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

Usual size		Usual access time
More than 2 GB	Virtual memory (disk)	3 - 15 ms
256 MB - 2 GB	Physical memory	100 - 150 ns
256 KB - 4 MB	2nd-level cache or LM	40 - 60 ns
16 - 64 KB	1st-level cache or LM	5 - 10 ns
32 words	Registers (processor)	1 ns

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, predictible, 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

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
   r2 := r1
   r1 := r2 * r2
   r1 := r1 - r2
r1 := r1 +1
```

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
   r2 := r1
   r1 := r2 * r2
   r1 := r1 - r2
r1 := r1 +1
```

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
```

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

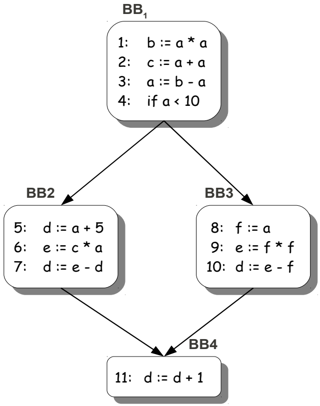
```
b := a * a
c := a + a
a := b - a
if (a < 10)
   d := a + 5
   e := c * a
   d := e - d
else
   f := a
   e := f * f
   d := e - f
d := d +1
```

Liveness of variables, live ranges

Program example

```
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
```

CFG

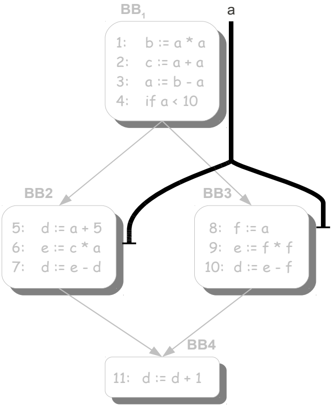


Liveness of variables, live ranges

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
```

CFG

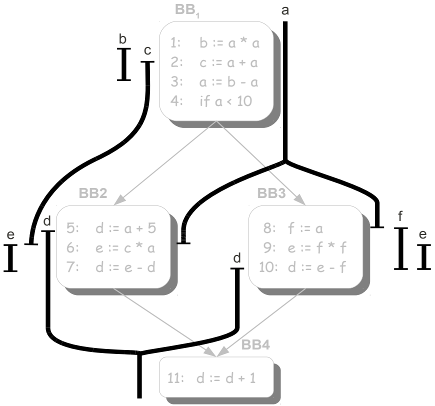


Liveness of variables, live ranges

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
```

CFG



Graph Coloring [Chaitin'81] (the classical approach)

Program example

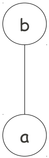
```
b := a * a
c := a + a
a := b − a
if (a < 10)
   d := a + 5
   e := c * a
   d := e − d
else
   f := a
   e := f * f
   d := e − f
d := d +1
```

3 available registers

Graph Coloring [Chaitin'81] (the classical approach)

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
```

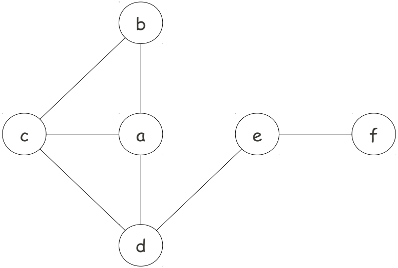


3 available registers

Graph Coloring [Chaitin'81] (the classical approach)

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
```

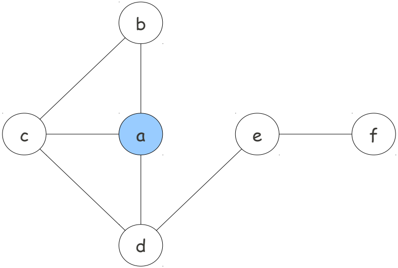


3 available registers

Graph Coloring [Chaitin'81] (the classical approach)

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
```

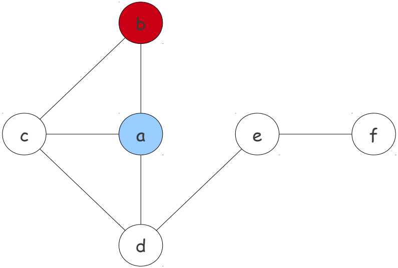


3 available registers

Graph Coloring [Chaitin'81] (the classical approach)

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
```

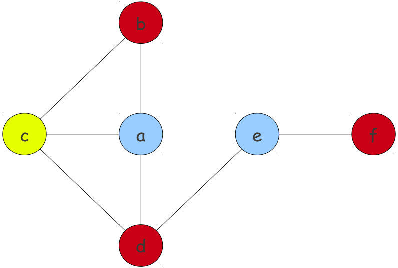


3 available registers

Graph Coloring [Chaitin'81] (the classical approach)

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
```

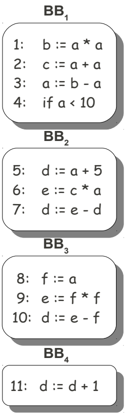


3 available registers

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

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
```

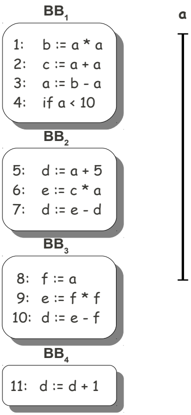


3 available registers

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

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
```

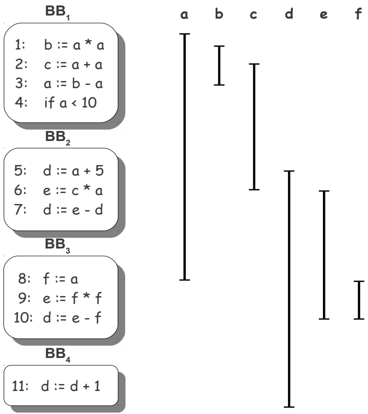


3 available registers

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

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
```

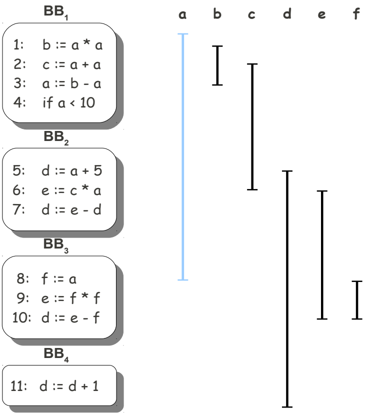


3 available registers

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

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
```

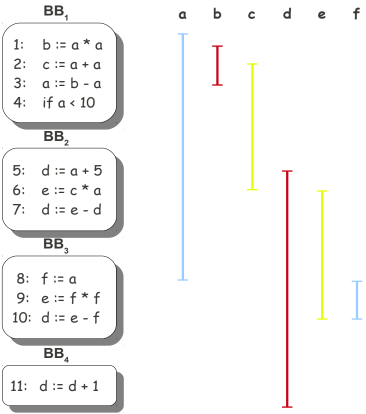


3 available registers

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

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
```



3 available registers

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

Procedure

Example

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

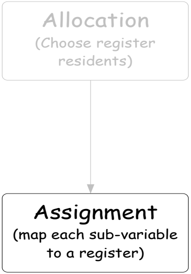
Procedure

Allocation
(Choose register residents)

Example

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

Procedure



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

```
d = ...
b = load …
b = b * d
a = load …
a = d / a
c = a / b
a = b + c
store c
store a
```

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

Procedure

Allocation
(Choose register residents)

↓

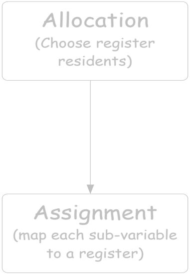
Assignment
(map each sub-variable to a register)

Example

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

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

Procedure



Example

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

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

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

maxlive

2 Available registers

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

Procedure

Allocation
(Choose register residents)

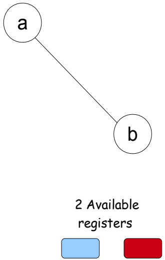
↓

Assignment
(map each sub-variable to a register)

Example

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

maxlive



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

Procedure

Allocation
(Choose register residents)

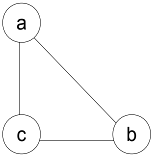
↓

Assignment
(map each sub-variable to a register)

Example

code	live	
1: d = ...	d	1
2: b = load …	b,d	2
3: b = b * d	b,d	2
4: a = load …	a,b,~~d~~	(3)
5: a = d / a	a,b	2
6: c = a / b	b,c	2
7: a = b + c	a,c	2
8: store c	a	1
9: store a		

maxlive



2 Available registers

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

Procedure

Allocation
(Choose register residents)

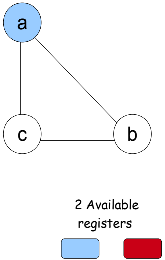
↓

Assignment
(map each sub-variable to a register)

Example

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

maxlive



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

Procedure

Allocation
(Choose register residents)

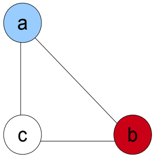
↓

Assignment
(map each sub-variable to a register)

Example

code	live	
1: d = ...	d	1
2: b = load …	b,d	2
3: b = b * d	b,d	2
4: a = load …	a,b,~~d~~	(3)
5: a = d / a	a,b	2
6: c = a / b	b,c	2
7: a = b + c	a,c	2
8: store c	a	1
9: store a		

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

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

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

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	live	
1: d = ...	d	1
2: b = load …	b,d	2
3: b = b * d	b,d	2
4: **a1** = load …	a,b,d	3
5: **a2** = d / **a1**	a,b	2
6: c = **a2** / b	b,c	2
7: **a3** = b + c	a,c	2
8: store c	a	1
9: store **a3**		

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	live	
1: d = ...	d	1
2: b1= load …	b1,d	2
3: b2= b1 * d	b2,d	2
4: a1= load …	a1,b2,d	3
5: a2= d / a1	a2,b2	2
6: c1= a2 / b2	b2,c1	2
7: a3= b2 + c1	a3,c1	2
8: store c1	a3	1
9: store a3		

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	live	
1: d = ...	d	1
2: b1= load …	b1,d	2
3: b2= b1 * d	b2,d	2
4: a1= load …	a1,b2,d	3
5: a2= d / a1	a2,b2	2
6: c1= a2 / b2	b2,c1	2
7: a3= b2 + c1	a3,c1	2
8: store c1	a3	1
9: store a3		

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	live	
1: d = ...	d	1
2: b1= load …	b1,d	2
3: b2= b1 * d	b2,d	2
4: a1= load …	a1,b2,~~d~~	3
5: a2= d / a1	a2,b2	2
6: c1= a2 / b2	b2,c1	2
7: a3= b2 + c1	a3,c1	2
8: store c1	a3	1
9: store a3		

maxlive

2 Available registers

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

Procedure

Allocation
(Choose register residents)
splitting

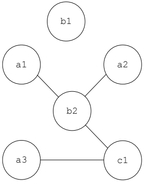
↓

Assignment
(map each sub-variable to a register)

Example

code	live	
1: d = ...	d	1
2: b1= load …	b1,d	2
3: b2= b1 * d	b2,d	2
4: a1= load …	a1,b2,~~d~~	3
5: a2= d / a1	a2,b2	2
6: c1= a2 / b2	b2,c1	2
7: a3= b2 + c1	a3,c1	2
8: store c1	a3	1
9: store a3		

maxlive



2 Available registers

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

Procedure

Allocation
(Choose register residents)
splitting

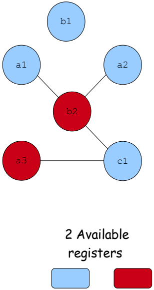
↓

Assignment
(map each sub-variable to a register)

Example

code	live	
1: d = ...	d	1
2: b1= load …	b1,d	2
3: b2= b1 * d	b2,d	2
4: a1= load …	a1,b2,~~d~~	3
5: a2= d / a1	a2,b2	2
6: c1= a2 / b2	b2,c1	2
7: a3= b2 + c1	a3,c1	2
8: store c1	a3	1
9: store a3		

maxlive



Outline

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

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

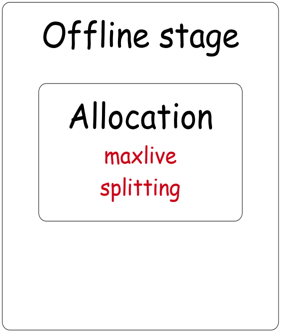
Global view of Split Register Allocation

Allocation

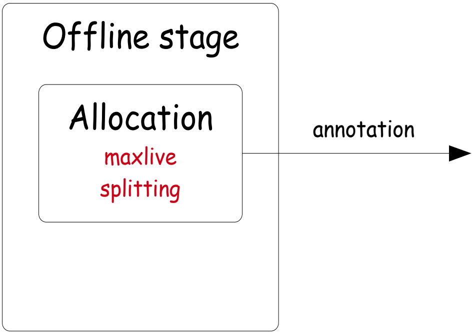
maxlive

splitting

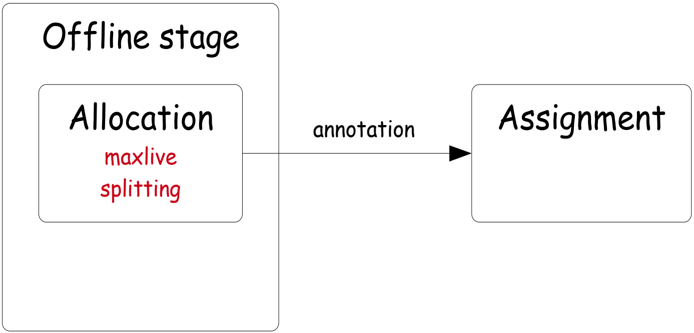
Global view of Split Register Allocation



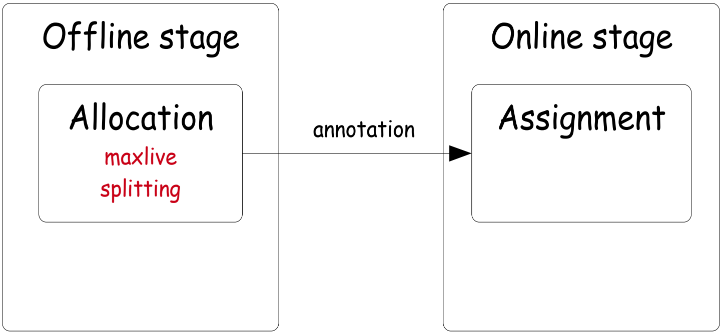
Global view of Split Register Allocation



Global view of Split Register Allocation



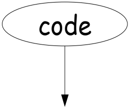
Global view of Split Register Allocation



Stages of Split Register Allocation

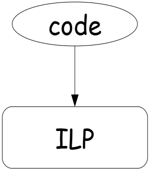
offline

Stages of Split Register Allocation



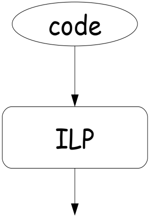
offline

Stages of Split Register Allocation



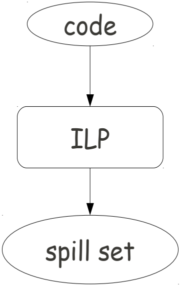
offline

Stages of Split Register Allocation



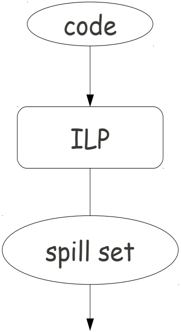
offline

Stages of Split Register Allocation



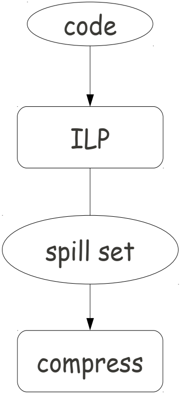
offline

Stages of Split Register Allocation



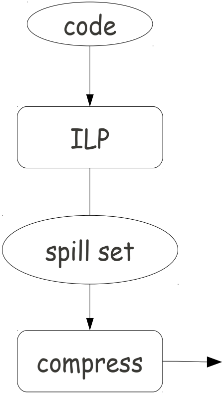
offline

Stages of Split Register Allocation



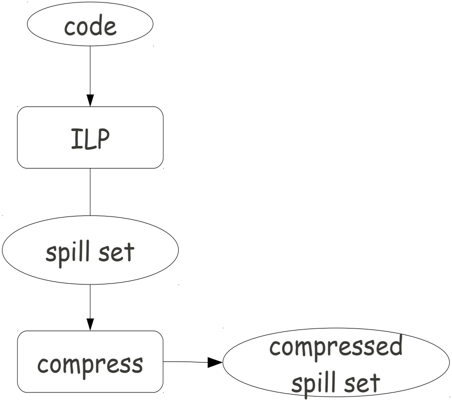
offline

Stages of Split Register Allocation



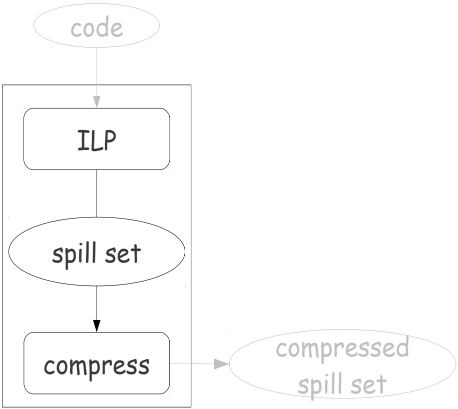
offline

Stages of Split Register Allocation



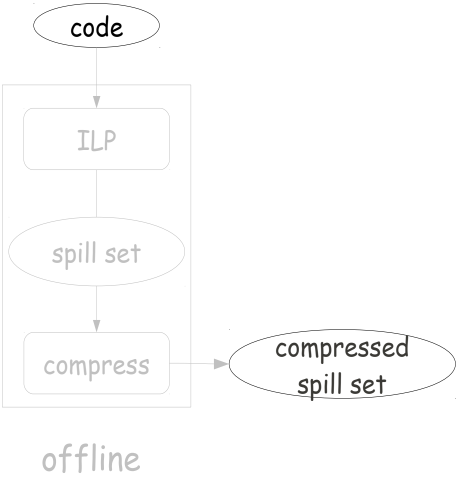
offline

Stages of Split Register Allocation

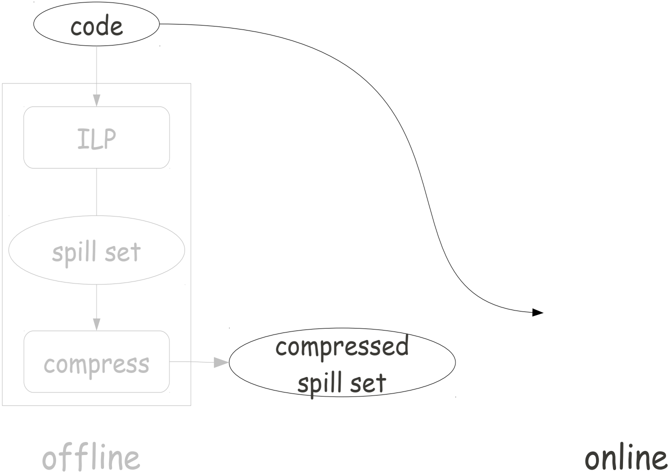


offline

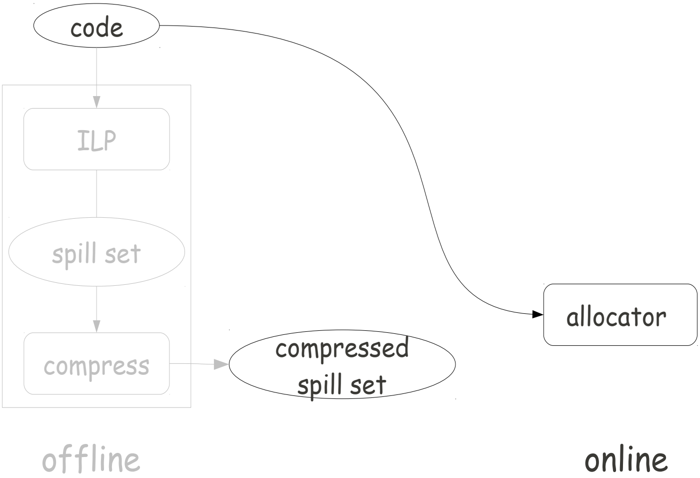
Stages of Split Register Allocation



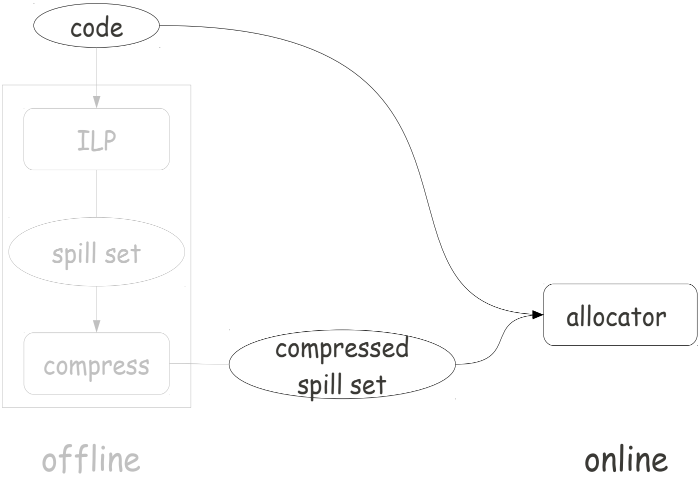
Stages of Split Register Allocation



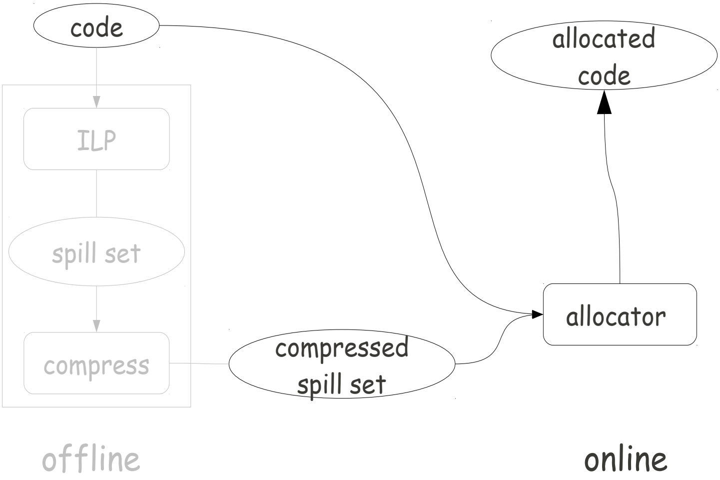
Stages of Split Register Allocation



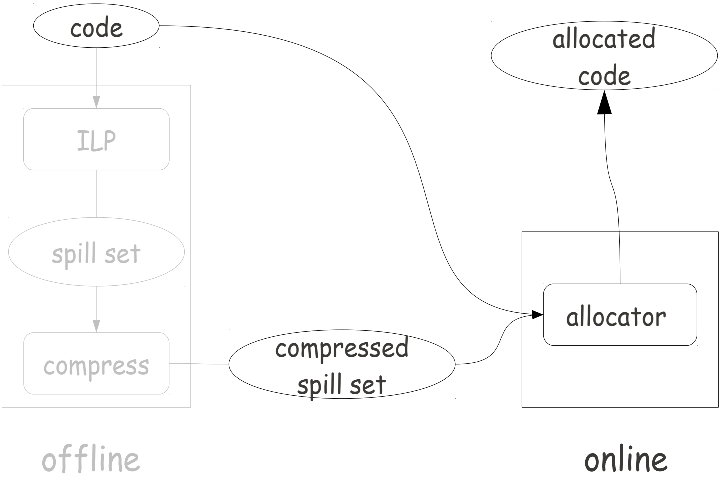
Stages of Split Register Allocation



Stages of Split Register Allocation



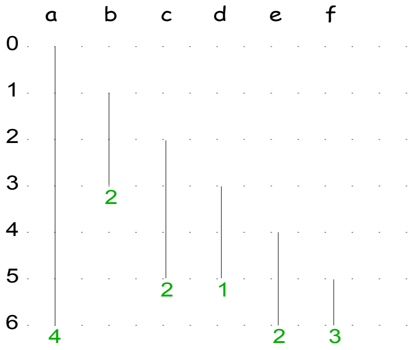
Stages of Split Register Allocation



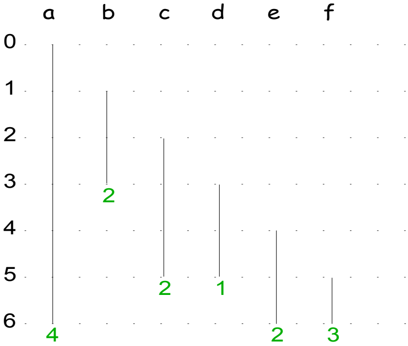
Compression

- 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

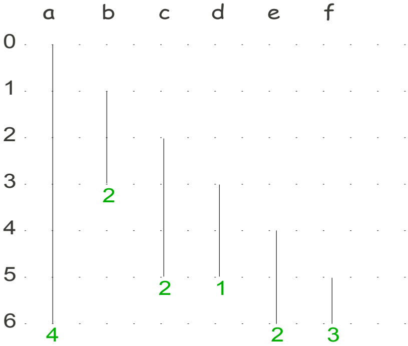


Compression Algorithm



2 available registers

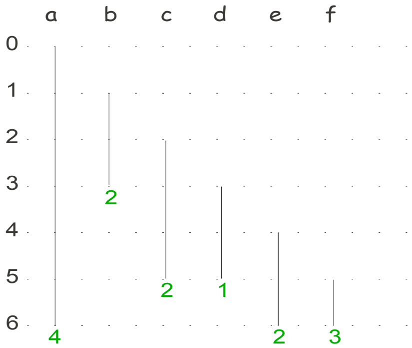
Compression Algorithm



2 available registers

Optimal spill set: {c,e}

Compression Algorithm

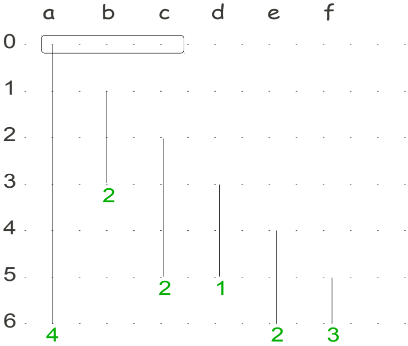


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

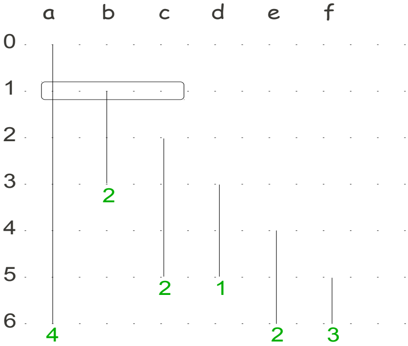


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

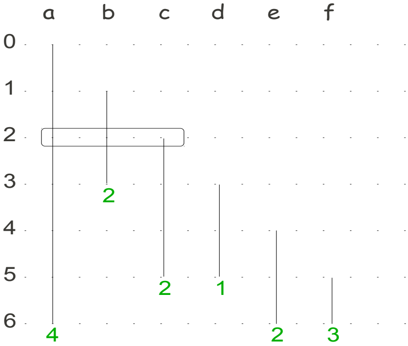


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

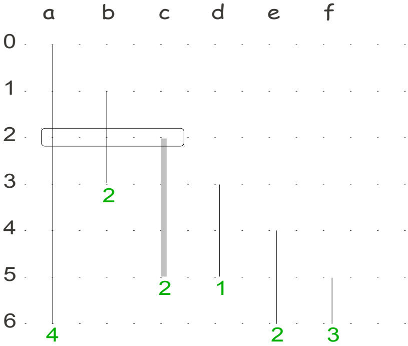


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

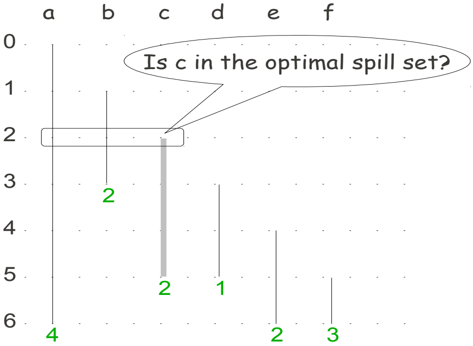


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

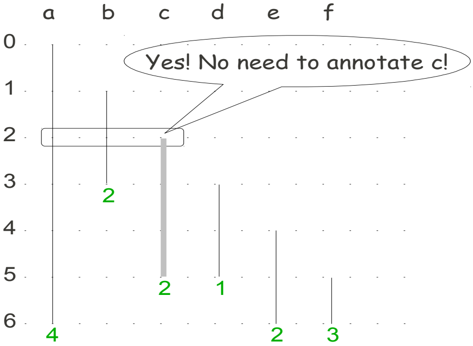


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

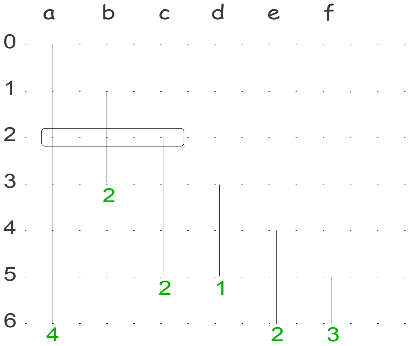


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

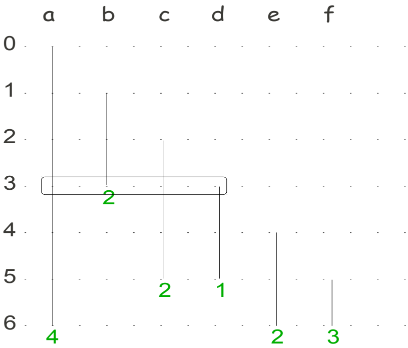


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

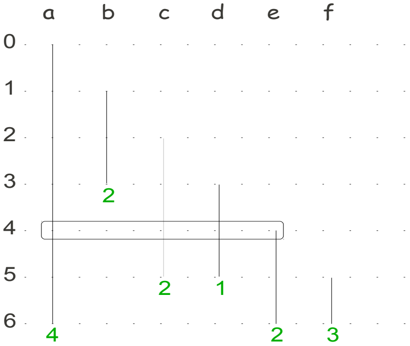


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

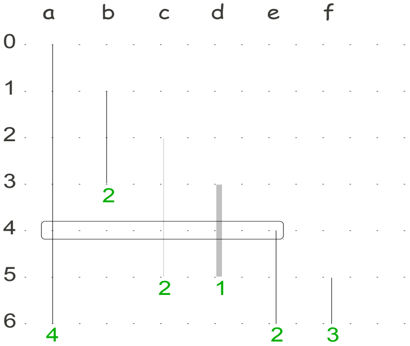


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

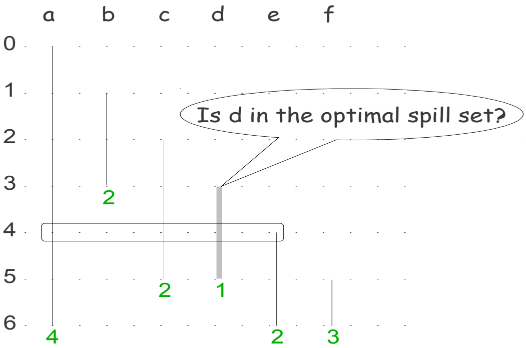


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

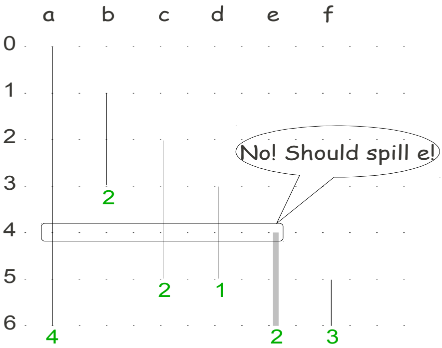


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

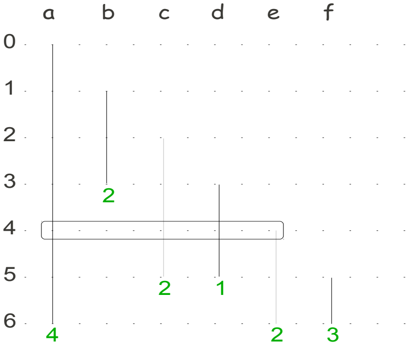


2 available registers

Optimal spill set: {c,e}

Annotated spills:

Compression Algorithm

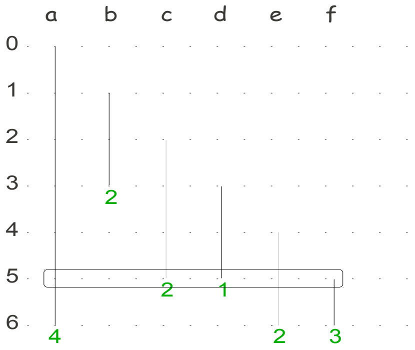


2 available registers

Optimal spill set: {c,e}

Annotated spills: {e}

Compression Algorithm

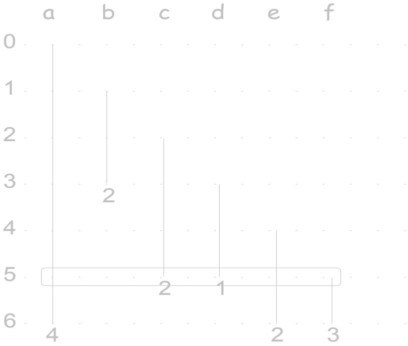


2 available registers

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:

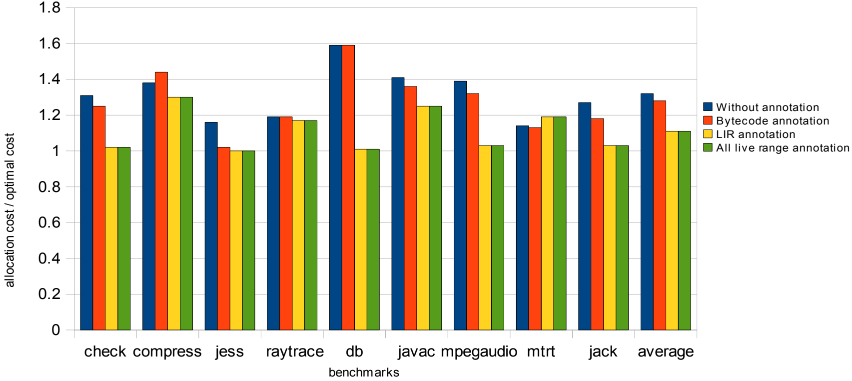
- JikesRvm 3.0.1
- CPLEX (ILP)
- SPEC JVM98 benchmarks
- x86_32

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

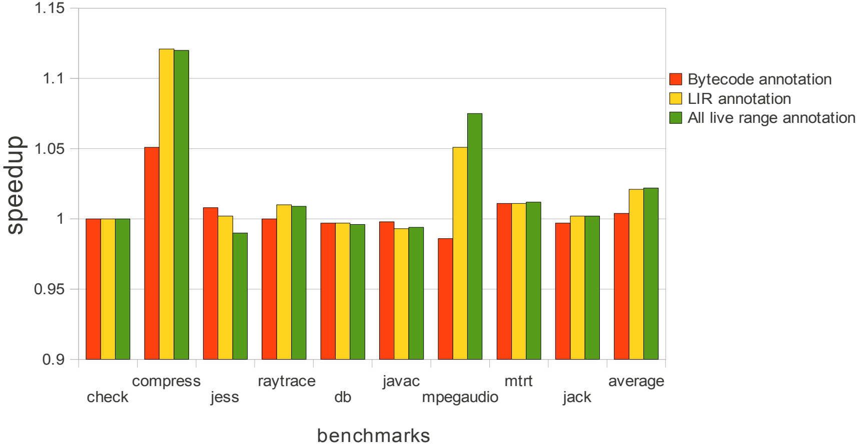
Experimental study: allocation cost



split compilation cost w.r.t. optimal cost

lower is better

Experimental study: speedups



speedup of annotated code

Outline

Two heuristics for the spill minimization problem

Input:

A register allocation problem where each varaiable 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

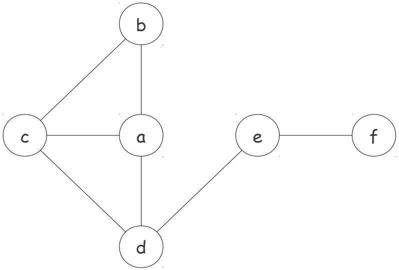
Basis:

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

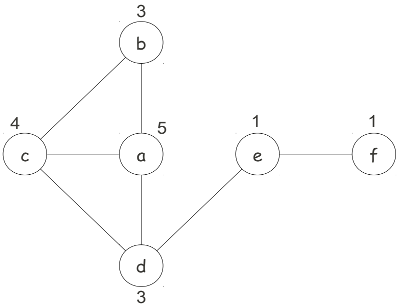
The solution:

- 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

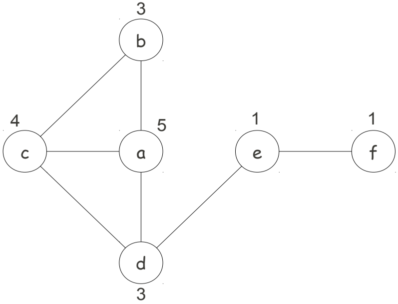
How the iterated-optimal allocator works



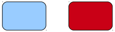
How the iterated-optimal allocator works



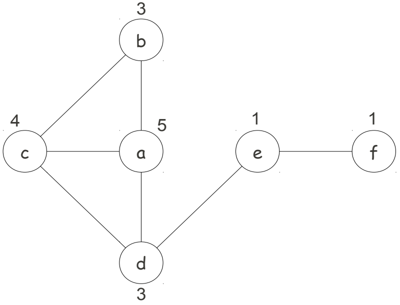
How the iterated-optimal allocator works



2 available registers

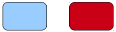


How the iterated-optimal allocator works

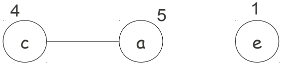


Allocated variables when one register is available:

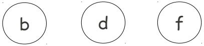
2 available registers



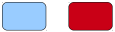
How the iterated-optimal allocator works



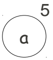
Allocated variables when one register is available:



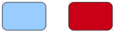
2 available registers



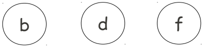
How the iterated-optimal allocator works



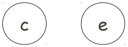
2 available registers



Allocated variables when one register is available:

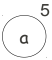


Allocated variables when a second register is available:

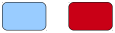


The cost of the allocation is **5**

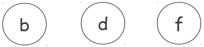
How the iterated-optimal allocator works



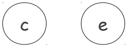
2 available registers



Allocated variables when one register is available:

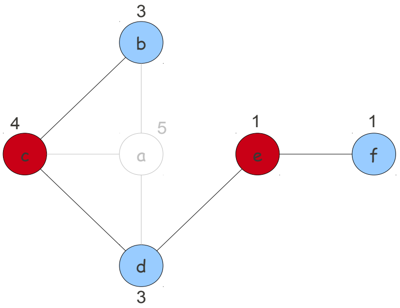


Allocated variables when a second register is available:

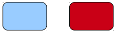


The cost of the allocation is **5**

How the iterated-optimal allocator works



2 available registers

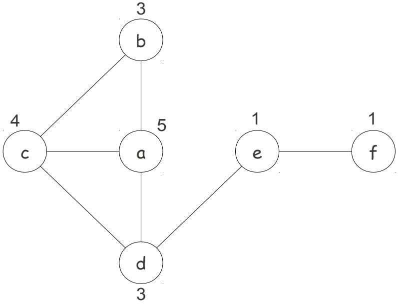


The clustering allocator

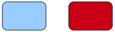
The solution:

- 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

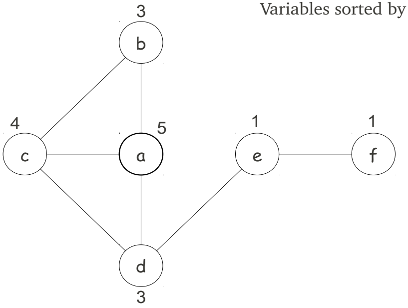
How the clustering allocator works



2 available registers

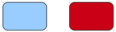


How the clustering allocator works

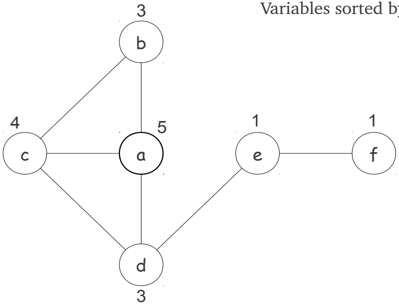


Variables sorted by decreasing cost: a, c, b, d, e, f

2 available registers



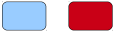
How the clustering allocator works



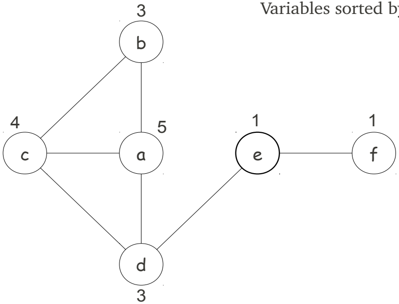
Variables sorted by decreasing cost: a, c, b, d, e, f

Cluster1: {a

2 available registers



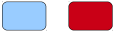
How the clustering allocator works



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

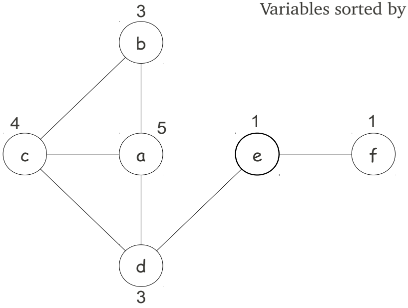
Cluster1: {a,e}

2 available registers



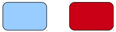
How the clustering allocator works

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

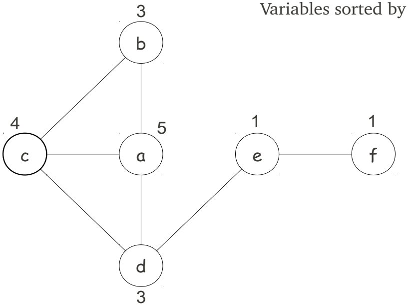


Cluster1: {a,e}

2 available registers



How the clustering allocator works

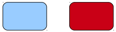


Variables sorted by decreasing cost: b, d, f

Cluster1: {a,e}

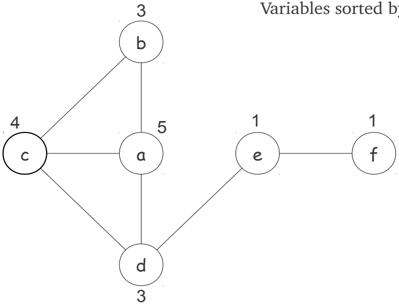
Cluster2: {c,

2 available registers



How the clustering allocator works

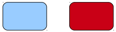
Variables sorted by decreasing cost: b, d



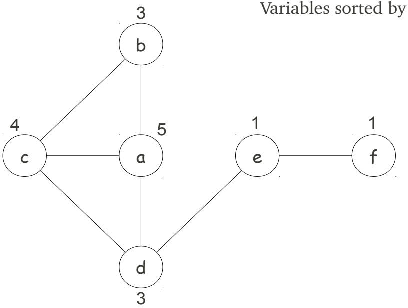
Cluster1: {a,e}

Cluster2: {c,f}

2 available registers



How the clustering allocator works



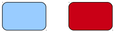
Variables sorted by decreasing cost:

Cluster1: {a,e}

Cluster2: {c,f}

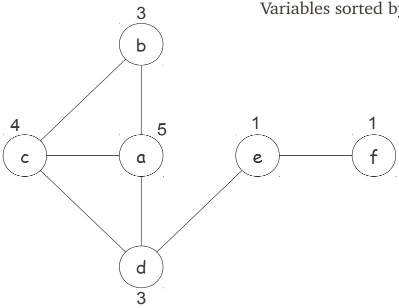
Cluster3: {b,d}

2 available registers



How the clustering allocator works

Variables sorted by decreasing cost:



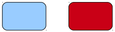
Cluster1: {a,e}

Cluster2: {c,f}

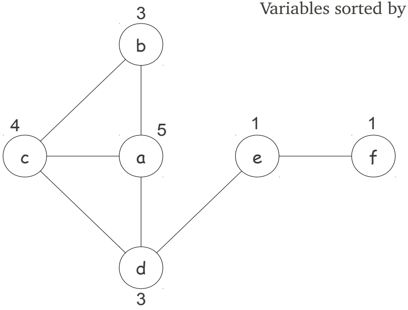
Cluster3: {b,d}

Clusters sorted by decreasing cost: cluster1, cluster3, cluster2

2 available registers



How the clustering allocator works



Variables sorted by decreasing cost:

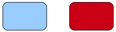
Cluster1: {a,e}

Cluster2: {c,f}

Cluster3: {b,d}

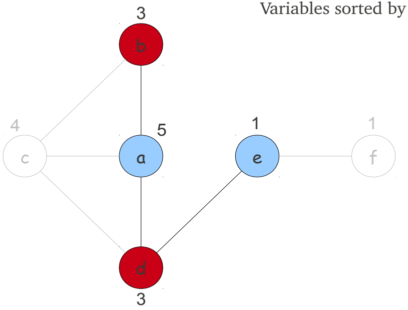
Clusters sorted by decreasing cost: cluster1, cluster3, cluster2

2 available registers



The cost of the allocation is **5**

How the clustering allocator works



Variables sorted by decreasing cost:

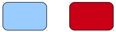
Cluster1: {a,e}

Cluster2: {c,f}

Cluster3: {b,d}

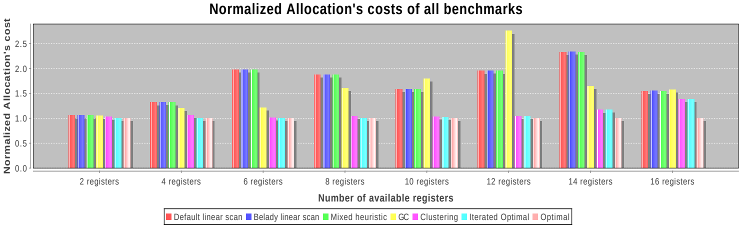
Clusters sorted by decreasing cost: cluster1, cluster3, cluster2

2 available registers



The cost of the allocation is **5**

Comparison about different allocators



Comparison about ILP-programs solving time

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

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

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?

Motivation 2/2

Reduce local-memory pressure

- through live range splitting: choice of decison 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)

```
//Nested within outer loops
for (i=0; i<N; i++)

  for (j=0; j<N; j++)
    C[i][j] = /* ... */;

F[0][0]=1;
F[0][1]=2;
...
...
F[2][1]=2;
F[2][2]=1;
```

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)

```
//Nested within outer loops
for (i=0; i<N; i++)

  for (j=0; j<N; j++)
    C[i][j] = /* ... */;

F[0][0]=1;
F[0][1]=2;
...
...
F[2][1]=2;
F[2][2]=1;
```

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)

```
//Nested within outer loops
for (i=0; i<N; i++)

  for (j=0; j<N; j++)
    C[i][j] = /* ... */;

F[0][0]=1;
F[0][1]=2;
...
...
F[2][1]=2;
F[2][2]=1;
```

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

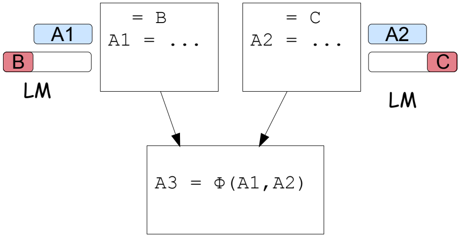
Pointer reconciliation

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

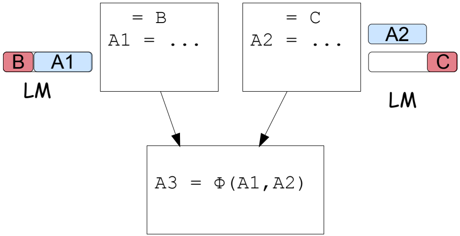


The approach 2/2

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

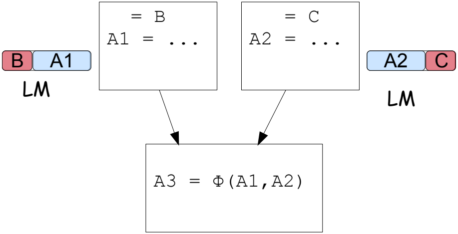


The approach 2/2

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

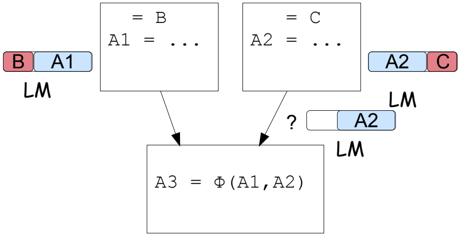


The approach 2/2

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

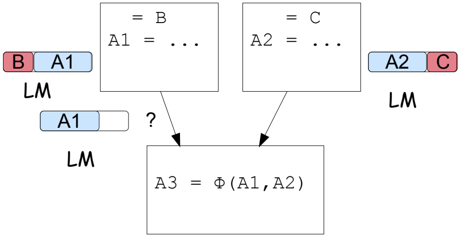


The approach 2/2

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

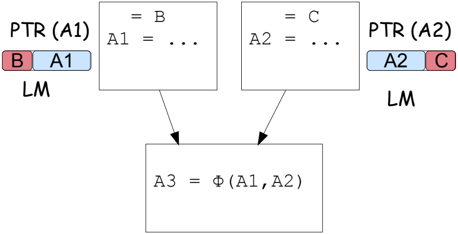


The approach 2/2

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

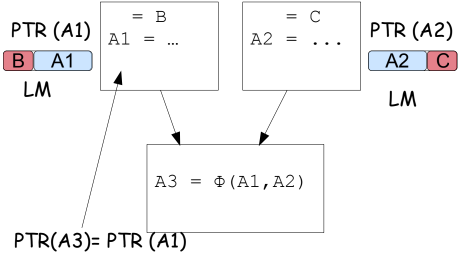


The approach 2/2

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

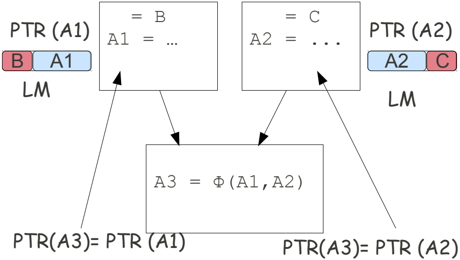


The approach 2/2

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



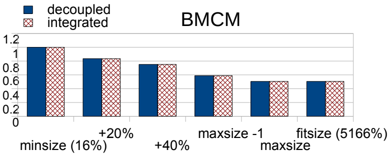
Outline

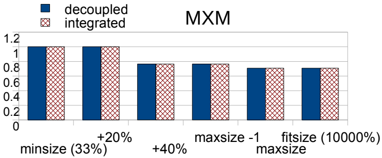
Benchmarks

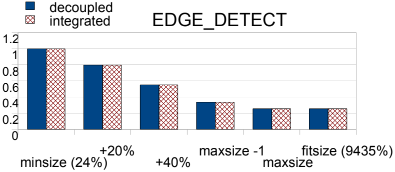
Benchmark	Brief description	Suite	Data size	arrays /blocks
Edge-Detect	Edge detection in an image	UTDSP	196644	4/385
D-FFT	256-point complex FFT	UTDSP	2032	7/7
Bmcm	Water molecular dynamics	Perfect Club	125240	10/310
MxM	Matrix multiplication	n.a.	120000	3/300

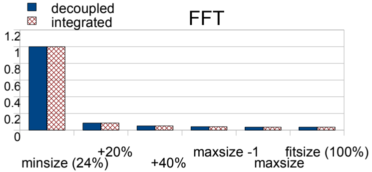
Constant	Latency
latency_LM	8
latency_MM	128
latency_move(s_V)	$8 + 2s_V$
latency_spill(s_V)	$128 + 4s_V$
latency_reload(s_V)	$128 + 4s_V$

Results


decoupled
integrated
BMCM
1.2
1
0.8
0.6
0.4
0.2
0
minsize (16%)
+20%
+40%
maxsize -1
maxsize
fitsize (5166%)


decoupled
integrated
MXM
1.2
1
0.8
0.6
0.4
0.2
0
minsize (33%)
+20%
+40%
maxsize -1
maxsize
fitsize (10000%)


decoupled
integrated
EDGE_DETECT
1.2
1
0.8
0.6
0.4
0.2
0
minsize (24%)
+20%
+40%
maxsize -1
maxsize
fitsize (9435%)


decoupled
integrated
FFT
1.2
1
0.8
0.6
0.4
0.2
0
minsize (24%)
+20%
+40%
maxsize -1
maxsize
fitsize (100%)

Outline

Local memory allocation and weighted Interval graph(WIG) coloring

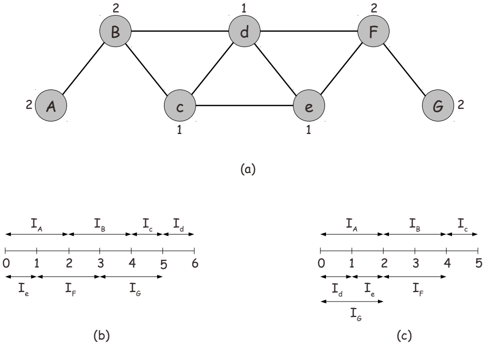
Linearized programs

- 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

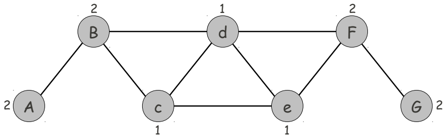
The shipbuilding problem

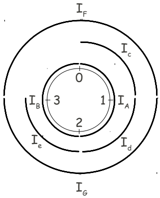


- Determining whether $\chi(G_w) \leq k$ is an NP-complete problem [Golumbic'04] even if G is an interval graph and w is restricted to the values 1 and 2

The submarine-building problem

- Assuming that loads and stores wrap around transparently we define the submarine building problem



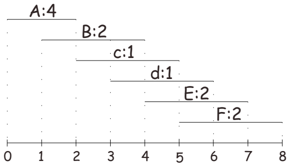


Complexity results of the submarine-building problem

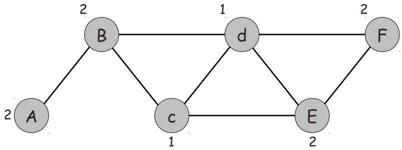
- Determining whether $\chi(G_w) \leq k$ is an NP-complete problem on interval graphs [Diouf'11]

- 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

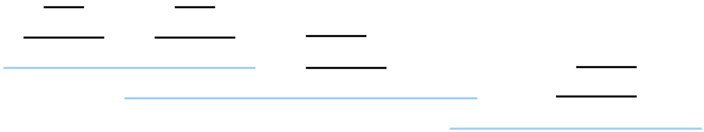


(a)



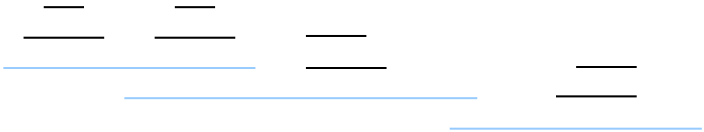
(b)

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

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

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