Decoupled Approaches to Register and Software-Controlled Memory Allocations

Boubacar Diouf$^{1,2}$

Directeur de thèse: Albert Cohen

$^1$Paris-Sud University

$^2$INRIA

15th December 2011 / PhD defense
The memory hierarchy

<table>
<thead>
<tr>
<th>Usual size</th>
<th>Usual access time</th>
</tr>
</thead>
<tbody>
<tr>
<td>More than 2 GB</td>
<td>3 - 15 ms</td>
</tr>
<tr>
<td>Virtual memory (disk)</td>
<td></td>
</tr>
<tr>
<td>256 MB - 2 GB</td>
<td>100 - 150 ns</td>
</tr>
<tr>
<td>Physical memory</td>
<td></td>
</tr>
<tr>
<td>256 KB - 4 MB</td>
<td>40 - 60 ns</td>
</tr>
<tr>
<td>2nd-level cache or LM</td>
<td></td>
</tr>
<tr>
<td>16 - 64 KB</td>
<td>5 - 10 ns</td>
</tr>
<tr>
<td>1st-level cache or LM</td>
<td></td>
</tr>
<tr>
<td>32 words</td>
<td>1 ns</td>
</tr>
<tr>
<td>Registers (processor)</td>
<td></td>
</tr>
</tbody>
</table>
Two software-controlled kind of memories

Registers
- The registers are fast and limited: all the values cannot reside in registers
- The use of registers must be optimized: register allocation

Local memories
- Many processors have Local Memories (DSP, GPUs, Cell SPU, many embedded processors)
- Fast, predictable, power efficient, smaller area cost
- Allocate the arrays to the local memory: Local memory allocation
Contributions of this thesis

Decoupled approach to register allocation

• split register allocation
• spill minimization problem

Decoupled approach to LM allocation

• experimental validation
• theoretical basis

The link between LM and register allocation as reported by Fabri [Fab’79]

Reconciling the two optimization problems

• A clustering allocator that works for both register allocation and LM allocation
Outline

Introduction

Register Allocation
- Register allocation techniques
- Split Register Allocation
- Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
- Decoupled allocation for linearized programs
- The (local memory) spill minimization problem

Conclusion
The goal of register allocation

Program example

\begin{align*}
b & := a \times a \\
c & := a + a \\
a & := b - a \\
\text{if } (a < 10) & \\
\quad d & := a + 5 \\
\quad e & := c \times a \\
\quad d & := e - d \\
\text{else} & \\
\quad f & := a \\
\quad e & := f \times f \\
\quad d & := e - f \\
d & := d + 1
\end{align*}

\begin{align*}
@b & := r_1 \times r_1 \\
r_2 & := r_1 + r_1 \\
r_1 & := @b - r_1 \\
\text{if } (r_1 < 10) & \\
\quad @d & := r_1 + 5 \\
\quad r_2 & := r_2 \times r_1 \\
\quad r_1 & := r_2 - @d \\
\text{else} & \\
\quad r_2 & := r_1 \\
\quad r_1 & := r_2 \times r_2 \\
\quad r_1 & := r_1 - r_2 \\
r_1 & := r_1 + 1
\end{align*}
The goal of register allocation

Program example

\[
\begin{align*}
b & := a \times a \\
c & := a + a \\
a & := b - a \\
\text{if} \ (a < 10) \\
\quad d & := a + 5 \\
\quad e & := c \times a \\
\quad d & := e - d \\
\text{else} \\
\quad f & := a \\
\quad e & := f \times f \\
\quad d & := e - f \\
\quad d & := d + 1
\end{align*}
\]

\[
\begin{align*}
@b & := r1 \times r1 \\
r2 & := r1 + r1 \\
r1 & := @b - r1 \\
\text{if} \ (r1 < 10) \\
\quad @d & := r1 + 5 \\
\quad r2 & := r2 \times r1 \\
\quad r1 & := r2 - @d \\
\text{else} \\
\quad r2 & := r1 \\
\quad r1 & := r2 \times r2 \\
\quad r1 & := r1 - r2 \\
\quad r1 & := r1 + 1
\end{align*}
\]
The goal of register allocation

Program example

\[
\begin{align*}
b & := a \times a \\
c & := a + a \\
a & := b - a \\
\text{if} \ (a < 10) \\
\quad d & := a + 5 \\
\quad e & := c \times a \\
\quad d & := e - d \\
\text{else} \\
\quad f & := a \\
\quad e & := f \times f \\
\quad d & := e - f \\
\quad d & := d +1
\end{align*}
\]

\[
\begin{align*}
@b & := r1 \times r1 \\
r2 & := r1 + r1 \\
r1 & := @b - r1 \\
\text{if} \ (r1 < 10) \\
\quad @d & := r1 + 5 \\
\quad r2 & := r2 \times r1 \\
\quad r1 & := r2 - @d \\
\text{else} \\
\quad r1 & := r1 \\
\quad r2 & := r1 \times r1 \\
\quad r1 & := r2 - r1 \\
\quad r1 & := r1 +1
\end{align*}
\]
The goal of register allocation

Program example

```
b := a * a
c := a + a
a := b - a
if (a < 10)
d := a + 5
e := c * a
d := e - d
else
  f := a
e := f * f
d := e - f
d := d +1
```

```
@b := r1 * r1
r2 := r1 + r1
r1 := @b - r1
if (r1 < 10)
  @d := r1 + 5
  r2 := r2 * r1
  r1 := r2 - @d
else
  r1 := r1
  r2 := r1 * r1
  r1 := r2 - r1
  r1 := r1 +1
```
Outline

Introduction

Register Allocation
- Register allocation techniques
  - Split Register Allocation
  - Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
- Decoupled allocation for linearized programs
- The (local memory) spill minimization problem

Conclusion
Liveness of variables, live ranges

Program example

```plaintext
b := a * a
c := a + a
a := b - a
if (a < 10)
    d := a + 5
e := c * a
d := e – d
else
    f := a
e := f * f
d := e – f
d := d +1
```
**Liveness of variables, live ranges**

Program example

\[
\begin{align*}
\text{b} & := a \ast a \\
\text{c} & := a + a \\
\text{a} & := b - a \\
\text{if } (a < 10) \\
\quad d & := a + 5 \\
\quad e & := c \ast a \\
\quad d & := e - d \\
\text{else} \\
\quad f & := a \\
\quad e & := f \ast f \\
\quad d & := e - f \\
\text{d} & := d + 1
\end{align*}
\]

CFG
Liveness of variables, live ranges

Program example

\[ b := a \times a \]
\[ c := a + a \]
\[ a := b - a \]
\[ \text{if} \ (a < 10) \]
\[ \quad d := a + 5 \]
\[ \quad e := c \times a \]
\[ \quad d := e - d \]
\[ \text{else} \]
\[ \quad f := a \]
\[ \quad e := f \times f \]
\[ \quad d := e - f \]
\[ d := d + 1 \]
Liveness of variables, live ranges

Program example

\[
\begin{align*}
    b & := a \times a \\
    c & := a + a \\
    a & := b - a \\
    \text{if } (a < 10) & \\
    \quad d & := a + 5 \\
    \quad e & := c \times a \\
    \quad d & := e - d \\
    \text{else} & \\
    \quad f & := a \\
    \quad e & := f \times f \\
    \quad d & := e - f \\
    d & := d + 1
\end{align*}
\]
Graph Coloring \cite{Chaitin'81} (the classical approach)

Program example

\begin{verbatim}
b := a * a
c := a + a
a := b - a
if (a < 10)
  d := a + 5
e := c * a
d := e - d
else
  f := a
e := f * f
d := e - f
d := d +1
\end{verbatim}

3 available registers

\begin{figure}
\centering
\includegraphics[width=\textwidth]{register_allocation.png}
\end{figure}
Graph Coloring [Chaitin’81] (the classical approach)

Program example

```plaintext
b := a * a  
c := a + a  
a := b - a  
if (a < 10)  
  d := a + 5  
e := c * a  
d := e - d  
else  
  f := a  
e := f * f  
d := e - f  
d := d + 1
```

3 available registers

[Diagram showing assignment variables and their relationships]
Graph Coloring [Chaitin’81] (the classical approach)

Program example

\[
\begin{align*}
b &:= a \times a \\
c &:= a + a \\
a &:= b - a \\
\text{if} \ (a < 10) \\
  \quad d &:= a + 5 \\
  \quad e &:= c \times a \\
  \quad d &:= e - d \\
\text{else} \\
  \quad f &:= a \\
  \quad e &:= f \times f \\
  \quad d &:= e - f \\
\end{align*}
\]

\[d := d + 1\]

3 available registers

[Diagram of a graph with nodes labeled b, c, a, e, f, and d connected by lines, indicating dependencies and register allocation.]
Graph Coloring [Chaitin’81] (the classical approach)

Program example

\[
\begin{align*}
    b &:= a \times a \\
    c &:= a + a \\
    a &:= b - a \\
    \text{if } (a < 10) \\
    &\quad d := a + 5 \\
    &\quad e := c \times a \\
    &\quad d := e - d \\
    \text{else} \\
    &\quad f := a \\
    &\quad e := f \times f \\
    &\quad d := e - f \\
    d &:= d + 1
\end{align*}
\]

3 available registers
**Graph Coloring** [Chaitin'81] (the classical approach)

Program example

```
\[ \begin{align*}
  &b := a * a \\
  &c := a + a \\
  &a := b - a \\
  &\text{if} \ (a < 10) \\
  &\quad d := a + 5 \\
  &\quad e := c * a \\
  &\quad d := e - d \\
  &\text{else} \\
  &\quad f := a \\
  &\quad e := f * f \\
  &\quad d := e - f \\
  &d := d +1
\end{align*} \]
```

3 available registers
Graph Coloring [Chaitin’81] (the classical approach)

Program example

\begin{verbatim}
  b := a * a
  c := a + a
  a := b - a
  if (a < 10)
    d := a + 5
    e := c * a
    d := e - d
  else
    f := a
    e := f * f
    d := e - f
  d := d + 1
\end{verbatim}

3 available registers

\begin{verbatim}
  ooo  
  ooo  
  ooo  
  ooo  
  ooo  
  o   
\end{verbatim}
Linear Scan [Poletto'99] (the JIT approach)

Program example

b := a * a

1: b := a * a

2: c := a + a

3: a := b - a

4: if (a < 10)

5: d := a + 5

6: e := c * a

7: d := e - d

else

8: f := a

9: e := f * f

10: d := e - f

11: d := d + 1

3 available registers
Linear Scan [Poletto'99] (the JIT approach)

Program example

\[
\begin{align*}
\text{b} & := \text{a} \times \text{a} \\
\text{c} & := \text{a} + \text{a} \\
\text{a} & := \text{b} - \text{a} \\
\text{if} \ (\text{a} < 10) & \text{ \ \ \text{d} := \text{a} + 5} \\
& \text{ \ \ \text{e} := \text{c} \times \text{a}} \\
& \text{ \ \ \text{d} := \text{e} - \text{d}} \\
\text{else} & \text{ \ \ \text{f} := \text{a}} \\
& \text{ \ \ \text{e} := \text{f} \times \text{f}} \\
& \text{ \ \ \text{d} := \text{e} - \text{f}} \\
\text{d} & := \text{d} + 1
\end{align*}
\]

3 available registers
Linear Scan [Poletto’99] (the JIT approach)

Program example

\[
\begin{align*}
  b & := a \times a \\
  c & := a + a \\
  a & := b - a \\
  \text{if} (a < 10) & \\
    d & := a + 5 \\
    e & := c \times a \\
    d & := e - d \\
  \text{else} & \\
    f & := a \\
    e & := f \times f \\
    d & := e - f \\
  d & := d + 1
\end{align*}
\]

3 available registers
**Linear Scan** [Poletto’99] (the JIT approach)

Program example

```
b := a * a

BB1
1: b := a * a
2: c := a + a
3: a := b - a
4: if a < 10
   1: b := a * a
   2: c := a + a
   3: a := b - a
   4: if a < 10
      1: b := a * a
      2: c := a + a
      3: a := b - a

BB2
5: d := a + 5
6: e := c * a
7: d := e - d

BB3
8: f := a
9: e := f * f
10: d := e - f

BB4
11: d := d + 1
```

3 available registers

```
b := a * a
c := a + a
a := b - a
if (a < 10)
d := a + 5
e := c * a
d := e – d
else
f := a
e := f * f
d := e – f
5: d := a + 5
6: e := c * a
7: d := e - d
8: f := a
9: e := f * f
10: d := e - f
11: d := d + 1
```
Linear Scan [Poletto'99] (the JIT approach)

Program example

\[ b := a \times a \]
\[ c := a + a \]
\[ a := b - a \]
\[ \text{if } (a < 10) \]
\[ \quad d := a + 5 \]
\[ \quad e := c \times a \]
\[ \quad d := e - d \]
\[ \text{else} \]
\[ \quad f := a \]
\[ \quad e := f \times f \]
\[ \quad d := e - f \]
\[ d := d + 1 \]

3 available registers
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

Example
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

Example

Allocation
(Choose register residents)
Decoupled Register Allocation [Appel’01, Hack’06, Bouchez’06]

Procedure

- Allocation (Choose register residents)
- Assignment (map each sub-variable to a register)

Example
Decoupled Register Allocation

[Appel'01, Hack'06, Bouchez'06]

Procedure

Allocation
(Choose register residents)

Assignment
(map each sub-variable to a register)

Example

code

1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez’06]

**Procedure**

- **Allocation**
  - (Choose register residents)

- **Assignment**
  - (map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

**Allocation**
(Choose register residents)

**Assignment**
(map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td></td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td></td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

- **Allocation** (Choose register residents)
- **Assignment** (map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>

2 Available registers

- Blue
- Red
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

- **Allocation** (Choose register residents)
- **Assignment** (map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>

Maxlive: 3

2 Available registers

---

**Boubacar Diouf**

**Decoupled Register and LM Allocation**
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

- **Allocation** (Choose register residents)
  - 1: d = ...
  - 2: b = load ...
  - 3: b = b * d
  - 4: a = load ...
  - 5: a = d / a
  - 6: c = a / b
  - 7: a = b + c
  - 8: store c
  - 9: store a

- **Assignment** (map each sub-variable to a register)
  - d
  - b,d
  - b,d
  - a,b,d
  - a,b
  - b,c
  - a,c
  - a

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>d</td>
</tr>
<tr>
<td>2:</td>
<td>b,d</td>
</tr>
<tr>
<td>3:</td>
<td>b,d</td>
</tr>
<tr>
<td>4:</td>
<td>b,d</td>
</tr>
</tbody>
</table>
| 5:   | a,b, x
| 6:   | a,b  |
| 7:   | b,c  |
| 8:   | a,c  |
| 9:   | a    |

**maxlive**

- 2 Available registers
Decoupled Register Allocation [Appel’01, Hack’06, Bouchez’06]

Procedure

Allocation
(Choose register residents)

Assignment
(map each sub-variable to a register)

Example

code

1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a

live

d
b,d
b,d
a,b,d
a,b
b,c
a,c
a

maxlive

2 Available registers

a

1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a

Decoupled Register Allocation

[Appel’01, Hack’06, Bouchez’06]
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

Allocation
(Choose register residents)

Assignment
(map each sub-variable to a register)

Example

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b,(\times)</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>a,b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td>a</td>
</tr>
</tbody>
</table>

maxlive

Available registers:
- Blue: 2
- Red: 1

2 Available registers

a

b

\(\times\)
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

### Procedure

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Choose register residents)</td>
<td>(map each sub-variable to a register)</td>
</tr>
</tbody>
</table>

### Example

**code**

1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a

**live**

<table>
<thead>
<tr>
<th>d</th>
<th>b, d</th>
<th>b, d</th>
<th>a, b, X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- a
- b
- c

maxlive: 2 Available registers

2 Available registers
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

Allocation (Choose register residents)

Assignment (map each sub-variable to a register)

Example

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>a,b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>

maxlive

2 Available registers

2 Available registers

Example

1: d =...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a

d
b,d
b,d
a,b,d
a,b
b,c
a,c
a

maxlive

2 Available registers

Example

1: d =...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a

d
b,d
b,d
a,b,d
a,b
b,c
a,c
a

maxlive

2 Available registers

Example

1: d =...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a

d
b,d
b,d
a,b,d
a,b
b,c
a,c
a

maxlive

2 Available registers
Decoupled Register Allocation [Appel’01, Hack’06, Bouchez’06]

Procedure

Allocation (Choose register residents)

Assignment (map each sub-variable to a register)

Example

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b, d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b, d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a, b, X</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a, b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b, c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a, c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td>a</td>
</tr>
</tbody>
</table>

maxlive

2 Available registers

a - b - c
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

1. **Allocation** (Choose register residents) **splitting**
2. **Assignment** (map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>

2 Available registers

---

**Decoupled Register and Local Memory Allocation**
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

Allocation
(Choose register residents)
splitting

Assignment
(map each sub-variable to a register)

Example

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>

2 Available registers
Decoupled Register Allocation [Appel’01, Hack’06, Bouchez’06]

### Procedure

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Choose register residents)</td>
<td>(map each sub-variable to a register)</td>
</tr>
</tbody>
</table>

#### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>1</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a = d / a</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Available Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
**Decoupled Register Allocation** [Appel’01, Hack’06, Bouchez’06]

**Procedure**

**Allocation**
(Choose register residents)

**splitting**

**Assignment**
(map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b = load ...</td>
<td>b,d</td>
</tr>
<tr>
<td>3: b = b * d</td>
<td>b,d</td>
</tr>
<tr>
<td>4: a1 = load ...</td>
<td>a,b,d</td>
</tr>
<tr>
<td>5: a2 = d / a1</td>
<td>a,b</td>
</tr>
<tr>
<td>6: c = a2 / b</td>
<td>b,c</td>
</tr>
<tr>
<td>7: a3 = b + c</td>
<td>a,c</td>
</tr>
<tr>
<td>8: store c</td>
<td>a</td>
</tr>
<tr>
<td>9: store a3</td>
<td></td>
</tr>
</tbody>
</table>

2 Available registers

---

*BOUBACAR DIOUF*
Decoupled Register Allocation [Appel’01, Hack’06, Bouchez’06]

**Procedure**

- **Allocation** (Choose register residents)
  - **splitting**
- **Assignment** (map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b1 = load ...</td>
<td>b1,d</td>
</tr>
<tr>
<td>3: b2 = b1 * d</td>
<td>b2,d</td>
</tr>
<tr>
<td>4: a1 = load ...</td>
<td>a1,b2,d</td>
</tr>
<tr>
<td>5: a2 = d / a1</td>
<td>a2,b2</td>
</tr>
<tr>
<td>6: c1 = a2 / b2</td>
<td>b2,c1</td>
</tr>
<tr>
<td>7: a3 = b2 + c1</td>
<td>a3,c1</td>
</tr>
<tr>
<td>8: store c1</td>
<td>a3</td>
</tr>
<tr>
<td>9: store a3</td>
<td></td>
</tr>
</tbody>
</table>

2 Available registers

---
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

- **Allocation** (Choose register residents)
  - **splitting**

- **Assignment** (map each sub-variable to a register)

**Example**

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b1 = load ...</td>
<td>b1,d</td>
</tr>
<tr>
<td>3: b2 = b1 * d</td>
<td>b2,d</td>
</tr>
<tr>
<td>4: a1 = load ...</td>
<td>a1,b2,d</td>
</tr>
<tr>
<td>5: a2 = d / a1</td>
<td>a2,b2</td>
</tr>
<tr>
<td>6: c1 = a2 / b2</td>
<td>b2,c1</td>
</tr>
<tr>
<td>7: a3 = b2 + c1</td>
<td>a3,c1</td>
</tr>
<tr>
<td>8: store c1</td>
<td>a3</td>
</tr>
<tr>
<td>9: store a3</td>
<td></td>
</tr>
</tbody>
</table>

**maxlive**

- 2 Available registers

---

**INTRODUCTION**

**REGISTER ALLOCATION**

- [Appel'01, Hack'06, Bouchez'06]

**LOCAL MEMORY**

---

**CONCLUSION**
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

Allocation
(Choose register residents)

splitting

Assignment
(map each sub-variable to a register)

Example

<table>
<thead>
<tr>
<th>Code</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>1</td>
</tr>
<tr>
<td>2: b1= load ...</td>
<td>2</td>
</tr>
<tr>
<td>3: b2= b1 * d</td>
<td>2</td>
</tr>
<tr>
<td>4: a1= load ...</td>
<td>3</td>
</tr>
<tr>
<td>5: a2= d / a1</td>
<td>2</td>
</tr>
<tr>
<td>6: c1= a2 / b2</td>
<td>2</td>
</tr>
<tr>
<td>7: a3= b2 + c1</td>
<td>1</td>
</tr>
<tr>
<td>8: store c1</td>
<td></td>
</tr>
<tr>
<td>9: store a3</td>
<td></td>
</tr>
</tbody>
</table>

maxlive

2 Available registers
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

**Procedure**

**Allocation**
(Choose register residents)

**splitting**

**Assignment**
(map each sub-variable to a register)

**Example**

**code**

1: \(d = \ldots\)
2: \(b_1 = \text{load} \ldots\)
3: \(b_2 = b_1 \times d\)
4: \(a_1 = \text{load} \ldots\)
5: \(a_2 = d / a_1\)
6: \(c_1 = a_2 / b_2\)
7: \(a_3 = b_2 + c_1\)
8: \(\text{store } c_1\)
9: \(\text{store } a_3\)

**live**

\(d\)
\(b_1, d\)
\(b_2, d\)
\(a_1, b_2, \times\)
\(a_2, b_2\)
\(b_2, c_1\)
\(a_3, c_1\)
\(a_3\)

maxlive

\(b_1\)
\(a_1\)
\(a_2\)
\(b_2\)
\(a_3\)
\(c_1\)

2 Available registers
Decoupled Register Allocation [Appel'01, Hack'06, Bouchez'06]

Procedure

- Allocation (Choose register residents) splitting
- Assignment (map each sub-variable to a register)

Example

<table>
<thead>
<tr>
<th>code</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: d = ...</td>
<td>d</td>
</tr>
<tr>
<td>2: b1 = load ...</td>
<td>b1, d</td>
</tr>
<tr>
<td>3: b2 = b1 * d</td>
<td>b2, d</td>
</tr>
<tr>
<td>4: a1 = load ...</td>
<td>a1, b2</td>
</tr>
<tr>
<td>5: a2 = d / a1</td>
<td>a2, b2</td>
</tr>
<tr>
<td>6: c1 = a2 / b2</td>
<td>b2, c1</td>
</tr>
<tr>
<td>7: a3 = b2 + c1</td>
<td>a3, c1</td>
</tr>
<tr>
<td>8: store c1</td>
<td>a3</td>
</tr>
<tr>
<td>9: store a3</td>
<td></td>
</tr>
</tbody>
</table>

maxlive

1 Available registers

2 Available registers
Outline

Introduction

Register Allocation
- Register allocation techniques
- Split Register Allocation
- Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
- Decoupled allocation for linearized programs
- The (local memory) spill minimization problem

Conclusion
Split compilation?

Just-in-time (JIT) compilation

1. portability (interpretation)
2. better performance (static compilation)

Annotation-enhanced JIT Compilation

1. reducing dynamic compilation time [Krintz’01]
2. improving performance of generated code [Jones’00]

Split compilation

de the goal is to split complex and target-dependent optimisations into two coordinated stages: offline (static compiler) and online (JIT compiler)
Global view of Split Register Allocation
Global view of Split Register Allocation
Global view of Split Register Allocation

Allocation
  maxlive
Global view of Split Register Allocation

Allocation
maxlive
splitting
Global view of Split Register Allocation

Offline stage

Allocation

maxlive
splitting
Global view of Split Register Allocation
Global view of Split Register Allocation

Offline stage

Allocation
  maxlive
  splitting

annotation

Assignment
Global view of Split Register Allocation

Offline stage

Allocation

maxlive
splitting

annotation

Online stage

Assignment

BOUBACAR DIOUF

DECOUPLED REGISTER AND LM ALLOCATION
Stages of Split Register Allocation

code

offline
Stages of Split Register Allocation

code

downwards

downwards

offline
Stages of Split Register Allocation

code

ILP

offline
Stages of Split Register Allocation

code

ILP

offline
Stages of Split Register Allocation

1. code
2. ILP
3. spill set
4. offline
Stages of Split Register Allocation

- **code**
- **ILP**
- **spill set**
- **offline**
Stages of Split Register Allocation

- code
- ILP
- spill set
- compress
- offline
Stages of Split Register Allocation

- code
- ILP
- spill set
- compress
- offline
Stages of Split Register Allocation

- code
- ILP
- spill set
- compress
- offline
- compressed spill set
Stages of Split Register Allocation

1. offline
2. ILP
3. spill set
4. compress
5. compressed spill set
Stages of Split Register Allocation

1. **code**
2. **ILP**
3. **spill set**
4. **compress**
5. **compressed spill set**

**offline**
Stages of Split Register Allocation

- **code**
  - **ILP**
    - **spill set**
      - **compress**
        - **compressed spill set**

**offline** → **online**
Stages of Split Register Allocation

1. **code**
2. **ILP**
3. **spill set**
4. **compress**
5. **offline**
6. **compressed spill set**
7. **allocator**
8. **online**
Stages of Split Register Allocation

- ** offline **
  - code
    - ILP
    - spill set
      - compress
    - compressed spill set
  - allocator

- ** online **
Stages of Split Register Allocation

- **Offline**
  - Code
  - ILP
  - Spill set
  - Compress
- **Online**
  - Allocated code
  - Allocator
  - Compressed spill set
Stages of Split Register Allocation

- **Offline**: ILP, spill set, compress
- **Online**: Allocated code, compressed spill set, allocator

The diagram shows the flow from code to allocated code, illustrating the stages of register allocation.
Compression

- The spill-Set produced by ILP can be used as annotation

- The goal is to reduce the annotation size:
  1. Run the online allocator
  2. Drop from the annotation any spilled variable that would be found by the online allocator
Compression Algorithm

```
  a  b  c  d  e  f
  0-   -   -   -   -   -   -
  1-   -   -   -   -   -   -
  2-   -   -   -   -   -   -
  3-  2   -   -   -   -   -
  4-   -   -   2   1   -   -
  5-   -  2   1   -   -   -
  6-  4   -   -   -   2   3
```

Compression Algorithm

2 available registers
Compression Algorithm

2 available registers
Optimal spill set: \{c,e\}
Compression Algorithm

2 available registers
Optimal spill set: \{c,e\}
Annotated spills:

\begin{tabular}{cccccccc}
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\end{tabular}
Compression Algorithm

2 available registers

Optimal spill set: {c,e}

Annotated spills:
Compression Algorithm

2 available registers
Optimal spill set: \{c,e\}

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:
Compression Algorithm

Is c in the optimal spill set?

Optimal spill set: \{c, e\}

2 available registers

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:

Yes! No need to annotate c!
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c, e\}

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills:

No! Should spill e!
Compression Algorithm

2 available registers

Optimal spill set: \{c,e\}

Annotated spills: \{e\}
Compression Algorithm

2 available registers
Optimal spill set: \{c,e\}
Annotated spills: \{e\}
Compression Algorithm

2 available registers
Optimal spill set: \{c,e\}
Annotated spills: \{e\}
Experimental study: Framework

Framework:

- JikesRvm 3.0.1
- CPLEX (ILP)
- SPEC JVM98 benchmarks
- x86_32
Experimental study: Annotation

- Preserving the information collected in the offline stage requires at most 0.26% of the live ranges to be annotated.

- The compression algorithm removes by average 95.71% of live ranges within the optimal spill set.
Experimental study: allocation cost

split compilation cost w.r.t. optimal cost
lower is better
Experimental study: speedups

speedup of annotated code

- Bytecode annotation
- LIR annotation
- All live range annotation

benchmarks:
- check
- compress
- jess
- raytrace
- db
- javac
- mpegaudio
- mtrt
- jack
- average
Outline

Introduction

Register Allocation
- Register allocation techniques
- Split Register Allocation
- Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
- Decoupled allocation for linearized programs
- The (local memory) spill minimization problem

Conclusion
Two heuristics for the spill minimization problem

**Input:**
A register allocation problem where each variable has an estimated spill cost

**Objective:**
We want to perform an allocation that minimizes the cost of all the spilled variables

**Our solutions:**
- Iterated-Optimal allocator
- Clustering allocator
The Iterated-Optimal Allocator

Basis:

• The spill minimization problem (spill everywhere) on SSA-programs is pseudo-polynomial [Bouchez'07]

The solution:

• for a set of variables and a fixed number of available register
• Iteratively find the optimal set of variables to allocate with a small number of registers
How the iterated-optimal allocator works

Diagram: A graph showing the allocation of variables to registers.
How the iterated-optimal allocator works

![Diagram of the allocator process]
How the iterated-optimal allocator works

2 available registers

---

Boubacar Diouf

Decoupled Register and LM Allocation
How the iterated-optimal allocator works

![Diagram of variable allocation]

Allocated variables when one register is available:

2 available registers

- Blue
- Red
How the iterated-optimal allocator works

2 available registers

Allocated variables when one register is available:

b d f
How the iterated-optimal allocator works

2 available registers

Allocated variables when one register is available:
- b
- d
- f

Allocated variables when a second register is available:
- c
- e

The cost of the allocation is 5
How the iterated-optimal allocator works

2 available registers

Allocated variables when one register is available:
- b
- d
- f

Allocated variables when a second register is available:
- c
- e

The cost of the allocation is 5
How the iterated-optimal allocator works

2 available registers

[Diagram showing the allocation process]
The clustering allocator

The solution:

- for a set of variables and a fixed number of available register
- iteratively approximate the set of variables to allocate with one register
- we call each of the group of variables to allocate to a register a cluster
How the clustering allocator works

2 available registers

A graph showing the allocation of registers. Each node represents a register, and the edges indicate the relationships between them. The numbers next to the nodes represent the allocation status.
How the clustering allocator works

Variables sorted by decreasing cost: a, c, b, d, e, f

2 available registers

[Diagram of nodes and edges with numbers indicating connections]
How the clustering allocator works

Variables sorted by decreasing cost: a, c, b, d, e, f

Cluster1: {a

2 available registers
How the clustering allocator works

Variables sorted by decreasing cost: c, b, d, e, f

Cluster1: {a,e}

2 available registers
How the clustering allocator works

Variables sorted by decreasing cost: c, b, d, f

Cluster1: \{a,e\}

2 available registers

- Blue
- Red
How the clustering allocator works

Variables sorted by decreasing cost: b, d, f

Cluster1: {a, e}
Cluster2: {c,

2 available registers
How the clustering allocator works

Variables sorted by decreasing cost: b, d

Cluster1: \{a, e\}
Cluster2: \{c, f\}

2 available registers

Cluster Diagram:

- Cluster1: \{a, e\}
- Cluster2: \{c, f\}
How the clustering allocator works

Variables sorted by decreasing cost:

Cluster1: \{a,e\}
Cluster2: \{c,f\}
Cluster3: \{b,d\}

2 available registers
How the clustering allocator works

Variables sorted by decreasing cost:

Cluster1: \{a,e\}
Cluster2: \{c,f\}
Cluster3: \{b,d\}

Clusters sorted by decreasing cost: cluster1, cluster3, cluster2

2 available registers
How the clustering allocator works

Variables sorted by decreasing cost:

Cluster1: \{a,e\}
Cluster2: \{c,f\}
Cluster3: \{b,d\}

Clusters sorted by decreasing cost: cluster1, cluster3, cluster2

The cost of the allocation is 5
How the clustering allocator works

Clusters sorted by decreasing cost: cluster1, cluster3, cluster2

2 available registers

The cost of the allocation is 5
Comparison about different allocators
Comparison about ILP-programs solving time

<table>
<thead>
<tr>
<th>Register count</th>
<th>Optimal (ms)</th>
<th>Iterated-Optimal (ms)</th>
<th>speedup (Iterated-optimal/Optimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 registers</td>
<td>2810</td>
<td>2880</td>
<td>0.98</td>
</tr>
<tr>
<td>4 registers</td>
<td>22998</td>
<td>7372</td>
<td>3.12</td>
</tr>
<tr>
<td>6 registers</td>
<td>74561</td>
<td>8846</td>
<td>8.43</td>
</tr>
<tr>
<td>8 registers</td>
<td>381755</td>
<td>9768</td>
<td>39.08</td>
</tr>
<tr>
<td>10 registers</td>
<td>1194311</td>
<td>10477</td>
<td>113.99</td>
</tr>
<tr>
<td>12 registers</td>
<td>3231582</td>
<td>11120</td>
<td>290.61</td>
</tr>
<tr>
<td>14 registers</td>
<td>4147764</td>
<td>11688</td>
<td>354.87</td>
</tr>
<tr>
<td>16 registers</td>
<td>4879200</td>
<td>12281</td>
<td>397.3</td>
</tr>
</tbody>
</table>

**Table:** Time spent in milliseconds (ms) to solve ILP-programs
Inclusion property

Inclusion Property

- Is the optimal spill set with n registers included in the optimal spill set with n-1 registers?

Experimental study

- Varying the number of registers from 2 to the maximal number where spilling is needed

Result

- Inclusion property holds for 99.83% of the SPEC JVM98’s methods
Inclusion property

Inclusion Property

• Is the optimal spill set with \( n \) registers included in the optimal spill set with \( n-1 \) registers?

Experimental study

• Varying the number of registers from 2 to the maximal number where spilling is needed

Result

• Inclusion property holds for 99.83% of the SPEC JVM98’s methods
Outline

Introduction

Register Allocation
- Register allocation techniques
- Split Register Allocation
- Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
- Decoupled allocation for linearized programs
- The (local memory) spill minimization problem

Conclusion
Motivation 1/2

Decoupled register allocation

- Allocation phase (rely on maxlive, choose register residents)
- Assignment: which register for which variable (polynomial under SSA)

Decoupling: isolate the hard problem of allocation (spilling)

Decoupled local memory allocation

- Allocation (rely on maxsize, choose local-memory residents)
- Assignment: which offset for which Array
  - Colorability?
  - Complexity?
Reduce local-memory pressure

- through live range splitting: choice of decision points where loads and stores are going to be inserted
- through loop-transformations: tiling, loop distribution, strip mining
The approach 1/2

Preliminary transformations

- Tiling
- Loop distribution
- Strip Mining

Allocation schemes

1. At every array instruction, finer decision points but may incur excessive complexity

2. Every time an array becomes alive (similar to SSA-based register allocation if arrays are renamed)

3. For the whole method (similar to spill everywhere problem)
The approach 1/2

Preliminary transformations

- Tiling
- Loop distribution
- Strip Mining

Allocation schemes

1. At every array instruction, finer decision points but may incur excessive complexity

2. Every time an array becomes alive (similar to SSA-based register allocation if arrays are renamed)

3. For the whole method (similar to spill everywhere problem)
The approach 1/2

Preliminary transformations

- Tiling
- Loop distribution
- Strip Mining

Allocation schemes

1. At every array instruction, finer decision points but may incur excessive complexity

2. Every time an array becomes alive (similar to SSA-based register allocation if arrays are renamed)

3. For the whole method (similar to spill everywhere problem)
The approach 1/2

Preliminary transformations

- Tiling
- Loop distribution
- Strip Mining

Allocation schemes

1. At every array instruction, finer decision points but may incur excessive complexity

2. Every time an array becomes alive (similar to SSA-based register allocation if arrays are renamed)

3. For the whole method (similar to spill everywhere problem)
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\begin{align*}
A1 &= B \\
A1 &= \ldots \\
B &= \text{LM} \\
A2 &= C \\
A2 &= \ldots \\
A3 &= \Phi(A1, A2) \\
&= B \\
&= C
\end{align*}
\]
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\begin{align*}
A_1 &= \ldots \\
A_2 &= \ldots \\
A_3 &= \Phi(A_1, A_2)
\end{align*}
\]
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\begin{align*}
B &= A1 = ... \\
A2 &= A3 = \Phi(A1, A2) \\
C &= B = A1 = ...
\end{align*}
\]
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\begin{align*}
\text{B} &\quad \text{A1} \\
\text{LM} &\quad = B \\
\text{A1} &\quad = \ldots \\
\text{LM} &\quad = C \\
\text{A2} &\quad = \ldots \\
\text{LM} &\quad A1 = B \\
\text{A1} &\quad \text{LM} \\
\text{LM} &\quad = C \\
\text{A2} &\quad \text{LM} \\
\text{LM} &\quad A3 = \Phi(A1, A2)
\end{align*}
\]
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\begin{align*}
\text{PTR (A1)} & : B \\
A1 & : \ldots \\
\text{LM} & : B
\end{align*}
\]

\[
\begin{align*}
\text{PTR (A2)} & : C \\
A2 & : \ldots \\
\text{LM} & : C
\end{align*}
\]

\[
A3 = \Phi(A1, A2)
\]
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\begin{align*}
\text{PTR (A1)} & = B \\
A1 & = ... \\
\text{LM} & = C \\
A2 & = ... \\
\text{PTR (A2)} & = B \\
A1 & = ...
\end{align*}
\]

\[
A3 = \Phi(A1, A2)
\]

\[
\text{PTR(A3)} = \text{PTR (A1)}
\]
The approach 2/2

Abstracted model

- Array blocks are like scalar variables in register allocation
- Extension of SSA to perform on array blocks
  - Not array SSA: no dataflow of individual array elements

Pointer reconciliation

\[
\text{PTR}(A1) = B \\
A1 = ... \\
\text{LM} \\
\text{PTR}(A2) = C \\
A2 = ... \\
\text{LM} \\
A3 = \Phi(A1, A2) \\
\text{PTR}(A3) = \text{PTR}(A1) \\
\text{PTR}(A3) = \text{PTR}(A2)
\]
Outline

Introduction

Register Allocation
  - Register allocation techniques
  - Split Register Allocation
  - Spill minimization problem

Local Memory
  - Motivation and approach
  - Experimental Validation
    - Decoupled allocation for linearized programs
    - The (local memory) spill minimization problem

Conclusion
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Brief description</th>
<th>Suite</th>
<th>Data size</th>
<th>arrays /blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge-Detect</td>
<td>Edge detection in an image</td>
<td>UTDSP</td>
<td>196644</td>
<td>4/385</td>
</tr>
<tr>
<td>D-FFT</td>
<td>256-point complex FFT</td>
<td>UTDSP</td>
<td>2032</td>
<td>7/7</td>
</tr>
<tr>
<td>Bmcm</td>
<td>Water molecular dynamics</td>
<td>Perfect Club</td>
<td>125240</td>
<td>10/310</td>
</tr>
<tr>
<td>MxM</td>
<td>Matrix multiplication</td>
<td>n.a.</td>
<td>120000</td>
<td>3/300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency_LM</td>
<td>8</td>
</tr>
<tr>
<td>latency_MM</td>
<td>128</td>
</tr>
<tr>
<td>latency_move($s_V$)</td>
<td>$8 + 2s_V$</td>
</tr>
<tr>
<td>latency_spill($s_V$)</td>
<td>$128 + 4s_V$</td>
</tr>
<tr>
<td>latency_reload($s_V$)</td>
<td>$128 + 4s_V$</td>
</tr>
</tbody>
</table>
Results

**BMCM**
- decoupled: Blue bar
- integrated: Pink bar
- minsize: (16%)
  - +20%
  - +40%
- maxsize: -1
- fitsize: (5166%)

**MXM**
- decoupled: Blue bar
- integrated: Pink bar
- minsize: (33%)
  - +20%
  - +40%
- maxsize: -1
- fitsize: (10000%)

**EDGE_DETECT**
- decoupled: Blue bar
- integrated: Pink bar
- minsize: (24%)
  - +20%
  - +40%
- maxsize: -1
- fitsize: (9435%)

**FFT**
- decoupled: Blue bar
- integrated: Pink bar
- minsize: (24%)
  - +20%
  - +40%
- maxsize: -1
- fitsize: (100%)
Outline

Introduction

Register Allocation
- Register allocation techniques
- Split Register Allocation
- Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
  - Decoupled allocation for linearized programs
  - The (local memory) spill minimization problem

Conclusion
Local memory allocation and weighted Interval graph (WIG) coloring

Linearized programs

- Given a numbered intermediate representation of a program (live range splitting performed)

- The live range of the arrays approximated as intervals

Equivalence

- The local memory allocation problem for a linearized program is equivalent to WIG coloring problem
The shipbuilding problem

- Determining whether $\chi(G_w) \leq k$ is an NP-complete problem [Golumbic'04] even if $G$ is an interval graph and $w$ is restricted to the values 1 and 2
The submarine-building problem

- Assuming that loads and stores wrap around transparently we define the submarine building problem.
Complexity results of the submarine-building problem

• Determining whether $x(G_w) \leq k$ is an NP-complete problem on interval graphs [Diouf'11]

• The problem is linear on proper interval graphs and on the Not-So-Proper (NSP) interval graphs, whereas the shipbuilding problem remains NP-complete on proper interval graphs [Diouf'11]
Proper interval graphs

(a)

(b)
The class of NSP interval graphs englobes the classes of proper interval graphs and of superperfect graphs defined by Li et al. [Li'11], that are used to decouple the local memory allocation.
NSP interval graphs

- The class of NSP interval graphs englobes the classes of proper interval graphs and of superperfect graphs defined by Li et al. [Li'11], that are used to decouple the local memory allocation.
The spill minimization problem

**Goal:**

- Given an estimated spill cost for each array
- We want to perform an allocation that minimizes the cost of all the spilled arrays (arrays placed in the main-memory)

**The arrays clustering allocator**

- Clusters the arrays into a list of cluster
- Each cluster is composed of batches that do not interfere among them
- An array is added into a batch if it interferes at least with one array already in the batch

**Results**

The results are satisfactory, but it is still a work in progress
Outline

Introduction

Register Allocation
- Register allocation techniques
- Split Register Allocation
- Spill minimization problem

Local Memory
- Motivation and approach
- Experimental Validation
- Decoupled allocation for linearized programs
- The (local memory) spill minimization problem

Conclusion
Conclusion and perspectives

- Split register allocation
- Two heuristics devoted to the spill minimization problem
- Experimental validation of a decoupled approach
- Theoretical foundations for a decoupled local memory allocation

Perspectives

- Automation of the proposed algorithms in a context of a SSA-based register allocator (e.g. LLVM, LAO, …)
- Extend the work to environments where many threads share the same local memory
- Consider programming models like (HMPP, OpenCL) offering more support for software-controlled local memories to PGAS (Partitionned Global Address Space) languages requiring more attention to the memory locality