Professur Praktische Informatik

OpenTuner: An Extensible Framework for Program Autotuning

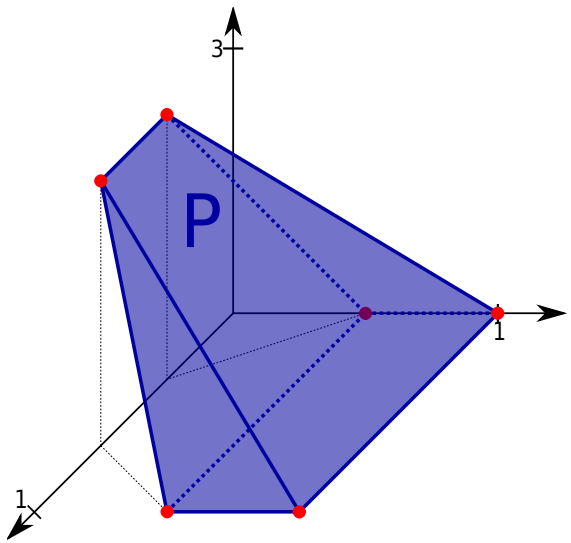
Matthias Tietz
Betreuung: Prof. Dr. Gudula Rünger, Dr. Michael Hofmann

20. November 2016

Gliederung

1. Einleitung

- Suchraum: Menge von Parametern die durchsucht werden soll
- geeignete Suchverfahren abhängig von der Beschaffenheit dieser Menge
- komplexe Struktur und Größe des Suchraums macht Handoptimierung oder vollständige Suche unmöglich (bzw. extrem ineffizient)
  $\rightarrow$ Nadel im Heuhaufen



- Ziele:
  - automatisierter und einfacher Optimierungsprozess
  - bessere und portierbare Performance von domänenspezifischen Programmen

Die 3 wesentlichen Anforderungen an ein Autotuning-Framework:

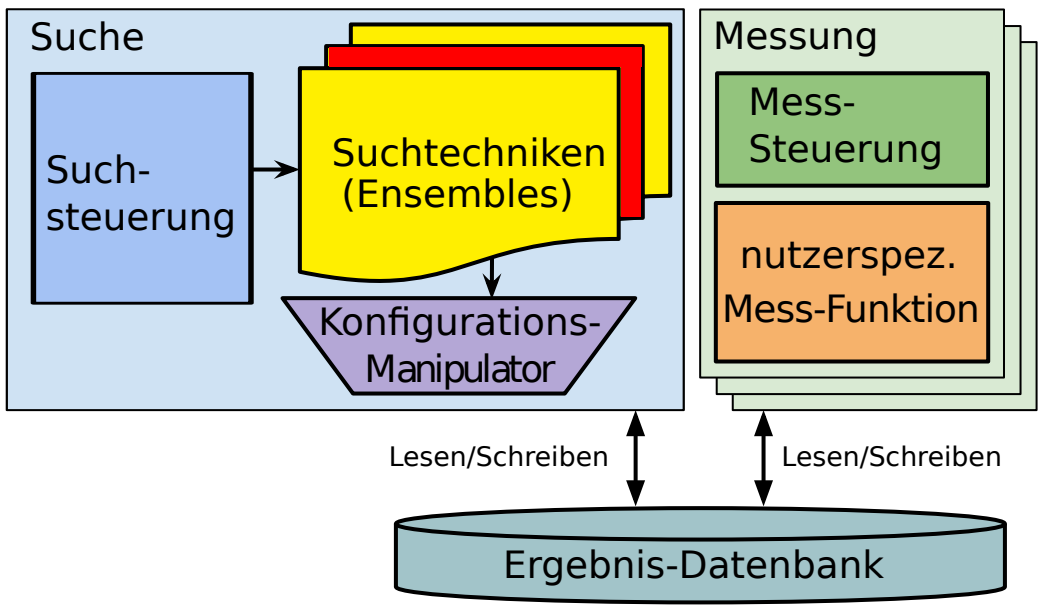
- 1. Eine passende Konfigurations-Repräsentation
  - Darstellung der domänenspezif. Datenstrukturen und Bedingungen
  - Qualität dieser Repräsentation entscheidend für Effizienz des Autotuners
- 2. Größe des validen Konfigurations-Raumes
  - durch Kürzen des Konfigurations-Raumes geht für viele Probleme gute Lösungen verloren (bei bisherigen Autotunern ist dies gängige Praxis, da vollständige Suche)
  - rießige Konfigurationsräume möglich $\rightarrow$ intelligente Suchtechniken notwendig
- 3. Landschaft des Konfigurations-Raumes
  - Suchräume in der Praxis meist sehr komplex
  - domänenspezif. Suchtechniken notwendig um optimale Lösung effizient zu ermitteln

Deshalb OpenTuner:

- Erstellen domänenspezifischer und multi-objective Programm-Autotuner
- vollständig anpassbare Konfigurations-Repräsentation
- erweiterbare Repräsentation für Suchtechniken und Datentypen
- Kombination mehrerer Suchtechniken (*Ensembles*), dynamische Zuweisung der Testanteile für die jeweiligen Suchtechniken
- einfache Schnittstelle zur Kommunikation mit dem zu optimierenden Programm

2. Das OpenTuner Framework

- Autotuning-Problem $\rightarrow$ Suchproblem
- Suchraum: Menge der Konfigurationen (Belegung von Parametern)
- Messung: 1 konkrete Konfig. wird gemessen: Ausführung $\rightarrow$ Ergebnis
- Möglichkeit mehrere Messungen parallel auszuführen



Verwendung

- 1. Suchraum definieren (Konfig.-Manipulator)
- 2. `run()`-Methode definieren: Auswerten der Konfig. im Suchraum $\rightarrow$ Ergebnis
- 3. Festlegen des Optimierungsziels
- Umsetzung mittels kleinem Python-Programm (OpenTuner API), Framework ist ausschließlich in Python geschrieben

Suchtechniken

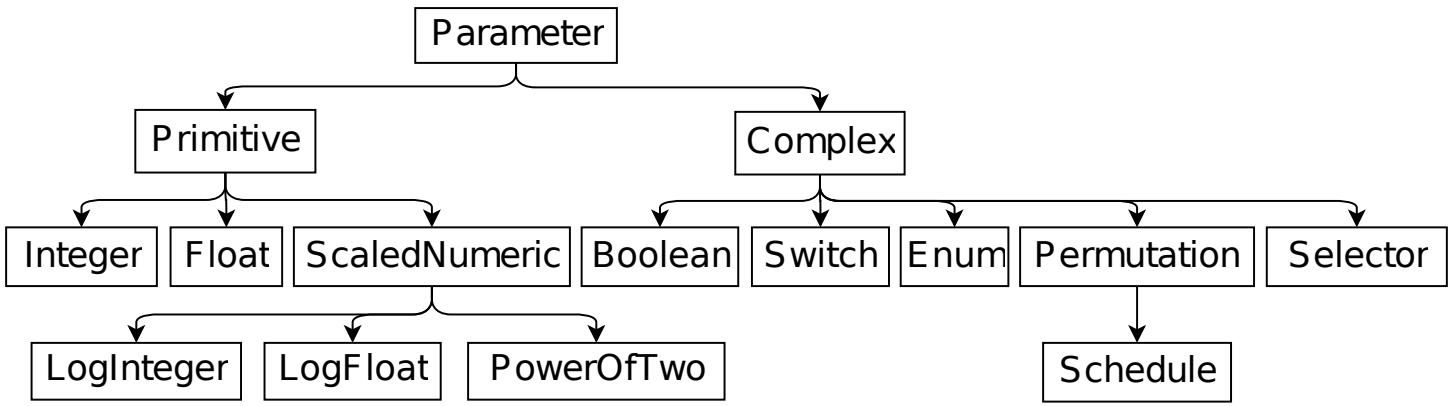
- OpenTuner stellt Suchtechniken für viele Suchraum-Typen bereit
- Ausführen mehrerer Suchtechniken gleichzeitig (Ensembles)
- dynamische Testzuweisung anhand Erfolges dieser Techniken
- erweiterbar: benutzerdefinierte Suchtechniken

Konfigurations-Manipulator

- Abstraktionsschicht zwischen Suchtechnik und roher Konfigurations-Struktur
- Liste der Parameter/Datenstruktur ist dynamisch erweiterbar
- Konfiguration wird als Dictionary verwaltet

Parameter-Typen

- jeder Parametertyp ist verantwortlich für Schnittstelle zwischen roher Parameterrepräsentation und stand. Ansicht dieses Parameters für die Suchtechnik
- Parameterrepräsentation und Abstraktion erweiterbar/konfigurierbar



Primitive Parameter

- numerische Werte mit Unter-/Obergrenze
- `Float` und `LogFloat (-Int)` gleiche Repräsentation in der Konfiguration, aber untersch. Ansicht des zugrundeliegenden Wertes für die Suchtechnik (skaliert)
- Grund: ohne Logskal. würde Effekt der Wertänderung mit steigender Parametergröße sinken
- ähnlich bei `PowerOfTwo` $\rightarrow$ Quadrat nur zulässiger Wert des Parameters

Komplexe Parameter

- haben ein variables Set an Manipulatoren, welche stochastische Änderungen an den Parametern vornehmen
- einfach domänenspezif. Strukturen zum Suchraum hinzuzufügen
- `Boolean, Switch` und `Enum` bewusst als komplex. Parameter, da Suchtechniken bei primitiven Parametern nach Gradients (Steigungen) suchen. Diese Parameter sind aber ungeordnete Sammlung $\rightarrow$ es existiert dafür kein Gradient.
- `Permutation` : Liste von Werte inkl. Manipulatoren zur randomisierten Änderung der Reihenfolge
- `Schedule` ist Sonderfall von `Permutation` : topolog. Sortierung nach jeder Änderung
- `Selector` : Mapping von Integer-Input auf Enum-Type (Darstellung als Baum)

Parameter-Interaktion

- Zusätzlich existieren erweiterbare Methoden für die Suchtechniken um zwischen mehreren Parametern zu interagieren. (z.B. Differenz-Funktion)

Optimierungsziele

- OpenTuner unterstützt mehrere Ziele, standardmäßig wird nach der `Zeit` optimiert
- `Genauigkeit, Energie, Größe` oder ein nutzerdef. Ziel
- Es können auch mehrere Ziele zugleich verfolgt werden, bspw. `Genauigkeit einhalten, gleichzeitig Zeit minimieren`

Suchen und Messen

- kommunizieren ausschließlich über die Ergebnis-Datenbank
- Motivation für die Unterteilung zwischen diesen beiden Modulen:
  - Ermöglichen der Parallelität zwischen mehreren Prozessen (Suchen und Messen)
  - Autotuning während Ausführung der Anwendung oder in Wartezeit (Online/Sideline Learning)
  - Mess-Modul einfach ersetzbar ohne das gesamte Framework zu modifizieren (Domäne: Embedded/Mobil $\rightarrow$ leichtgewichtiges Mess-Modul)

Ergebnis-Datenbank

- vollfunktionale SQL-Datenbank
- alle grundlegenden DB-Typen unterstützt, default: SQLite
- Abfragen und Eintragen der Ergebnisse in einer Vielzahl von Möglichkeiten
- nützlich für die Performance-Beobachtung der Suchtechniken

3. Konkrete Anwendungen der Entwickler

Vorstellen eines Beispiels aus einer bestimmten Kategorie

a) GCC/G++ Flags

- klassische Parameter-Optimierung
- unterstützte Flags: `g++ --help=optimizers`
- Parameter inkl. zulässiger Wertebereiche: `params.def` (gcc source code)
- Implementierung des Autotuners:
  - 1. Erstellen des `configuration manipulator` : Set der Parameter (Suchraum)
  - 2. Erstellen der `run`-Funktion
  - 3. Festlegen des Optimierungsziels

```
import opentuner                                                    1
from opentuner import ConfigurationManipulator                      2
from opentuner import EnumParameter                                 3
from opentuner import IntegerParameter                              4
from opentuner import MeasurementInterface                          5
from opentuner import Result                                        6
                                                                    7
GCC_FLAGS = [                                                       8
  'align-functions', 'align-jumps', 'align-labels',                 9
  'branch-count-reg', 'branch-probabilities',                       10
  # ... (176 total)                                                 11
]                                                                   12
                                                                    13
# (name, min, max)                                                  14
GCC_PARAMS = [                                                      15
  ('early-inlining-insns', 0, 1000),                                16
  ('gcse-cost-distance-ratio', 0, 100),                             17
  # ... (145 total)                                                 18
]                                                                   19
```

```
class GccFlagsTuner(MeasurementInterface):

  def manipulator(self):
    """
    Define the search space by creating a
    ConfigurationManipulator
    """
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('opt_level', 0, 3))
    for flag in GCC_FLAGS:
      manipulator.add_parameter(
        EnumParameter(flag,
                      ['on', 'off', 'default']))
    for param, min, max in GCC_PARAMS:
      manipulator.add_parameter(
        IntegerParameter(param, min, max))
    return manipulator
```

```
  def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then
    return performance
    """
    cfg = desired_result.configuration.data
    gcc_cmd = 'g++ raytracer.cpp -o ./tmp.bin'
    gcc_cmd += ' -O{0}'.format(cfg['opt_level'])
    for flag in GCC_FLAGS:
      if cfg[flag] == 'on':
        gcc_cmd += ' -f{0}'.format(flag)
      elif cfg[flag] == 'off':
        gcc_cmd += ' -fno-{0}'.format(flag)
    for param, min, max in GCC_PARAMS:
      gcc_cmd += ' --param {0}={1}'.format(
        param, cfg[param])

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0
    run_result = self.call_program('./tmp.bin')
    assert run_result['returncode'] == 0
    return Result(time=run_result['time'])

if __name__ == '__main__':
  argparser = opentuner.default_argparser()
  GccFlagsTuner.main(argparser.parse_args())
```