Generating Compiler Optimisations using Rosser

Richard Warburton, Warwick University Computer Science Department

July 1, 2008

Motivation

Why implement an optimisation generator?

- TRANS *ideally* supports experimentation

Motivation

Why implement an optimisation generator?

- TRANS *ideally* supports experimentation
- Existing optimisation specifications often difficult or impossible to compare formally

Motivation

Why implement an optimisation generator?

- TRANS *ideally* supports experimentation
- Existing optimisation specifications often difficult or impossible to compare formally
- Questions over efficiency of generated optimisations.

Overview

- `soot` system used as basis

Overview

- `soot` system used as basis
- Specifications refined

Overview

- `soot` system used as basis
- Specifications refined
  - rewrites become pattern matches and replacement

Overview

- `soot` system used as basis
- Specifications refined
  - rewrites become pattern matches and replacement
  - temporal operators reduced

Overview

- `soot` system used as basis
- Specifications refined
  - rewrites become pattern matches and replacement
  - temporal operators reduced
- `transform` method corresponds to action

Overview

- `soot` system used as basis
- Specifications refined
  - rewrites become pattern matches and replacement
  - temporal operators reduced
- `transform` method corresponds to action
- `condition` method corresponds to side condition

Dimple

- Represents certain aspects of the program using Binary Decision Diagrams

Dimple

- Represents certain aspects of the program using Binary Decision Diagrams
- Suitable for CTL based dataflow analysis

Dimple

- Represents certain aspects of the program using Binary Decision Diagrams
- Suitable for CTL based dataflow analysis
- Uses JEDD as output language

Relations

- `<from,to,edgetype> Edges`

Relations

- `<from,to,edgetype> Edges`
- `Assign, IfStmt, ReturnsValues`

Relations

- `<from,to,edgetype> Edges`
- `Assign, IfStmt, ReturnsValues`
- `<eq, l, r, op> Expr`

Side Conditions

```
comp-s true          =   [res = ⊤]
comp-s False         =   [res = ⊥]
comp-s conlit(v)     =   [temp1 = ⊤, res = temp1{v} >< meth.Conlit{c}]
comp-s varlit(v)     =   [temp1 = ⊤, res = temp1{v} >< meth.Varlit{v}]
comp-s ¬ φ           =   comp-s φ @ [res = ⊤ - pred]
comp-s n |= φ        =   comp-s φ @ [res = (at =>) pred{n,at} <> pred{at,n}]
comp-s φ ∧ ψ         =   comp-s φ @ comp-s ψ @ [res = pred1 & pred2]
comp-s φ ∨ ψ         =   comp-s φ @ comp-s ψ @ [res = pred1 | pred2]
```

Node Conditions

```
comp-t EX φ   =   comp-t φ @
                  [temp1 = (et=>)meth.Edges,
                  res = (to=>at) pred{at} <> temp1{from} ]
```

Until

```
temp1 = (et=>) meth.Edges;
acc = pred2;
do {
  prev = acc;
  temp2 = (from=>) pred1{at} <> temp1{to};
  acc |= pred2 & temp2
} while(prev != acc);
res = acc
```

What is Performance?

Aim: compare *Performance* with hand-written optimisations.

What is Performance?

Aim: compare *Performance* with hand-written optimisations.

Definition

The Effectiveness of an optimization is a measure of the extent to which it improves the performance of the program being optimized.

What is Performance?

Aim: compare *Performance* with hand-written optimisations.

Definition

The Effectiveness of an optimization is a measure of the extent to which it improves the performance of the program being optimized.

Definition

The Efficiency of an optimization is a measure of the performance properties of the optimization.

What is Performance?

Aim: compare *Performance* with hand-written optimisations.

Definition

The Effectiveness of an optimization is a measure of the extent to which it improves the performance of the program being optimized.

Definition

The Efficiency of an optimization is a measure of the performance properties of the optimization.

Applied: Dead Code Elimination, Common Subexpression elimination, Lazy Code Motion

Scimark 2 Benchmark

- Part of Spec JVM 2008

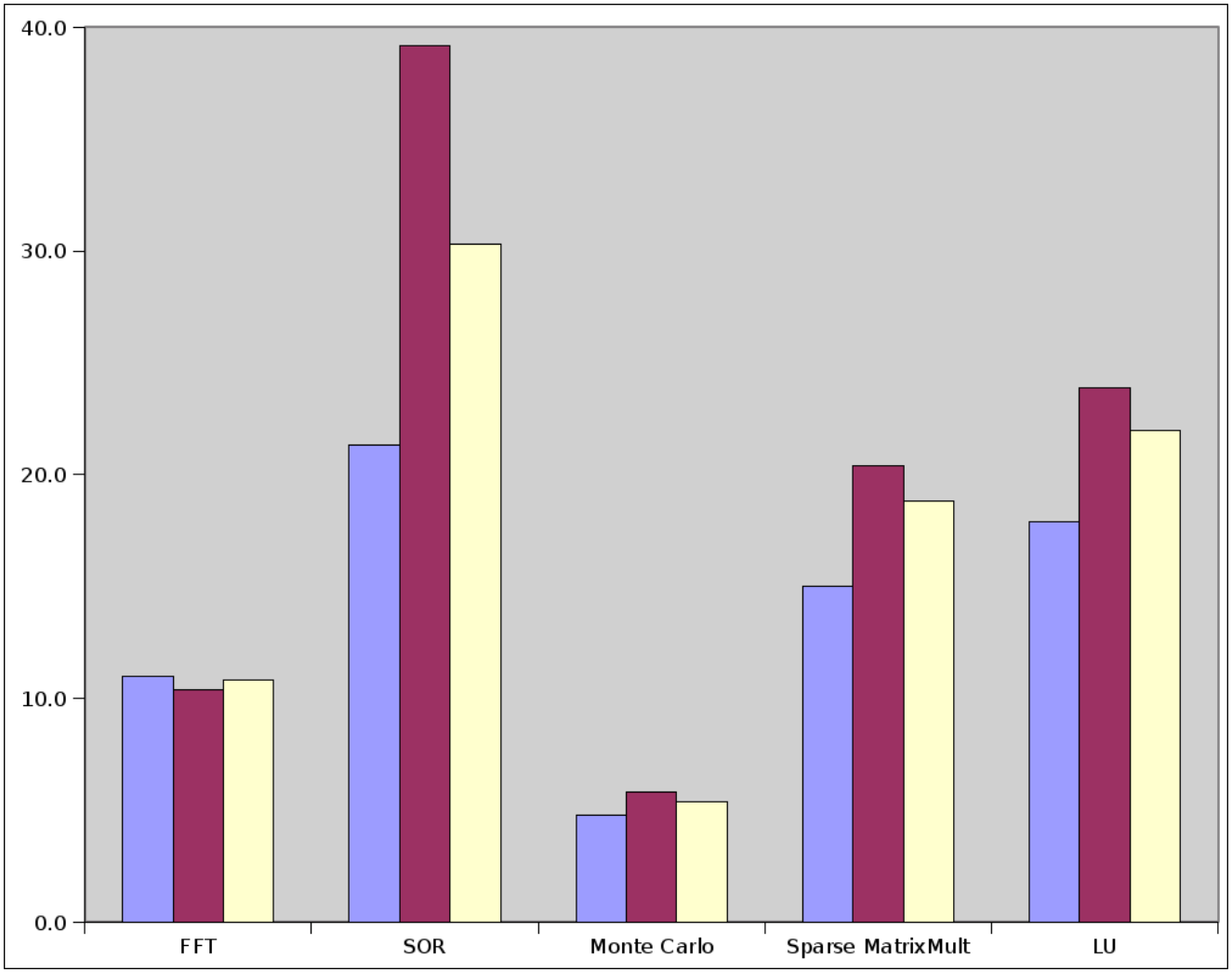
Scimark 2 Benchmark

- Part of Spec JVM 2008
- Commonly Used Java benchmark

Scimark 2 Benchmark

- Part of Spec JVM 2008
- Commonly Used Java benchmark
- Performance of Scientific Application Kernels.

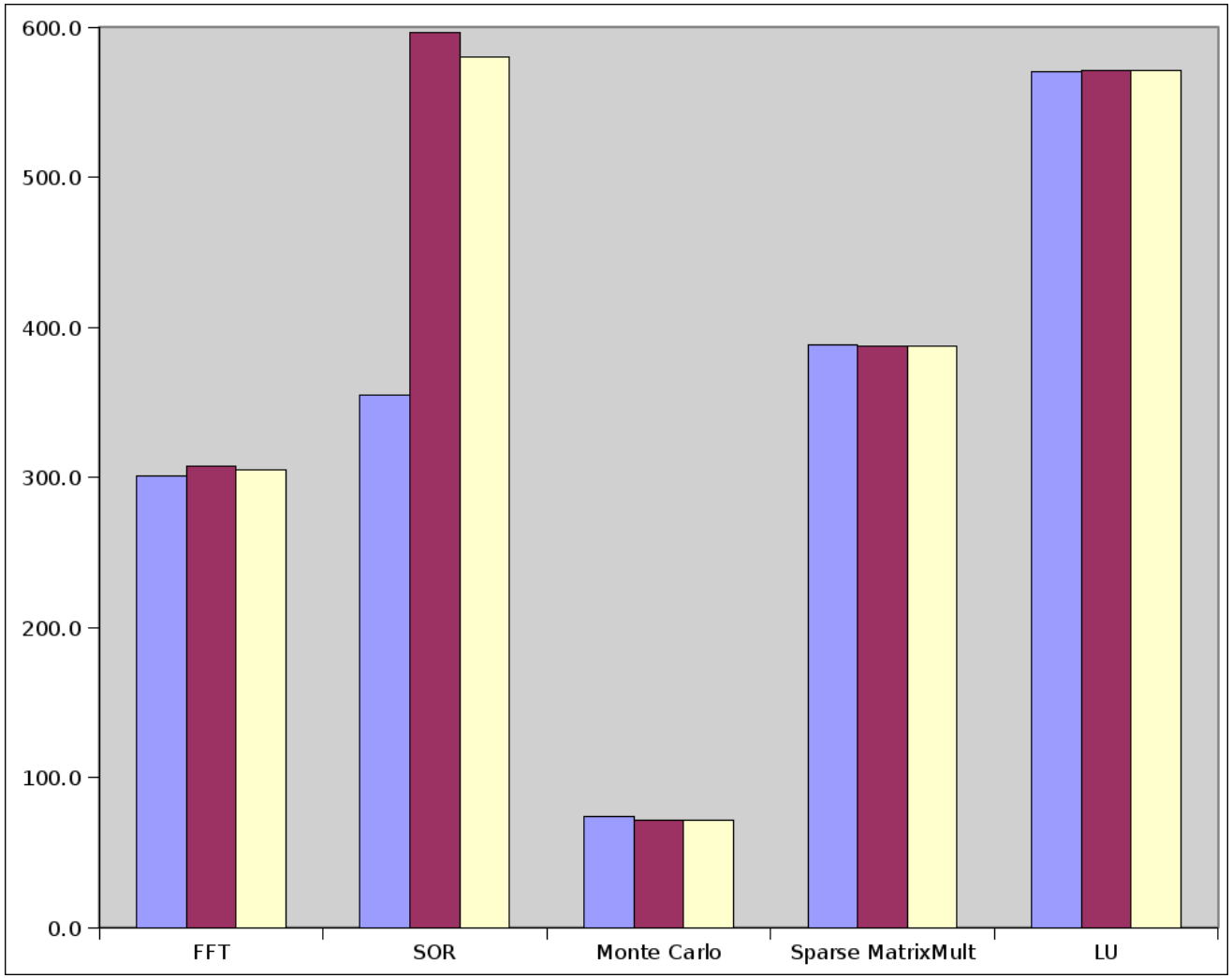
SableVM



The End

Any~~one for Pub~~ Questions?

SUN JVM


600.0
500.0
400.0
300.0
200.0
100.0
0.0
FFT
SOR
Monte Carlo
Sparse MatrixMult
LU

Efficiency

- Soot: 15 seconds for scimark

Efficiency

- Soot: 15 seconds for scimark
- Rosser: 270 seconds for scimark

Efficiency

- Soot: 15 seconds for scimark
- Rosser: 270 seconds for scimark
- but 131/133 Methods optimised in 30 seconds

Efficiency

- Soot: 15 seconds for scimark
- Rosser: 270 seconds for scimark
- but 131/133 Methods optimised in 30 seconds
- ∴ Pathological cases in 2 remaining methods

Efficiency

- Soot: 15 seconds for scimark
- Rosser: 270 seconds for scimark
- but 131/133 Methods optimised in 30 seconds
- ∴ Pathological cases in 2 remaining methods
- Usually: 2x slower than hand-coded

Efficiency

- Soot: 15 seconds for scimark
- Rosser: 270 seconds for scimark
- but 131/133 Methods optimised in 30 seconds
- ∴ Pathological cases in 2 remaining methods
- Usually: 2x slower than hand-coded
- Pathologically: $>$ 1000x slower