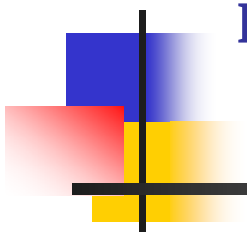
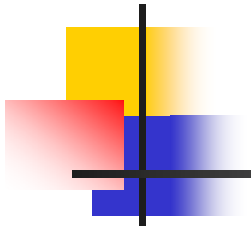


Erfahrungen bei der Auswahl von maschinenunabhängigen Optimierungen im Bereich des mobilen Codes

Wolfram Amme

Friedrich-Schiller-Universität, Jena, Deutschland

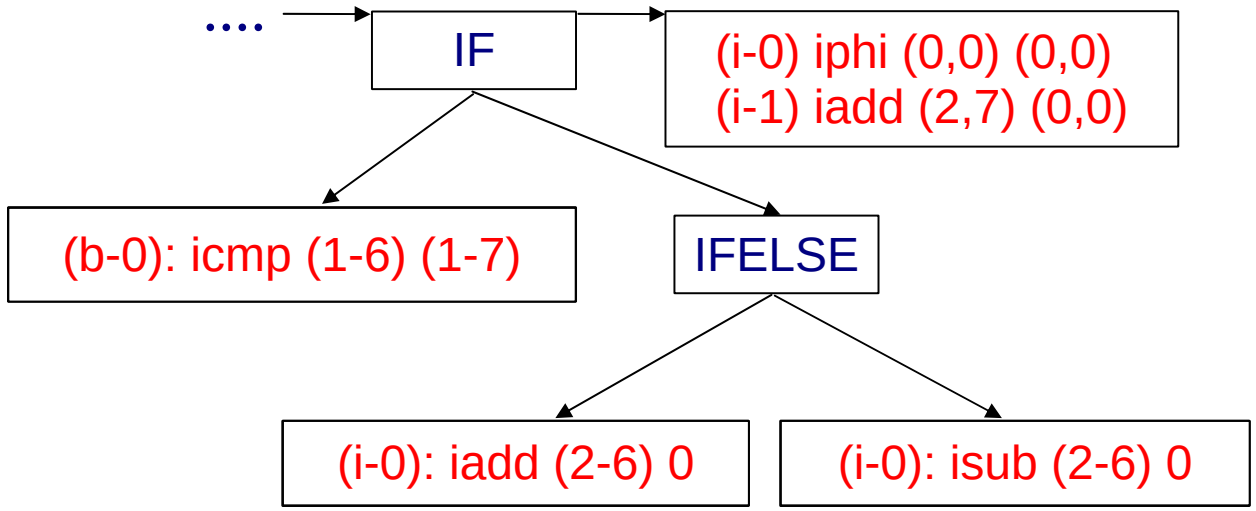


SafeTSA-Projekt: Einführung

- Projekt-Inhalt: Erstellung einer sicheren und effizienten Zwischencoderepräsentation für mobilen Code
- SafeTSA-Format: ein auf der SSA-Form basierendes Zwischencodeformat
  - gewährleistet die Typ- und Referenzsicherheit per Konstruktion
  - erlaubt die Durchführung von Optimierungen bereits bei der Erstellung des Zwischencodeformats
  - unnötige Nullreferenz- und Indexüberprüfungen können bereits auf der Produzentenseite entfernt werden
  - kann nach dem Laden auf der Konsumentenseite ohne weitere Transformation direkt vom JIT-Compiler verarbeitet werden
- SafeTSA ist als Eingabeformat für JIT-Compiler wesentlich besser geeignet als Java Bytecode

SafeTSA-Darstellung: Beispiel

```
....
if  (i <= j)
    i = i + 1;
else
    i = i - 1;
j = j + i;
```

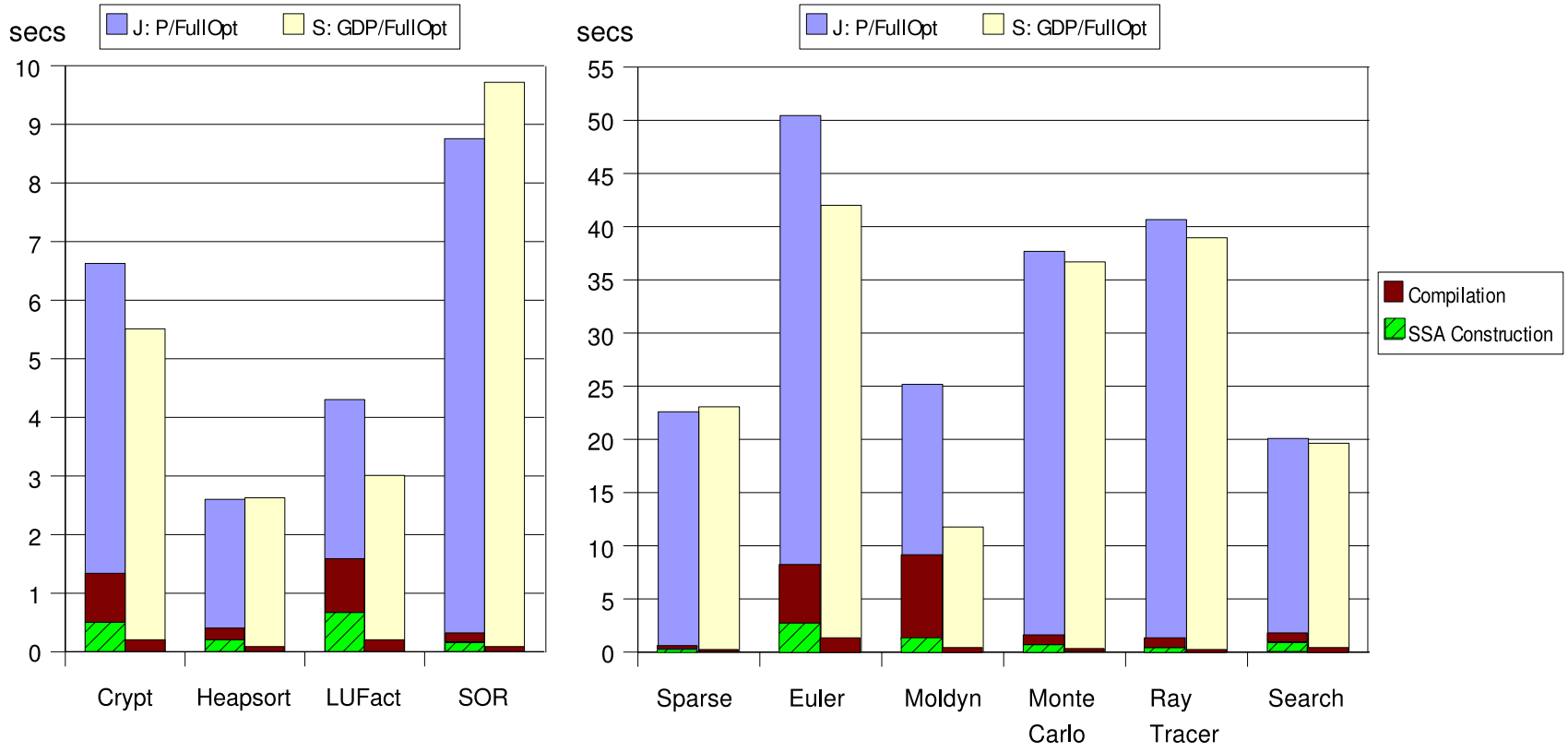


SafeTSA ist ein baumartiges Zwischencodeformat
- **abstrakter Syntaxbaum + Symboltabelle**
- **kodierte SSA-Form-Instruktionen in den Basisblöcken**

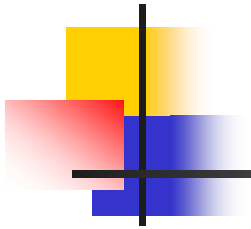
SafeTSA: Implementierung

- **eine vollständige Produzentenseite die Java-Programme in SafeTSA-Dateien übersetzt**
  - Erweiterung von ***Pizza´s Java Compiler***
  - optional: *maschinenunabhängige Optimierungen*,sowie *verifizierbare Programmannotationen*
- **Konsumentenseite wurde in Form einer Erweiterung von *IBM´s JikesRVM* realisiert**
  - Ergebnis ist eine JVM mit der eine JIT-Übersetzung sowohl für Java Bytecode- als auch SafeTSA-Programme ausgeführt werden kann
  - Zielarchitekturen: PowerPC- und IA32-Architektur
  - Erweiterung: *optimierender JIT-Compiler der SafeTSA-Darstellung in das interne Format der JikesRVM überführt*

Laufzeitverhalten: SafeTSA versus Java-Bytecode



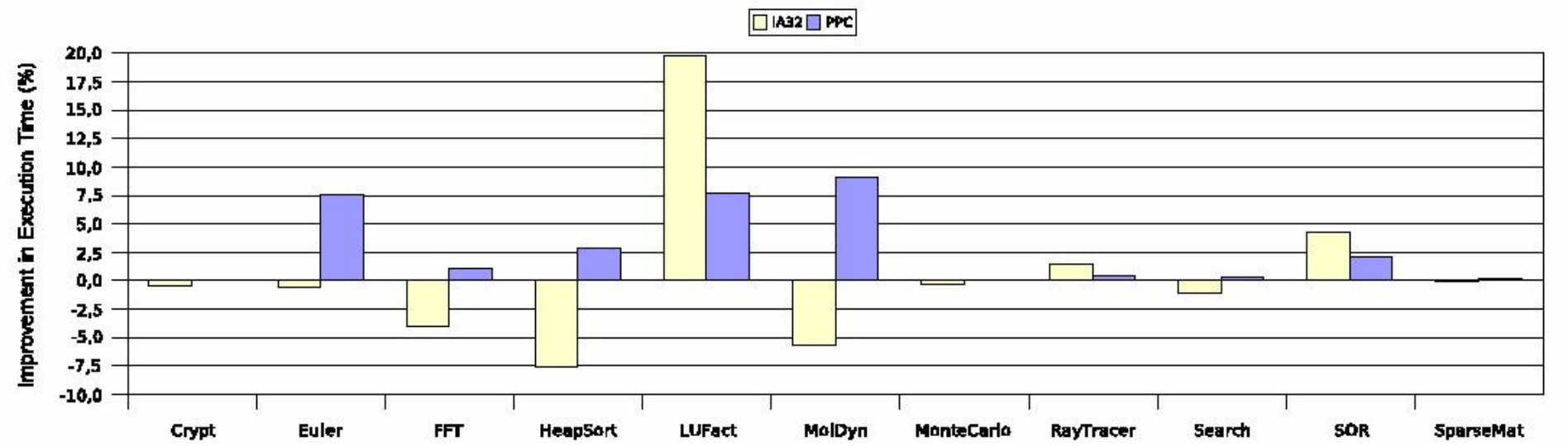
Festlegen maschinenunabhängiger Optimierungen



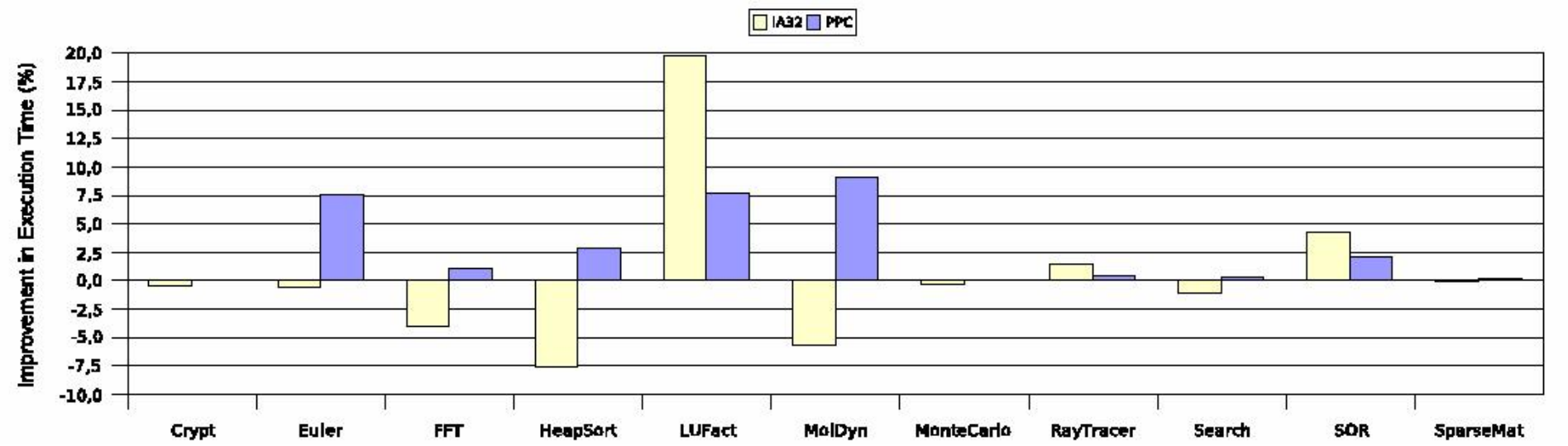
- auf der Produzenten implementierte Optimierungen
  - Konstantenfaltung und -weitergabe, Entfernen von nutzlosen Code
  - Entfernen gemeinsamer Teilausdrücke, Entfernen überflüssiger Ladebefehlen, Global Code Motion, Global Value Numbering
- Ausführung von optimierten und unoptimierten Java-Grande-Benchmark-Programmen auf den verschiedenen Architekturen
  - PowerPC: CISC-Architektur, 32 (allgemeine) Register
  - IA32: RISC-Architektur, 8 allgemeine und 8 Floating-Point-Register
- Beobachtung: für alle Optimierungen existiert zumindest ein Programm, das auf zumindest einer Architektur zur Laufzeitverschechterung führt
  - globale Unstrukturierungen erschweren eine gute Registerallokierung
  - Änderung von Programmen kann Cache-Verhalten negativ beeinflussen

Trotzdem: eine ausgewählte Verwendung solcher Optimierungen kann sinnvoll sein

Entfernen gemeinsamer Teilausdrücke: über Basisblöcke hinweg



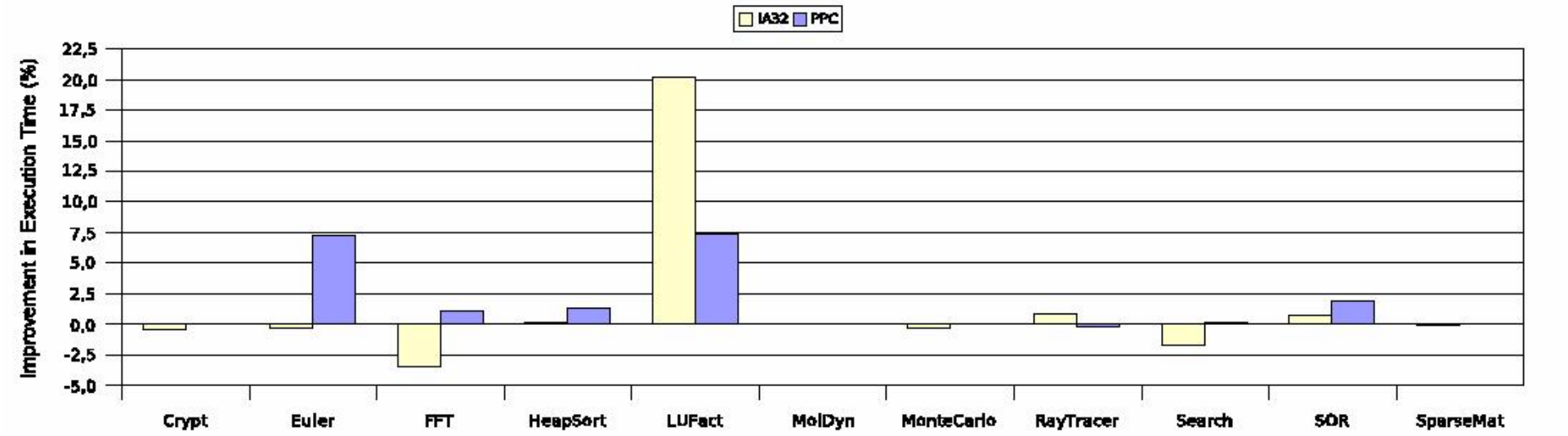
Entfernen gemeinsamer Teilausdrücke: über Basisblöcke hinweg



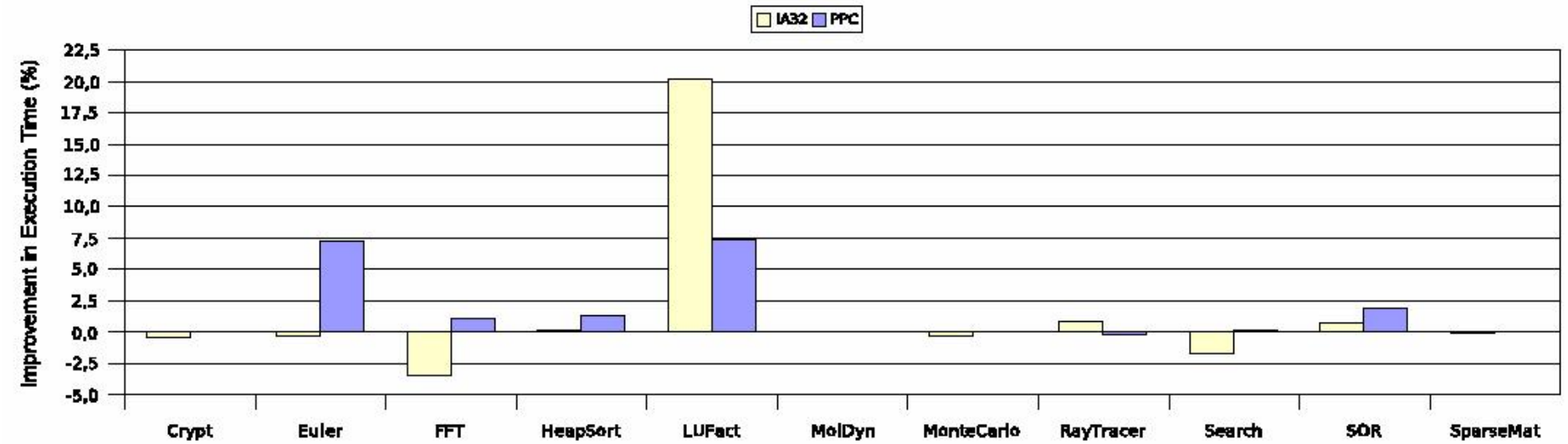
Ursachen der Laufzeitverschlechterungen

- vergrösserte Lebensbereiche führen zur vermehrten Speicherung von Registerinhalten
- zusätzliches Problem: JikesRVMs Registerallokierer weist bevorzugt Variablen mit kurzen Lebenbereichen die Register zu

Entfernen gemeinsamer Teilausdrücke: innerhalb von Basisblöcken

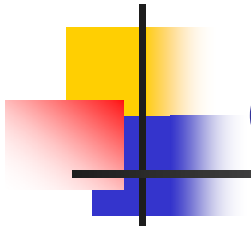


Entfernen gemeinsamer Teilausdrücke: innerhalb von Basisblöcken



Gründe für die Laufzeitverbesserungen

- Eliminierung von Indexüberprüfungen
- Eliminierung von Nullreferenzüberprüfungen spielte keine Rolle

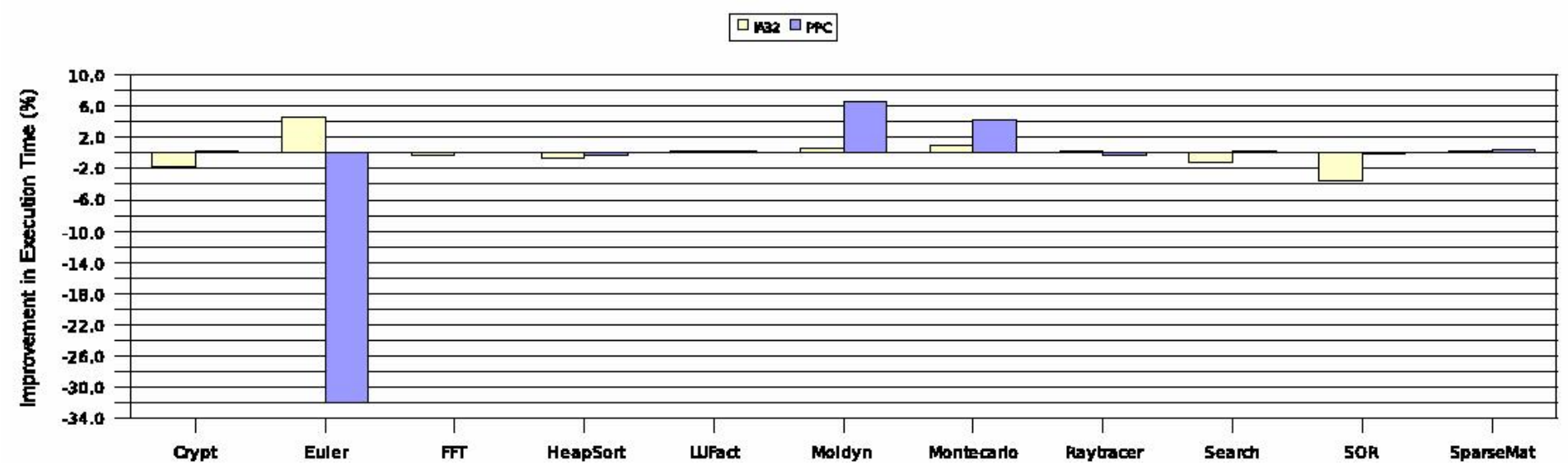


Global Code Motion

Instruktionen werden an geeignetere Programmstellen verschoben

- bevorzugt aus Schleifen heraus an weniger häufig ausgeführte Programmstellen
- ist dies nicht möglich, werden die Instruktionen derart positioniert, dass die Lebenbereiche minimiert werden

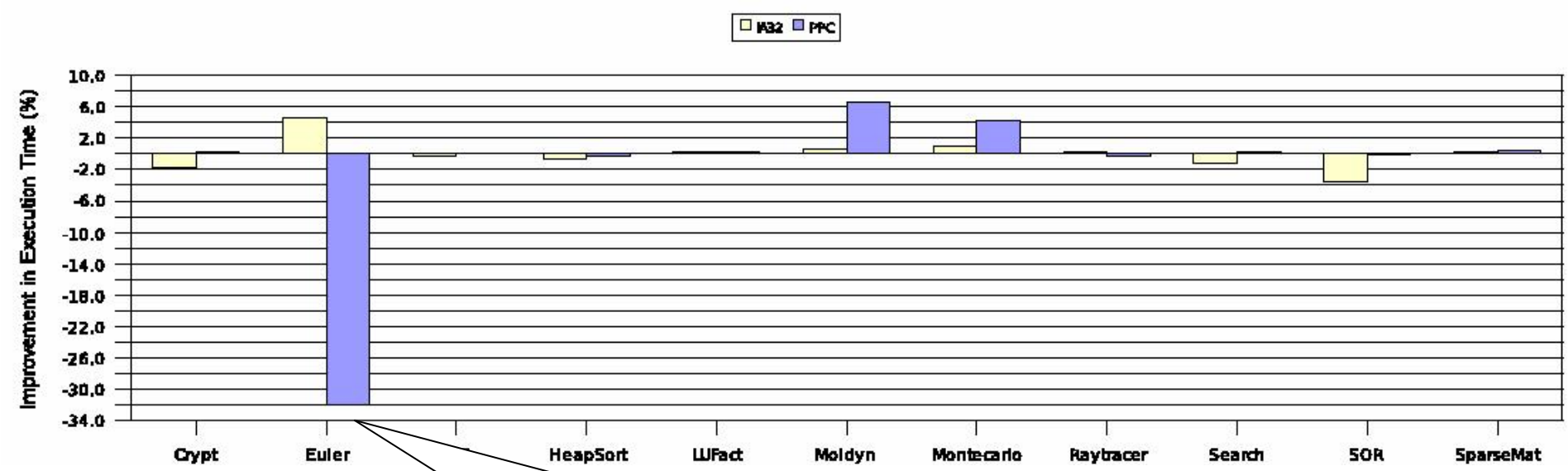
Global Code Motion



Instruktionen werden an geeignetere Programmstellen verschoben

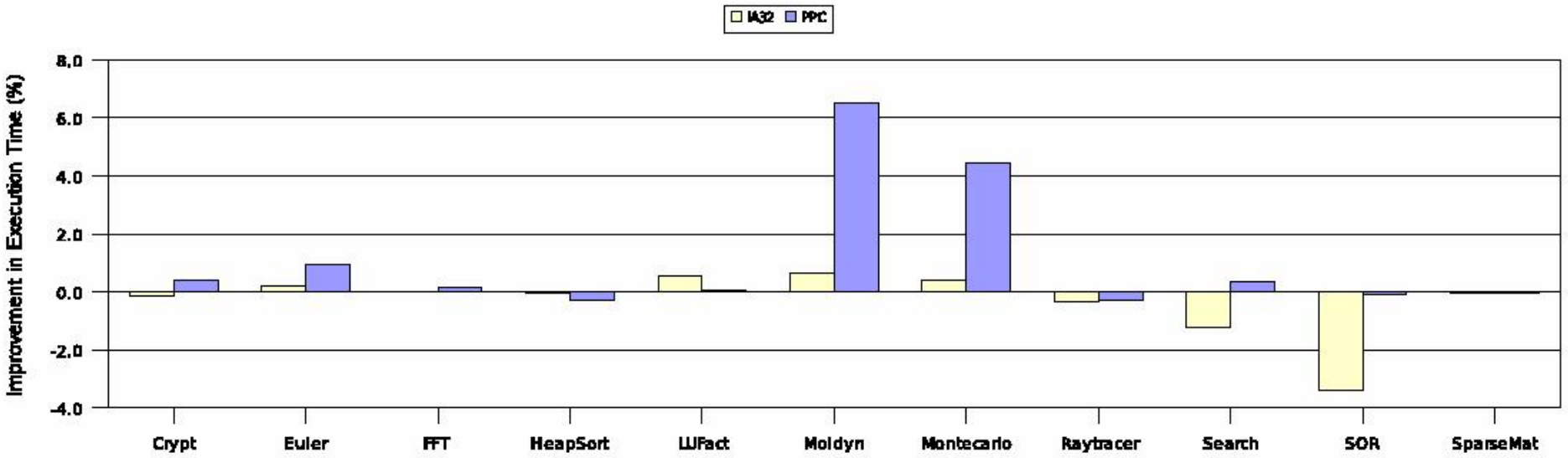
- bevorzugt aus Schleifen heraus an weniger häufig ausgeführte Programmstellen
- ist dies nicht möglich, werden die Instruktionen derart positioniert, dass die Lebenbereiche minimiert werden

Global Code Motion



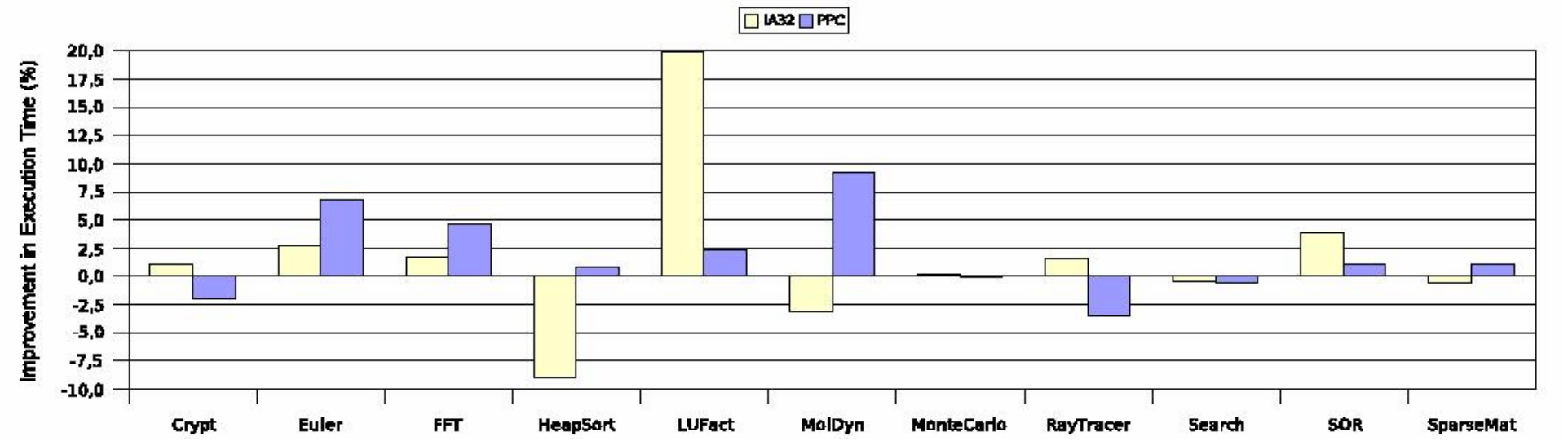
- 89 Instruktionen werden aus Schleifen herausgeschoben
- davon 22 arith. Ausdrücke i+1 mit denen innerhalb der Schleife auf Reihungen zugegriffen wird

Global Code Motion
Eingeschränkte Version



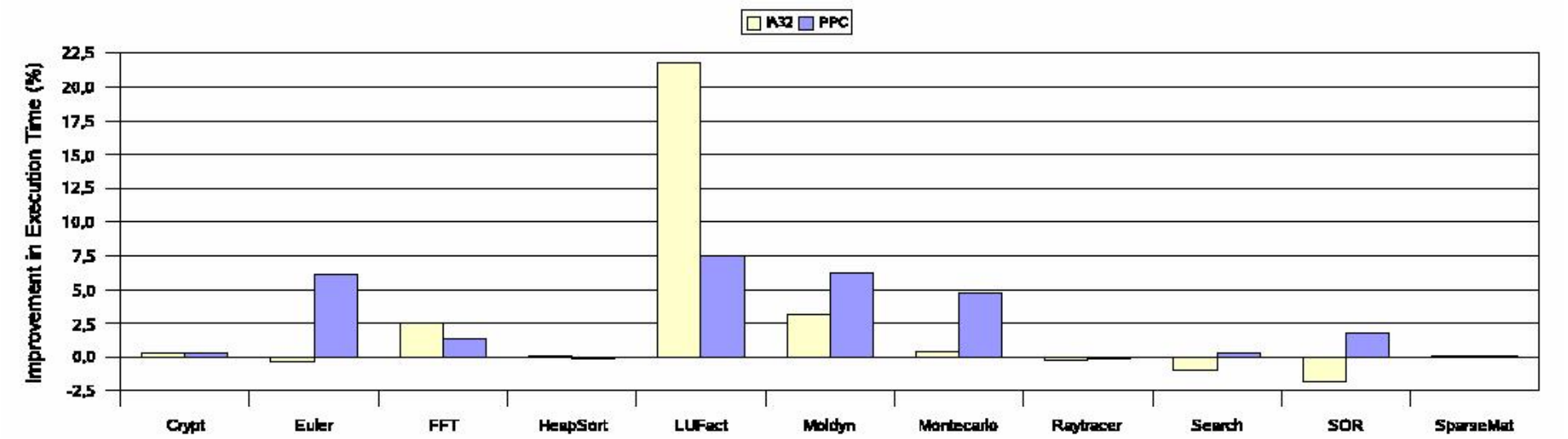
Auf das Verschieben von schleifeninvarianten Ausdrücken wird verzichtet

Global Code Motion + Global Value Numbering



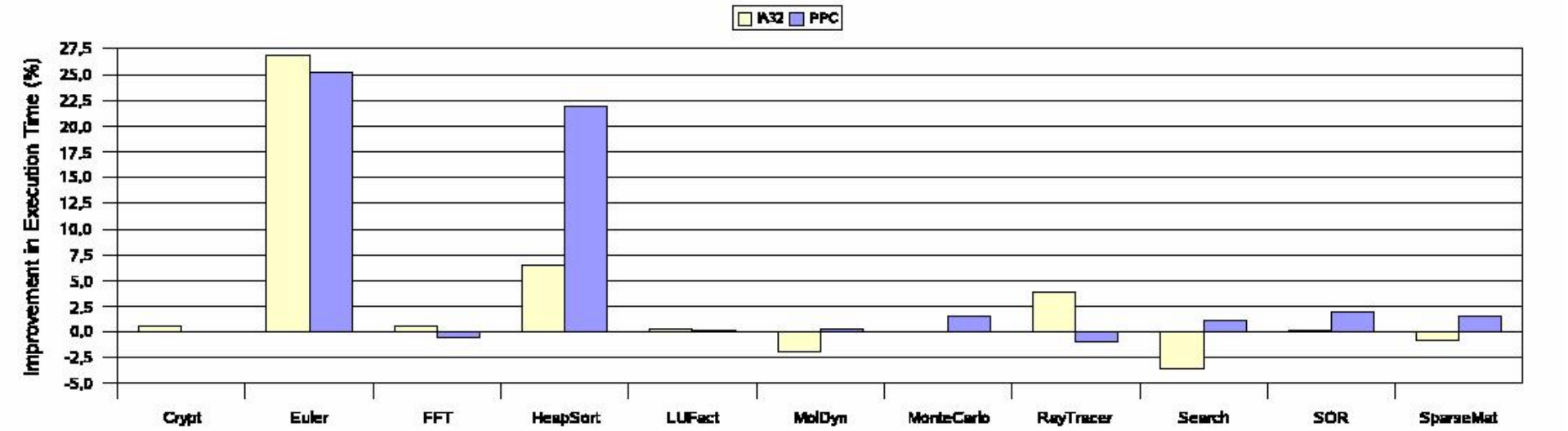
Ähnliches Verhalten wie Entfernen gemeinsamer Teilausdrücke (global)

Global Code Motion (eingeschränkt) + Entfernen gemeinsamer Teilausdrücke (lokal)



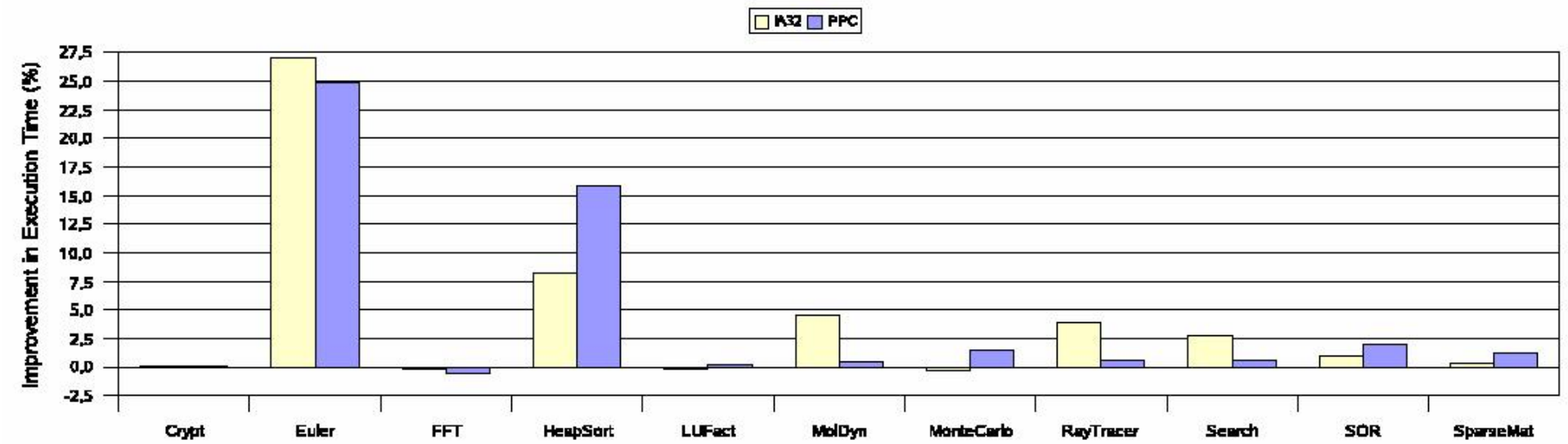
Leichte Verbesserung von Entfernen gemeinsamer Teilausdrücke (lokal)

Eliminierung von Ladebefehlen (global)



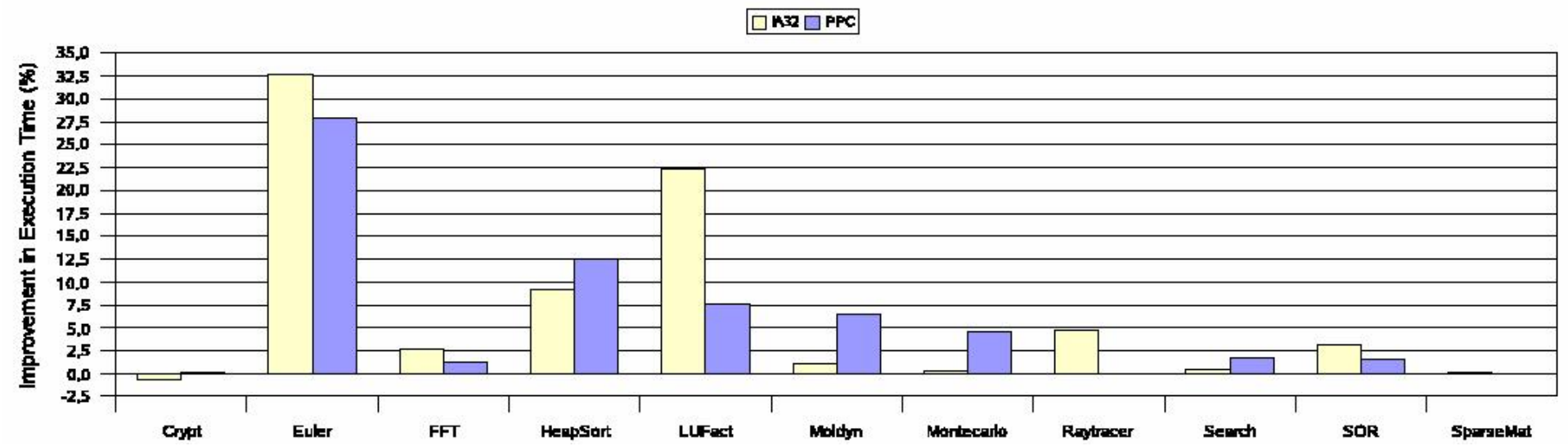
- **zielt auf Entfernen unnötiger Feld- und Reihungszugriffe**
- **sehr konservative Form des Algorithmus**
  - Speicherdefinitionen, Funktionsaufrufe und Zugriff auf synchronisierte Variablen werden als Gesamtspeicherdefinition angesehen

Eliminierung von Ladebefehlen (lokal)



- **zielt auf Entfernen unnötiger Feld- und Reihungszugriffe**
- **sehr konservative Form des Algorithmus**
  - Speicherdefinitionen, Funktionsaufrufe und Zugriff auf synchronisierte Variablen werden als Gesamtspeicherdefinition angesehen

Laufzeitverbesserung: Eingebundene maschinenunabhängige Optimierungen



Integration maschinenunabhängiger Optimierungen

- alle nicht auf globalen Umstrukturierungen basierenden Optimierungen
  Konstantenfortpflanzung und -weitergabe, Entfernen von nutzlosen Code, Entfernen gemeinsamer Teilausdrücke und Ladebefehle (jeweils lokal)
- sowie eingeschränkte Version der Global Code Motion