Linear-Scan-Registerallokation und die native OCaml-Toplevel

Marcell Fischbach

Übersicht

1 Einleitung

2 Linear-Scan-Algorithms

3 Die native OCaml-Toplevel

4 Performance

5 Fazit

Einleitung

Entwicklungsziele eines Compilers

- Schneller Compiler
- Schneller Code
- Konkurrierendes Entwicklungsziel

Einleitung

Entwicklungsziele eines Compilers

- Schneller Compiler
- Schneller Code
- Konkurrierendes Entwicklungsziel

Wahl des Compilers abhängig von Einsatzgebiet

- Offline $\rightarrow$ schneller Code
- Online $\rightarrow$ schneller Compiler

Registerallokation

Was ist Registerallokation?

- Schritt der Codegenerierung eines Compilers
- Arbeitet auf *Intermediate-Language*
- Pseudoregister für jeden Wert (Variablen, Constanten, Zwischenergebnisse)
- Vergabe und Zuweisung von physikalischem Register an Pseudoregister

Registerallokation ...

Warum Registerallokation?

- Instruktion auf Registeroperand erheblich schneller
- Somit Steigerung der Performance
- Diverse Instruktionen erfordern Registeroperanden

Registerallokation ...

Warum Registerallokation?

- Instruktion auf Registeroperand erheblich schneller
- Somit Steigerung der Performance
- Diverse Instruktionen erfordern Registeroperanden

Problem

- Sehr viele Pseudoregister, wenige physikalische Register
- Nur heurischtische Lösungen
- Übliches Verfahren: Graph-Coloring

Übersicht

1 Einleitung

2 Linear-Scan-Algorithms

3 Die native OCaml-Toplevel

4 Performance

5 Fazit

Der grundlegende Algorithmus

Jedes Pseudoregister

- wird durch ein Intervall repräsentiert,
- welches die Liveness des Pseudoregisters widerspiegelt

Der grundlegende Algorithmus

Jedes Pseudoregister

- wird durch ein Intervall repräsentiert,
- welches die Liveness des Pseudoregisters widerspiegelt

Jedem Intervall

- wird ein Register zugeordnen, jedoch
- niemals zwei überlappenden Intervallen das Selbe

Der grundlegende Algorithmus ...

Liste `active`

- alle Intervalle mit zugewiesenem Register
- Größe durch Anzahl physikalischer Register begrenzt

Der grundlegende Algorithmus ...

Liste `active`

- alle Intervalle mit zugewiesenem Register
- Größe durch Anzahl physikalischer Register begrenzt

Chronologische bearbeitung jeden Intervalls

- Entfernen aller abgelaufenen Intervalle aus `active`
- Platz in `active` $\rightarrow$ Register zuweisen, Intervall aufnehmen
- Kein Platz $\rightarrow$ letztes Intervall *spillen*

Intervalldarstellung

Ein Intervall besteht aus

- dem Pseudoregister, welches es repräsentiert
- den Grenzen $[begin, end]$ und
- Ranges, welche ebenfalls auf $[begin, end]$ definiert sind.

Intervalldarstellung

Ein Intervall besteht aus

- dem Pseudoregister, welches es repräsentiert
- den Grenzen $[begin, end]$ und
- Ranges, welche ebenfalls auf $[begin, end]$ definiert sind.

Jeder Instruktionsindex unterteilt sich in

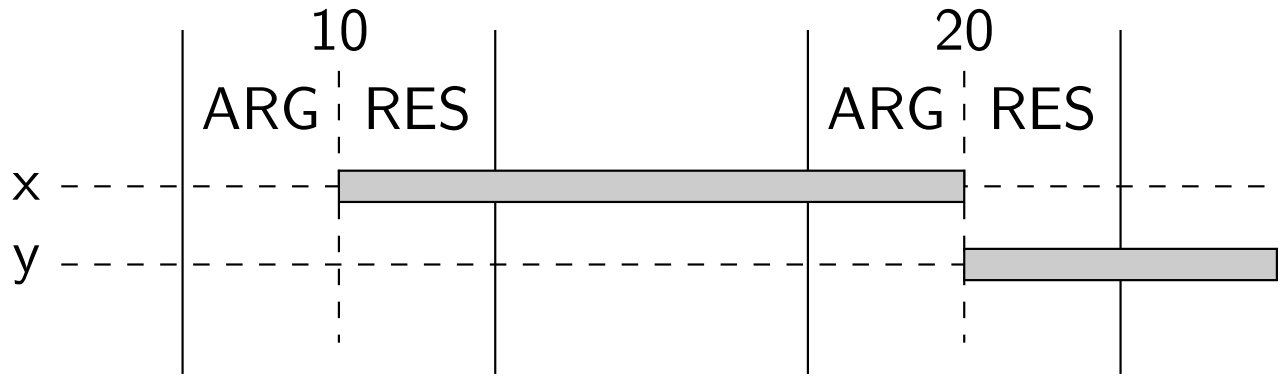
- *ARG* - den Argumentslot und
- *RES* - den Resultslot

Intervalldarstellung - Beispiel

Für folgenden Code

```
10: x := 10;
20: y := 2 * x;
```

entstehen folgende Intervalle



Instruktionsbaum

Eine OCaml-Instruktion enthält

- Resultat
- Argumente
- Live
- Instruktionsbeschreibung
- Next

Traversierung des Instruktionsbaums

Instruktionsbeschreibung definiert den Baum

- `Iend` - Ende einer Instruktionsfolge
- `Iifthenelse` - Instruktionsfolgen für *Then* und *Else*
- `Iswitch` - Für jeden *Case* eine Instruktionsfolge
- `Iloop` - Eine Instruktionsfolge für den *Loopkörper*
- `Icatch` & `Itrywith` - Zwei Instruktionsfolgen für den *Körper* und den *Handler*

Erzeugen und Updaten der Intervalle

Für jeden Instruktion werden *Result*, *Argument* und *Live* betrachtet und

- ein neues Intervall mit Range angelegt, wenn keins existiert
- die letzte Range erweitert, wenn möglich
- eine neue Range angelegt, wenn keine Erweiterung möglich

Fixe Intervalle

Fixe Pseudoregister

- bilden festes Mapping auf physikalische Register
- werden benutzt, wenn best. Register benötigt werden
- z.B. schreibt `Idiv` Resultat nach `eax` und `edx`

Für jedes fixe Pseudoregister wird ein fixes Intervall angelegt.

Zerstörte Intervalle

Manche Instruktionen überschreiben den Inhalt von Registern

- So überschreibt `Idiv` die Register `eax` und `edx`
- Keine Zuweisung solcher Register an Intervalle
- Da *falscher* Inhalt überschrieben würde

Zerstörte Intervalle

Manche Instruktionen überschreiben den Inhalt von Registern

- So überschreibt `Idiv` die Register `eax` und `edx`
- Keine Zuweisung solcher Register an Intervalle
- Da *falscher* Inhalt überschrieben würde

Lösung

- Punktuelle Ranges in den fixen Intervallen
- Sie verhindern die Zuweisung eines solchen Registers

Drei Listen

Abweichend von dem Originalalgorithmus werden drei Listen verwaltet

- `active` - alle aktuellen Intervalle
- `inactive` - nicht abgelaufene Intervalle
- `fixed` - alle fixen Intervalle

Sequentielle Abarbeitung der Intervalle

Für jedes Intervall

- alle abgelaufenen Intervalle aus `active`, `inactive` und `fixed` entfernen
- ein freies Register vergeben, wenn dies nicht möglich
- ein Register zum *spillen* auswählen

Vergabe eines freien Registers

Um ein freies Register zu emitteln

- Registerpool mit allen Register
- alle aus `active` aus dem Pool entfernen
- alle Überschneidenden aus `inactive` aus dem Pool entfernen
- alle Überschneidenden aus `fixed` aus dem Pool entfernen
- wenn Pool leer $\rightarrow$ keine freie Vergabe möglich
- sonst $\rightarrow$ erstes Register aus Pool zuweisen

Aktuelles Intervall in `active` aufnehmen.

Vergabe bei blockierten Registern

Wenn kein Register verfügbar

- Ein Register muss *gespillt* werden
- Intervall mit spätestem Endzeitpunkt auswählen
- Entweder aktuelles Intervall `current`
- oder letztes Intervall `last` aus `active`

Vergabe bei blockierten Registern

Wenn kein Register verfügbar

- Ein Register muss *gespillt* werden
- Intervall mit spätestem Endzeitpunkt auswählen
- Entweder aktuelles Intervall `current`
- oder letztes Intervall `last` aus `active`

Wenn aktuelles Intervall

- `current` $\leftarrow$ Stackslot

Vergabe bei blockierten Registern

Wenn kein Register verfügbar

- Ein Register muss *gespillt* werden
- Intervall mit spätestem Endzeitpunkt auswählen
- Entweder aktuelles Intervall `current`
- oder letztes Intervall `last` aus `active`

Wenn aktuelles Intervall

- `current` $\leftarrow$ Stackslot

Wenn letztes Intervall aus `active`

- `current` $\leftarrow$ Register von `last`
- Entferne `last` aus `active` und füge `current` hinzu.

Konfliktauflösung

Nach Registervergabe können illegale Speicheroperanden auftreten. Z.B.:

- Speicher-Speicher-Move `mov [rax], [rbx]` (*i386*, *amd64*)
- Speicher-Operand bei Instruktion $\neq$ Load/Store (*PPC*, *ARM*)

Konfliktauflösung

Nach Registervergabe können illegale Speicheroperanden auftreten. Z.B.:

- Speicher-Speicher-Move `mov [rax], [rbx]` (*i386*, *amd64*)
- Speicher-Operand bei Instruktion $\neq$ Load/Store (*PPC*, *ARM*)

Lösung:

- *Reload* - Funktion des OCaml-Compilers
- Einfügen von Load/Store Instruktionen
- $\Rightarrow$ Registerallokation erneut ausführen

Übersicht

1 Einleitung

2 Linear-Scan-Algorithms

3 Die native OCaml-Toplevel

4 Performance

5 Fazit

Was ist die OCaml-Toplevel?

Bei dem OCaml-Compiler

- Wandlung von ML-Code in Bytecode $\rightarrow$ Datei
- Wandlung von ML-Code in Maschinencode $\rightarrow$ Datei

Was ist die OCaml-Toplevel?

Bei dem OCaml-Compiler

- Wandlung von ML-Code in Bytecode → Datei
- Wandlung von ML-Code in Maschinencode → Datei

Bei der OCaml-Toplevel

- Wandlung von ML-Code in Bytecode → Ausführen
- Wandlung von ML-Code in Maschinencode → Ausführen

Workflow der nativen OCaml-Toplevel

Erzeugung von Maschinencode

- ML-Code → Assembler → Datei
- Toolchain Assembler `as` → Datei
- Toolchain Linker `ln` → Datei
- Runtimelinker
- Ausführen

Motivation

Aus dieversen Gründen kein attraktives Verhalten

- Viele IO-Operationen des Systems sind langsam
- Benötigt externe Toolchain
- Diese ist evtl. nicht Installiert
- Oder für die Nutzungsumgebung nicht vorhanden

Diese Gründe führten zu der Implementirung des `ocamlnatjit`

Struktur des Emitters

Neues und altes Verfahren sollte möglich sein

- Es wurde eine abstrakte Klasse `emitter` entworfen
- Für jede Instruktion des Instruction-Sets eine Methode z.B. `method mov`

Struktur des Emitters

Neues und altes Verfahren sollte möglich sein

- Es wurde eine abstrakte Klasse `emitter` entworfen
- Für jede Instruktion des Instruction-Sets eine Methode z.B. `method mov`

Zwei Implementierungen des `emitters`

- `asm_emitter` für die Nutzung der Toolchain
- `obj_emitter` für die `ocamlnatjit`

Darstellung der Operanden

Jede Instruktion hat Operanden unterschiedlichen Typs

- Register - z.B. `%eax`
- Speicher - z.B. `10(%eax, %ebx, 4)`
- Konstanten - z.B. `$10`
- Symbole - z.B. `ocaml_test_fact`

Konkrete Instruktion hängt von Operandentypen ab.

Symbole

Symbole markieren Positionen im Speicher

- Beim Schreiben im Speicher aktuelle Position merken
- Bei definition von Symbol aktuelle Position zu Symbol speichern

Symbole

Symbole markieren Positionen im Speicher

- Beim Schreiben im Speicher aktuelle Position merken
- Bei definition von Symbol aktuelle Position zu Symbol speichern

Bei Referenzierung von Symbol

- Symbolauflösung noch nicht durchführbar
- Relative Position zu Beginn des Datenstroms
- Speichern der Referenzposition und des Symbols

Allozieren des Speichers

Mittel von OCaml bieten nicht genügend Hardwarenähe

- Allozieren von Speicher erfolgt in `C`
- Benötigter Speicher wird mit `mmap` geholt
- Für alle relativen Position können nun Absolute bestimmt werden
- Auflösen der Symbole $\Rightarrow$ Relocation

Relative und direkte Relocation

Auflösen der Symbole zu Speicheradressen

- Relative Relocation
  - In allen Sprunginstruktionen verwendet
  - Differenz zwischen Referenzierung und Symboldefinition

Relative und direkte Relocation

Auflösen der Symbole zu Speicheradressen

- Relative Relocation
  - In allen Sprunginstruktionen verwendet
  - Differenz zwischen Referenzierung und Symboldefinition
- Direkte Relocation
  - Bei Zugriffen auf Datenspeicher verwendet
  - Absolute Speicheradresse

Ausführen

Alle Opcodes wurden bisher in OCaml-Strings geschrieben

- Schreiben der Strings in den vorbereiteten Speicher
- Setzen den Zugriffsrechte: Lesen, Schreiben und Ausführen

Ausführen

Alle Opcodes wurden bisher in OCaml-Strings geschrieben

- Schreiben der Strings in den vorbereiteten Speicher
- Setzen den Zugriffsrechte: Lesen, Schreiben und Ausführen

Zum Schluss

- Registrieren der Daten und Text Sektionen in OCaml-Runtime
- Übergabe des Einsprungpunkts an OCaml-Runtime
- $\Rightarrow$ Ausführen

Übersicht

1 Einleitung

2 Linear-Scan-Algorithms

3 Die native OCaml-Toplevel

4 Performance

5 Fazit

Performance

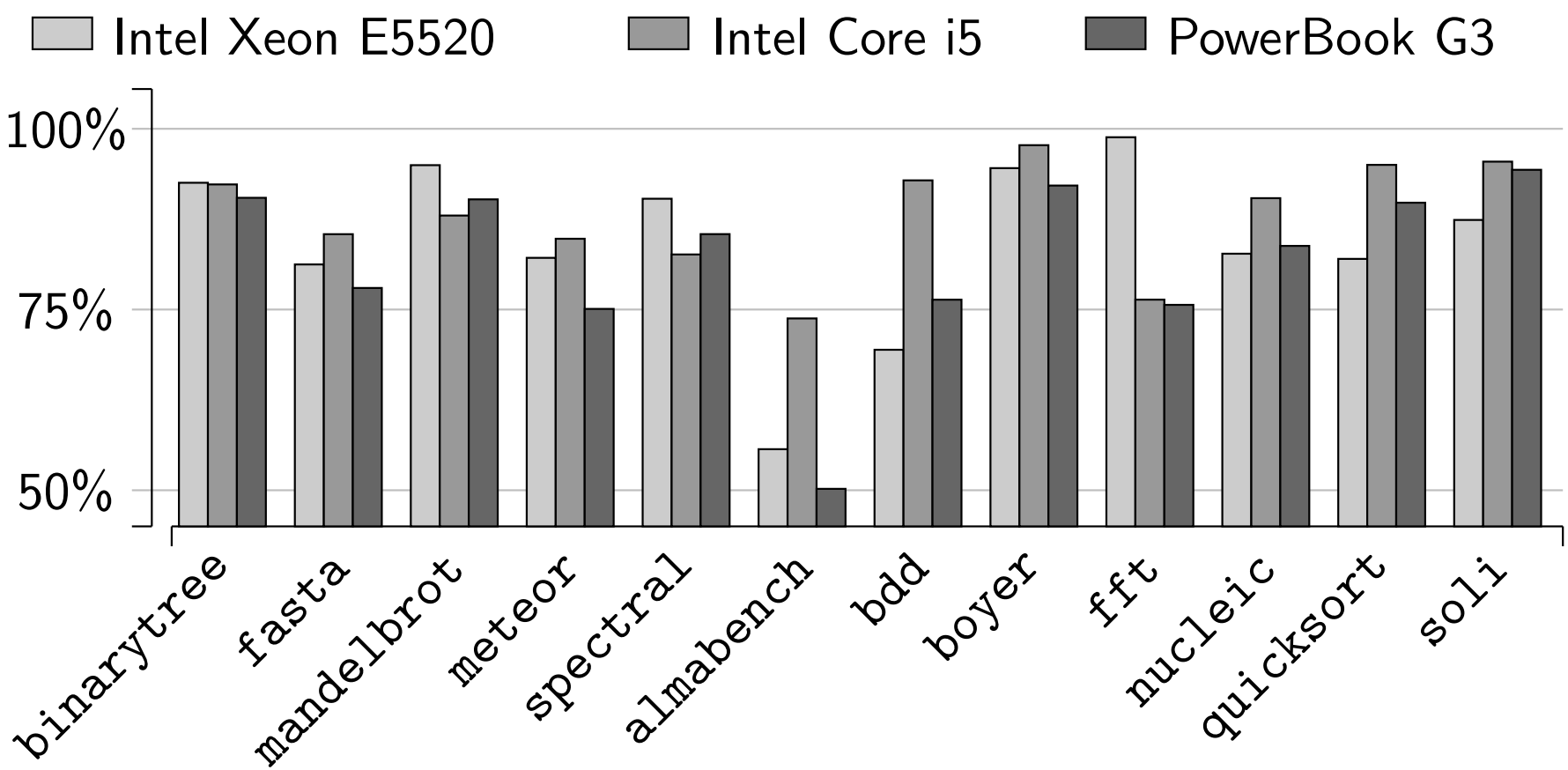
Untersuchte wurden folgende Gebiete

- Compiletime
- Runtime
- native Toplevel

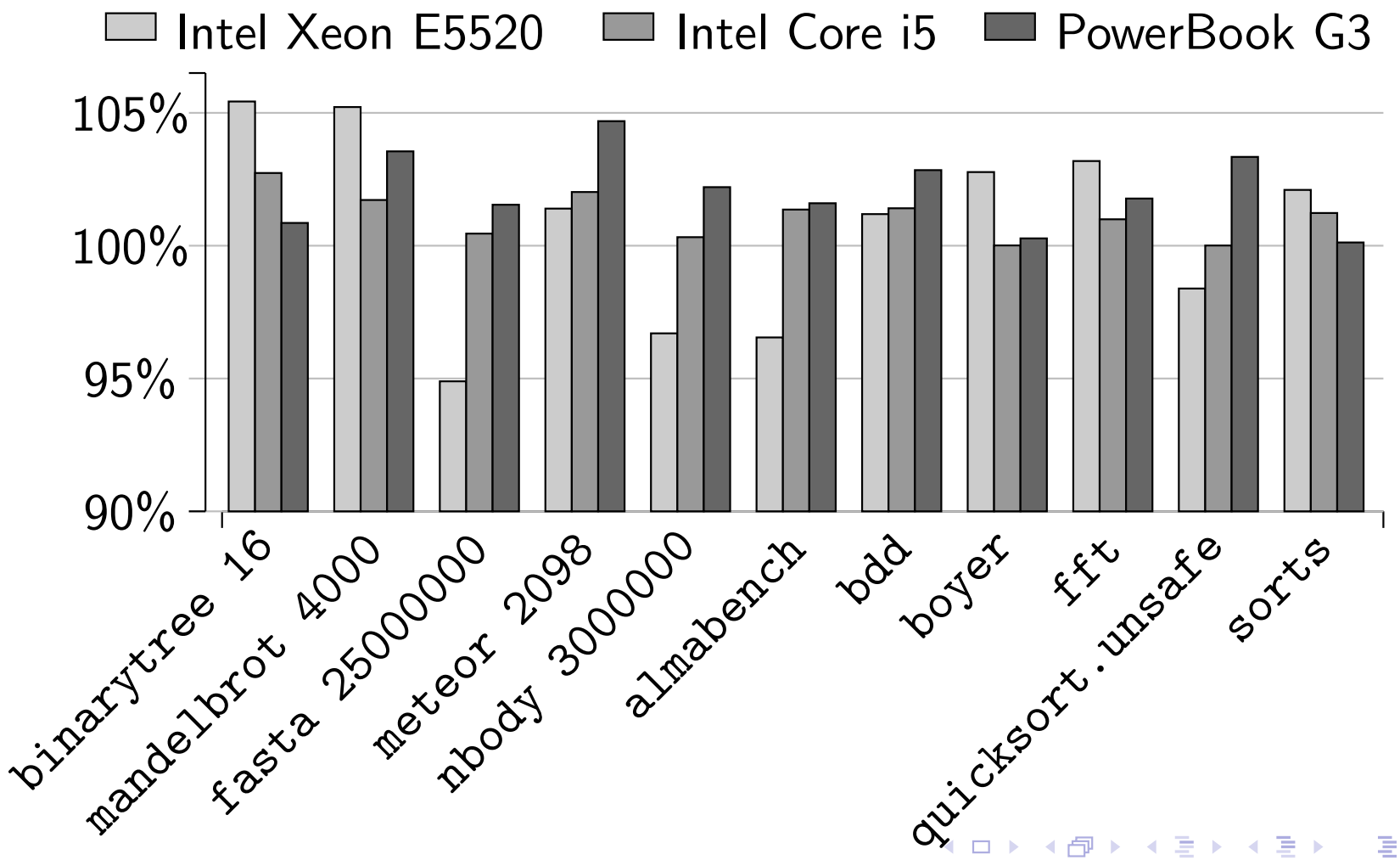
Alle Tests wurden auf folgenden Architekturen durchgeführt

- `amd64`: Intel(R) Xeon(TM) E5520 2.26GHz CPUs
- `i386`: Intel(R) Core(TM) i5-2300 @ 2.8 GHz CPU
- `PPC`: Macintosh PowerBook G3 PPC 740/750

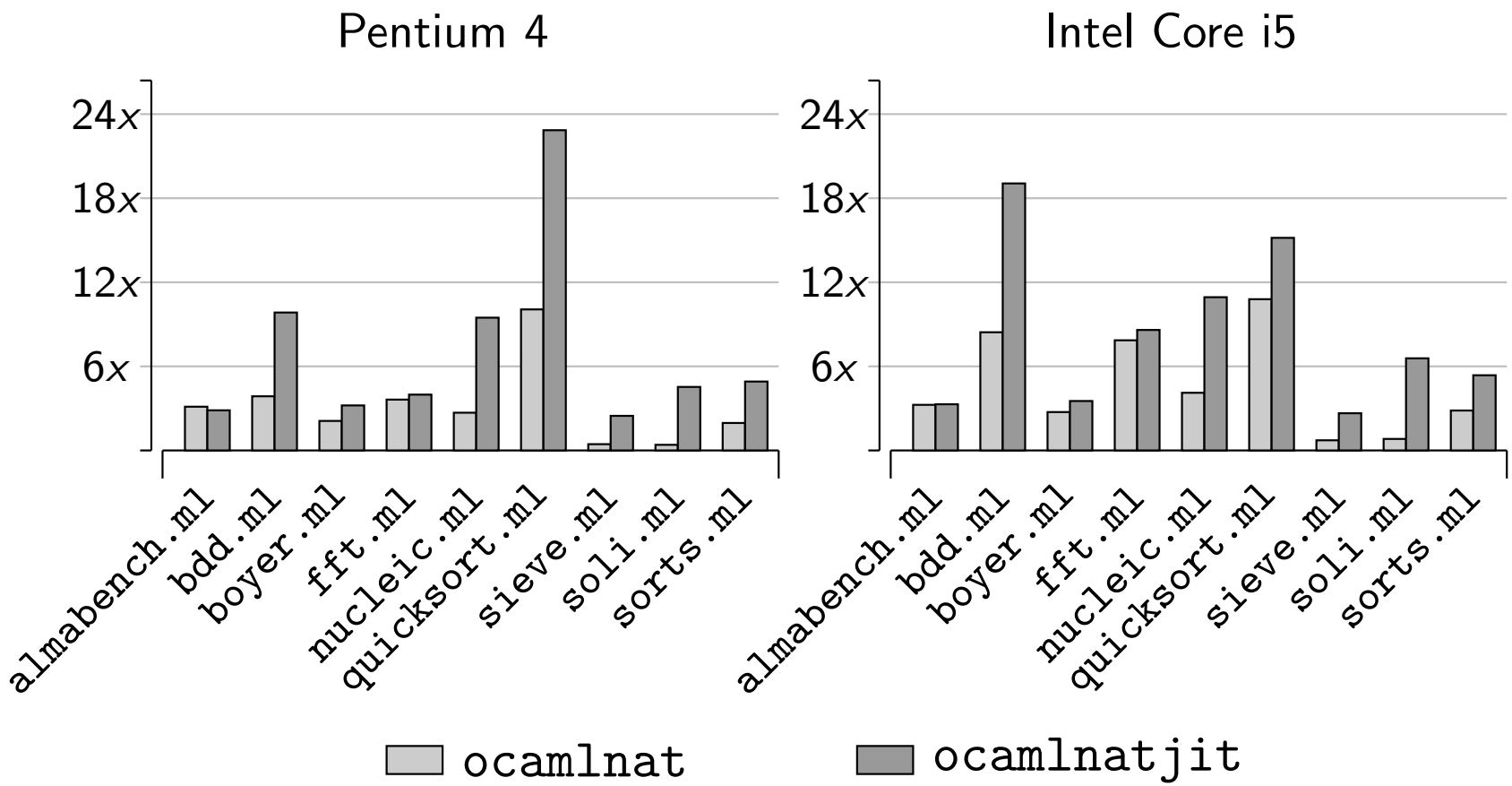
Compiletime



Runtime


Intel Xeon E5520
Intel Core i5
PowerBook G3
105%
100%
95%
90%
binarytree 16
mandelbrot 4000
fasta 25000000
meteor 2098
nbody 3000000
almabench
bdd
boyer
fft
quicksort.unsafe
sorts

Native OCaml-Toplevel



Übersicht

1 Einleitung

2 Linear-Scan-Algorithms

3 Die native OCaml-Toplevel

4 Performance

5 Fazit

Fazit

Linear-Scan-Algorithmus

- Compiletime in allen Fällen schneller als mit Graph-Coloring
- Runtime lässt sich mit der des Graph-Coloring vergleichen

Fazit

Linear-Scan-Algorithmus

- Compiletime in allen Fällen schneller als mit Graph-Coloring
- Runtime lässt sich mit der des Graph-Coloring vergleichen

Native OCaml-Toplevel

- Erwartungen weit übertroffen
- Portierung auf *amd64* gab Messungen bis zu 100facher Geschwindigkeit auf *iMac*

Vielen Dank für Ihre Aufmerksamkeit