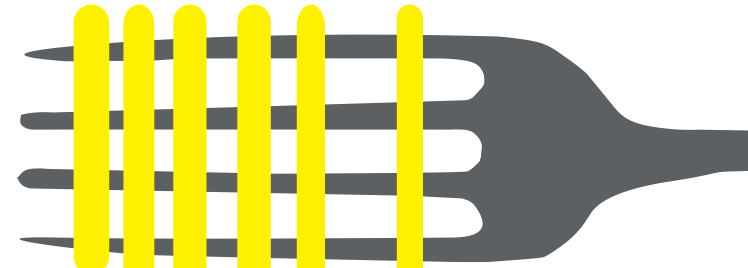
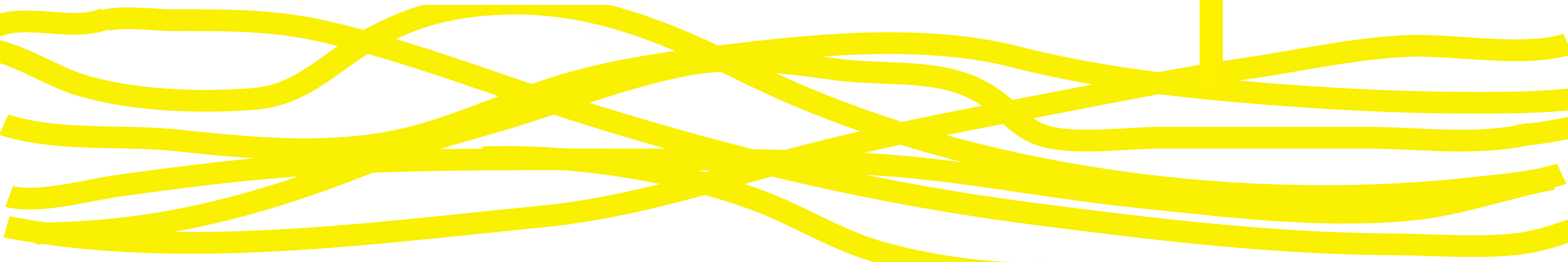


Code Refactoring

von Hauke Schulz



Überblick

- Definition
- Motivation
- Durchführung
- Qualitätstest
- Refactoring im Hochleistungsrechnen
- Zusammenfassung

Was ist Code Refactoring

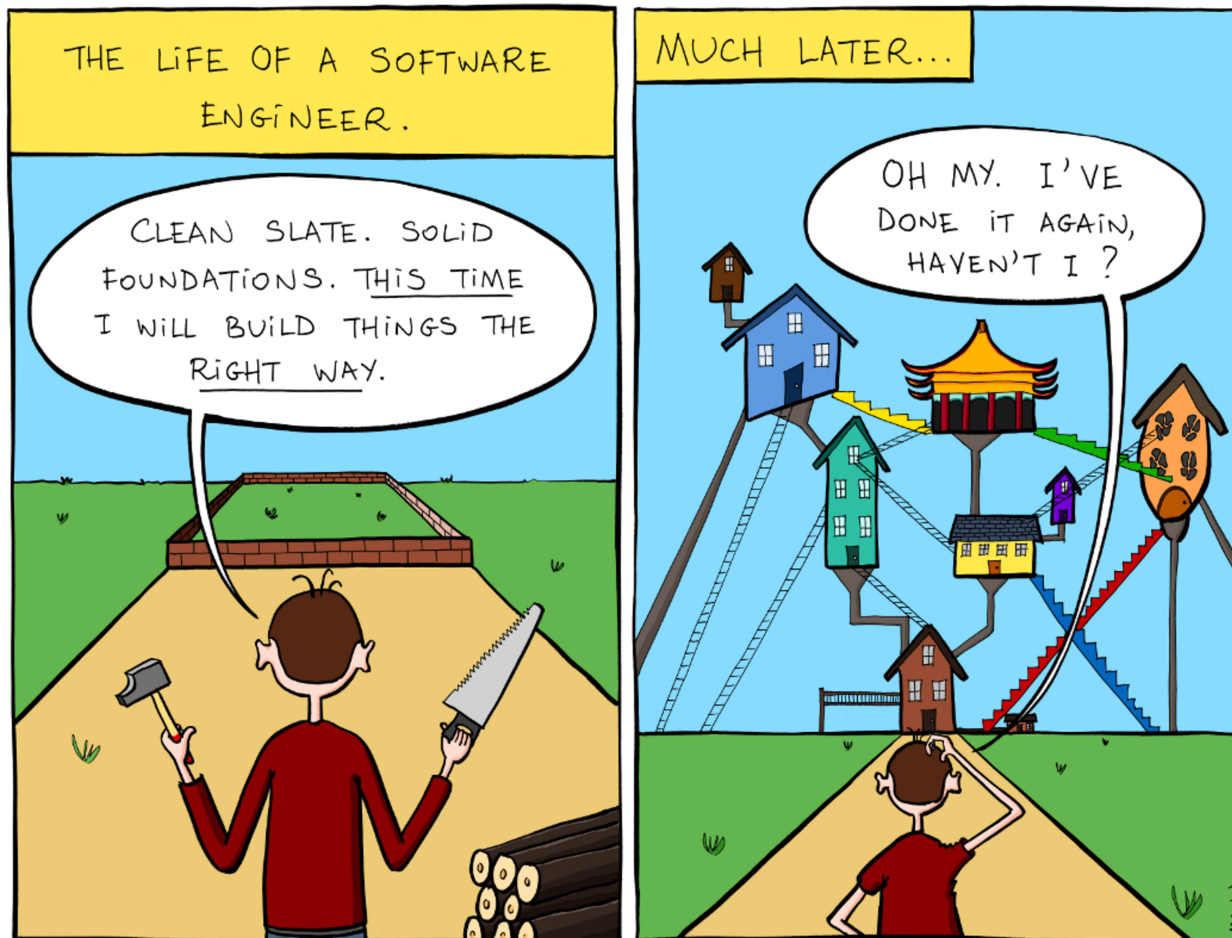
> *"Refactorings do not change the behavior of a program; that is, if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same. Refactorings are behavior preserving so that, when their preconditions are met, they do not "break" the program."*

William F. Opdyke (1992)

Warum ist Code Refactoring wichtig?

- Verständnis
  - Besseres Verständnis von Algorithmen
  - Nachvollziehbar von interessierten Wissenschaftlern
- Anpassbarer
  - Plattform unabhängig ausführbar
  - Neue wissenschaftliche Fragestellungen leichter zu implementieren
- Fehlerbekämpfung

Warum ist Code Refactoring wichtig?



Comic von Manu Cornet / Creative Common Lizenz

Wie sieht optimaler Code aus?

Original Quelltext	Refactored (Lesbarkeit)	Refactored (Leistung)

program main
 dimension c(4,4,4)
 n=4
 DO i=1,n
 DO j=1,n
 DO k=1,n
 c(i,j,k) = i+j+k
 c(i,j,k) = c(i,j,k)/3
 END DO
 END DO
 END DO
 print*,c
end program
	program main
	 implicit none
	 integer :: i,j,k,MAX=4
	 integer :: c(MAX,MAX,MAX)
	 DO i=1,MAX
	  DO j=1,MAX
	   DO k=1,MAX
	    call mean(c,i,j,k,MAX)
	   END DO
	  END DO
	 END DO
	 print*,c
	contains
	 subroutine mean(c,i,j,k,MAX)
	  implicit none
	  integer :: i,j,k,MAX
	  integer :: c(MAX,MAX,MAX)
	  c(i,j,k) = (i+j+k)/3
	 end subroutine mean
	end program main
		program main
		 implicit none
		 integer :: i,j,k,MAX=4,i1,j1
		 integer :: c(MAX,MAX,MAX)
		 DO i1=1,floor(real(MAX-1)/2)*2+1,2
		  DO j1=1,floor(real(MAX-1)/2)*2+1,2
		   DO i=max(1,i1),min(MAX,i1+1)
		    DO j=max(1,j1),min(MAX,j1+1)
		     DO k=1,n
		      call mean(c,I,j,k)
		     END DO
		    END DO
		   END DO
		  END DO
		 END DO
		 contains
		  subroutine mean(c,i,j,k)
		
		…

Wie sieht optimaler Code aus?

- SOLID – Prinzipien
  - Single responsibility
    - jede Klasse soll nur eine einzige Verantwortung haben
  - Open-closed
    - Software-Einheiten wie Module und Klassen, sollen offen für Erweiterungen sein, ihr Verhalten jedoch bestehen bleiben
  - Liskov substitution
    - ein Programm, das Objekte einer Basisklasse verwendet, muss auch mit Objekten der davon abgeleiteten Klasse korrekt funktionieren
  - Interface segregation
    - Interfaces sollen überschaubar sein, und nicht zu komplex. Komplexität sollte an Benutzer angepasst sein.
  - Dependency inversion
    - Abhängigkeiten sollten immer von konkreteren Modulen niedriger Ebenen zu abstrakten Modulen höherer Ebenen gerichtet sein

Durchführung

- Manuell
  - Händisch ohne hilfreiche Tools
- Semi-automatisch
  - Hilfreiche Tools zum Abarbeiten einzelner Refactoring-Schritte
- Automatisch
  - Vollautomatische Werkzeuge, die kein Wissen des Benutzers benötigen

Durchführung

- Manuell
  1. Identifikation von Stellen, die Refactoring benötigen
  2. Entscheidung über Art des Refactorings
  3. Verhaltenstest / Unit-Test
  4. Durchführung des Refactorings
  5. Durchführung von Qualitätstests
  6. Wartung von möglicherweise abhängigen Programmen und Produkten (Dokumentation, Voraussetzungen,...)

Durchführung

- Semi-automatisch
  1. Identifikation von Stellen, die Refactoring benötigen
  2. Entscheidung über Art des Refactorings
  3. ~~Verhaltenstest / Unit-Test~~
  4. ~~Durchführung des Refactorings~~
  5. Durchführung von Qualitätstests
  6. Wartung von möglicherweise abhängigen Programmen und Produkten (Dokumentation, Voraussetzungen,...)

Durchführung

- Voll-automatisch
  1. ~~Identifikation von Stellen, die Refactoring benötigen~~
  2. ~~Entscheidung über Art des Refactorings~~
  3. ~~Verhaltenstest / Unit-Test~~
  4. ~~Durchführung des Refactorings~~
  5. Durchführung von Qualitätstests
  6. Wartung von möglicherweise abhängigen Programmen und Produkten (Dokumentation, Voraussetzungen,...)

Tools

- Clone analyst tool
  - Finden doppelter Codebausteine
- Photran
  - Umbennen, extract procedure,lokale Variable, Subprogram refactorings, Module refactorings, Loop refactorings (zusammenfügen, umkehren, aufteilen, ausrollen), entfernen obsoleter Sprachbestandteile (z.B. GOTO)
- CamFort
  - Entfernen von veralteten FORTRAN-Methoden, Einführung von Strukturdatentypen
- SPAG
  - Konvertiert F77 zu F90 und beherrscht: Entfernung von unbenutztem Code, Umsetzung von IMPLICIT NONE, Namensänderungen, Groß-/Kleinschreibung für bestimmte Variablentypen(lokal, global)

Refactoring mit Photran

Extract Procedure

Changes to be performed

Extract Procedure

Example.f95 - Refactoring_Example

Example.f95

Original Source

```
program main
    implicit none
    real :: c
    dimension c(4,4,4)
    n=4
    DO i=1,n
        DO j=1,n
            DO k=1,n
                c(i,j,k) = i+j+k
                c(i,j,k) = c(i,j,k)/3
            END DO
        END DO
    END DO
    print*,c
end program
```

Refactored Source

```
program main
    implicit none
    real :: c
    dimension c(4,4,4)
    n=4
    DO i=1,n
        DO j=1,n
            DO k=1,n
                call mean(c)
            END DO
        END DO
    END DO
    print*,c
contains

    subroutine mean(c)
        implicit none
        real :: c(4,4,4)
        c(i,j,k) = i+j+k
        c(i,j,k) = c(i,j,k)/3
    end subroutine

end program
```

< Back | Cancel | OK

Qualitätstest

- Metriken
  - Lines of Code (LOC) (-)
  - Average Method Size (AMS) (-)
  - Depth of Inheritance Tree (DIT) (+)
    - Je tiefer eine Klasse in der Hierarchie, umso wahrscheinlicher ist die Wiederverwendung von abgeleiteten Methoden
    - Tiefere Hierarchien sind komplexer aufgrund zusätzlicher Methoden/Klassen
  - Number of Children (NOC) (+)
    - Höhere Anzahl von Subroutinen → höhere Vererbung/Wiederverwendung
    - Höhere Anzahl erhöht die Wahrscheinlichkeit von fehlerhaften Abstraktionen
  - Lack of Cohesion in Methods (LCOM) (-)
    - Kohäsion ist ein Maß für die Kapselung
    - Mängel von Design-Fehlern lassen sich aufspüren

Qualitätstest

- Metriken
  - Coupling Between Object classes (CBO)(-)
    - Geringere Verknüpfung führt zu einfacherer Wiederverwendung in anderen Anwendungen
    - Höhere Verknüpfungen führt zu sensiblen Designs, die eine Wartung erschweren
    - Testen wird mit steigender Verknüpfung komplizierter
  - Weighted Methods per Class (WMC)(-)
    - Anzahl / Komplexität zeigt an, wie groß der Aufwand für die Weiterentwicklung und Wartung der Klasse ist
    - Anzahl von Methoden innerhalb einer Klasse deutet auf eine hohe Spezialisierung hin und somit auf geringe Wiederverwertbarkeit
  - Response for a Class (RFC)(-)
    - Gibt die Anzahl der möglichen Testszenarien an
    - Je mehr Methoden aufgerufen werden können, umso komplexer ist die Klasse

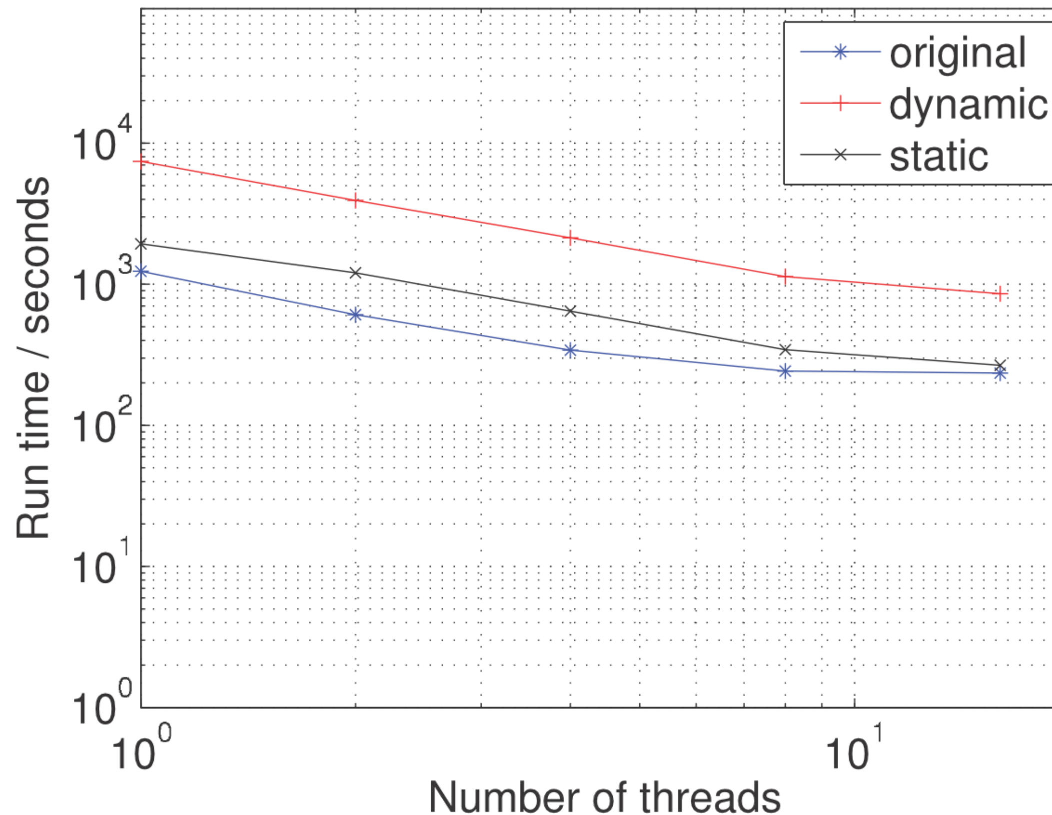
Qualitätstest

- Ergebnisse des Refactorings von HAParaNDA

Metrik	Original Code	Refactored	Differenz
LOC	105.96	61.23	-42 %
AMS	14.41	6.91	-52 %
DIT	0.04	0.91	+2175 %
NOC	0.08	0.85	+963 %
LCOM	21.55	32.20	+49 %
CBO	4.76	4.47	- 6%
WMC	17.88	10.47	- 41 %
RFC	12.80	26.60	+ 108 %

Kallen, Malin, and Sverker Holmgren. „Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application.“ (2014)

Qualitätstest



aus Kallen, Malin, and Sverker Holmgren. "Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application." (2014)

Optimierungsrahmen

- Laufzeit (Echtzeit-Software, HLR)
- Stromverbrauch, Speicherbedarf (eingebettete Software, HLR)
- Sicherheit (Sicherheitssoftware)

Qualitätstest

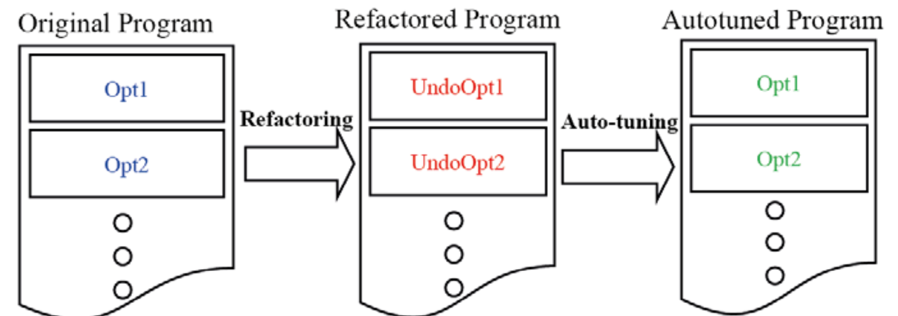
$$Kosten = Kosten_{Entwicklung} + Kosten_{Wartung} + Kosten_{Betrieb} + \cdots$$

$$Kosten_{Wartung} = Kosten_{Verständnis} + Kosten_{Testen} + Kosten_{Implementation}$$

- Bisher gezeigter Code für HLR durch Optimierungen unübersichtlich
- In jenem Falle können die Betriebskosten die Wartungskosten des unübersichtlichen Codes um ein vielfaches übersteigen → Geschwindigkeit vor Lesbarkeit
- Autotuning verschafft Abhilfe → Lesbarkeit trotz Geschwindigkeit

Refactoring in HPC

- Anwendungen sind maschinenoptimiert
- Maschinen ändern sich → Code muss an neue Maschinen angepasst werden.
- Ansatz
  - Refactoring wird benutzt, um die Maschinenoptimierung rückgängig zu machen
  - z.B. loop unrolling, loop tiling
  - Autotuning für erneute Maschinenoptimierung



aus Wang, Chunyan, et al. „Combining Code Refactoring and Auto-Tuning to Improve Performance Portability of High –Performance Computing Applications" (2015)

Autotuning mit HMPPCG

- Benutzung von pragma-Anweisungen
- z.B. #pragma hmppcg unroll { <var>:<factor> } [, contiguous|split|changestep]...

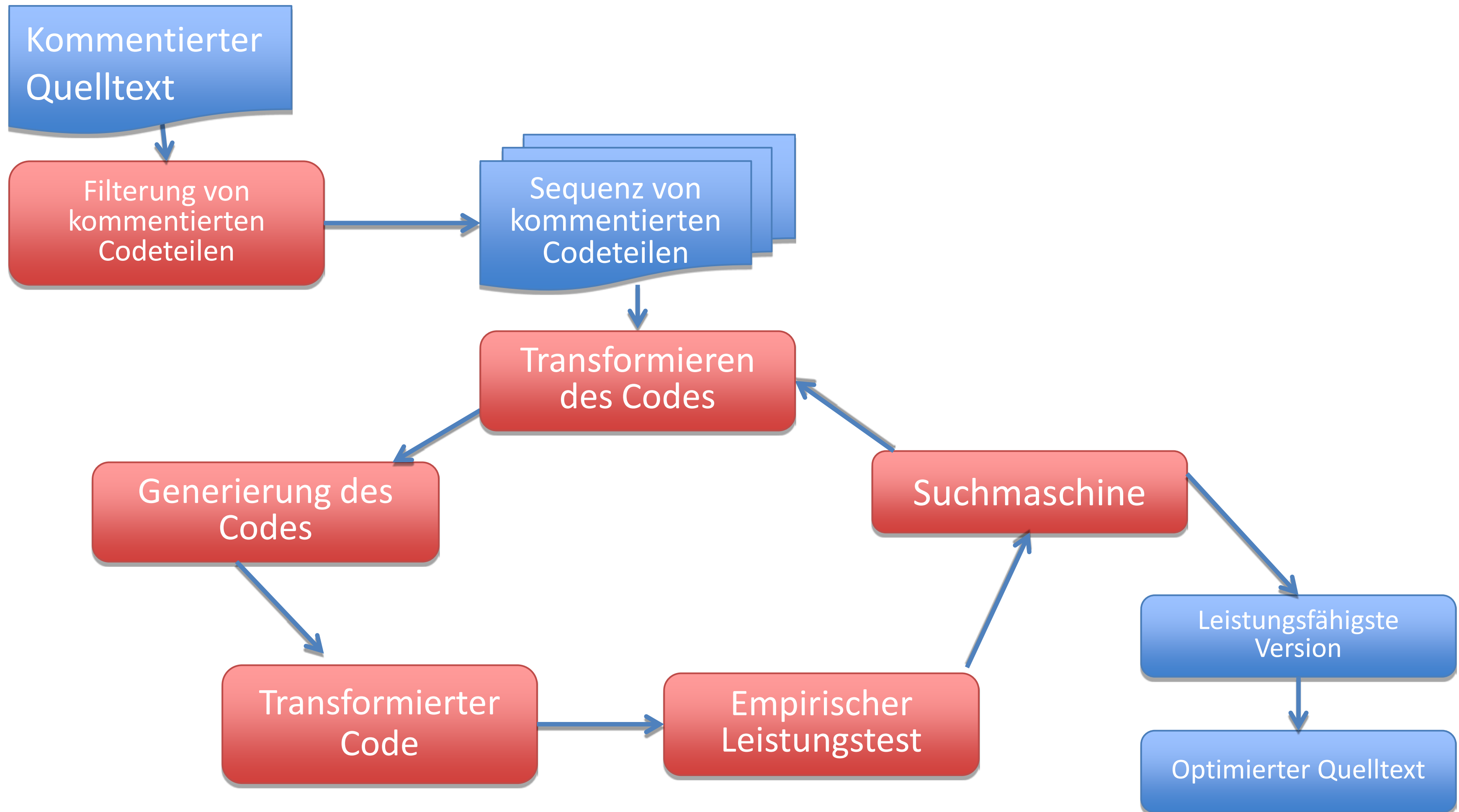
vor

```
#pragma hmppcg unroll i:4, contiguous
for( i = 0 ; i < n ; i++ ) {
  v1[i] = alpha * v2[i] + v1[i];
}
```

danach

```
for (i_1 = 0, __hmppcg_end = (n / 4) - 1; i_1 <= __hmppcg_end; i_1 += 1)
  {
    v1[4 * i_1] = (alpha * (v2[4 * i_1])) + (v1[4 * i_1]);
    v1[(4 * i_1) + 1] = (alpha * (v2[(4 * i_1) + 1])) + (v1[(4 * i_1) + 1]);
    v1[(4 * i_1) + 2] = (alpha * (v2[(4 * i_1) + 2])) + (v1[(4 * i_1) + 2]);
    v1[(4 * i_1) + 3] = (alpha * (v2[(4 * i_1) + 3])) + (v1[(4 * i_1) + 3]);
}
```

Autotuning

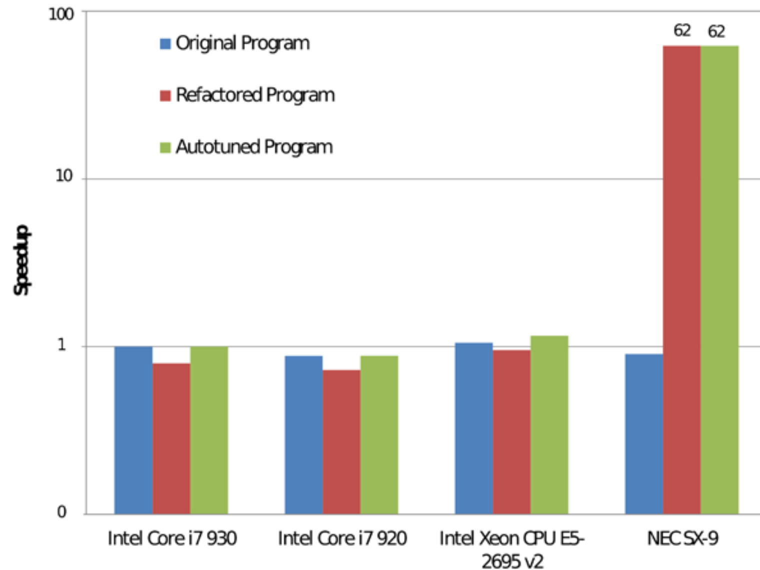


nach Hartano et al., Annotation-Based Empirical Performance Tuning Using Orio (2009)

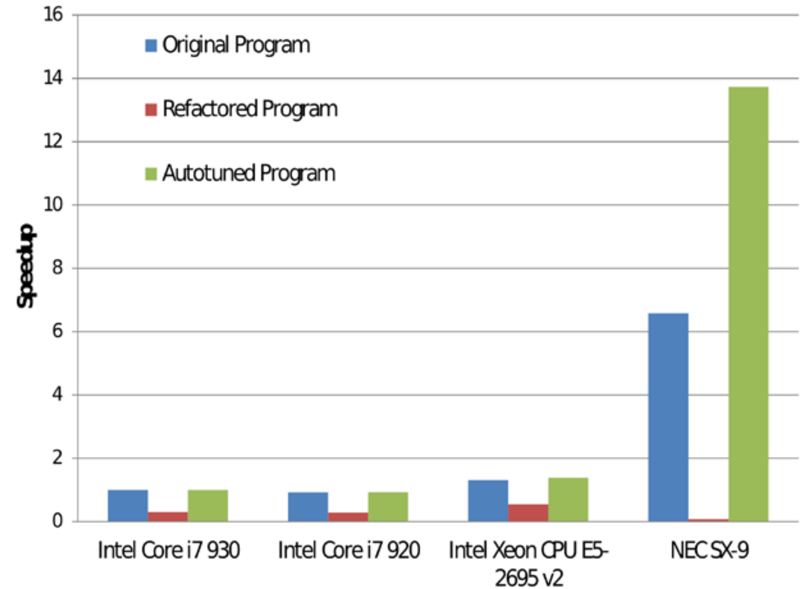
Autotuning mit HMPPCG

- Kombinationen aus Schleifenoptimierungen
- Jede Optimierung hat einen definierten Suchraum
  - Faktoren für ‚loop unroll / tile' (z.B. 1-8,16,32)
  - Kontinuierliches/Geteiltes ‚unrolling'
  - Permutation der Schleifenindizes
  - Hinzufügen/Entfernen von parallelen Bereichen

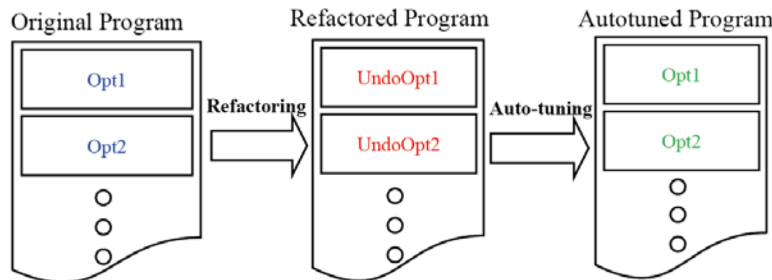
Refactoring in HPC



Speedup bei Loop Unrolling



Speedup bei Loop Tiling



aus Wang, Chunyan, et al. „Combining Code Refactoring and Auto-Tuning to Improve Performance Portability of High –Performance Computing Applications“ (2015)

Zusammenfassung

- Refactoring minimiert den Wartungsaufwand und maximiert die Verständlichkeit des Codes
- Diverse Tools helfen beim Refactoring
- Semi-automatische Tools werden bevorzugt eingesetzt (vollautomatisches Refactoring ist nur in Spezialfällen brauchbar)
- Refactoring ist auch bei laufzeitkritischen Anwendungen anwendbar mithilfe von Autotuning

Botschaft

- Schreibt Programme mit kleinen, klaren Funktionen
  - Das Gedächtnis des Lesers kann nur um die 7 Variablen behalten
- Benutze bedeutungsvolle, unterscheidbare und konsistente Bezeichnungen
  - *p* als Variablenname ist nur ein Bruchteil so hilfreich wie *pressure*
  - benutze *temp* nicht für *temporary* UND *temperature*
- Formatiere deinen Code konsistent
  - Z.B. Style Guide
- Automatisiere Arbeitsschritte (make files, etc.)
- Mache immer nur kleine Arbeitsschritte und kontrolliere deine Arbeit
- Benutze ein System zur Versionskontrolle

Quellenangabe

- Mens, Tom, and Tom Tourwé. „A survey of software refactoring“ *Software Engineering, IEEE Transactions on* 30.2 (2004): 126-139.
- Wang, Chunyan, et al. „Combining Code Refactoring and Auto-Tuning to Improve Performance Portability of High –Performance Computing Applications“ Computation tools 2015, Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking
- Barnes, Nicholas, and David Jones. "Clear climate code: Rewriting legacy science software for clarity." *Software, IEEE* 28.6 (2011): 36-42.
- Kimura, Shuhei, et al. "Move code refactoring with dynamic analysis." *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012.
- Kallen, Malin, and Sverker Holmgren. "Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application." *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014.
- Sokol, Francisco Zigmund, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. "Does the Act of Refactoring Really Make Code Simpler? A Preliminary Study." *4th Brazilian Workshop on Agile Methods*. 2013.
- Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." *Software Engineering, IEEE Transactions on* 20.6 (1994): 476-493.
- Hartono, Albert, Boyana Norris, and Ponnuswamy Sadayappan. "Annotation-based empirical performance tuning using Orio." Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009.