

# Vergleich des Oberon x86-Compilers mit dem GNU gcc

Projekt fuer die Vorlesung 37-235:  
Computer Systems Performance Analysis & Benchmarking

Corsin Decurtins

Nicky Kern

## **Zusammenfassung:**

Bei einem groben Vergleich der Laufzeit von Oberon-Programmen, die mit dem Oberon-Compiler fuer Native ix86-Oberon bzw. dem gcc uebersetzt wurden, stellte sich heraus, dass die Leistungen teilweise beträchtlich differieren. In einem Grossteil der Fälle schnitt der Oberon-Compiler schlechter ab, als der gcc.

In diesem Projekt soll detaillierter festgestellt werden, wo die Stärken und Schwächen der beiden Compiler liegen. Zudem soll festgestellt werden, mit welchen Optimierungen die Leistung des Oberon-Compilers am leichtesten verbessert werden kann.

# 1. Einführung

## ***Problemstellung***

In einem "Spielzeugvergleich" wurden Oberon-Compiler und gcc miteinander verglichen. Dafür wurden die Oberon-Module mit einem Oberon-to-C-Compiler nach C übersetzt und dann mit dem gcc in x86-Maschinencode übersetzt. Bei diesem Vergleich stellte sich heraus, dass die Leistung der Programme, die mit dem Oberon-Compiler kompiliert worden waren, teilweise deutlich schlechter war, als der Programme, die (via Oberon-to-C-Compiler) mit dem gcc kompiliert wurden.

Mit diesem Projekt soll nun herausgefunden werden, wo genau Stärken und Schwächen der Oberon-Compilers liegen. Weiterhin ist von Interesse, welchen Einfluss, welche Optimierungen des gcc auf die Leistung des übersetzten Codes haben. Das schlussendliche Ziel ist, herauszufinden, welche Optimierung sich für eine Implementierung im Oberon-Compiler anbietet, um mit möglichst geringem Aufwand eine möglichst grosse Verbesserung der Leistung zu erreichen.

## ***Vorgehen***

Es liegen einige Standard-Benchmarking-Kernel vor, welche bereits im "Grobtest" zur Anwendung kamen. Da sie die Menge der üblichen Programmierkonstrukte recht gut abdecken, wird hier wieder darauf zurückgegriffen.

Diese Programme werden nun mit dem Oberon-to-C-Compiler oo2c [oo2c] nach C übersetzt. Das Vorgehen der Evaluation selber ist ein dreistufiges: In einem ersten Schritt soll versucht werden, herauszufinden, inwiefern Compiler und Kernel interagieren. Da die Interaktion recht stark ist, werden im nächsten Schritt alle Kernel separat betrachtet. In diesem wird nun ein erster Vergleich der Leistung des Oberon-Compiler mit der des gcc gezogen. Im letzten Schritt werden die einzelnen Optimierungen des gcc selber detaillierter untersucht.

## ***Verwendete Kernel***

Die verwendeten Kernel sind die folgenden. Bei jedem Kernel wurde kurz erwähnt, welche Programmkonstrukte den grössten Einfluss haben.

1. Dhystone  
Integer-Mix
2. Permutation  
– Stark rekursiv.  
– Array-Zugriffe.
3. Towers of Hanoi  
– Funktionsaufrufe  
– Array-Zugriffe
4. 8-Queens-Problem  
– Funktionsaufrufe  
– Array-Zugriffe

5. Matrix-Multiplikation
  - Floating-Point-Arithmetik
  - Array-Zugriffe
6. Integer Matrix-Multiplikation
  - Integer-Arithmetik.
  - Array-Zugriffe.
7. Tree-Sort
  - Funktionsaufrufe
  - Speicherallokation
8. Bubble-Sort
  - Array-Zugriffe
  - Schleifenoptimierungen
9. Quick-Sort
  - Funktionsaufrufe
  - Array-Zugriffe
10. Puzzle
  - Schleifen
  - Matrizen (Arrays)
11. Fast Fourier Transformation
  - Floating Point Code

## 2. gcc-Optimierungen im Detail

Bevor näher auf die Compiler und ihre Unterschiede und Gemeinsamkeiten eingegangen wird, sollen hier die einzelnen Optionen des gcc aufgelistet und kurz erläutert werden.

### ***Optimierender/Nichtoptimierender Compilerlauf***

Der GCC unterscheidet zwischen einem optimierenden Lauf und einem nicht-optimierenden. In letzterem werden gar keine Optimierungen durchgeführt. In einem optimierenden Durchlauf werden einige Optimierungen standardmässig ausgeführt, während andere über separate Flags steuerbar sind.

Leider werden mehrere Transformationen im optimierenden Durchlauf durchgeführt, die nicht separat beeinflussbar sind. Es handelt sich dabei um:

#### 1. Local Common Subexpression Elimination

Diese Optimierung versucht identische Teilausdrücke, die mehrfach berechnet werden, zu finden und zu eliminieren. Die Implementation des gcc arbeitet nicht auf "normalen" Basisblöcken, d.h. auf solchen, die durch Sprungziele und Sprunginstruktionen begrenzt werden, sondern auf etwas erweiterten, die über konditionale Sprünge hinaus arbeitet, und nur durch Sprungziele begrenzt wird.

Da diese Variante der CSE verhältnismässig einfach und schnell ist, wird sie im gesamten

Compilerlauf auch mehrfach angewendet (nach verschiedenen anderen Optimierungen).

## 2. Local Constant Propagation

In diesem Schritt werden Konstanten in die ausgeführten Berechnungen hineinpropagiert. Das Ziel ist, einen möglichst grossen Teil der Berechnungen schon während der compile-Zeit durchführen zu können, so dass zur Laufzeit des eigentlichen Programms möglichst wenig berechnet werden muss.

## 3. Jump Optimizations

Es wird versucht, Sprünge zu optimieren:

*Dead Code Elimination* entfernt Blöcke, die nicht mehr aufgerufen werden können.

*Jump Threading* entfernt Sprünge auf Sprünge.

Zusätzlich wird noch versucht, bedingte Sprünge auf bedingte Sprünge zu entfernen. Das funktioniert allerdings nur, wenn die Bedingungen gleich, oder invers sind.

Diese Optimierung wird normalerweise vor der CSE durchgeführt. Werden in der CSE Sprünge verändert, so wird sie nach der CSE noch einmal gemacht.

## **Globale CSE**

Die globale CSE ist das blockübergreifende Analogon zur lokalen CSE. Konkret implementiert der gcc eine Partial Redundancy Elimination nach Morel–Renvoise (s. [Muchnick]) durch.

## **Function Inlining**

Function Inlining versucht, mittels geeigneter Heuristiken, genügend "kleine" Funktionen zu finden, und sie direkt an die Stelle des Aufrufs einzufügen, statt einen normalen Funktionsaufruf in den Code einzufügen.

## **Instruction Scheduling**

In diesem Schritt wird versucht, Instruktionen so umzuordnen, dass auf Ressourcen in der CPU (Integer–Pipeline, FP–Unit, Speicherzugriffe, etc.) so wenig, wie möglich gewartet werden muss.

## **Loop Optimizations**

Schleifen sind interessante Objekte für eine Optimierung, da sie sehr häufig ausgeführt werden. So bietet der gcc auch eine ganze Reihe von Schleifenoptimierungen:

### 1. Invariant Code Motion

Konstante oder sich in verschiedenen Schleifendurchläufen nicht verändernde Berechnungen werden aus der Schleife entfernt.

### 2. Strength Reduction

Mittels algebraischem Transformationen wird versucht, die notwendigen Berechnungen in der eigentlichen Schleife zu minimieren. Zum Beispiel wird die Schleife

```
for(i=0; i<10; i++)  
    sum += i*10+5;
```

wird zu:

```
for(t=5; t<100; t+=10)
    sum += t;
```

### 3. Loop Unrolling

Durch "Aufrollen" einer Schleife, kann der Mehraufwand für Schleifenkopf, sowie die Behandlung von Abbruchbedingungen pro Schleifendurchlauf reduziert werden. Auch ist es durch Loop Unrolling möglich, dass das Instruction Scheduling besser funktioniert, weil mehr "Masse" zum bearbeiten vorhanden ist.

## ***Vorgegebene Optimierungsstufen O0–O3***

Der gcc bietet drei vorgegebene Optimierungsstufen:

1. O0:

Diese Stufe entspricht einem nicht-optimierenden Compilerdurchlauf

2. O1:

In dieser Stufe wird ein optimierender Compilerlauf durchgeführt, ohne allerdings weitere Optimierungen hinzuschalten. D.h. es wird im wesentlichen eine lokale CSE sowie einige Sprungoptimierungen zusätzlich ausgeführt.

3. O2:

In dieser Stufe sind alle Optimierungen bis auf Function Inlining aktiviert.

4. O3:

Es werden dieselben Optimierungen wie in O2 und zusätzlich noch Function Inlining ausgeführt.

## **3. Ein erster Test**

Um einen Ueberblick über Kernel und Compiler zu bekommen, sollte in einem ersten Test herausgefunden werden, in welchem Mass Compiler und Kernel miteinander interagieren. Dafür wurden Kernel und Compiler einander gegenübergestellt.

Dafür wurden alle Kernel mit den verschiedenen Compilern übersetzt. Neben dem Oberon-Compiler wurde der gcc mit den Optimierungsstufen 0 bis 3 (je höher, desto stärker wird optimiert) verwendet. Diese Optimierungsstufen sind vom gcc vorgegeben und wurden für diesen ersten Test nicht weiter differenziert.

	Oberon	gcc -O0	gcc -O1	gcc -O2	gcc -O3	Mittel	Effekt
perm	79	107	96	99	71	90,4	-101,58
towers	73	136	78	69	59	83	-108,98
queens	68	89	66	66	66	71	-120,98
intmm	58	86	34	38	37	50,6	-141,38
mm	67	126	44	51	47	67	-124,98
puzzle	350	738	393	393	393	453,4	261,42
quick	81	148	73	71	70	88,6	-103,38
bubble	112	239	88	78	79	119,2	-72,78
tree	94	139	85	83	96	99,4	-92,58
fft	156	400	109	72	72	161,8	-30,18
dhry	899	1237	723	741	537	827,4	635,42
Mittel	185,18	313,18	162,64	160,09	138,82	191,98	
Effekt	-6,8	121,2	-29,35	-31,89	-53,16		
Variation explained by Compiler:		6,40%					
Variation explained by Kernel:		85,31%					
Unexplained variation:		8,29%					

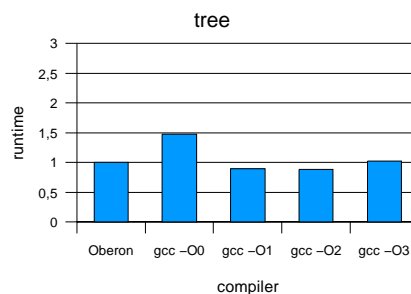
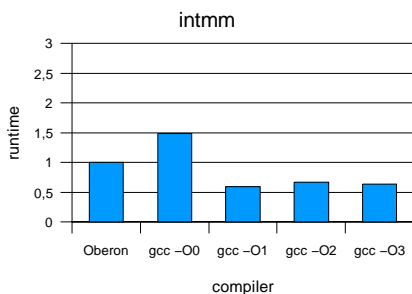
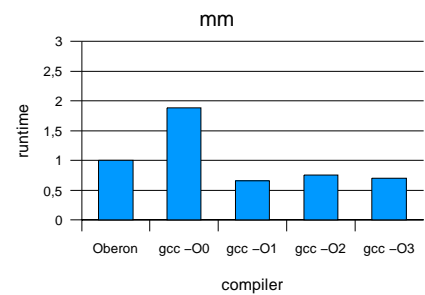
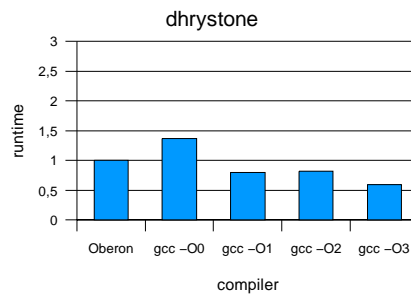
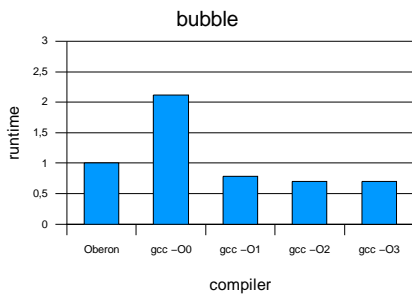
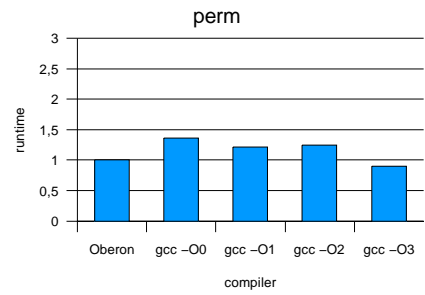
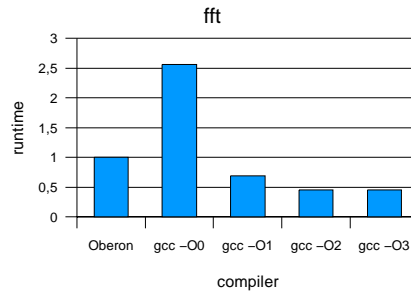
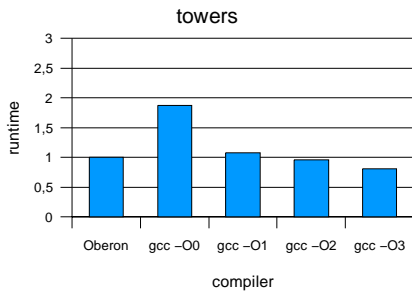
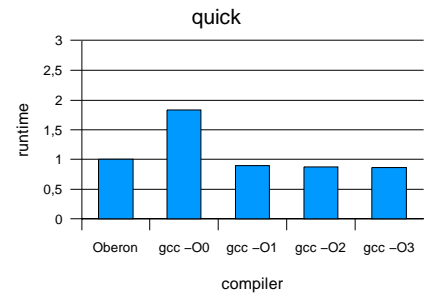
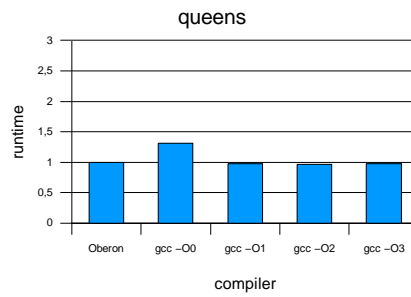
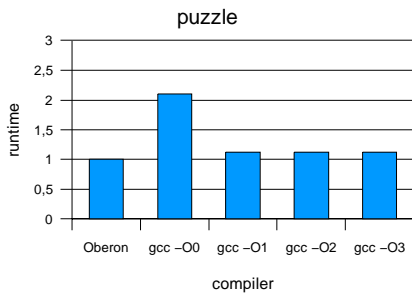
Erwartungsgemäss ist der durch die Kernel erzeugte Anteil an der Varianz der Messwerte mit 85.5% am grössten. Der Compiler ist nur für etwa 6% der Varianz der Messwerte verantwortlich. Die restlichen 8.5% der Varianz müssen somit mit den Interaktionen zwischen Kernel und Compiler zusammenhängen.

Um nun die Wirkung der einzelnen Optimierungsstufen beurteilen zu können, müssen die Kernel offenbar getrennt weiterbetrachtet werden.

## 4. Die gcc Optimierungsstufen

Als nächstes wollen wir uns einen Ueberblick über die Optimierungen und ihren Einfluss auf die Gesamtperformance verschaffen. Die Optimierungen sind in 4 grobe Klassen eingeteilt. Diese Klassen entsprechen genau den 4 Optimierung-Stufen (O0, O1, O2 und O3) des gcc. Die Performance-Wert sind auf den Oberon-Compiler normiert.

Als erstes fällt einem auf, dass der Oberon-Compiler anscheinend gar nicht so schlecht ist. Diese Schlussfolgerung ist allerdings ein bisschen gefährlich. Der Performance-Wert hängt selbstverständlich nicht nur vom Compiler, sondern auch vom Betriebssystem etc. ab.



Als erstes ist wohl zu bemerken, dass der Oberon-Compiler in *jedem* Kernel schneller ist, als der nicht-optimierende gcc. Da die optimierten gcc-Läufe ein deutlich differenzierteres Bild ergeben ist, sei diese Tatsache für den Augenblick zur Seite gestellt.

Eine auffällige Differenz ergibt sich bei *Fliesspunkt-intensivem Code*. Sowohl in der *fft* als auch bei FP-Matrix-Multiplikation schneidet der gcc deutlich besser ab.

Hingegen scheinen *Funktionsaufrufe* im Oberon-Code deutlich effizienter zu sein: im *perm*-Kernel schneidet der gcc auch bei eingeschalteten Optimierungen schlechter ab, als der Oberon-Compiler. Erst durch das Hinzuschalten von Function Inlining kann der gcc mit dem Oberon-Compiler mithalten.

Eine Schwäche des Oberon-Compiler scheinen Array- bzw. Matrix-Zugriffe zu sein. In allen Kernels, die Array-Zugriffe testen (mm, intmm, quick und bubble) schneidet der Compiler schlechter ab, als der gcc. Bemerkenswert ist hier, dass der gcc sehr stark von den Basis-Optimierungen (lokale CSE und Sprungoptimierungen) zu profitieren scheint.

Interessant ist, dass Paar Quicksort vs. Bubble-Sort zu betrachten. Während beide viele Array-Zugriffe durchführen (und daher der gcc schneller sein sollte, als der Oberon-Compiler), ist das Bild beim rekursiven Quicksort viel ausgeglichener als beim iterativen Bubblesort.

Prinzipiell lässt sich sagen, dass der gcc-Code mit eingeschalteten Basis-Optimierungen sehr häufig schneller ist als der des Oberon-Compiler, es sei denn der Kernel enthält einen grossen Anteil an Prozeduraufrufen. Der weitere Leistungsgewinn (-O2 und -O3) ist häufig sehr klein oder gar negativ!

Die logische Vorgehensweise wäre jetzt eigentlich, sich die Basis-Optimierungen genauer anzuschauen. Leider ist das aber nicht möglich. Der aktuelle gcc erlaubt es nicht, diese Optimierungen einzeln zu aktivieren und zu deaktivieren. Wir können also keine genaue Aussage darüber machen, welche Optimierungen in diesem Paket den grössten Effekt erzielt.

Anstattdessen werden wir im nächsten Schritt die separat einschaltbaren Optimierungen genauer betrachten, um herauszufinden, ob es darunter eine gibt, die einen überragenden Einfluss hat.

## 5. Optimierungen im Detail

Ziel dieses Experiments ist, herauszufinden, welche der "weitergehenden" Optimierungen des gcc, welchen Einfluss auf die Leistung des Codes haben. Dafür wurden, ausgehend vom optimierenden Compiler-Lauf, die einzelnen Optionen variiert.

### ***Aufteilung der gcc-Optimierungen im Experiment***

Im Experiment wurde der aktuell verfügbare egcs 1.1.2 verwendet. Für die Berechnung der Beiträge der einzelnen Optimierungen, wurden sie in "Paketen" zusammengefasst. Leitfaden bei der Zusammenstellung war, möglichst ähnliche Optimierungen zusammenzufügen, bzw. mehrere gcc-Optionen, die verschiedene Varianten derselben Optimierung steuern, zusammenzunehmen. Mit diesem Ansatz wurde versucht, die Anzahl Parameter, die zu variieren sind, in einem vernünftigen Rahmen zu halten. Sollte sich herausstellen, dass ein Paket sehr starken Einfluss hat, oder stark mit einem anderen interagiert, so können sie in einem zweiten Schritt noch aufgeteilt werden.

Im folgenden sind die Pakete jeweils mit den genauen getesteten Compilerschaltern angegeben:

#### 1. Local Common Subexpression Elimination

Es ist zwar nicht möglich, die LCSE aus- oder einzuschalten, es gibt aber dennoch eine ganze Reihe von Optionen, die das Verhalten der LCSE steuern. Sie wurden alle in

diesem Paket zusammengefasst.

Die Schalter sind:

- ffunction-cse
- fforce-mem
- fforce-addr
- fcse-follow-jumps
- fcse-skip-blocks

## 2. Global Common Subexpression Elimination

Schaltet die GCSE ein oder aus. Schalter:

- fgcse

## 3. Function Inlining

Es gibt die Möglichkeit, Function Inlining nur für Funktionen einzuschalten, die im Quelltext als "inline" markiert sind. Da aber der Vergleich zum Oberon-Compiler gesucht ist, und es diese Möglichkeit in Oberon nicht gibt, wird nur zwischen gar kein FI und heuristischem FI (d.h. der Compiler versucht, herauszufinden, bei welcher Funktion es sich lohnt, sie direkt einzufügen) unterschieden.

Compilerschalter kein FI:

- fno-inline

Compilerschalter heuristisches Inlining:

- finline

## 4. Instruction Scheduling

Der gcc bietet die Möglichkeit, das Scheduling nach der Registervergabe ein zweites Mal durchzuführen, um zu verhindern, dass Spilling Code den Schedule zerstört. Für den ersten Schritt wurden beide Durchläufe aktiviert, mit der Option, sie später noch aufzutrennen.

Compilerschalter:

- fschedule-insns
- fschedule-insns2

## 5. Loop Optimizations

Alle Schleifenoptimierungen in einem Paket:

- fstrength-reduce
- funroll-loops
- fmove-all-movables
- freduce-all-givs
- frerun-cse-after-loop (durch -funroll-loops impliziert)
- frerun-loop-opt

## 6. Diverse

Eine ganze Reihe "kleinerer" Optimierungen, die sich im Wesentlichen mit Spezialitäten des Assemblercodes beschäftigen. Es handelt sich dabei um die folgenden:

-fregmove:

Optimiert das Verschieben von Registern.

-fomit-frame-pointer:

Optimiert den Stackframe von Prozeduren, indem kein Frame-Pointer angelegt wird.

-fthread-jumps:

Optimiert konditionale Sprünge auf konditionale Sprünge.

-fdefer-pop

Versucht, das Freigeben von Stackframes am Stück zu machen, statt jedes Element auf dem Stack einzeln freizugeben.

## **Konstante Optionen**

Der GCC bietet die Möglichkeit, strikte Annahmen über die Typen von Objekten und Adressreferenzen zu machen (-fstrict-aliasing). Da Oberon ein recht starkes Typing hat, kann man davon ausgehen, dass Modifikationen von Optimierungen, die aufgrund dieses Schalters aktiviert werden, auch im Oberon-Compiler Sinn machen. Es gibt daher keinen Grund, diese Option auszuschalten, da ein Oberon-Compiler nicht auf schwächere Typisierungsbedingungen Rücksicht nehmen muss.

## **Design**

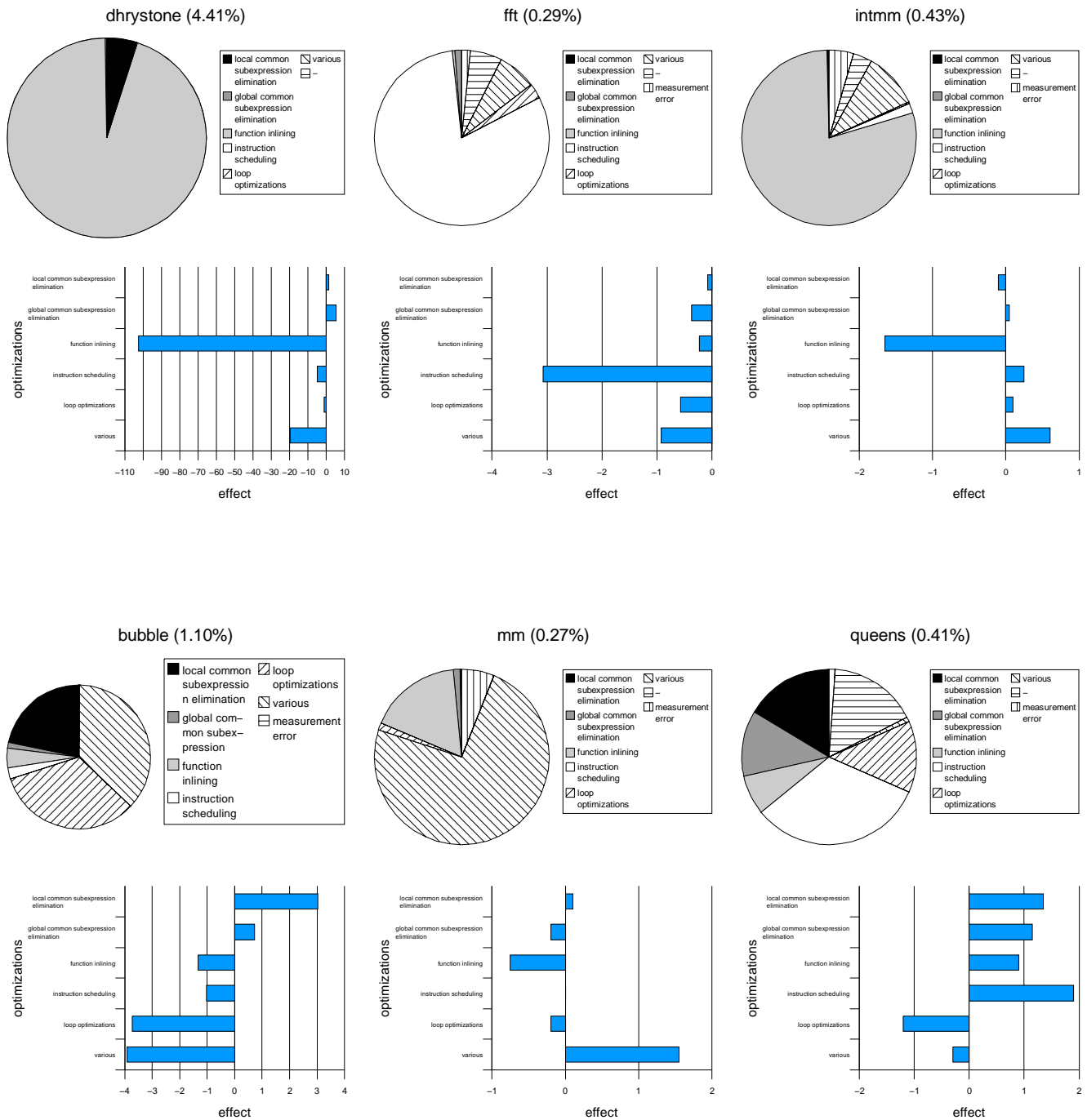
Um zuverlässige Daten zu erhalten, müssen wir die Messungen replizieren. Wir haben uns für 5 Wiederholungen entschieden. Sollten die Messungen grössere Schwankungen enthalten, hätten wir weitere Messungen machen können. Wie sich herausstellte, war das aber nicht nötig.

Eine mögliche Design-Variante wäre ein Full-Factorial-Design mit Replikation. In unserem Fall wäre die Laufzeit wahrscheinlich keine Hindernis gewesen. Wir haben uns trotzdem gegen diese Variante entschieden, da wir immerhin schon 6 Faktoren haben und die Analyse der Daten sehr komplex und unübersichtlich werden würde.

Stattdessen haben wir uns entschieden, ein  $2^{6-3} * 5$  Design zu verwenden.

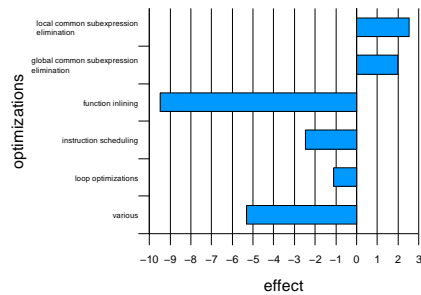
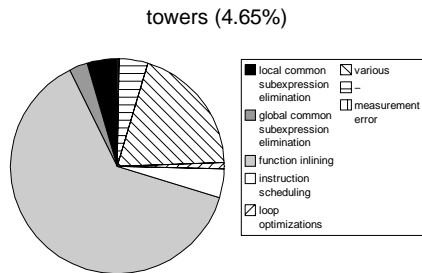
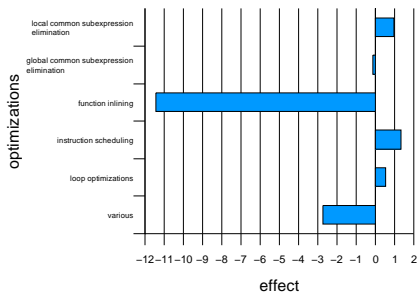
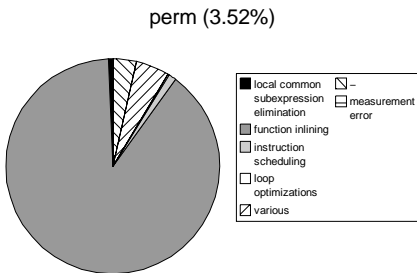
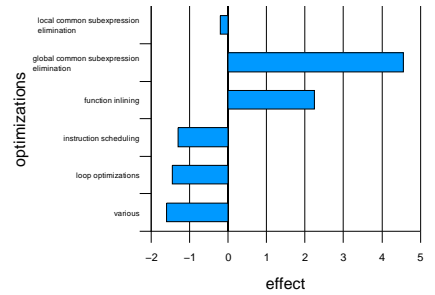
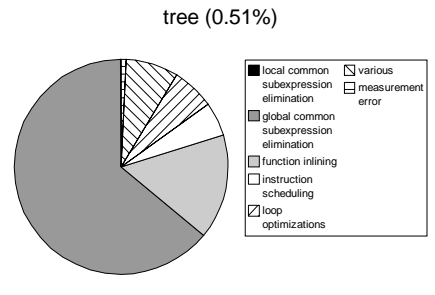
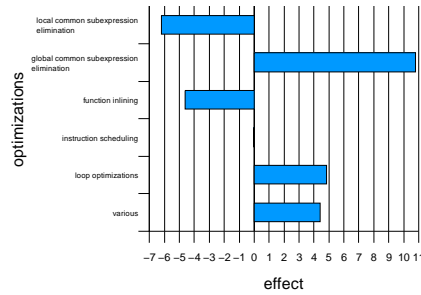
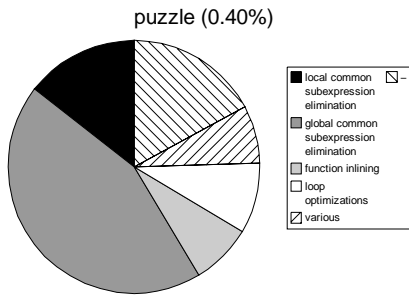
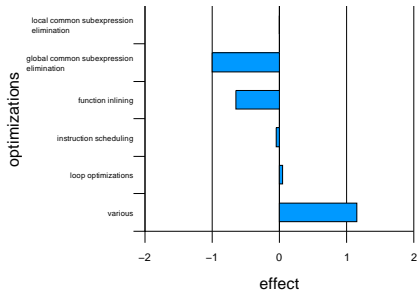
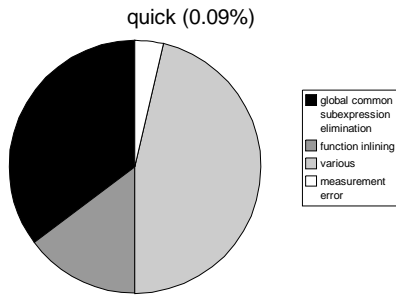
## **Resultate**

In den folgenden Diagrammen sind einmal die Effekte (Balkendiagramme) und der Einfluss auf die Varianz (Kuchendiagramme) dargestellt. Der Wert in Klammern auf den Kuchendiagrammen bezeichnet die maximale Varianz, die durch die Optimierungen zu erreichen war. Effekte, die kleiner als 0.1 % waren, wurden in der graphischen Darstellung nicht berücksichtigt.



Als erstes ist wohl zu bemerken, dass bei keinem der getesteten Kernel die Optimierungen mehr als etwa 4% an Varianz bewirken konnten. So gesehen sind die, durch die Optimierungen zu erreichenden, Leistungssteigerungen nur noch mässig interessant.

Es lässt sich sehr schlecht eine Optimierung ausmachen, die immer oder wenigstens häufig einen grossen Einfluss auf die Leistung hat. Während einmal die Variation der local CSE sich stark bemerkbar macht (bubble), hat sie bei einigen anderen Kernel kaum (mm, intmm) bzw. sogar negativen Einfluss (puzzle). Eine ähnliche Argumentation lässt sich leider für alle anderen Optimierungen ebenfalls führen.



Sinnvolle Schlussfolgerungen lassen sich also aus diesen Daten allerdings kaum ziehen. Interessant ist allerdings zu sehen, dass die Optimierungen im Verhältnis zur Performance nur sehr wenig bringen. Der Anteil liegt in den meisten Fällen unter 1%!

## Schlussfolgerungen

Trotz der Tatsache, dass die präsentierten Zahlen mit Vorsicht genossen werden müssen, weil Effekte durch oo2c und Betriebssystem nur schlecht quantifiziert werden können, lassen sich gewisse Trends feststellen.

Der Oberon-Compiler kann in einigen Bereichen problemlos mit dem gcc an Code-Qualität mithalten. Insbesondere Funktionsaufrufe sind seine Stärke.

Auf der anderen Seite sind sowohl Floating-Point intensiver Code, als auch Array-Zugriffe nicht sehr überzeugend. Hier kann der gcc mit der lokalen CSE und den Sprungoptimierungen sehr viel an Leistung aus dem Code herausholen.

Interessanterweise ist der Einfluss der übrigen Optimierungen verhältnismässig gering. So liegen ihre Effekte im Bereich von einigen, wenigen Prozent. Zudem ist es fast unmöglich, Optimierungen anzugeben, die in allen Fällen mehr Leistung bringen. Der Effekt der Optimierungen ist sehr stark vom Kernel abhängig. Eine genauere Untersuchung über die Zusammensetzung von "durchschnittlichem" Oberon-Code liegt aber ausserhalb des Bereichs dieses Berichts. Der einzige Kernel, den man als durchschnittlich bezeichnen könnte ist wohl der Dhrystone Benchmark. Hier zeigt sich dann aber auch, dass, bis auf Function Inlining, keine der zusätzlichen (d.h. über die Basisoptimierungen hinausgehenden) Optimierungen durchschlagenden Erfolg hat. Dies zeigt sich sowohl in den absoluten Zahlen des zweiten Tests, als auch in der genaueren Analyse des dritten Tests.

Abschliessend lässt sich wohl sagen, dass die einzige Optimierung, die ein vernünftiges "Preis/Leistungsverhältnis" bezüglich Leistungssteigerung und Implementationsaufwand die Common Subexpression Elimination (mit Constant/Copy Propagation) und den Sprungoptimierungen (evtl. zusammen mit einer Dead-Store/Code-Elimination) ist. Da die Implementation all dieser Optimierungen auf ähnlichen Datenflussalgorithmen basiert, sollte es verhältnismässig wenig Mehraufwand sein alle drei, statt nur einer davon zu implementieren.

## Referenzen

- [Muchnick] Steven S. Muchnik: Advanced Compiler Design & Implementation  
Morgan Kaufmann Publishers, 1997.
- [Dragon] Aho, Sethi, Ullman: Compilers, Principles, Techniques, and Tools  
Addison Wesley, 1988
- [oo2c] <http://www.uni-kl.de/OOC/>
- [Jain} Raj Jain: The Art of Computer Systems Performance Analysis  
John Wiley & Sons, 1991