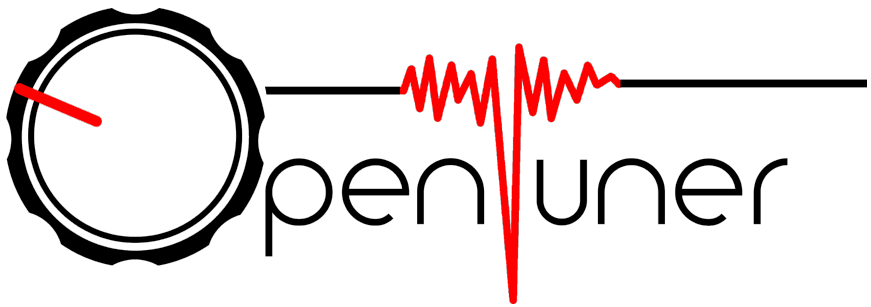
OpenTuner: An Extensible Framework for Program Autotuning

Hauptseminar WS 2016/2017 - Bachelor Informatik

Matthias Tietz

10.02.2017

Gliederung



Das Paper

- Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, Saman Amarasinghe
- Massachusetts Institute of Technology, 2014

Motivation

- Tuning-Prozess automatisierbar
- bestehende Autotuning-Frameworks domänenspezifisch (ATLAS, FFTW)

OpenTuner:

- Optimierungsverfahren komfortabler und oft effizienter
- mehrere Optimierungsziele komibinierbar, hohe Erweiterbarkeit
- Performance-Portabilität

verwendete Begriffe:

- **Parameter**: Variablen eines bestimmten Typs
- **Konfiguration**: konkrete Zuweisungen von Parametern
- **Konfigurations-Manipulator**: Bearbeiten der Parameter in Konfig.

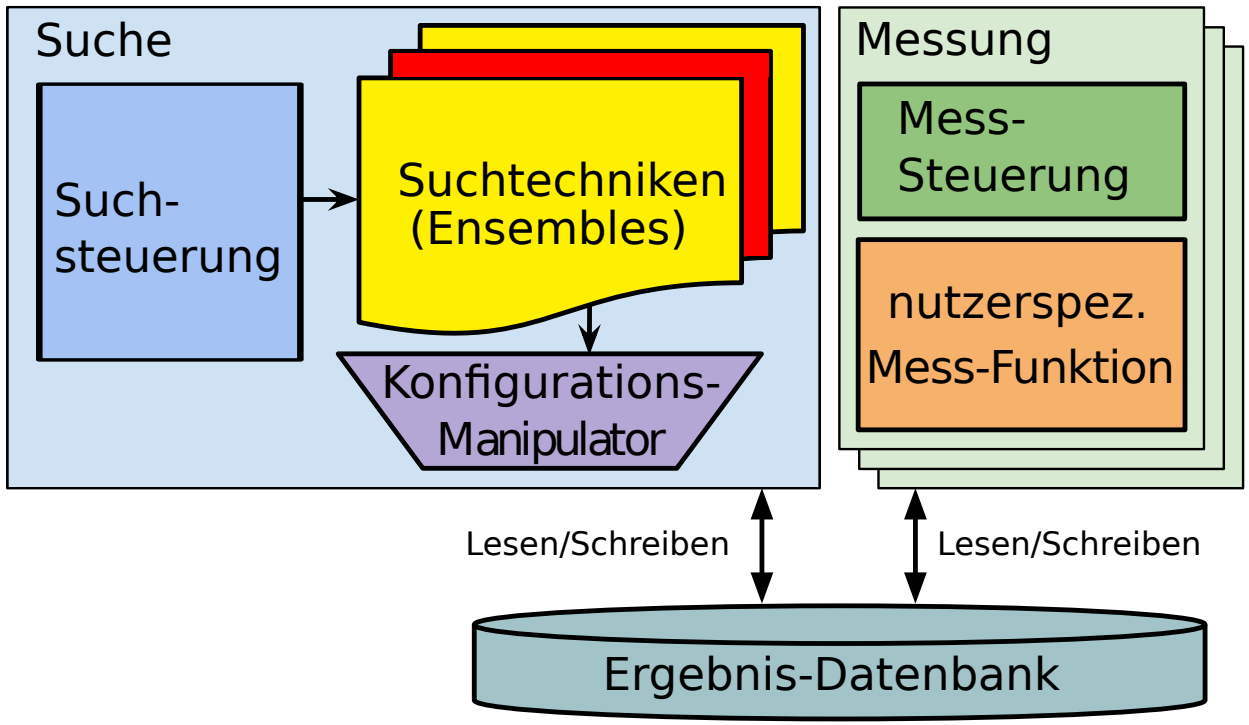
- **Suchraum**: zu durchsuchende Menge von Konfigurationen
- **Suchtechnik**: Suchraum erforschen (KM), Anfragen für Messung (Ergebnis)
- **Suchprozess**: Lesen/Schreiben von Konfigurationen

- **Messprozess**: konkrete Konfiguration ausführen
- **Messresultat**: Ergebnis der Messung, abhängig von Optimierungsziel(en)
- **Ergebnisdatenbank**: Festhalten aller Ergebnisse während Tuning-Vorgangs

Herausforderungen bei der Entwicklung eines Autotuning-Frameworks:

- 1. Eine passende Konfigurations-Repräsentation
  - Darstellung der domänenspezif. Datenstrukturen und Bedingungen
  - Qualität der Repräsentation entscheidend für Effizienz des Autotuners
- 2. Größe des gültigen Konfigurations-Raumes
  - Kürzen des Konfigurations-Raumes $\rightarrow$ Verlieren guter Lösungen
  - rießige Konfigurationsräume möglich $\rightarrow$ intelligente Suchtechniken notwendig
- 3. Beschaffenheit des Konfigurations-Raumes
  - Suchräume in der Praxis meist sehr komplex
  - domänenspezif. Suchtechniken notwendig

2. OpenTuner Framework



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)

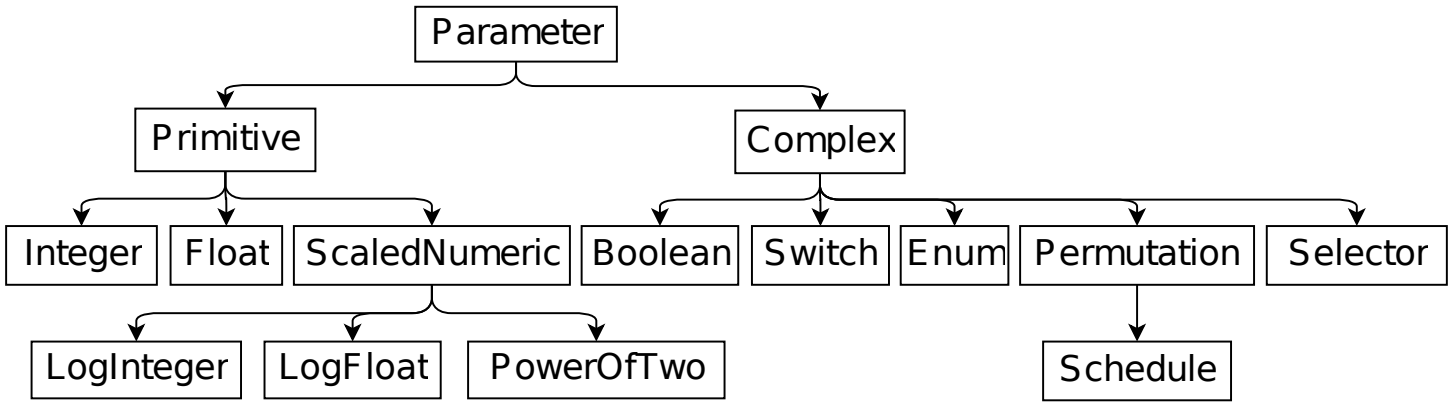
Suchtechniken

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

Konfigurations-Manipulator

- Abstraktionsschicht zwischen Suchtechnik und Konfigurations-Struktur
- Zugriff auf Parameter-Objekte: Lesen/Schreiben per Suchtechnik

Parameter-Typen



Primitive Parameter

- numerische Werte mit Unter-/Obergrenze
- `Float` und `LogFloat (-Int)` gleiche Repräsentation in Konfiguration, aber zugrundeliegender Wert für Suchtechnik skaliert
- Grund: ohne Logskal. würde Effekt der Wertänderung mit steigender Parametergröße sinken

Komplexe Parameter

- haben variables Set an Manipulatoren $\rightarrow$ stochastische Parameter-Änderungen
- `Boolean, Switch` und `Enum` bewusst als komplex. Parameter, Darstellung als ungeordnete Sammlung $\rightarrow$ es existiert kein Gradient (wie bei prim. Param.)
- `Permutation` : Liste von Werte inkl. Manipulatoren zur randomisierten Änderung der Reihenfolge
- `Schedule`, `Selector`

Optimierungsziele

- OpenTuner unterstützt mehrere Ziele, default: `time`
- `accuracy, energy, size, confidence` oder ein nutzerdef. Ziel
- Es können auch mehrere Ziele zugleich verfolgt werden, bspw. *Genauigkeit einhalten, gleichzeitig Zeit minimieren*

3. Anwendungsbeispiele

GCC/G++ Flags

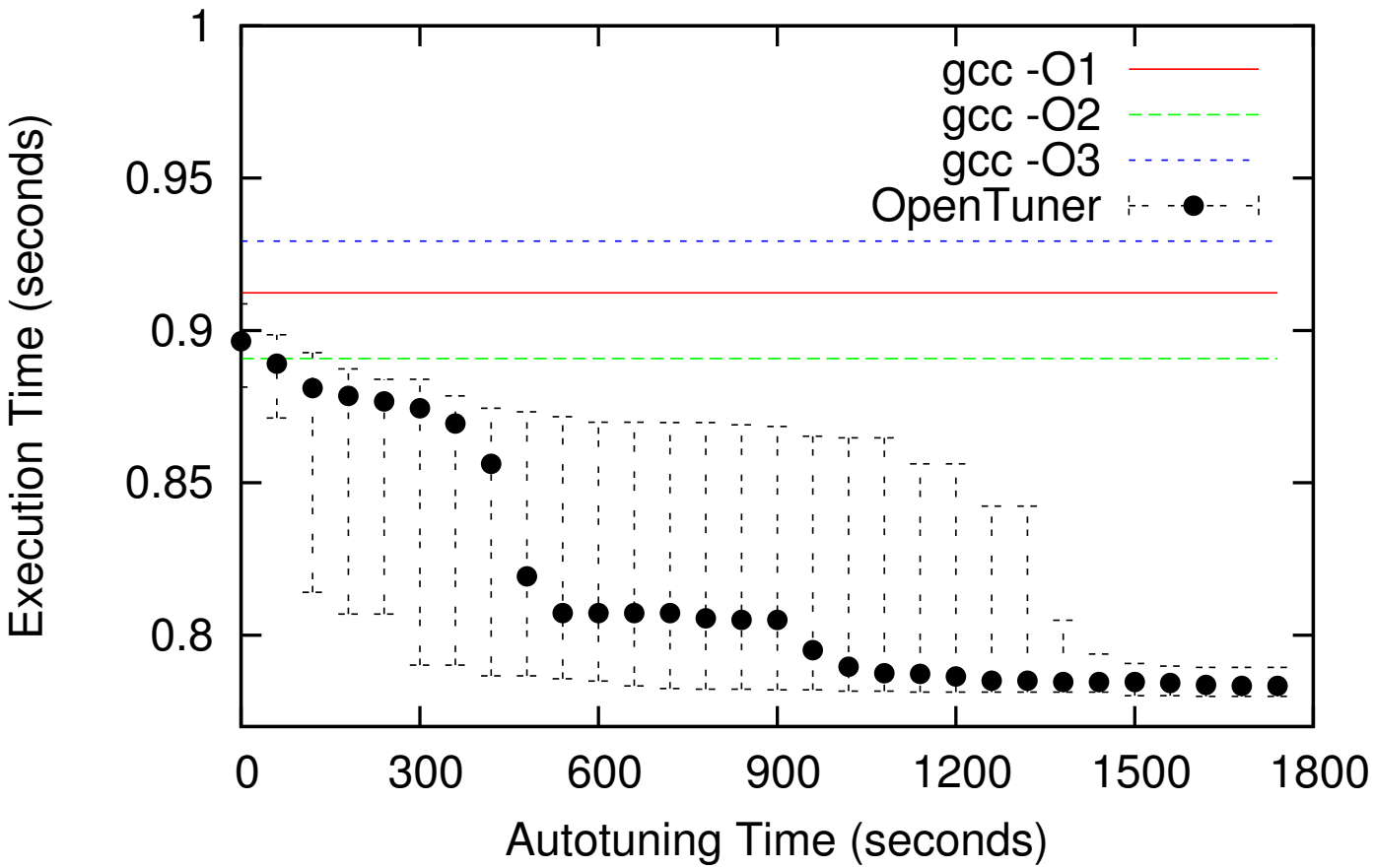
- klassische Parameter-Optimierung
- unterstützte Flags: `g++ --help=optimizers`
- Parameter inkl. zulässiger Wertebereiche: `params.def` (gcc source code)

- Implementierung des Autotuners:
  - 1. Erstellen des `configuration manipulator`
  - 2. Erstellen der `run`-Funktion
  - 3. Festlegen des Optimierungsziels

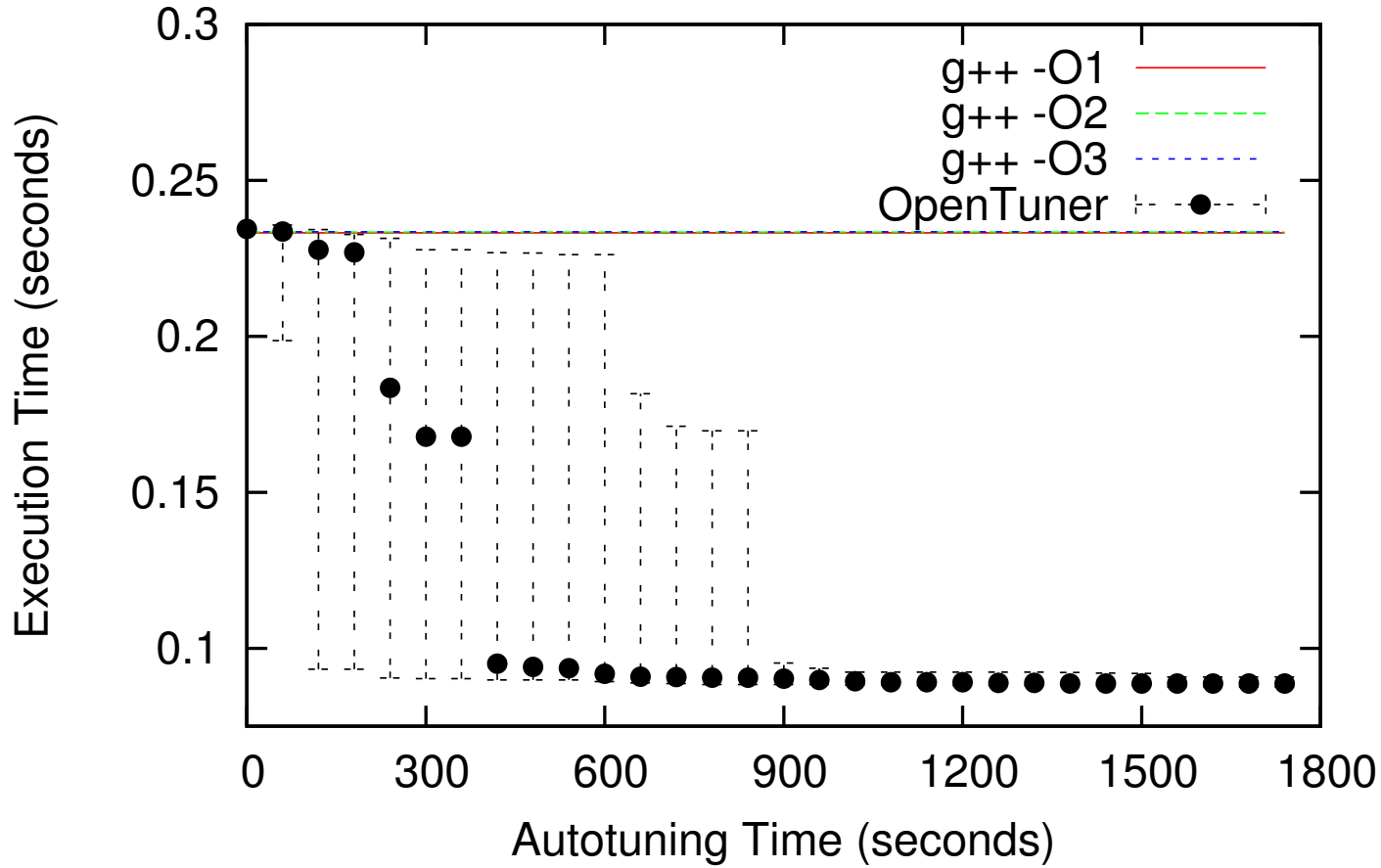
```
import opentuner
from opentuner import ConfigurationManipulator
from opentuner import EnumParameter
from opentuner import IntegerParameter
from opentuner import MeasurementInterface
from opentuner import Result

GCC_FLAGS = [
  'align-functions', 'align-jumps', 'align-labels',
  'branch-count-reg', 'branch-probabilities',
  # ... (176 total)
]

# (name, min, max)
GCC_PARAMS = [
  ('early-inlining-insns', 0, 1000),
  ('gcse-cost-distance-ratio', 0, 100),
  # ... (145 total)
]
```

```
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'])
```


Execution Time (seconds)
1
0.95
0.9
0.85
0.8
0
300
600
900
1200
1500
1800
Autotuning Time (seconds)
gcc -O1
gcc -O2
gcc -O3
OpenTuner

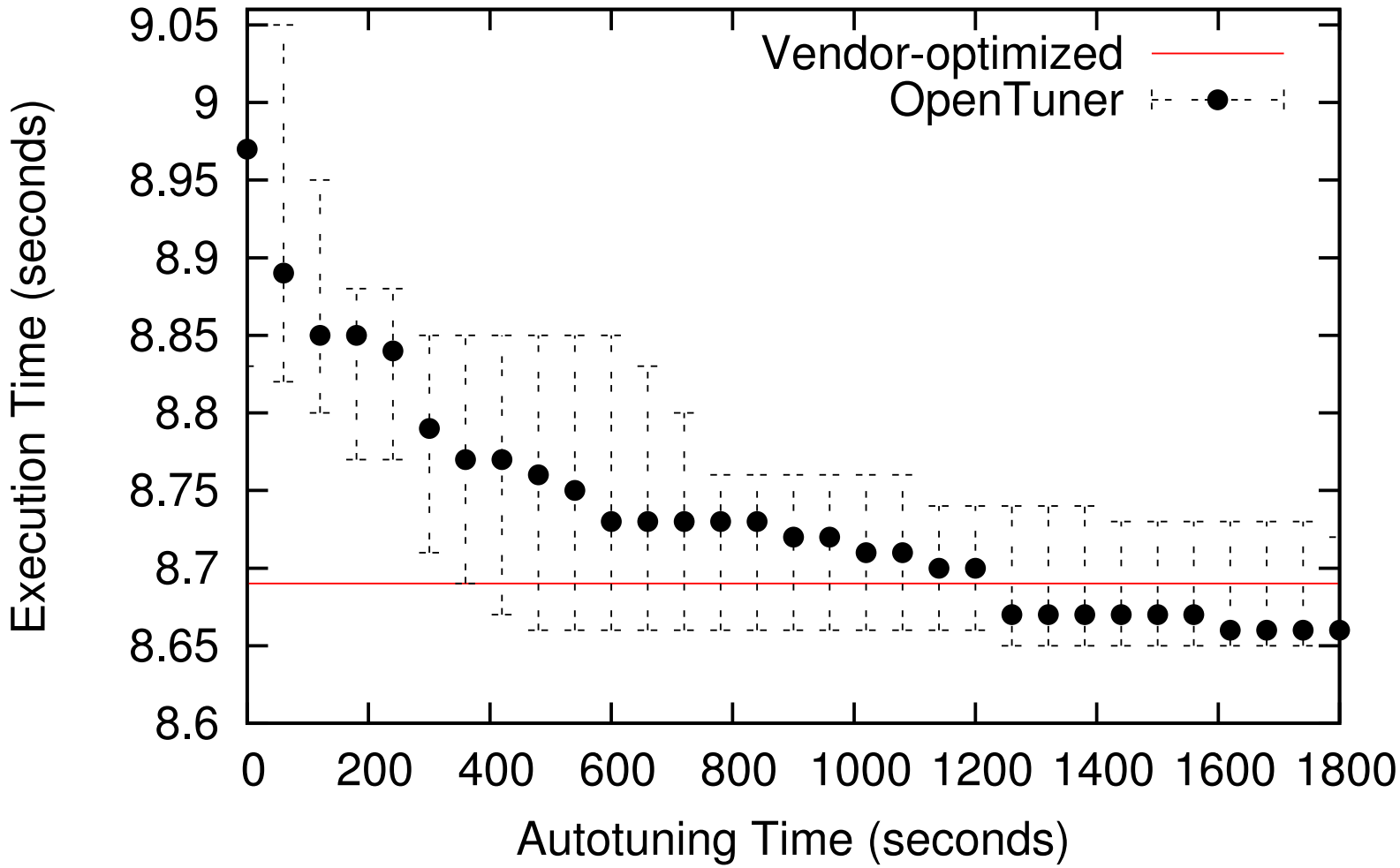
(a) fft.c



(b) matrixmultiply.cpp

High Performance Linpack Benchmark (HPL)

- Bestimmen der `floating point performance` von Rechnern
- Kriterium für Wahl der Top 500 Supercomputer
- HPL misst Geschwindigkeit für Lösen eines großen LGS
- HPL besitzt eingebauten Autotuner (vollst. Suche)
- ca. 15 versch. Parameter (u. a. Matrix-Blockgröße)
- größter Performance-Einfluss: Blockgröße (`IntegerParameter`)
- OpenTuner im Vergleich zu HPL-Implementierung von Intel


Vendor-optimized
OpenTuner
Execution Time (seconds)
Autotuning Time (seconds)
9.05
9
8.95
8.9
8.85
8.8
8.75
8.7
8.65
8.6
0
200
400
600
800
1000
1200
1400
1600
1800

Super Mario

- Abschließen des ersten Levels durch Sequenz von Tasten-Eingaben (Konfig.)
- Tasten-Eingaben $\rightarrow$ Speichern in Datei $\rightarrow$ Abspielen in NES-Emulator
- Wiedergabe im Emulator schneller als in Echtzeit
- Auswerten mehrerer Instanzen des Emulators parallel

- Ziel: Maximieren der #Pixel die Mario nach rechts läuft
- Suchraum: Tendenz horizontal nach rechts bewegen
- Eingaben-Modellierung:
  - welche Taste
  - zu welchem Zeitpunkt (Frame)
  - wie lange gedrückt wird

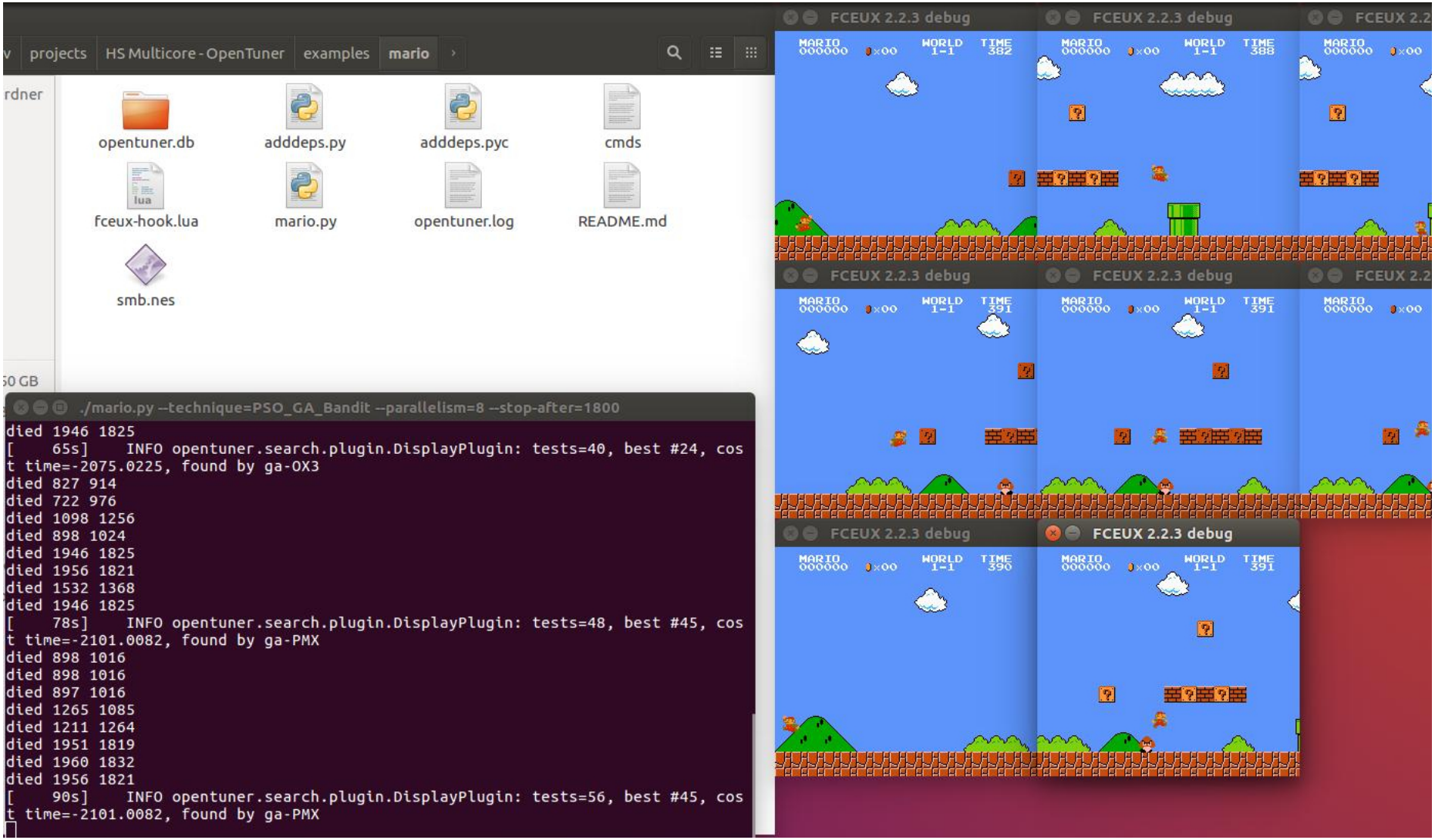


Abbildung: Tuning-Prozess

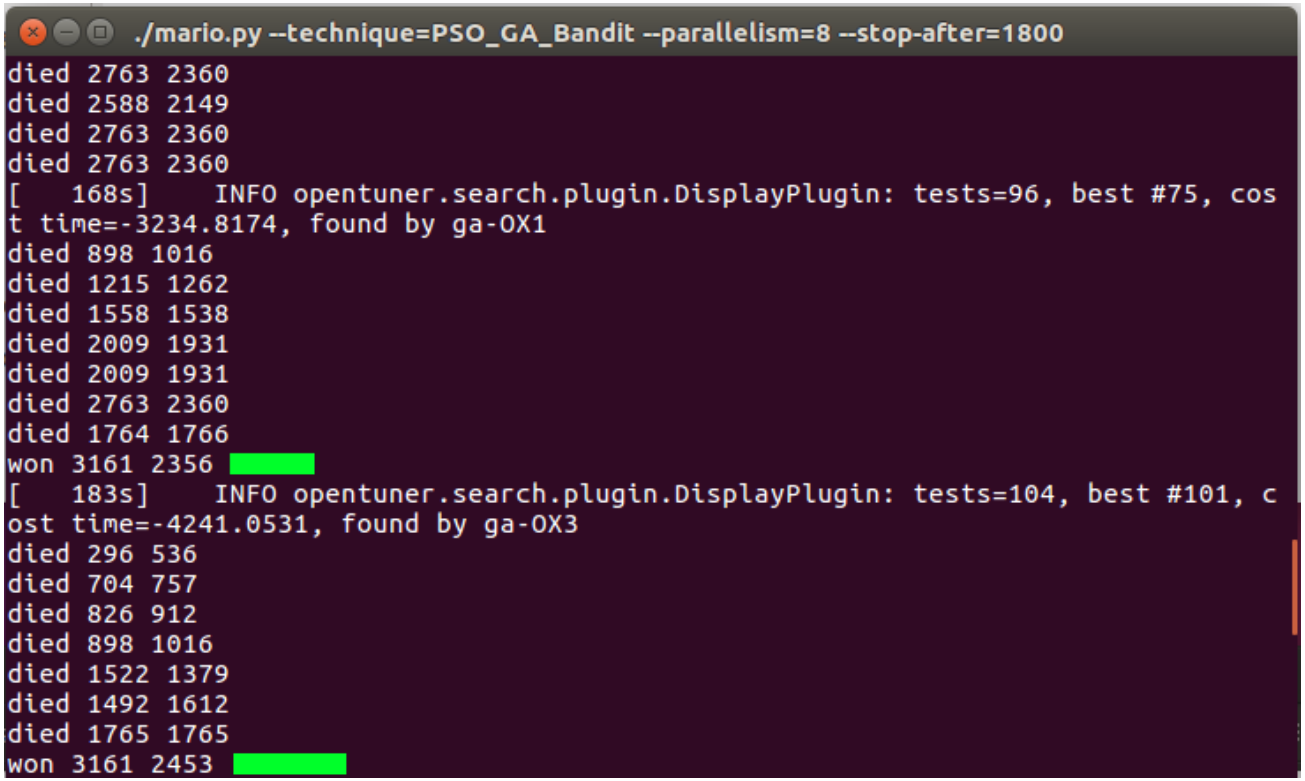


Abbildung: Ziel erreicht

Demo: First Level of Super Mario

4. Aktueller Stand des Projekts

- `www.github.com/jansel/opentuner`, MIT-Lizenz
- Entwicklung 2014-2015
- gute Dokumentation, Support via Issues

PRO

- Flexibilität und Erweiterbarkeit
- einfache und klare Handhabung
- große Anzahl von Beispielanwendungen
- Unterstützung mehrerer Optimierungsziele (Kombination)
- bessere Ergebnisse als bisherige Autotuner

CONTRA

- Optimierungsgrad variiert stark (Bsp.)
- Trade-off: Zeitaufwand $\leftrightarrow$ Nutzen

Quellen

 Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, Saman Amarasinghe. *OpenTuner: An Extensible Framework for Program Autotuning.* 2014.

 Shoaib Kamil. *Hands on session.* PLDI 2015

 `www.github.com/jansel/opentuner`

 `www.opentuner.org`