Moderne Compiler-Optimierungen

Alte und neue Tricks für den kleinsten und schnellsten Code

André Schmitz, Green Hills Software

Compiler Optimierung ist ein alter Hut und bekannt solange es Compiler gibt. Trotzdem kommen jedes Jahr neue Versionen von Compilern auf den Markt die nochmal z.B. 5% kleineren oder 10% schnelleren Code generieren. Wie kann das sein? Sind die bisherigen Compiler etwa schlecht oder verwendet der Compiler Hersteller Tricks die nicht Standard konform sind? Compiler Optimierung ist von Natur her eine komplexe Angelegenheit und der Erfolg der Optimierung hängt sehr stark von dem zu optimierenden Source Code ab. Dieser Beitrag beleuchtet grundsätzliche Konzepte von optimierenden Compilern für C und C++, zeigt Beispiele sowohl von altbekannten als auch von sehr aktuellen Optimierungen auf und hinterfragt den Sinn dieser Optimierungen. Außerdem werden Fallstricke aufgezeigt beim Versuch Code manuell zu optimieren.

Warum Optimierung

Gerade in Embedded Software Projekten besteht das Ziel möglichst schnellen oder kleinen Code zu generieren. Je schneller Code ausgeführt wird, desto schneller kann man auf Events regieren, was die Reaktionszeit und die Verwendbarkeit verbessert. Je schneller eine Aufgabe erledigt ist, desto eher kann man in den Strom-Spar-Modus wechseln, was zur Energieeinsparung und bei batteriegetriebenen Geräten zur Verlängerung der Batterielaufzeit genutzt werden kann. Kleinerer Code erlaubt es mehr Funktionalität in dem gleichen Speicher eines Gerätes unter zu bringen.

Manchmal versuchen Softwareentwickler bereits optimierten Source-Code zu schreiben. Das kann von Vorteil sein, in manchen Fällen kann es aber auch die Qualität des Programms verschlechtern, weil der Optimierer des Compilers diesen manuell optimierten Code dann nicht mehr so gut optimieren kann, was unterm Strich schlechtere Optimierung bedeuten kann (siehe unten).

Grundsätzlich kann man bei Compiler Optimierungen unterscheiden zwischen der Target abhängigen Optimierung, bei der man auf Instruktionsebene arbeitet, und der Target unabhängigen Optimierung, bei der auf Source-Code Ebene optimiert wird. Dabei gibt es wiederum Optimierungen, die sowohl die Größe als auch die Geschwindigkeit des Programms verbessern, also das Programm sowohl kleiner als auch schneller machen. Es gibt aber auch gegensätzliche operierende Maßnahmen, d.h. einige Optimierungen verkleinern den Code auf Kosten der Geschwindigkeit oder umgekehrt. Schauen wir uns nachfolgend mal einige Beispiele an.

Beispiele von Compiler Optimierungen

Typische und altbekannte Source Level Optimierungen sind zum Beispiel “Loop Unrolling” oder “Function Inlining”. Beide haben das Ziel auf Kosten der Codegröße die Geschwindigkeit des Programms zu erhöhen. Eine weitere gängige Optimierung ist die "Common Subexpression Elimination" (CSE), bei der mehrere redundante Berechnungen, die in jedem Programmpfad auftreten, durch eine Berechnung ersetzt wird (siehe Beispiel in Bild 1).

Original code:

```
int a = val1();
int b = val2();
foo(a * b);
if (a > 0)
   c = a * b;
else
   d = a * b;
```

Optimized (saves 1 multiply)

```
int a = val1();
int b = val2();
temp = a * b;
foo(temp);
if (a > 0)
   c = temp;
else
   d = temp;
```

Bild 1: Common Subexpression Elimination

Ein neuerer Algorithmus, der aggressiver ist als CSE ist "Partial Redundancy Elimination" (PRE). Bei diesem reicht es, wenn die Berechnung nur in einem Pfad redundant ist, und er kommt auch mit Schleifen zurecht. CSE und PRE helfen auch bei vielen redundanten Speicher Berechnungen. So kann zum Beispiel

```
arr[i] + arr[i+3]
```

durch

```
ptr = arr + i; ptr[0] + ptr[3];
```

ersetzt werden, was die ein oder andere Instruktion spart. Weitere altbekannte Optimierungen wären zum Beispiel "Dead Code Elimination" (DCE), wo der Compiler versucht Code Komponenten zu löschen, die für das Ergebnis einer Funktion keine Relevanz haben, oder auch "Constant Propagation".

Original code:

```
if (cond == true) {
   max = 3.1;
   set_params(x);
} else {
   max = 4.0;
   set_params(x);
}
```

Optimized code:

```
if (cond == true) {
   max = 3.1;

} else {
   max = 4.0;

}
set_params(x);
```

Bild 2: Busy Code Motion

Bei der "Busy Code Motion" (BCM) wird die Größe des Programms reduziert indem der Compiler versucht ähnliche Instruktionen in mehreren Blöcken in nur einem Block zu reduzieren. (siehe Bild 2). Diese Idee wird in der Linker Optimierung, die im nächsten Kapitel beschrieben wird, noch weitergetrieben. Bei der Target abhängigen "Instruction Scheduling" Optimierung wird das Wissen über das Scheduling von Instruktionen in der CPU Pipeline verwendet. Während eine Instruktion in der Pipeline ausgeführt wird, beginnt schon die Ausführung der nächsten Instruktion. Und wenn die nächste Instruktion auf das Ergebnis der vorherigen warten muss, kommt es zu Pipeline „Stalls“. Der Compiler sortiert im

Rahmen der Möglichkeiten die Instruktionen so, dass es so selten wie möglich zu Pipeline Stalls kommt.

Ein weiterer Bereich der Optimierung befasst sich mit der Vektorisierung, wobei man in der Regel zwischen manueller und automatischer Vektorisierung unterschiedet. Gerade in diesem Bereich passiert in den letzten Jahren sehr viel, nicht zuletzt, weil viele moderne CPUs neuerdings entsprechende SIMD Instruktionen (Single Instruction Multiple Data) mitbringen die eine Vektorisierung erlauben, wie z.B. Power Architecture AltiVec, ARM NEON oder Intel SSE. Manuelle Vektorisierung kann zu sehr guten Verbesserungen der Performance führen, erfordert aber, dass der Entwickler hier selbst entscheidet, was man wie parallelisieren kann und was dafür zu tun ist. Er muss neue Datentypen und Compiler Intrinsics lernen und anwenden und sich Gedanken über Alignment und Aliasing machen.

Automatische Vektorisierung hat das gleiche Ziel und verspricht auf den ersten Blick den Entwickler zu entlasten, ist aber meist nicht wo einfach zu verwenden wie man sich das als Entwickler wünscht. Ähnlich wie bei automatischer Parallelisierung von Programmen im Allgemeinen muss eine saubere Alias Analyse durchgeführt werden, und das ist sehr schwierig in den Sprachen C und C++. Daher sollte der Entwickler auch bei der automatischen Vektorisierung dem Compiler mithilfe von Pragmas oder anderen Schlüsselworten klar machen wo es sicher ist zu Vektorisieren.

Weitere Bereiche für Optimierung betreffen gerade in jüngster Zeit zum Beispiel den Bereich der Register Allokierung, oder im Fall von C++ das Löschen nicht verwendeter virtueller Funktionen oder der Umgang mit C++ Exceptions. Außerdem werden mehr und mehr spezielle CPU Instruktionen direkt vom Compiler oder mithilfe von Intrinsics unterstützt. Intermodulare Optimierung hat ein sehr großes Potential, erfordert aber, dass der Compiler zweimal über jede Quelldatei geht. Beim ersten Mal wird zunächst die Struktur des Modules erkannt und gemerkt, und im zweiten Durchlauf wird jedes Modul unter Zuhilfenahme der Strukturinformationen von allen anderen Modulen tatsächlich übersetzt und optimiert.

In manchen Projekten ist die Geschwindigkeit das wichtigste und die Code Größe egal, in anderen Fällen ist die Code Größe das wichtigste, ganz egal wie langsam dadurch der Code wird. Aber meistens will man irgendwie einen Mittelweg, und man wählt dann eine Optimierungseinstellung die ein Kompromiss aus Geschwindigkeit und Größe darstellt. Dieser Mittelweg ist aber nicht unbedingt der beste Weg, denn meist verbringt das Programm recht viel CPU Zeit in nur sehr wenig Code, und ein großer Teil des Codes wird nur wenig ausgeführt. Wie optimiert man so etwas dann am besten? Die optimale Lösung dafür ist die Profiler basierte Optimierung, bei der die tatsächliche Optimierung parametrisiert wird durch die Ergebnisse des Profilings zur Laufzeit.

Profiling erkennt welche Teile des Programms wie oft aufgerufen werden und wieviel Zeit man dort verbringt. Damit erkennt man die Bereiche, in denen sehr viel Zeit verbracht wird und bei denen Geschwindigkeitsoptimierung daher sehr viel bringt, und solche Bereiche in denen man nur wenig Rechenzeit verbringt und bei denen es daher sinnvoll sein kann auf Größe zu optimieren, da hier der eventuelle Trade-Off bzgl. Geschwindigkeit vernachlässigbar ist. In Bild 3 sieht man zum Beispiel, dass die meiste CPU Zeit in nur zwei Funktionen verbracht wird

(minSpanningTreePrims und minSpanningTreeKruskals). Diese beiden Funktionen kann man also extrem auf Geschwindigkeit optimieren, auch wenn diese dadurch deutlich größer werden. Alle anderen könnten vermutlich eher mit einer Optimierung auf Größe leben. Funktionen, die sehr häufig aufgerufen werden, auch wenn Sie nur wenig Zeit kosten, wören vermutlich gute Kandidaten für Inlining.

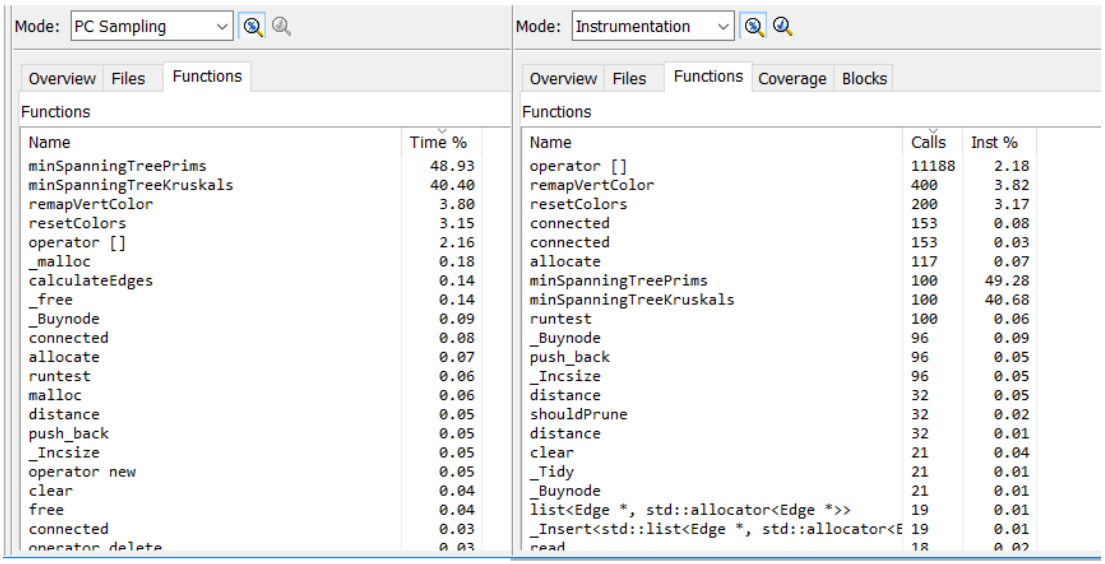

Bild 3: Profiling Ergebnisse (CPU-Zeit und Anzahl der Aufrufe)

Letztlich kann der Entwickler mithilfe des Profiling Ergebnisses die Hot-Spots seines Programmes erkennen und die Optimierungs-Optionen in der Build Umgebung entsprechend anpassen.

Linker Optimierung

Die wenigstens Entwickler wissen, dass auch der Linker Optimieren kann. Warum macht das Sinn? Der Linker muss alle Module anfassen und sieht als letzter im Build Prozess welche Funktionen und Daten tatsächlich verwendet werden. Er kann also am Ende alle nicht verwendeten Funktionen und Daten einfach beim Linken weglassen. Natürlich ist etwas Vorsicht geboten bei der Verwendung von zur Laufzeit festgelegten Funktionszeigern, aber da kann man dem Linker entsprechend Hinweise geben, was er auf keinen Fall entfernen darf. Außerdem kann der Linker im gesamten Programm sehen, ob ggf. bestimmte Instruktionssequenzen an verschiedenen Stellen vorkommen, die man ggf. durch eine Subroutine ersetzen kann. Diesen Ansatz kann man auch als „Outlining“ (Gegenteil von Inlining) bezeichnen.

Typische Fallstricke

Nun gibt es natürlich auch Dinge, die man vermeiden sollte, damit der Compiler möglichst gut Optimieren kann. Manche Entwickler denken sich, was der Compiler will kann ich besser, oder wollen einfach nur bestimmte Aspekte eines Programms effizient in Assembler implementieren. Grundsätzlich kann handgeschriebener Assembler Code natürlich deutlich besser sein als das was ein Compiler aus einer Hochsprache wie C oder C++ an Instruktionen generiert. Kritisch wird es aber dann, wenn der Entwickler Inline Assembler verwendet, also mitten im C-Code einzelne Instruktionen von Hand einfügt. Diese Inline Assembler Instruktionen gelten nämlich

automatisch als Optimierungs-Barriere für den Compiler. Die ganzen oben erwähnten Tricks wie CSE, BCM, DCE, etc. dürfen über die Grenze der eingefügten Assemblerinstruktion hinweg nicht angewendet werden. Der Versuch einer manuellen Optimierung kann also ganz böse nach hinten losgehen.

Weitere Probleme können auch durch die falsche Annahme darüber entstehen, wie ein Compiler Daten im Speicher ablegt und wie Datenzugriffe optimiert werden dürfen. Compiler und Linker dürfen Variablen im Speicher beliebig umsortieren, solange sie nicht als Elemente einer Struktur oder Attribute eines Klasse deklariert sind. Lokale Variablen kann der Compiler auf den Stack oder in Register legen, und Zugriffe auf Variablen könnten im besten Fall auch komplett wegoptimiert werden (siehe DCE weiter oben). Will man das Entfernen der Zugriffe verhindern, so bietet sich die Verwendung des „volatile“ Type Qualifiers an, denn dieser sagt dem Compiler, dass die zugrundeliegende Variable auch außerhalb der Kontroller des compilierten Codes geändert werden kann.

Zusammenfassung
Wie man sieht gibt es sehr viele verschiedene Optimierungen, die sehr unterschiedliche Ziele haben und auf sehr unterschiedlichen Source Code oder Instruktionssequenzen operieren. Von den Optimierungen sind also abhängig vom zugrundeliegenden Code und von der zugrunde liegenden Hardwarearchitektur sehr unterschiedliche Ergebnisse zu erwarten. Damit sollte auch einleuchten, dass eine Verbesserung einer Optimierung immer s

Referenzen
[1] http://compileroptimizations.com/index.html

Autor

Andre Schmitz erhielt sein Diplom in Physik 1997 an der Universität Bonn. Anschließend entwickelt er bei der FhG Steuerungs- und Simulations-Software für Autonome Roboter. Von 2000 bis 2005 entwickelte Herr Schmitz Embedded Software für UMTS Kommunikationssysteme. Seit 2005 ist Herr Schmitz bei Green Hills Software für die technische Unterstützung von Kunden und die Durchführung von Schulungen zuständig. Herr Schmitz ist seitdem regelmäßig Referent bei diversen Fachkonferenzen.

Kontakt
Internet: www.ghs.com
Email: aschmitz@ghs.com