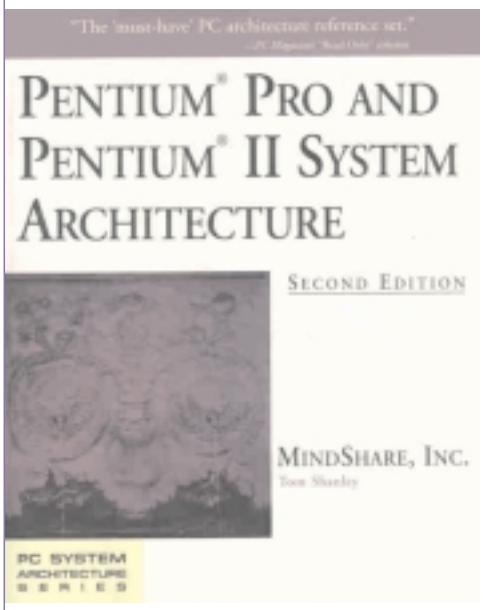
Code-Optimierung für Pentium Pro und Pentium II

Norbert Bartos

Moderne Compiler optimieren den Assemblercode für den speziellen Zielprozessor zum Teil sehr effizient. Aufwendigere Optimierungen können üblicherweise durch einen entsprechenden Switch ein- bzw. ausgeschaltet werden, Basisoptimierungen werden aber automatisch durchgeführt. Wie der Autor im Rahmen der heurigen Abendschulmatura feststellen musste, verhindert eine Standardoptimierung sogar manchmal das Funktionieren der in C geschriebenen Lösung einer Prüfungsaufgabe. Das Problem konnte nur durch den Einstieg in den Assembler und durch die Beantwortung der Frage "Was hat der Compiler aus meinem C-Programm gemacht ?" gelöst werden. Man ist daher gut beraten, über die Optimierungstechniken Bescheid zu wissen. Das anschließende Kapitel ist aus folgendem Buch übernommen:

Pentium Pro und Pentium II System Architecture, 2nd Ed., Tom Shanley, MindShare Inc., Addison-Wesley, 1998, 588 Seiten, ATS 696,-, ISBN 0-201-30973-4



Das Buch kann als Standardwerk für Hardware-Designer und Low-Level-Software-Entwickler angesehen werden. Der Kapitelaufbau ist ausgesprochen übersichtlich. Es wird jeweils zu Beginn auf die wichtigsten Aspekte des vorhergehenden Abschnittes hingewiesen, danach folgt die aktuelle Kapitelübersicht, danach ein kurzer Einblick in das Folgekapitel und letztlich der detaillierte Inhalt des aktuellen Abschnittes. So bleibt für die Leserschaft immer die Übersicht gewahrt.

Der folgende Abschnitt gibt einige Hinweise auf von Intel empfohlene Codeoptimierungen, um eine hohe Systemperformanz zu erzielen.

a) Reduktion der Anzahl der Branches

Durch das verwendete Verfahren der *Branch-Prediction*, welches wegen der *Instruction-Pipeline* notwendig ist, kann es bei Fehlvorhersagen zu Leistungseinbuβen im System kommen, da dann Pipeline-Stalls (Bubbles) entstehen. Eliminieren Sie daher, wo immer möglich, die Branches, beispielsweise durch Verwendung der Befehle CMOV (*Conditional Move*) und FCMOV (*Conditional Floatingpoint Move*).

b) Berücksichtigung des *Branch-Prediction-Algorithmus* beim Programmentwurf

Durch die Berücksichtigung des Vorhersagealgorithmus und die entsprechende Anpassung der Programmstruktur an diesen, kann eine Performanzsteigerung erzielt werden.

c) Erkennen und Vermeiden von unvorhersehbaren Branches

Indirekte Branches, wie bei Switch-Statements, Computed-GOTOs oder Calls durch Pointer, sind nicht vorhersagbar. Ersetzen Sie daher die obigen Strukturen, wo immer möglich, durch eine Reihe von wohldurchdachten *Conditional-Branches*, für die der *Prediction-Algorithmus* wieder anwendbar ist.

d) Keine Vermischung von Code und Daten

Vermischen Sie nicht Code und Daten in der selben Cache-Line, da diese dann klarerweise im Instruction- und im Data-Cache residiert. Wenn in weiterer Folge die Daten modifiziert werden, nimmt der Prozessor natürlich an, dass ein selbstmodifizierender Code vorliegt, was somit, wegen des administrativen Overheads in diesem Fall, wieder zu einem Performanz-Verlust führt.

e) Ausrichtung der Daten

Richten Sie Datenobjekte (2 oder 4 Byte) immer auf die entsprechenden Grenzen aus, ansonsten sind zusätzliche Speicheroperationen notwendig. Noch ärger ist es, wenn ein Datensatz auf einer Page-Grenze liegt, sodass beim Page-Fault gleich zwei Pages eingelesen werden müssen.

f) Vermeidung von direkten Code- und Datenabhängigkeiten

Stellen Sie sicher, dass die Distanz zwischen zwei abhängigen Befehlen möglichst groβ ist, um ein Umreihen der Befehle im Rahmen der superskalaren Ausführung durch den Prozessor zu ermöglichen.

g) *Context-Switch* in Software

Lassen Sie den automatischen Kontextwechsel durch den Prozessor nur dort ausführen, wo er unbedingt notwendig ist. In diesem Fall werden alle Register in das Task-State-Segment des unterbrochenen Tasks kopiert und alle Register vom Task-State-Segment des aufzurufenden Tasks restauriert. Das ist aufwendig und in vielen Fällen auch gar nicht notwendig. Häufig sind nur einige wenige Register freizumachen, bzw. zu restaurieren, was schneller durch eine Implementation in Software gemacht werden kann.

h) Elimination von Partial-Stalls in der Pipeline

Wenn ein volles Register (EAX, EBX, ECX oder EDX) gelesen wird, nachdem ein Teil dieses Registers beschrieben wurde, kann es zu einem Stall von über 7 Taktzyklen kommen. Es wird zuerst der Abschluss der Schreiboperation abgewartet, bis das Lesen beginnt. Das ist besonders beim Abarbeiten von 16-Bit-Applikationen der Fall, für die der Prozessor nicht optimal geeignet ist.

Wenn die Wahrheit zu Schwach ist,
sich zu verteidigen,
muss sie zum Angriff übergehen.

Der Mensch, das sonderbare Wesen: mit den Füβen im Schlamm, mit dem Kopf in den Sternen.

Wer einen Fehler gemacht hat und ihn nicht korrigiert, begeht den zweiten.

Man muss auf den Nebenmann achten, nicht auf den Vordermann.

Grundsätze sind ein Korsett, das mit der Zeit immer enger wird.

Keiner kann sich rühmen,
dass er etwas ohne Menschen
zu machen imstande ist.

Wer zu handeln versäumt, ist noch keineswegs frei von Schuld. Niemand erhält seine Reinheit durch Teilnahmslosigkeit.