IA64 vs x86_64

Felix von Leitner
felix-ccc@fefe.de

28. Dezember 2002

Zusammenfassung
Was sollte man als Assembler-Hacker über IA64 und X86_64 wissen?

Einführung

Prozessoren arbeiten auf Befehlsebene. Bei RISC sind alle Befehle gleich lang (gewöhnlich so breit wie die Register). Bei x86 sind Befehle variabel lang.

Heutige Prozessoren erreichen ihre Geschwindigkeit durch Pipelining und Superskalarität.

Bei Pipelining wird die Befehlsausführung auf n Schritte aufgeteilt (z.B. *Laden*, *Dekodieren+Dispatch*, *Ausführen* und *Ergebnisse schreiben*; das waren die Pipeline-Schritte beim PowerPC G3), und dann überlappt man die Ausführung.

Während von diesem Befehl *Dekodieren+Dispatch* läuft, läuft vom nachfolgenden Befehl die *Laden*-Phase.

Pipelining

Offensichtlich kann man den nächsten Befehl nur laden, wenn man weiß, welches der nächste Befehl ist. Daher haben Prozessoren ausgeklügelte Vorhersage-Mechanismen. Damit raten sie den Ausgang von Sprüngen und führen dann dort die Befehle aus.

Das nennt man spekulative Ausführung. Die aktuelle Athlon-Generation hat bei üblichen Anwendungen über 90% Trefferquote bei der Sprungvorhersage. Wenn ein Sprung falsch vorhergesagt wird, müssen alle Befehle in der Pipeline verworfen werden. Hinzu kommt, daß einem das gewöhnlich erst gegen Ende der Pipeline auffällt.

Trotzdem: je höher die Taktrate, desto weniger kann man pro Zyklus machen. Daher wird mit höherem Takt auch die Pipeline immer länger (Athlon: 11, P4: 20!).

Pipelining

90% richtrige Vorhersage klingt gut, aber rechnen wir mal nach!

Jeder 10. Sprung wird falsch vorhergesagt. Bei einer Pipeline-Tiefe von 20 gehen jedes Mal 20 Befehle verloren. Nehmen wir an, daß jeder 10. Befehl ein Sprung ist.

Dann ist jeder 100. Befehl ein falsch vorhergesagter Sprung, der 20 Takte killt. Das sind 20% der Gesamt-Performance des Prozessors, die hier verloren gehen!

In der Praxis ist es nicht ganz so schlimm, weil die Prozessoren die meisten Zeit auf den Speicher oder Eingaben warten.

Sprungdichte

Wie hoch der Anteil der Sprünge an Code ist, hängt natürlich von der Anwendung ab. Besonders wenige Sprünge haben typische DSP- und Mathe-Anwendungen wie Bildbearbeitung, Fouriertransformationen, wissenschaftliche Simulationen und Krypto-Routinen.

Besonders viele und dazu schlecht vorhersagbare Sprünge haben Compiler, Interpreter und insbesondere Spiele-AI wie Schach- oder Go-Programme (hier können schon mal 20-50% falsche Vorhersage vorkommen).

Hyperthreading

Hyperthreading heißt, daß der Prozessor zwei Maschinen-Kontexte hat, d.h. zwei Register-Files, zwei Dekoder. Der Software gegenüber sieht das aus wie zwei Prozessoren. Wenn der eine Prozessor stallt (z.B. wegen eines schlecht vorhergesagten Sprunges), werden halt die Befehle des anderen Threads ausgeführt.

Daher kann man beim Pentium 4 sehen, daß von Hyperthreading vor allem gcc und gnuchess profitieren.

Kann man sich mit Hyperthreading die Sprungvorhersage sparen?

Nein. Die meisten Leute fahren jeweils nur einen Thread auf ihrem System.

Superskalarität

Moderne Prozessoren sind superskalar, d.h. sie sind wie ein Mediamarkt aufgebaut. Im 1. Stock sind Kameras, im 2. Stock sind Bücher, im 3. Stock sind Stereoanlagen, im 4. Stock sind Kühlschränke. Die Größe der einzelnen Bereiche ergeben sich aus dem an den Produkten gemachten Gewinn.

Genau so ist das bei Prozessoren. Es gibt einen Eingang zur Straße; den macht man so breit, wie es geht. Dahinter gibt es funktionale Einheiten für Integer-Berechnungen, für Fließkomma-Berechnungen, für SIMD und für den Speicher-Zugriff.

Nun dauert Laden aus dem Speicher nicht immer gleich lang, d.h. man kann das nicht in eine Pipeline pressen. Daher hat man mehrere Pipelines unterschiedlicher Länge mit Puffern dazwischen.

AMD Hammer

Der Hammer ist im Grunde einfach ein aufgebohrter Athlon. AMD hat die Entwickler gefragt, was sie bei x86 am meisten stört. Als häufigste Antworten kamen:

1. Gebt uns mehr Register!
2. Der x87 FPU-Stack stinkt!
3. Wir wollen 64-bit Adressen, aber ohne den Code-Bloat!
4. Gebt uns ein anständiges ABI!
5. PC-relative Adressierung jetzt!

Hammer: Register, x87

Hammer hat 16 64-bit Integer-Register. 32-bit Operationen auf 64-bit Register setzen die oberen 32 Bit auf Null. Das Linux ABI wurde angepaßt, um bis zu 6 Argumente in Registern zu übergeben.

Floating-Point macht man mit der SSE2-Einheit, die beim Hammer auch 16 Register hat. Das Linux ABI wurde angepaßt, um bis zu 9 Argumente in den SSE2-Registern zu übergeben. Für `long double` gibt es auch noch den x87 Register Stack, aber das ist ja eher selten.

Dadurch entfällt auch die unsägliche `-ffloat-store` Problematik mit den Genauigkeiten.

Hammer: 64-bit Adressen

Wie läd man große Adressen? Wenn man da 64-bit Werte zwischen die Instruktionen schreibt, kann am Ende der Dekoder in der CPU nur zwei Instruktionen pro Zyklus lesen statt der üblichen 4-5!

Bei MIPS hat man das besonders abstoßend gelöst (SPARC und Alpha sehen genau so aus):

```
0:   3c1c0000        lui     gp,0x0
                     0: R_MIPS_HI16  _gp_disp
4:   279c0000        addiu   gp,gp,0
                     4: R_MIPS_LO16  _gp_disp
8:   0399e021        addu    gp,gp,t9
c:   8f820000        lw      v0,0(gp)
                     c: R_MIPS_GOT16 blub
```

Hammer: 64-bit Adressen

Bei ARM hat man das Problem sehr elegant gelöst:

```
        ldr     r3, .L2
[...]
.L2:
        .word   blub
```

Oh wie cool! Man schreibt die Konstante einfach neben den Code in den Speicher und läd sie dann mit PC-relativer Adressierung in das Register.

Beim Hammer macht man das genau so, nur daß das Standard Memory Model zusätzlich vorsieht, daß Variablen nahe dem Code sind. Daher reicht das Standard 32-bit Offset aus.

```
        movq    blub(%rip), %rax
```

Hammer: PIC Code

Ein häufiges Problem für Compilerbauer und Assembler-Hacker ist PIC-Code. Das braucht man in dynamisch gelinkten Programmen und Libraries. Dabei läuft der Zugriff über die GOT, eine Tabelle mit Offsets. So sieht ein PIC-Zugriff aus:

```
0:   48 8b 05 00 00 00 00    mov    0(%rip),%rax
                     3: R_X86_64_GOTPCREL    blub+0xfffffffffffffffc
7:   48 8b 00                mov    (%rax),%rax
```

Auf x86 ruft PIC-Code erst mal eine Dummy-Routine auf, die ihre Return-Adresse vom Stack holt und zurückliefert. Dann wird das Offset zur GOT addiert und der Wert geladen. Und weil gcc das in ebx tut, muß das auch noch auf dem Stack gesichert werden.

Hammer: Syscalls

Wenn man schon dabei ist, kann man auch Syscalls optimieren. Hammer benutzt hierfür nicht mehr `int 0x80` wie x86, sondern eine explizite `syscall` Instruktion. Das ist viel schneller, übergibt aber keinen brauchbaren Stack.

Dafür gibt es beim Opteron einen Per-CPU-Ministack, einfach ein paar Register. Es gibt eine `swapgs` Instruktion, die `gs` mit dem Wert aus dem Mini-Stack tauscht. Die führt man halt einmal vorher und einmal nachher aus und ist fertig.

Auch hinterlegt `syscall` den PC aus dem User Space in diesem Bereich, d.h. es sind sehr schnelle Syscalls möglich. Linux hinterlegt dort auch Daten wie den aktuell laufenden Prozeß.

Hammer: gcc

```
int bar(int a,int b,int c) { return foo(a,b,c,0); }
```

```
bar_x86:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   16(%ebp)
        pushl   12(%ebp)
        pushl   8(%ebp)
        call    foo
        addl    $16, %esp
        leave
        ret
```

```
bar_hammer:
        xorl    %ecx, %ecx
        jmp     foo
```

Hammer: Virtuelle Syscalls

Einige Syscalls fragen lediglich read-only eine Variable aus dem Kernel ab, z.B. gettimeofday() oder time().

In Netzwerk-Servern ruft man diese Funktionen üblicherweise dauernd auf, um Timeouts zu implementieren.

Bei Linux auf Hammer sind diese daher als vsyscalls implementiert, d.h. sie führen nicht wirklich einen Syscall aus, sondern springen eine Kernel-Funktion direkt an, die der Kernel in alle User-Space Prozesse einblendet. Das spart den Overhead für Syscalls vollständig.

Und jetzt: IA64

Im Vergleich zu anderen CPUs kommt Itanium, von Fachleuten liebevoll Itanic getauft, aus einer Paralleldimension (die wo Spock einen Bart hat).

Andere Prozessoren haben viel Aufwand damit, Instruktionen so parallel auszuführen, daß das Ergebnis stimmt. Instruktionen müssen warten, bis in den Quellregistern das Ergebnis der Instruktion davor steht.

Compiler treiben viel Aufwand, der CPU die Arbeit zu erleichtern, indem Instruktionen so bei der CPU ankommen, daß möglichst viele unabhängig von denen davor ist. So muß die CPU nicht viel umsortieren. Der Compiler hat über das Programm ja auch mehr Überblick als die CPU.

IA64: EPIC

Bei IA64 ist jetzt die Innovation, daß die CPU einfach Instruktionen gar nicht mehr selber analysiert, um zu schauen, ob sie parallel ausgeführt werden können. Stattdessen schreibt der Compiler das an die einzelnen Instruktionen heran, welche parallel ausgeführt werden können und welche nicht.

Die CPU kann sich dann den Aufwand sparen und mit dem Platz dann z.B. eine Integer-Einheit mehr unterbringen, oder mehr Cache oder so.

Der Vorteil gegenüber VLIW ist, daß die Programme nicht neu optimiert werden müssen für die Nachfolger-CPU. Der Compiler macht die Analyse nur einmal, was parallel ausgeführt werden kann, und holt so viel Parallelität aus dem Programm heraus, wie irgend möglich. Wenn die CPU zwei Integer-Einheiten hat, und fünf Befehle parallel ausgeführt werden können, nimmt die CPU halt nur zwei davon. Der Nachfolger kann dann vielleicht schon drei.

IA64: Bundles

Befehle werden als Bundles abgelegt. Bundles sind immer gleich lang (16 Bytes) und bestehen aus drei Instruktionen.

Es gibt verschiedene Kombinationen, die als Bundle definiert sind.

Unbenutzte Teile eines Bundles muß man mit NOPs füllen.

IA64: EPIC

Wenn man schon dabei ist, kann man auch gleich die Sprungvorhersage wegoptimieren. Bei IA64 hat man daher mit den Predicates einen der PA-RISC Cancellation ähnlichen Mechanismus übernommen: Bei IA64 gibt es Prädikate bei den Instruktionen.

Es gibt 64 Prädikate, Und das Prädikat 0 ist immer wahr. Alle Instruktionen hängen von einem Prädikat ab, und werden nur ausgeführt, wenn das Prädikat wahr war.

IA64: EPIC

IA64 hat so viele Register, daß man bei Sprüngen einfach *beide* Varianten ausführt.

Wenn klar ist, wie der Sprung hätte genommen werden sollen, schmeißt man die Ergebnisse der anderen Variante weg. Damit das klappen kann, hat der IA64 echt viele Register (128 Integer plus 128 Fließkomma). Wenn man das konsequent durchführt, hat man auch plötzlich echt viele parallelisierbare Instruktionen (die beiden Alternativen sind natürlich unabhängig).

Als Assembler-Programmierer ist die Haupt-Umstellung, daß man jetzt explizit nach (fast) jeder Instruktion sagen muß, daß die nächste unabhängig ist.

IA64: EPIC

```
C:
    return a==5?8:-3;

IA64:
    cmp4.eq p7,p6=5,r32 ;;
    (p06) mov r8=-3
    (p07) mov r8=8
    br.ret.sptk.many b0 ;;
```

Essentiell zwei Befehle, das sieht gut aus. Der Nachteil: die Kodierung der Befehle ist immer in dreier-Bündeln, d.h. der cmp4-Befehl ist mit zwei NOPs kodiert. Für diese Funktion belegt IA64 daher 32 Bytes. Hammer belegt nur 17 Bytes.

IA64: EPIC

Noch krasser sieht man das bei dem folgenden, eigentlich perfekt parallelisierbaren Beispiel, wo IA64 eigentlich besonders deutlich führen sollte:

```
return a=='\r' || a=='\n' || a==' ' || a=='\t';
```

Auf IA64 sind das 128 Bytes, auf dem Hammer 30.

Man kann also gut sehen, daß die Itanic ihre große Speicherbandbreite auch echt braucht, weil sie die meiste Zeit damit beschäftigt ist, NOPs oder Befehle zu lesen, die später verworfen werden. Und gcc generiert hier *guten* Code, der alle Bundles maximal füllt!

IA64: EPIC

Macht das ganze überhaupt Sinn?

EPIC ist gut dafür geeignet, die funktionalen Einheiten mit Instruktionen zu versorgen. Leider werden später die meisten davon verworfen. gcc ist nicht unbedingt der tollste Compiler für IA64, aber auch mit Intels Compiler sieht es nicht viel besser aus. Statistiken bei SPECint haben ergeben, daß bei manchen Teilen 80% der Befehle verworfen werden!

Im Endeffekt ist IA64 also eine hocheffiziente Heizung. Vom Look und Feel her wird eine 900 MHz Itanic 2 Maschine von meinem Notebook um mehr als den Faktor 2 ausperformt (gemessen an der make-Zeit für die diet libc).

IA64: die Zukunft

Wenn der Pentium 4 einen Pipeline-Stall hat, dann werden die gerade nicht benutzten Einheiten wenigstens abgeschaltet, um Strom zu sparen. Dadurch sind die Kühlprobleme auf Servern überhaupt in den Griff zu bekommen.

Der IA64 heizt einfach nur die ganze Zeit vor sich hin und führt dabei die Anwendungen nicht effizienter aus. In einzelnen Fällen kann man beim IA64 mit Handoptimierung einiges reißen, und man kann der Architektur ihr Hack Value nicht absprechen.

Intel hat für IA64 Hyperthreading angekündigt. Das ist im Grunde eine Bankrotterklärung.