Dynamischer Dispatch in C++

Markus Raab
uni@markus-raab.org

16. Dezember 2008

Beim Einsatz objektorientierter Techniken wird oftmals Dispatch mit *Virtual Function Tables* eingesetzt, um zur Laufzeit die richtige virtuelle Methode zu finden. In diesem Paper will ich der Frage nachgehen, wieviel dieser Dispatch auf superskalaren Computerarchitekturen kostet, wo Dispatch in C++ mit welchen Aufwand auftritt und in wie weit der Dispatch in C++ mittels verschiedener Techniken vermeidbar ist. Zudem will ich beleuchten, welche Compileroptimierungen die Situation wesentlich verbessert haben oder gute Ergebnisse versprechen.

1 Einführung

Dynamischer Dispatch bezeichnet eine Technik, die dafür zuständig ist, zur Laufzeit die richtige Methode von ein oder mehreren Klassen zu bestimmen. Diese Funktionalität wird in manchen objektorientierten Programmen so intensiv verwendet, dass ein beträchtlicher Anteil der Zeit im Dispatch Code verbracht wird.

C++ ist eine sehr interessante Sprache, um Dispatch genau zu studieren. Das Modell, welches der Sprache für Speicher und Ausführung zugrunde liegt, ist ein sehr realistisches Modell für die meisten Computer[16]. Die Sprache hat den besonderen Vorteil, dass man Dispatch für einzelne Funktionen ausschalten kann, es ist aber durch ein gemeinsames *Interface* auch sehr einfach für alle Klassen und Methoden Dispatch zu verwenden. Da *virtual* explizit notiert werden muss, hat man einen Anhaltspunkt wo der Programmierer es für nötig hielt dynamischen Dispatch einzusetzen.

1.1 Kosten

Direkte Kosten von dynamischem Dispatch beschreibt genau jene Zeit, die aufgewendet werden muss, um die Empfängerklasse zur Laufzeit zu bestimmen. In manchen Sprachen bedeutet das sehr hohen Aufwand, da erst zur Laufzeit aufgrund des Namens

der Methode und des statischen Objektes die Methode gesucht wird[19][3]. Diesen Fall wollen wir hier nicht berücksichtigen, sondern es geht um die wesentlich effizientere Situation beim Einsatz von VFT (Siehe Kapitel 2).

Um direkte Kosten zu ermitteln, kann man die Gesamtlaufzeit minus der Laufzeit eines idealen Programmes, welches keine Zeit im Dispatch Code verbringt, verwenden.

Die *indirekten Kosten* stammen von jenen nicht durchgeführten Optimierungen die ausbleiben müssen, weil die Funktion zur Übersetzerzeit noch nicht bekannt war und deshalb Techniken wie z.B. *inlinen* nicht durchgeführt werden können[11]. Diese Kosten werden in der weiteren Betrachtung ignoriert.

1.2 Interfaces

Interfaces in C++ sind Klassen welche nur aus *pure virtual* Methoden bestehen.

Stroustrup ist sehr starker Befürworter, *Interfaces* in vielen Kontexten zu verwenden[16], weil sie einfach zu handhaben, effizient und stark getypt sind. Sie ermöglichen es verschiedene Implementationen bereit zu stellen und damit den Benutzer komplett von der Implementation zu abstrahieren.

Ergänzend ist zu sagen, dass es nicht unbedingt sinnvoll ist, ein Interface zu bauen wenn es nur einen Implementor gibt. Ich würde eher eine Herangehensweise wie bei *Templates*[12] empfehlen. Dabei implementiert man zuerst den speziellen Fall. Sieht man dass ein weiterer Fall mit gleichem Interface auftritt, konvertiert man ersteren zu ein Interface welches zwei Mal implementiert wird. Voraussetzung dafür ist aber, dass das Objekt immer per Zeiger oder Referenz übergeben wurde, ein Vorgehen welches unabhängig davon empfohlen wird, da es effizienter ist und `const` das Sprachmittel in C++ ist um auszudrücken dass ein Parameter nicht modifiziert wird.

Ein Teil muss dann allerdings im Programm geändert werden und zwar die Allokation. Hierbei ist aber gleich zu überdenken, ob man durch das FACTORY PATTERN[8] nicht gleich zusätzliche Flexibilität gewinnen will.

1.3 Generische Programmierung

Wir wollen nun untersuchen ob durch generische Programmierung auch dynamischer Dispatch eingespart werden kann. Es stimmt natürlich, dass in C++ kein dynamischer Laufzeitoverhead für generische Programmierung auftritt. In einigen Situationen kann man auch Datenstrukturen entweder mit Generizität oder Subtyping implementieren, z.b. `list`. Allerdings kann man den Effekt das sich jedes Element der Liste unterschiedlich, aber doch kompatibel zum Obertyp, verhält nur mit Subtyping erreichen.

Will man eine homogene Liste, so hat man mit Generizität genau das richtige Konzept gewählt und erspart sich sämtliches Dispatching. Abzuraten ist dabei allerdings von einer künstlichen Klasse `Object`, welche ein Obertyp aller Klassen ist. Hier wird *Casting* dann unvermeidlich und man verschiebt genannte Probleme auf die Laufzeit.

2 Implementierung

Dieses Kapitel wollen wir den Implementierungstechniken für Dynamischer Dispatch widmen. Eine Standardmöglichkeit ist die *Virtual Function Table (VFT)*. Dabei bekommt jede Klasse eine Tabelle, welche Zeiger auf ihre Methoden hat, und jedes Objekt einen Zeiger auf die Tabelle. VFT wurden anfänglich von Simula[4] benutzt und ist heutzutage - mit Optimierungen - die bevorzugte C++ Dispatch Methode[7].

Mit Einfachvererbung reicht ein Array von Methodenzeigern aus, wobei der Compiler den *Selektor* für jeden Methodennamen kennt. Dieser Selektor wird dann auch bei allen abgeleiteten Klassen verwendet, zusätzlich ergänzt mit weiteren virtuellen Methoden in dieser Klasse. Der Dispatch Prozess selber besteht aus folgenden Schritten:

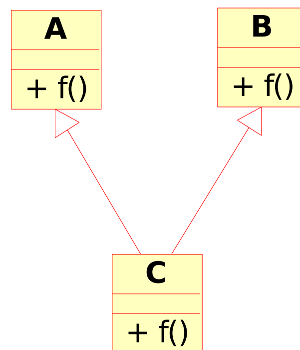
1. Laden der VFT der Klasse
2. Laden der Funktionsadresse durch den Selektor
3. Aufrufen der Methode

2.1 Mehrfachvererbung

```
class A
{
public:
    virtual void f();
};

class B
{
public:
    virtual void f();
};

class C: public A, public B
{
public:
    virtual void f();
};
```

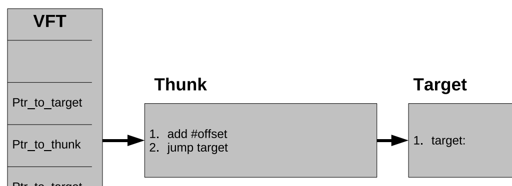


Bei Mehrfachvererbung[17] ist es notwendig, mehrere virtuelle Tabellen pro Klasse zu generieren, um Mehrdeutigkeiten aufzulösen. Definieren die zwei Klassen `A`, `B` virtuelle Methoden, so können nicht von beiden ihre Selektoren in eine neue VFT einer Klasse `C` übernommen werden. Deshalb werden zwei VFT für `Ca` und `Cb` erzeugt.

Besondere Vorsicht muss dabei dem `this` pointer zukommen. Methoden der Klasse `B` verlangen einen `this` pointer auf ein `B` Objekt. Deshalb muss der `this` pointer, wenn die Methode über `C` aufgerufen wird, modifiziert, über `B` aufgerufen, aber gleich gelassen werden.

Eine naheliegende Herangehensweise ist es, in der VFT auch ein `delta` für jede Methode einzuführen welches die notwendige Informationen zur Modifikation enthält. Der Vorteil dieser Methode ist, dass dafür ohne Probleme C-Code generiert werden kann. Die Nachteile sind allerdings, dass die VFT beträchtlich, im Fall `sizeof(int) = sizeof(funcptr)` sogar verdoppelt wird. Die Objektgröße wird auch auf `int`

limitiert. Das schwerwiegenste Problem ist aber, dass zwei zusätzliche Operationen bei jedem Aufruf von virtuellen Methoden hinzukommen, in denen aber fast immer 0 hinzuaddiert wird.



Diese Probleme werden mit Hilfe der *Thunks* vermieden. Statt `delta` in der `VFT` einzutragen, diesen Wert zu laden und dann hinzuaddieren, wird wo nötig direkt eine parameterlose Funktion aufgerufen. Diese Funktion, der *Thunk*, modifiziert den `this` pointer mit an dieser Stelle bekanntem Offset und ruft die richtige Methode auf. Dadurch gibt es bei der Einfachvererbung keinen zusätzlichen Aufwand.

3 Superskalare Prozessoren

Um zu verstehen, warum die zusätzliche Anzahl der besprochenen Instruktionen pro Aufruf nicht den Kosten entsprechen, müssen wir uns mit superskalaren Prozessoren beschäftigen. Instruktionen werden vom Cache geholt und in einem Instruktionsbuffer geladen. Bei jedem Taktzyklus werden nun mehrere Instruktionen zu den richtigen *functional units* verteilt und parallel ausgeführt. Die *issue width* bestimmt dabei die maximale Anzahl der parallel ausgeführten Instruktionen.

3.1 Abhängigkeiten

Die *issue width* ist aber für einen Prozessor nur das theoretische Maximum von gleichzeitig ausgeführten Instruktionen. Diese wird zusätzlich durch die Anzahl der *functional units* und durch Abhängigkeiten in den Daten limitiert. Es kann nämlich nur eine Instruktion die unabhängig, oder in der alle Daten für die Parameter vorhanden sind, ausgeführt werden.

Leider gibt es noch eine zusätzliche Abhängigkeit, nämlich die des Kontrollflusses. So kann noch nicht mit Gewissheit bestimmt werden ob eine Instruktion nach einem bedingten Sprung ausgeführt wird, ohne die Bedingung zu überprüfen. Im schlimmsten Fall muss der *Instruktionenbuffer* komplett neu geladen werden nachdem das Ziel feststeht.

3.2 Obere Grenze

Wir halten fest: Sind N Instruktionen unabhängig, so können diese in nur einem Zyklus ausgeführt werden. Ist jede hingegen von der vorigen abhängig, so werden N Zyklen benötigt. Dabei ist aber noch nicht berücksichtigt, dass Instruktionen auf bestimmter Hardware mehrere Zyklen benötigen können. Deshalb ist die Zahl der In-

struktionen ein schlechter Maßstab für Ausführungszeit auf superskalaren Prozessoren.

Nach dem bis jetzt gesagten können wir annehmen die Kosten betragen $2L + B$. Dabei sind L die Kosten der Ladezyklen, jeweils zum Laden des *VFT* und der Funktionsadresse, und B die Zeit um nach dem Sprung den Instruktionenbuffer neu zu füllen. Diese Kosten sind allerdings nur eine obere Grenze, der nächste Absatz erklärt warum:

Da Verzweigungen in Programmen sehr häufig vorkommen - jede fünfte oder sechste Anweisung ist ein Branch[9] - sind Hardwarehersteller ersucht, die Kosten dafür möglichst niedrig zu halten. Die Basistechnik ist dabei die Sprungvorhersage. Hierbei wird ausgenützt, dass die meisten Sprünge bei wiederholter Ausführung an die gleiche Stelle gehen. Um das zu erreichen lädt man das vorige Ergebnis des bedingten Sprunges von einem sog. Sprungbuffer, bevor überhaupt klar ist ob dort hin gesprungen wird. Von dieser vermeintlichen Zieladresse werden dann sofort Instruktionen geladen.

Die Technik kann sogar noch weiter ausgebaut werden, dass diese geladenen Instruktionen sogar ausgeführt werden. Allerdings muss hardwaremäßig sichergestellt sein, dass alles rückgängig gemacht wird, falls die Vorhersage falsch war. Daraus folgt dass unter Umständen keine Kosten B auftreten wenn der Sprung richtig vorhergesagt wurde, aber auch sehr hohe Kosten möglich sind wenn viel rückgängig gemacht werden muss. Dadurch ergibt sich erstaunlicherweise, dass die Kosten der Dispatch Sequenzen - obwohl immer exakt die gleichen Instruktionen ausgeführt werden - abhängig von dem umliegenden Code sind[5].

4 Compiler

Neben den bis jetzt genannten Möglichkeiten zur Optimierung, kann ein Compiler durch Analyse des Programmes virtuellen Dispatch wieder entfernen, wenn er nicht notwendig ist. Wir wollen der Frage nachgehen in wie weit das durchführbar ist und wie aufwändig es ist. Dabei können sowohl die Programmgeschwindigkeit als auch die Größe der ausführbaren Datei verbessert werden.

In [2] wird gezeigt, wie ein optimierender Compiler mit dem am besten empfundenen Algorithmus Rapid Type Analysis bis zu 71% der virtuellen Funktionsaufrufe und 25% der Programmgröße im Durchschnitt einspart. Der große Nachteil ist allerdings, dass ein kompletter Aufrufgraph des gesamten Programmes und aller *Libraries* bestimmt werden muss was genau entgegen des Paradigmas kleinerer Übersetzungseinheiten läuft. Der zusätzliche Ausführungsaufwand des Compilers beträgt allerdings nur ca. 6%.

Eine sehr einfache Technik, die sich auf der Ebene des Linkers abspielt kann sich positiv auf die Codegröße auswirken. In ihr wird einfach verglichen ob ein Symbolname in allen Objektdateien nur einmal vorkommt. Dann kann man den virtuellen Aufruf durch einen Direkten ersetzen. Schade ist hier allerdings dass dann auf weitere Optimierungen wie z.B. *Inlining* verzichtet werden muss. Als Alternative kann

sehr einfach auf große statische Kompilate zugunsten *shared libraries* verzichtet werden, wodurch auch weitere Vorteile [15][14] lukriert werden können.

Es ist auch möglich mittels Source-to-Source Compiler viele Virtual Function Calls komplett zu eliminieren[1]. Hier sind auch bis zu 40% (18% im Durchschnitt) Verbesserungen der Performance möglich. Dabei war das Setup mit *all-virtual* und anschließender Optimierung schneller als das ursprüngliche Programm.

5 Diskussion

Wir haben in einem kurzen Rundgang die wesentlichen Aspekte von C++, VFT und Computerarchitekturen im Bezug auf dynamischen Dispatch betrachtet. In einigen der getesteten Programme gibt es durchaus noch viel Optimierungspotenzial, wo durch Verzicht von `virtual` oder durch verbesserte Optimierungen im Compiler noch viel ausgeschöpft werden kann.

Hauptsächlich gestützt habe ich mich auf das Paper [5] welches viele, noch immer gültige, Einsichten auf das Thema Dispatch und Computerarchitekturen enthält. Es hat sich aber ein Fehler eingeschlichen: auf Seite 7 wird behauptet dass alle Funktionen bis auf Operatoren und Destruktoren virtuell deklariert wurden. Bei den Operatoren wird nur das Meßergebnis ein wenig verfälscht. Lässt man hingegen die `virtual` Deklaration beim Destruktor weg, so wird zur Laufzeit nicht der Richtige ausgeführt[18][10]. Dadurch wurden wahrscheinlich nicht alle Ressourcen freigegeben, wodurch die Ausführungszeit wieder verkürzt wird. Die Benchmarks von *all-virtual* sind somit als untere Schranken zu sehen.

Ich denke nicht dass die Kosten des Dispatches, auch theoretisch nicht, auf 0 sinken können. Das ist deshalb nicht möglich, weil es mehrere Empfänger geben kann, und damit auch Programmlogik und Entscheidungen darin codiert werden können.

Aufgrund der Art und Weise wie *virtual function calls* gesucht wurden[5], ist anzunehmen, dass die meisten Erkenntnisse auch für andere Sprachen, z.B. Java[13] gelten.

Es gibt durchaus auch andere Ansätze die komplett ohne VFT auskommen, z.B. bei SmallEiffel[20]. Die Abhängigkeitsstrukturen sind aber bei den meisten Ansätzen gleich und das Laufzeitverhalten deshalb auf superskalaren Computerarchitekturen[6] ähnlich.

Literatur

[1] AIGNER, G. und U. HOELZLE: *Eliminating Virtual Function Calls in C++ Programs*. Lecture Notes in Computer Science, Seiten 142–166, 1996.

[2] BACON, D.F. und P.F. SWEENEY: *Fast static analysis of C++ virtual function calls*. In: *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 324–341. ACM New York, NY, USA, 1996.

[3] CHAMBERS, C., D. UNGAR und E. LEE: *An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes*. Higher-Order and Symbolic Computation, 4(3):243–281, 1991.

[4] DAHL, O.J. und B. MYHRHAUG: *SIMULA 67 Implementation Guide*, 1969.

[5] DRIESEN, K. und U. HÖLZLE: *The direct cost of virtual function calls in C++*. ACM SIGPLAN Notices, 31(10):306–323, 1996.

[6] DRIESEN, K., U. HOLZLE und J. VITEK: *Message Dispatch on Modern Computer Architectures*. ECOOP '95 Proceedings, August 1995.

[7] ELLIS, M.A. und B. STROUSTRUP: *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1990.

[8] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Design*. Professional Computing Series. Addison-Wesley, 1995.

[9] HENNESSY, JOHN L. und DAVID A. PATTERSON: *Computer architecture (2nd ed.): a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[10] HEROLD, HELMUT, M. KLAR und S. KLAR: *C++, UML und Design Patterns*. Addison-Wesley Verlag, 2005.

[11] HÖLZLE, URS und DAVID UNGAR: *Optimizing dynamically-dispatched calls with run-time type feedback*. In: *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, Seiten 326–336, New York, NY, USA, 1994. ACM.

[12] JOSUTTIS, N. und D. VANDERVOORDE: *C++ Templates, The Complete guide*. Addison-Wesley, 2003.

[13] LEE, J., B.S. YANG, S. KIM, K. EBCIOĞLU, E. ALTMAN, S. LEE, Y.C. CHUNG, H. LEE, J.H. LEE und S.M. MOON: *Reducing virtual call overheads in a Java VM just-in-time compiler*. ACM SIGARCH Computer Architecture News, 28(1):21–33, 2000.

[14] LEVINE, J.R.: *Linkers and Loaders*. Morgan Kaufmann, Oktober 1999.

[15] ORR, D.B., J. BONN, J. LEPREAU und R. MECKLENBURG: *Fast and Flexible Shared Libraries*. In: *Proc. of the Summer 1993 USENIX Conf*, Seiten 237–251, 1993.

[16] STROUSTRUP, B.: *An overview of C++*. In: *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, Seiten 7–18. ACM New York, NY, USA, 1986.

[17] STROUSTRUP, BJARNE: *Multiple inheritance for C++*. In: *Proceedings of the Spring 1987 European Unix Users Group Conference*, 1987.

[18] STROUSTRUP, BJARNE: *Die C++ Programmiersprache*. Addison-Wesley, 4 Auflage, 2000.

[19] VITEK, J. und R.N. HORSPOOL: *Compact Dispatch Tables for Dynamically Typed Object Oriented Languages*. Lecture Notes in Computer Science, Seiten 309–325, 1996.

[20] ZENDRA, O., D. COLNET und S. COLLIN: *Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler*. ACM SIGPLAN Notices, 32(10):125–141, 1997.