performanz im Vergleich zum IXP2800 und einem optimalen, festen Prozessor (Quelle: [1])