



Matthias Withopf

Virtuelles Tandem

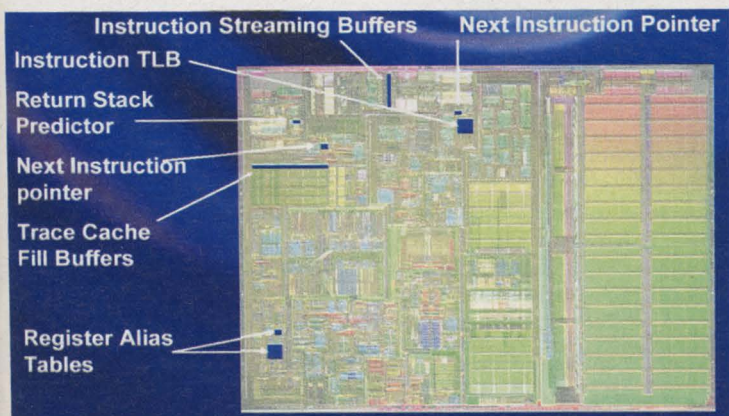
Hyper-Threading im neuen Pentium 4 mit 3,06 GHz

Bereits Anfang des Jahres hat Intel die Xeon-Prozessoren für Server mit der Hyper-Threading-Technik ins Rennen geschickt. Mit dem neuen Pentium-4-Modell soll diese Technik jetzt auch den Massenmarkt erobern. Beim Hyper-Threading fährt im Prozessor quasi ein zweiter Fahrer mit, doch wie beim Tandem hängt das Vorwärtkommen davon ab, ob dieser auch fleißig mit in die Pedalen tritt, sich faul ziehen lässt oder sogar bremst.

2 · 32 ist auch 64 – mit dieser Milchmädchenrechnung wollte Intel zum Weihnachtsgeschäft den zu diesem Zeitpunkt eigentlich erwarteten 64-Bit-Prozessoren von AMD etwas Zugkräftiges entgegensetzen. Und so errieten Intels Marketing-Strategen das schöne und kaum noch steigerungsfähige Schlagwort 'Hyper-Threading' (HT) für ein Verfahren, mit dem der Prozessorhersteller schon lange geheimniskrämerisch unter dem Codenamen 'Jackson Technology' herumexperimentiert hat und welches allgemein unter 'Simultaneous Multithreading' (SMT) bekannt ist.

Bei SMT arbeiten mehrere logische Prozessoren in einem Chip, bei Intels Hyper-Threading

sind es derzeit derer zwei. Beide rechnen weitgehend unabhängig voneinander, was helfen kann, den Durchsatz zu steigern. Dazu packt Intel nicht etwa zwei komplette Prozessor-Cores auf den Silizium-Chip (das 'Die'), sondern spendiert nur einen zusätzlichen Registersatz sowie einen eigenen Interrupt-Controller (APIC) und einige interne Verwaltungsstrukturen. Sowohl die Anzahl der Rechenwerke als auch Zahl und Größe der Caches bleiben gleich, beide logische Prozessoren nutzen diese gemeinsam. Dadurch ist es kein 'echtes' Zwei-Prozessor-System, sondern nur ein virtuelles aus Sicht von Software und Betriebssystem. Diese Erweiterung benötigt nicht einmal mehr als



Nur minimale Erweiterungen sind nötig (hier blau markiert) und schwupps hat man zwei logische Prozessoren.

ein Prozent an zusätzlichen Transistoren und erhöht die Die-Fläche lediglich um fünf Prozent. Interessanterweise hat die Konkurrenz bei AMD nach eigenen Angaben den gleichen Anteil der Die-Fläche in die 64-Bit-Architekturverbreiterung investiert, ebenfalls in der Hoffnung auf höhere Performance.

Für den Chiphersteller ist es sehr vorteilhaft, dass der Mehraufwand so gering ausfällt, denn dadurch erhöhen sich die Produktionskosten praktisch nicht. Auch wenn die ersten Prozessoren üblicherweise recht teuer vermarktet werden, ist doch langfristig abzusehen, dass Hyper-Threading keinen Aufpreis bedeutet. Bei Performance-Vorteilen, die Intel mit bis zu 35 Prozent angibt, ein willkommenes Geschenk an die Anwender.

Bislang versuchten die Prozessorbauer, die Anwender mit möglichst hohen Gigahertz-Zahlen zu ködern. Vor allem Intel hat das mit dem Pentium 4 erfolgreich vorexerziert und jetzt mit 3 GHz Takt einen neuen Rekord vorgelegt. Allerdings werden höhere Frequenzen zunehmend schwieriger. Ein Taktzyklus dauert jetzt nur noch 0,33 Nanosekunden – in dieser kurzen Zeit schafft selbst das Licht im Vakuum gerade mal 10 Zentimeter und die Wellen in Leitungen, die noch etwas langsamer sind, pflanzen sich vielleicht 6 Zentimeter fort.

Gefragt sind also intelligente Verfahren statt Gigahertz-Brute-Force und offenbar liegt der Schlüssel zur Performance in der Parallelverarbeitung. Diesen Trend verfolgt Intel nicht erst seit Neuestem, sondern

schon seit der Einführung des Pentium vor circa zehn Jahren. Der Pentium konnte im Gegensatz zum betagten 486 'superskalar' arbeiten, was bedeutet, dass er mehrere Instruktionen pro Takt gleichzeitig ausführen kann. Das alleine nützt allerdings noch nicht viel, denn meist ist der Programmcode so strukturiert, dass sich aufeinander folgende Instruktionen – bedingt durch Abhängigkeiten der Befehle und Daten – nicht parallel ausführen lassen. Daher waren auch die beiden Pipelines des Pentium nur selten ausgelastet.

Sein Nachfolger Pentium Pro brachte Abhilfe durch 'Out-of-Order-Execution' (OOO), bei der der Prozessor Instruktionen nicht zwingend in der Reihenfolge ausführt, in der sie im Programmcode stehen, sondern nach Bedarf so umordnet, dass sich möglichst viel Parallelität nutzen lässt. Man spricht dabei von 'Instruction Level Parallelism' (ILP). Aber auch das ist begrenzt. Studien haben gezeigt, dass sich in üblichem Code maximal 2,5 bis drei Instruktionen pro Takt zur gleichzeitigen Ausführung finden lassen. Und schon dafür sind etliche Tricks nötig, etwa so genanntes Register-Renaming oder spekulative Ausführungen von Befehlen. Diese führen aber zu sehr komplexen Systemen, die sich nur mit enorm viel Aufwand validieren lassen. Die Fehlerlisten moderner Prozessoren sprechen Bände davon.

Um diesem Dilemma zu entkommen, bietet es sich an, die verzweifelte Suche nach Parallelität nicht dem Prozessor zu überlassen, sondern schon im

Vorfeld dem Compiler oder dem Programmierer aufzubürden. Bei IA-64 (Itanium) beispielsweise versucht der Compiler, möglichst viele unabhängige Sequenzen zu entdecken und diese dem Prozessor in Bündeln zu übergeben (EPIC: Explicit Parallel Instruction Computing).

Zusätzlich kann man die Software in weit größerem Maßstab in logisch voneinander unabhängige Teile zerlegen, die als Threads gleichzeitig ablaufen können. Im Gegensatz zu ILP spricht man hier von 'Thread Level Parallelism' (TLP). Obwohl das Verwalten mehrerer Threads (Multithreading) für heutige Betriebssysteme keine Herausforderung mehr darstellt, bleibt der erhoffte Geschwindigkeitsvorteil allerdings aus, wenn mehrere Threads um die Ausführung auf nur einem Prozessor konkurrieren. Das ändert sich, wenn man zu Mehrprozessorsystemen mit 'Symmetrischem Multi-Processing' (SMP) übergeht. Doch dafür sind spezielle, teure SMP-Boards nötig und nicht zuletzt fallen zusätzliche Kosten für mehrere SMP-taugliche (und daher meist viel teurere) Prozessoren an.

Warten, warten

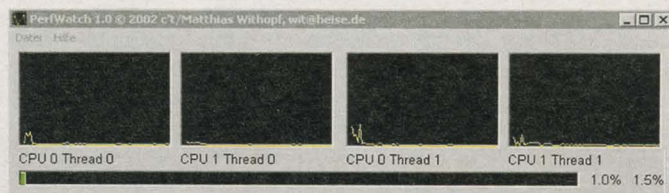
Man könnte annehmen, mit einem SMP-System das Optimum erreicht zu haben. Allerdings zeigt sich in der Praxis, dass diese Rechner oft nur zu einem verschwindend geringen Teil ausgelastet sind, sodass das Preis/Performance-Verhältnis sehr ungünstig ausfällt. Um die wirkliche Auslastung der Prozessoren feststellen zu können, reicht es keinesfalls aus, auf den

Windows Task-Manager zu schauen. Die meiste Zeit verplempern nämlich die Prozessoren mit Warten, Warten, Warten – zumeist auf den Hauptspeicher. Der Task-Manager bewertet diese unendlichen Wartezeiten aber auch als Arbeit. Für ihn ist der Prozessor nur dann 'idle', wenn der ganze Thread gerade auf Eis liegt.

Um die echte interne Auslastung eines Pentium-4- bzw. Xeon-Prozessors festzustellen, muss man schon tiefer in seine Innereien hineinschauen. Hierzu haben wir das Utility 'Perf-Watch' (siehe Softlink) entwickelt, das die Performance-Zähler des Prozessors benutzt.

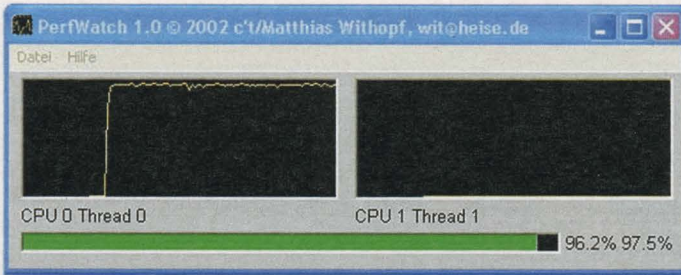
Die Messungen mit Perf-Watch werfen ein trauriges Bild auf die tatsächliche interne Prozessorauslastung. Denn selbst der rechenintensive SPEC2000-Benchmark langweilt einen Prozessorkern über weite Strecken, ganz zu schweigen von einem möglicherweise vorhandenen zweiten, der bei diesem Benchmark vollkommen ungenutzt bleibt. Da hilft es auch nicht, hoch optimierende Intel-Compiler einzusetzen, auch damit steigt die interne Last über den gesamten Verlauf der Suite selten über 30 Prozent, mehr als 60 Prozent werden nie erreicht. Bei höherem Takt wartet der Prozessor einfach nur schneller.

Damit wird klar, dass es nicht reicht, auf die Fähigkeiten des Prozessors zu vertrauen, die Arbeit zu parallelisieren, sondern dass man das brachliegende Potenzial auf andere Art nutzen muss, wenn man die interne Auslastung des Prozessors verbessern will. Und genau dazu sind mehrere Threads in Form des feinkörnigen Multi-



Der erste Eindruck täuscht: Der Rechner befindet sich nicht etwa im Tiefschlaf, sondern berechnet gerade einen Routenplan (Test 181.mcf) der rechenintensiven SPEC-2000-Benchmark-Suite auf einem Dual-Xeon-System unter Windows .NET Server RC1. Die interne Gesamtauslastung der beiden Prozessoren beträgt ein Prozent, der Windows Task-Manager zeigt hingegen Vollast für eine CPU an.





Bei geschickter Programmierung lässt sich der Pentium 4 bis fast zum theoretischen Maximum ausreizen, hier mit einem SSE-Programm, das jeweils vier Single-Precision-Gleitkommazahlen pro Instruktion verarbeitet. Hyper-Threading bringt in einem solchen (seltenen) Fall keinen Vorteil.

threading nützlich, bei denen die Threads nicht um ganze Prozessoren, sondern um die vorhandenen Rechenwerke innerhalb eines Prozessors konkurrieren. Dieses Konkurrenzverhalten bei SMT wird allerdings auch immer wieder als Gegenargument ins Feld geführt, da es ja nicht viel nützt, wenn sich die zwei Threads mehr behindern als nützliche Arbeit verrichten. Das tritt beispielsweise auf, wenn bereits ein Thread so gut programmiert ist, dass er die Einheiten stark auslastet und er dank geschicktem Cache-Management und Prefetches kaum Speicherwartzeiten erleidet. Richtig hoch optimierter Code lässt also wenig Raum für Lückenfüller wie Hyper-Threading.

Auslastungs-Szenario

Tatsächliche Performance-Messungen ergeben aber ein anderes Bild. Da zeigt sich, dass auch bei rechenintensiven Prozessen in fast allen Fällen genügend Prozessorkapazität frei

bleibt, die ein zweiter logischer Prozessor nutzen könnte.

Zunächst stellt sich die Frage, wie man die interne Prozessorauslastung definiert, und vor allem, wie sie sich messen lässt. Dazu ist ein kleiner Abstecher zu dem von Intel als NetBurst-Architektur bezeichneten Aufbau der Pentium-4- und Xeon-Prozessoren notwendig. Eines der wichtigsten Merkmale ist die Art und Weise der Codeausführung. Die x86-Instruktionen werden nämlich nicht direkt verstanden, sondern beim Dekodieren in (durchschnittlich zwei) Mikro-Operationen (μ ops) zerlegt, die einfacher strukturiert sind und sich besser in Hardware ausführen lassen. Die fertig dekodierten μ ops legt der Pentium 4 in einem als 'Execution Trace Cache' bezeichneten Speicher ab. Das erspart viel überflüssige Dekodierarbeit, wie sie beim Vorgänger Pentium III immer wieder an der Tagesordnung war, der beispielsweise bei kleinen Schleifen mit viel Mühe dieselben Instruktionen dekodieren musste.

Eine der wichtigsten Kenngrößen der aktuellen NetBurst-Implementierung ist, dass maximal drei μ ops pro Taktzyklus abgeschlossen (retired) werden können. Bei der aktuellen Taktfrequenz von 3,06 GHz sind das also mehr als neun Milliarden Operationen pro Sekunde. Dieses theoretische Optimum lässt sich aber nur erreichen, wenn keine Wartezeiten auftreten. Um Stau-Problemen auf die Spur zu kommen, sind im Pentium 4 interne Zähler (Performance Counter) eingebaut, die akribisch Buch führen, sowohl über die bewältigte Arbeitsmenge, aber auch über auftretende Missstände. Diese Performance Counter liest das Utility 'Perf-Watch' aus und stellt sie grafisch dar.

Gründe für lange Wartezeiten und schlechte interne Auslastung gibts viele, einer der wichtigsten ist der im Vergleich zum Prozessor extrem langsame Hauptspeicher. Trotz bis zu dreistufiger Cache-Hierarchie kommt es häufig vor, dass sich die aktuell benötigten Daten dort nicht befinden (Cache Miss) und nachgeladen werden müssen. Es dauert 'eine halbe Ewigkeit', bis die ersten Daten ankommen, eine Zeit, die als Latenz des Speichers bezeichnet wird. Um eine Vorstellung davon zu bekommen, wie das Missverhältnis zwischen CPU und Speicher ist, haben wir mit einem kleinen Messprogramm die Wartezeit für einen Cache Miss ermittelt.

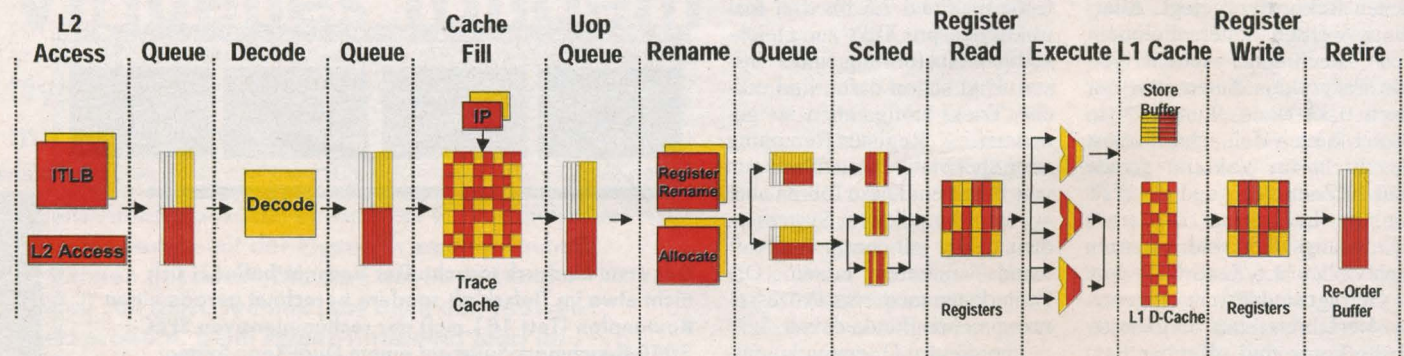
Auf dem verwendeten Testsystem (ASUS-P4PE-Board mit Intel-845PE-Chipsatz und DDR-2700/2-0-3-3-Speicher) betrug diese gemessene Latenzzeit 124 ns. Das entspricht 380 Pro-

zessortakten, in denen der Prozessor jeweils maximal drei μ ops ausführen könnte, also insgesamt 1140 Mikro-Operationen. Obwohl die Out-of-Order-Ausführung von NetBurst bis zu 126 Instruktionen gleichzeitig in Bearbeitung haben kann, sind selbst in diesen seltenen Fällen fast 90 Prozent des Potenzials vergeudet – ein Potenzial, das von Hyper-Threading nutzbar gemacht werden könnte.

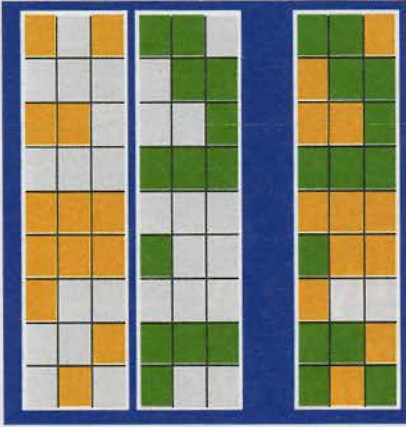
Bei SMP-Systemen ist übrigens die Speicherlatenz im Regelfall noch höher, auf dem Dual-Xeon-Testsystem ergaben sich ungefähr 20 Prozent längere Latenzzeiten. Der Grund liegt hier vor allem in zusätzlichen Prozessoraktivitäten, die die Datenkonsistenz zwischen den Prozessoren sicherstellen (Cache-Kohärenz mit aufwendigem Bus Snoop und Broadcast). Daneben haben SMP-Systeme meist auch größere Speichersysteme mit registered DIMMs, die längere Zugriffszeiten benötigen.

Um zu verstehen, wie Hyper-Threading funktioniert, muss man sich vergegenwärtigen, dass jede Mikro-Operation beim Pentium 4 eine ausgesprochen lange, mindestens 20-stufige Pipeline durchläuft. Jede Pipeline-Stufe erledigt dabei nur einen Bruchteil der Arbeit, beispielsweise das Holen des μ op-Codes aus dem Trace Cache (Fetch) oder das Verteilen der Operation auf die verfügbaren Rechenwerke (Scheduling). Es gibt sogar zwei Stufen, die nur als Wartestufen vorgesehen sind, um in dieser Zeit die Daten von der einen Chipseite auf die andere zu transportieren.

Es gelingt jedoch fast nie, die Pipeline ohne Stocken am Lau-



Die beiden Hyper-Threading-Prozessoren haben eigene Registersätze sowie getrennte Reorder- und Store-Buffer. Den Rest der Prozessor-Ressourcen teilen sie sich brüderlich. Wenn keine Wartezeiten auftreten, werden die Instruktionen abwechselnd (rot und gelb) aus dem Trace-Cache in die Pipeline hineingeschoben.



Links die geringe Ausnutzung der Prozessor-Pipeline für zwei separate Threads, normalerweise mit vielen Lücken (graue Felder). Rechts ist die Pipeline von zwei Threads gleichzeitig belegt, doch selbst damit gibt's noch Lücken. [2]

fen zu halten, da dauernd kleinere und größere Wartezeiten auftreten (Pipeline stalls). Das gibt dann Produktionslücken im Ablaufplan (die grauen Kästchen in der Abbildung oben), die sich wie Luftblasen durch die einzelnen Stufen schieben, ohne einen sinnvollen Beitrag zu leisten. Je mehr Lücken es gibt, umso ineffizienter arbeitet der Prozessor. Mit Hyper-Threading kann man solche Lücken durch Arbeit für einen zweiten, unabhängigen Programmteil (Thread) belegen. Dadurch führt der Prozessor den ersten Thread zwar nicht schneller aus, es steigt also nicht die absolute Performance, aber der Durchsatz erhöht sich, indem mehr Arbeit pro Zeiteinheit ausgeführt werden kann.

Leider messen die meisten Benchmark-Programme nur die absolute Leistung eines Systems mit Single-Thread-Anwendungen und nicht dessen Durchsatz und können deswegen Hyper-Threading nicht so recht würdigen. Das ist wohl auch der wesentliche Grund, warum Hyper-Threading zunächst nur für Server-Systeme vorgesehen war, wo im Unterschied zu klassischer Desktop-Software nahezu ausschließlich der Durchsatz ausschlaggebend ist. Aber auch Desktop-Software ist nicht auf Dauer konstant, auch das Benutzerverhalten ändert sich im Laufe der Zeit und so kommt dem Durchsatz mehr Bedeutung zu. Es wird also Zeit, dass sich neue Benchmarks mit dieser Situation angemessen auseinandersetzen.

Bei der Wahl des Betriebssystems ist wichtig, dass der Scheduler, der die aktiven Threads der Prozesse auf die verfügbaren CPUs verteilt, schon einmal von Hyper-Threading gehört hat.

Denn die Prozessoren sind nicht alle gleichwertig, das 'S' in SMT bedeutet eben nicht 'symmetrisch' wie in SMP, sondern 'simultaneous'. Insbesondere dann, wenn zwei oder mehr physische Prozessoren vorhanden sind – also bei den Xeons –, sollte der Scheduler die Unterschiede zwischen logischem und physischem Prozessor berücksichtigen und der Auswahl-Algorithmus einer freien physischen CPU gegenüber einer logischen den Vorzug geben, die ihre Prozessorressourcen ja mit ihrer Zwilling-CPU schwesterlich teilen muss.

Statt also zwei Prozesse auf beide logische CPUs eines Prozessors zu verteilen, soll der Scheduler – falls sie denn vorhanden sind – lieber zwei reale Prozessoren rekrutieren. Ältere SMP-Betriebssysteme wie Windows NT und 2000 haben dabei ihre Schwierigkeiten, worunter die Performance leidet.

Heimtückische Bremse

Geeignet sind hingegen Linux (ab Kernel 2.4.18) und Windows XP, die Home Edition reicht für den Pentium 4. Man muss den zusätzlichen logischen Prozessor nicht extra lizenzieren, das Betriebssystem zählt nur die physischen Prozessoren zur Lizenzierung. Hier sei allerdings nicht verschwiegen, dass der Multiprozessor-Kernel durch aufwendigere Synchronisationsmechanismen gegenüber der Uniprozessor-Variante etwas bremst. Diesen Nachteil gilt es durch Hyper-Threading wieder aufzuholen.

Eine weitere Hyper-Threading-Besonderheit im Unterschied zu SMP ist das Leerlaufverhalten der logischen CPUs. Wenn ein Thread auf irgend-

etwas wartet – gemeinhin auf das Ergebnis eines anderen Thread –, so kann die wartende CPU die arbeitende Zwillingsschwester massiv in ihrer Arbeit behindern, nämlich dann, wenn die Threads mit 'Busy Waiting' über eine globale Variable synchronisiert sind. Sie liest dann laufend in einer Schleife eine Speicherstelle aus und blockiert damit völlig unnötig einige Funktionseinheiten. Diese furchtbare Synchronisationsmethode war zwar schon immer der Feind sinnvoller Multithreading-Programmierung, allerdings fiel ihre negative Wirkung bei SMP-Systemen bis auf überflüssigen Stromverbrauch nicht so sehr auf, da hier die gegenseitige Beeinflussung der Prozessoren gering ist. Und so kommt es, dass zuhause so genannte 'Multithreaded Programme' verbreitet sind, die diesen Namen nicht verdienen und bei denen sich mangels Kooperation der Threads der Hyper-Threading-Vorteil schnell ins Gegenteil verkehrt. Hier zeigt sich also ein weiteres Problem für ein korrektes Benchmarking.

Um die Bremswirkung zu vermeiden, ist dringend angesagt, die Synchronisationsmöglichkeiten des Betriebssystems zu benutzen, also beispielsweise EnterCriticalSection unter Windows oder pthread_mutex_lock aus der Pthread-Bibliothek unter Unix, um den Zugriff auf gemeinsame Ressourcen zu regeln. Nicht nur für HT ist es natürlich noch besser, Lock-freie Algorithmen zu verwenden, die es beispielsweise für die Listenverwaltung gibt und die mit atomaren Operationen (Interlocked-Funktionen in Windows) arbeiten.

Hyper-Threading ist wie SIMD (MMX, SSE et cetera) eine Technik, von der Programme nicht automatisch profitieren. Während sich aber SIMD vorwiegend an Multimedia-Anwendungen richtet und sich für viele Algorithmen nicht eignet, sieht das bei HT günstiger aus. Sobald mindestens zwei Programmteile gleichzeitig aktiv sind, bringt es bereits einen Vorteil. Dafür muss die Software jedoch nicht zwingend Threads verwenden, zwei gleichzeitig laufende Anwendungen (Multiprocessing) reichen aus. Im Windows Task-Manager lässt

sich überprüfen, dass selbst bei einem frisch gestarteten Rechner daran kein Mangel ist.

Kontext

Um den Gewinn durch Hyper-Threading in der Praxis zu beleuchten, haben wir ein Testprogramm für Windows entwickelt, das verschiedene Aufgaben absolviert, beispielsweise die arithmetische Kompression und Dekompression von Speicherblöcken, die zuvor mit Zufallszahlen gefüllt wurden. Neben diesem Integer-Test steht auch die Berechnung der Quadratwurzel mittels Heron-Verfahren (Gleitkomma-Code) und die Bild-Dekompression mit Intels hoch optimierter JPEG-Library ijl15.dll (nutzt MMX- und SSE2-Instruktionen) zur Auswahl. Die verschiedenen Tests lassen sich als parallel laufende Threads kombinieren, alternativ ist aber auch möglich, jeweils nur einen Test zu starten, dafür aber zwei Instanzen derselben Anwendung. Um nicht auf den Scheduler des Betriebssystems angewiesen zu sein, haben wir die Threads manuell auf Prozessoren verteilt. Die dazu notwendige Funktion SetThreadAffinityMask kennt Windows ab NT 3.5. Die Linux-Gemeinde konnte sich noch nicht einigen, diese Funktionalität in den User-Kernel zu integrieren, im Entwicklerkernel sind allerdings bereits die Syscalls set_affinity und get_affinity eingeplant.

Der einfachste Fall ist es, jeweils gleiche Tests in zwei Threads zu kombinieren, die auf die beiden logischen Prozessoren verteilt sind. Da bei Hyper-Threading die Anzahl der Rechenwerke nicht erhöht ist, sondern die vorhandenen gemeinsam genutzt werden, ist klar, dass die beiden Aufgaben nicht in voller Geschwindigkeit ablaufen können, sondern sich gegenseitig etwas ausbremsen. Bei Hyper-Threading ist also anders als bei SMP nicht direkt der Gewinn messbar, sondern eher der 'Bremsfaktor', der in dem Integer-Test dazu führt, dass beide Threads nur noch mit 63,5 Prozent der ursprünglichen Geschwindigkeit laufen. Allerdings summiert sich die erledigte Arbeit, wodurch sich insgesamt ein Gewinn von 27 Prozent ergibt.

Alias-Probleme

Um zu überprüfen, wie zufrieden der Prozessor mit dieser Lösung ist, kam Intels VTune [1] zum Einsatz. Dieses Tool liest die bereits erwähnten Zähler des CPU-internen Performance-Monitors aus und hilft, Software zu optimieren, indem es Schwachstellen sichtbar macht. Und prompt zeigt sich auch ein Problem, nämlich so genannte Aliasing-Konflikte des Cache. Dahinter verbirgt sich die ineffiziente Ausnutzung der Cache-Einträge, wenn sich häufig benutzte Einträge gegenseitig verdrängen (Thrashing), was die gefürchteten und zeitraubenden Cache Misses zur Folge hat.

Die NetBurst-Architektur arbeitet, wie mancher Serverprozessor auch (Alpha 21264, UltraSparc etc.) beim L1-Datencache mit einer Technik, die sich 'virtually indexing, physically tagged' nennt. Die virtuelle Adressierung ermöglicht beim Pentium 4 einen besonders schnellen Zugriff (nur zwei Takte Latenz), allerdings speichert der Prozessor von der virtuellen Adresse des Cache-Eintrags nur deren niederwertigen 20 Bits. Daraus folgt, dass Daten, die sich in zwei Cache-Einträgen (von jeweils 64 Byte pro Eintrag) befinden, die im virtuellen Adressraum genau 1 MByte auseinander liegen, um denselben Eintrag konkurrieren (Alias-Konflikt).

Die bisherigen Pentium-4 und Xeon-Modelle haben dieses Problem sogar noch in verschärfter Form, da sie sogar nur die unteren 16 Adressbits auswerten, was zu '64K-Aliasing-Konflikten' führt. Da Windows beim Erzeugen von Threads standardmäßig die Stackbereiche genau ein Megabyte auseinander legt, schlägt das Problem in beiden Fällen voll zu, sowohl in Form von 64-KByte- als auch 1-MByte-Konflikten. Im Test führen beide Threads identischen Code aus, daher benutzen sie die gleichen Stack-Zugriffsmuster und konkurrieren so intensiv um dieselben Cache-Einträge, was die Performance deutlich senkt. Intel empfiehlt den Programmierern eine simple Vermeidungsstrategie mit der `_alloca`-Funktion, die eine variable Speichermenge auf dem Stack alloziert. Wenn es gelingt, in den Thread-Funktionen die

Hyper-Threading Gewinn

Benchmark	Beschleunigung	interne Prozessorauslastung
2x Kompression (Integer)	+43 Prozent	40 Prozent → 60 Prozent
2x Heron (Gleitkomma)	+44 Prozent	12 Prozent → 17 Prozent
2x JPEG-Dekompression (MMX+SSE2)	+16 Prozent	35 Prozent → 41 Prozent
1x Integer + 1x Gleitkomma	+70 Prozent	42 Prozent

Stackpointer um ein Vielfaches der Größe eines Cache-Eintrags (Cache Line) von 64 Bytes gegeneinander zu verschieben, ist diese Angelegenheit so gut wie erledigt. Diese Optimierung durch eine zusätzliche Zeile im Quelltext wird prompt belohnt, indem die Performance eines einzelnen Thread von 63,5 Prozent auf 71,5 Prozent steigt und damit der Vorteil durch Hyper-Threading von 27 Prozent auf immerhin 43 Prozent Mehrleistung. Von linearer Skalierung ausgehend entspricht diese 'Mehrarbeit' einer virtuellen Beschleunigung des Pentium 4 auf 4,4 GHz.

Wenn man verschiedene Aufgaben kombiniert, kann das Ergebnis noch günstiger ausfallen, da es weniger Konkurrenz um dieselben Funktionseinheiten im Prozessor gibt. Beim gleichzeitigen Ablaufen des Integer- und Gleitkomma-Codes des Testprogramms stört sich die beiden Threads relativ wenig. Vor allem der Thread mit Gleitkomma-Code, der hauptsächlich rechnet und kaum auf den Hauptspeicher zugreift, lief nahezu ungestört weiter (95 Prozent), während der Integer-Thread ein wenig mehr abgebremst wurde (75 Prozent). Da werden zwar Äpfel und Birnen zusammengerechnet, aber grob ergibt sich für dieses Mischobst ein Leistungsgewinn von 70 Prozent.

HT ungleich HT

Beim neuen Pentium 4 (C-Step) spielt es übrigens keine

Rolle, ob die beiden Aufgaben als zwei Threads in einem Programm oder als zwei separate Anwendungen in verschiedenen Prozessen umgesetzt sind, das Ergebnis ist in beiden Fällen mit etwa 40 Prozent höherem Gesamtdurchsatz gleich. Ganz anders sieht es beim Hyper-Threading-Vorfahren Xeon (mit B-Step) aus. Dieser erleidet einen geradezu dramatischen Einbruch beim zweimaligen Aufruf des Kompressionstests auf ein Drittel – und zwar nicht nur für jeden Prozessor, sondern insgesamt.

Ursache ist auch hier wieder das Aliasing-Problem, diesmal betrifft es jedoch nicht nur den Stack, sondern auch die Daten, denn das Testprogramm fordert die Speicherblöcke für die Kompression und Dekompression vom Heap an. Bei der Thread-Lösung werden diese Blöcke im gleichen virtuellen Adressraum angefordert und liegen dann hintereinander, bei zwei Instanzen liegen sie jedoch an identischen virtuellen Adressen, sozusagen übereinander mit hohem Störpotenzial. Benchmarks, die wie etwa SPECrate mehrmals dasselbe Programm anstarten, liefern daher auf dem Xeon miserable Ergebnisse [4], weil sie genau diese Worst-Case-Situation mit mehreren Programmen ausstoppen. Zwar gibt es einen Weg, wie man mithilfe des Microsoft-Tools EditBin (gehört zu Visual C++) von fertigen Programmen zumindest die Stack-Größe nachträglich ändern kann, was schon eine spürbare Verbesserung auf dem Xeon be-

wirkt. In die Gewinnzone führt das HT jedoch auch nicht.

Für Server, bei denen meist nur eine Multithreaded-Anwendung läuft, ist dieses Verhalten dennoch tolerabel. Für Desktops hingegen wäre es ziemlich desaströs – und so haben die Intel-Entwickler am Pentium-4-Design herumgebastelt und die Cacheverwaltung so optimiert, dass der P4 und mit ihnen die neueren Xeons sogar jetzt besser an zwei Prozesse angepasst ist als an zwei Threads. Wie allerdings die Mannen um Glenn Hinton das hingezaubert haben, ist derzeit noch ein wohlbehütetes Geheimnis.

Hier zeigt sich übrigens, warum ein Programm – auch wenn es auf einem SMP-System erfolgreich getestet wurde – nicht zwangsläufig für Hyper-Threading optimal sein muss. Denn die Cache-Aliasing-Problematik tritt bei zwei 'echten' CPUs nicht auf, da hier jede ihren eigenen Cache verwaltet und es nicht zu solchen Konflikten kommen kann.

Von Quanten und Spins

Das Testprogramm stellt einen idealisierten Fall dar, denn beide Aufgaben laufen voneinander unabhängig, eine Abstimmung (Synchronisation) der Arbeitsteile ist nicht erforderlich. In der Praxis ist das jedoch meist anders. Wenn mehrere Programmteile an einer gemeinsamen Aufgabe wirken, ist eine gegenseitige Abstimmung nötig. Bei Windows ist die 'Critical Section' die einfachste Methode, um sicherzustellen, dass nur ein Thread einen bestimmten Bereich im Programm ausführen kann. Dazu ruft er die Windows-Funktion `EnterCriticalSection` auf und gibt ihn so schnell wie möglich wieder mit `LeaveCriticalSection` frei. Tritt der günstigste Fall ein, dass noch kein anderer Thread die

Die Synchronisation in Spin-Wait mit der PAUSE-Instruktion. Der Spin-Lock-Variablen sollte man aus Performancegründen gleich eine ganze Cacheline gönnen. Aber bitte nicht selbst kodieren, sondern die zuständigen Betriebssystemfunktionen benutzen!

```
KiAcquireSpinLock:
start:
    lock     bts     dword ptr [ecx],0 ; setze Bit 0 und teste es
           jnb     wait_loop          ; war es schon gesetzt? -> warten
           ret                     ; neu gesetzt, SpinLock ist jetzt belegt

wait_loop:
    test     dword ptr [ecx],1 ; ist SpinLock gerade frei geworden?
           je      start             ; ja -> versuche, es zu belegen
           pause   ; warten, derzeit ca. 44 T
           jmp     wait_loop         ; neuer Anlauf...

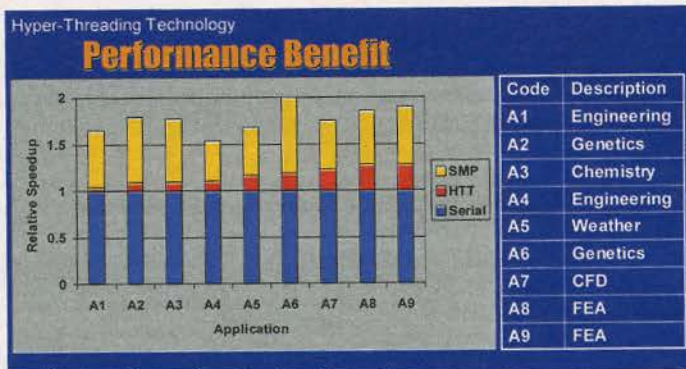
KiReleaseSpinLock:
    mov     byte ptr [ecx],0 ; SpinLock freigeben, Bit 0 löschen
    ret
```


selbe Critical Section angefordert hat, kostet das nur wenige Instruktionen im User-Modus, der Kernel wird dazu nicht bemüht. Anders sieht es aus, wenn der Bereich bereits belegt ist. Dann wird es notwendig, dass der Thread wartet, bis der konkurrierende Programmteil den Bereich wieder verlassen und die Critical Section freigegeben hat. So lange legt sich der wartende Thread schlafen mit dem Wunsch ans Betriebssystem, geweckt zu werden, wenn's weiter gehen soll.

Diese Methode geht zwar sparsam mit den Ressourcen um, es kann aber dauern, denn das Betriebssystem verteilt nur in bestimmten Intervallen Prozesse auf die CPUs. Dieses Intervall – es heißt unter Windows 'Quantum', andere Systeme sprechen von 'Time Slice' oder Zeitscheibe – ist bei den Professional/Workstation-Versionen zwischen 20 und 60 Millisekunden einstellbar, bei den auf Durchsatz optimierten Server-Versionen liegt es sogar bei 120 ms. Das bedeutet, dass hier der Scheduler den wartenden Thread im ungünstigsten Fall 1/8 Sekunde später aufweckt, nachdem die Critical Section frei geworden ist. Da Critical Sections üblicherweise nur extrem kurze Codeteile schützen, ist der zeitliche Nachteil des Wartens per Kontextwechsel gewaltig.

Zur Abhilfe hat Windows mit dem Service Pack 3 von NT4 eine Optimierung für Multiprozessorsysteme geschaffen, nämlich den 'Spin Count', und verwendet ihn selbst beispielsweise zur Serialisierung der Heap-Verwaltung. Dabei wird bewusst ein kurzes 'Busy Waiting' (Abfrageschleife) in Kauf genommen. Jede Critical Section kann einen Zähler haben, der angibt, wie oft eine belegte Critical Section zu prüfen ist, bevor sich der wartende Thread schlafen legt. Wird der Bereich frei, während noch der Spin Count beim Warten runtergezählt wird, wechselt er sofort den Besitzer, ohne die Wait-Funktion und den Scheduler zu bemühen. In günstigen Fällen kann das mehrere Millionen Mal schneller gehen.

So weit, so gut, doch Hyper-Threading führt diese Optimierung wieder ad absurdum, denn dieses 'Spinnen' des logischen



Hyper-Threading (rot) kann mit echten Dual-Prozessoren (gelb) zwar nicht mithalten, aber dafür bekommt man bis zu 25 Prozent Performance-Steigerung nahezu umsonst.

Prozessors lastet ihn weitgehend aus; er bremsst auch die andere logische CPU, sodass diese langsamer vorankommt, was wiederum die erwünschte Freigabe der Critical Section weiter verzögert – ein Teufelskreis. Um dem zu entkommen, hat Intel der NetBurst-Architektur eine neue Instruktion spendiert: PAUSE. Damit signalisiert ein Programm dem Prozessor ein gewolltes Busy Waiting, das das Warten deutlich langsamer laufen lässt. Das schont die Prozessor-Ressourcen und den Stromverbrauch, außerdem vermeidet es die 'Memory Ordering Violation', die beide Prozessor-Threads noch zusätzlich ausbremst. Diesen Konflikt gibt es zwar schon seit dem Pentium Pro, seine Schädlichkeit ist aber beim Pentium 4 um den Faktor 25 größer. Das Busy Waiting ohne PAUSE ist einer der Gründe, warum eine Anwendung auf Windows 2000 womöglich schlechter läuft als unter XP, selbst wenn man nicht auf den Betriebssystem-Scheduler vertraut und die Threads manuell auf die Prozessoren verteilt.

Die PAUSE-Instruktion lässt sich auf allen x86-Prozessoren verwenden, ob von Intel oder von der Konkurrenz, denn sie setzt sich aus einem NOP mit zusätzlichem REP-Präfix zusammen, was Nicht-NetBurst-Prozessoren schlichtweg ignorieren. Mit einem Testprogramm haben wir die Wirkung von PAUSE vermessen. Obwohl der Windows Task-Manager weiterhin eine 100-Prozent-Auslastung anzeigt, sinkt die interne Prozessorauslastung in einer Schleife von 80 Pro-

zent auf acht Prozent. Während der Pause tut der logische Prozessor viele Takte rein gar nichts, vor allem stört er den anderen Prozessor weniger. Die mit PAUSE versehene Schleife läuft dann logischerweise deutlich langsamer, nach unseren Messungen etwa um den Faktor 20.

Fazit

Hyper-Threading, Intels Version von Simultaneous Multi-Threading, ist eine taugliche Methode, um die Leistung des Prozessors in Bezug auf den Durchsatz zu erhöhen – zumindest dann, wenn neue Software auf die spezifischen Befindlichkeiten von Hyper-Threading Rücksicht nimmt. Man bekommt HT nahezu zum Nulltarif – wen es stört, der kann es einfach im BIOS-Setup abschalten.

Derzeit kann man noch nicht erwarten, dass bestehende Programme bereits dafür optimiert sind. Wer hauptsächlich fertige Anwendungen einsetzt, wird anhand der konkreten Software-Versionen prüfen müssen, inwieweit Hyper-Threading für ihn Vorteile bringt. Intel wird in der Werbung vor allem den Parallelbetrieb verschiedener Anwendungen in den Vordergrund stellen, wo sich bei passender Zusammenstellung in der Tat deutliche, real spürbare Performance-Verbesserungen zeitigen. Das quadriert allerdings das Benchmark-Dilemma, denn nun kommt es nicht allein auf die ohnehin problematische Auswahl einer Test-Software an, sondern auf eine sinnvolle Mischung gleichzeitig ablaufender unterschiedlicher Aufgaben.

Wer Software entwickelt, wird sich intensiv mit dem Thema Optimierung befassen müssen. Denn wer die Software nicht an die besonderen HT-Gegebenheiten anpasst, verschenkt reichlich Potenzial. Man darf nicht den Fehler begehen, wegen nicht optimierter Anwendungen Hyper-Threading zu disqualifizieren. Ohne Tools wie VTune [1] zu konsultieren, um prozessorinterne Konflikte auszuschließen, die die Performance auf einen Bruchteil schrumpfen lassen, bleibt jede Aussage über schlechte Hyper-Threading-Performance reine Spekulation. Und das Argument, dass 'echte' Dual-Prozessorsysteme den Hyper-Threading-Kollegen klar überlegen sind, zieht auch nicht, da hierfür Aufwand und Kosten in unvergleichlich viel höheren Regionen rangieren.

Von den regelmäßigen Takterhöhungen der Prozessoren kommt in der Praxis immer weniger beim Benutzer an, denn die wichtigste Ursache für die schlechte interne Prozessorauslastung sind Wartezeiten auf dem Hauptspeicher. Je höher der CPU-Takt, umso mehr steigt dieses Missverhältnis. Hyper-Threading hilft hier, einen Teil der ansonsten verlorenen Performance zurückzugewinnen. Es ist nach Intels eigenen Aussagen derzeit noch relativ einfach gestrickt. Das lässt noch einiges erwarten, wenn Intel sein Versprechen wahr macht und es in weiterentwickelter Form in alle Prozessoren der Pentium-4-Linie einbaut. Und schließlich ist der wohl angesehenste SMT-Experte Joel Emer via DEC und Compaq vor einiger Zeit als Fellow bei Intel gelandet. (as)

Literatur

- [1] VTune, kostenloser Download der 30-Tage-Version
- [2] Intel Technology Journal zum Thema HyperThreading, www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf
- [3] Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, <http://developer.intel.com/design/pentium4/manuals/248966.htm>
- [4] Andreas Stiller, Gipfelstürmer, c't 6/02, S. 224