Bitwidth Aware Global Register Allocation

By Sriraman Tallam and Rajiv Gupta

Presented by Will Wendorf, Ben Stoler, and Bing Schaefer

Outline

- Motivation
- Implementation
  - Live Range Analysis
  - Packing Variables into Registers
- Results
- Evaluation

Motivation

Consider the following code. How many registers are need?

```
int main() {
    char a = ‘a’, b = ‘b’, c = ‘c’, d = ‘d’;
    cout << a << b << c << d << “\n”;
}
```

Motivation

Consider the following code. How many registers are need?

4 registers

```
int main() {
    char a = ‘a’, b = ‘b’, c = ‘c’, d = ‘d’;
    cout << a << b << c << d << “\n”;
}
```

Problems with Register Underutilization

- Having a large register file causes higher power usage
  - 10 - 25% more power
- Higher register pressure causes more memory spills
  - Causes extra load and stores
- May require multiple cycles to access physical register content

Motivation

Registers are underutilized if we constrain each register to one variable

- Especially with 64-bit, only fully utilized by doubles or long ints
- e.g. 63/64 bits wasted for boolean variables

Ex. for (int i = 0; i < 28; ++i) { cout << i << endl; }

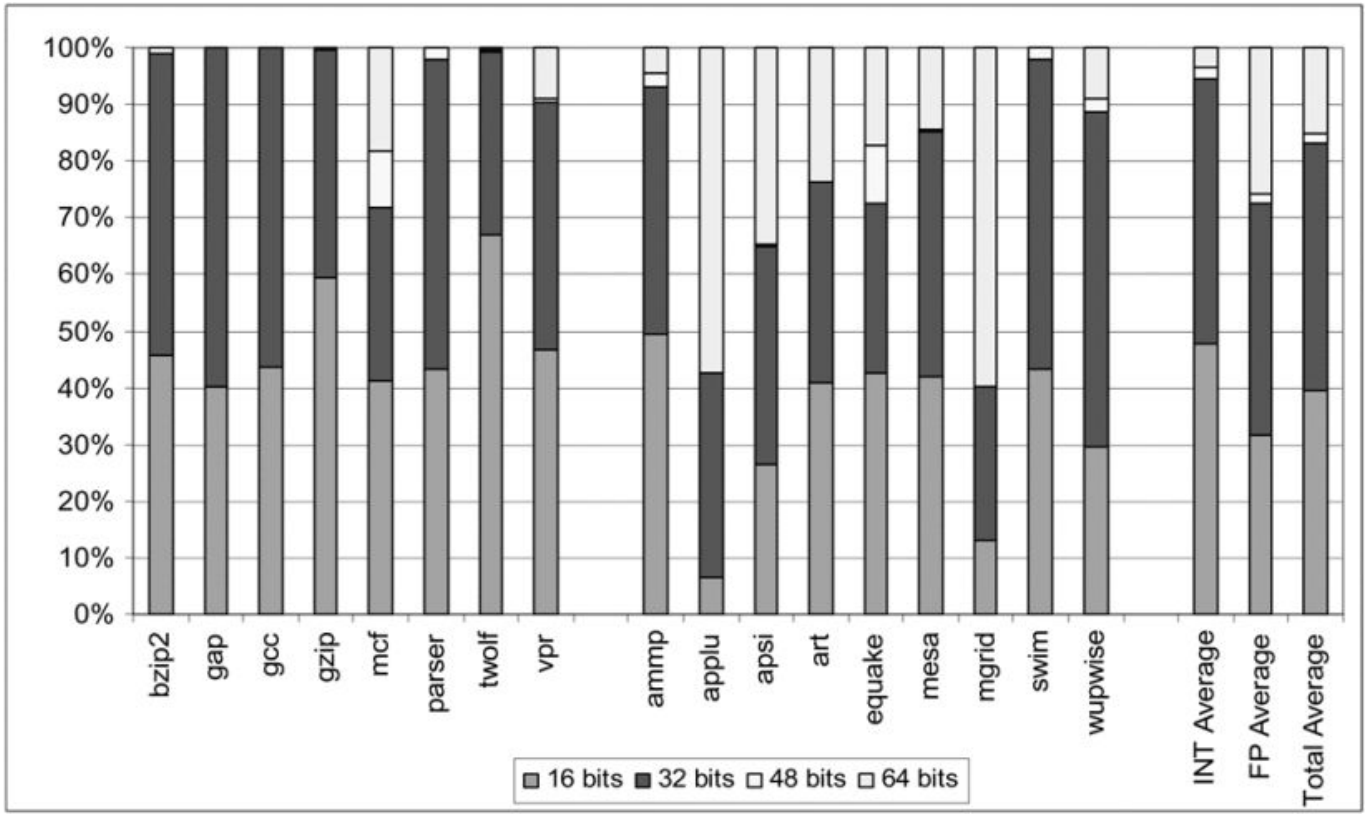
Motivation

Registers are underutilized if we constrain each register to one variable

- Especially with 64-bit, only fully utilized by doubles or long ints
- e.g. 63/64 bits wasted for boolean variables

Ex. for (int i = 0; i < 28; ++i) { cout << i << endl; }

SPEC 2000 Benchmark Suite Width Distribution



Solution? Register Packing!

Multiple results can be stored within a single physical register!

```
int main() {
    char a = ‘a’, b = ‘b’, c = ‘c’, d = ‘d’;
    cout << a << b << c << d << “\n”;
}
```

1 32-bit register:

‘a’	‘b’	‘c’	‘d’

Outline

- Motivation
- Implementation
  - Live Range Analysis
  - Packing Variables into Registers
- Results
- Evaluation

2 Steps of Register Packing

1. Live Range Analysis
    - Which bits are live at certain points?
2. Packing Variables into Registers
    - How do we assign variables into physical registers?

Live Range Analysis

Goal: Determine which bits in which variables are live at point *p*.

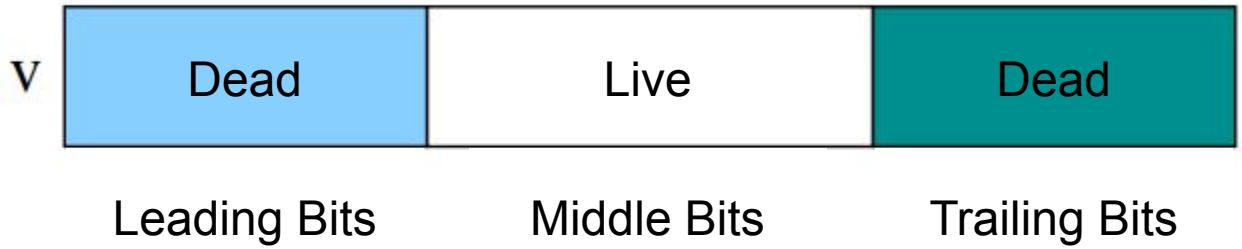
1. Apply forward analysis to find leading and trailing zero bits
2. Apply backward analysis to find dead bits

```
1 int A, B, C
2 A = 2147483647
3 C = A
4 A = A >> 16
5 B = A
```

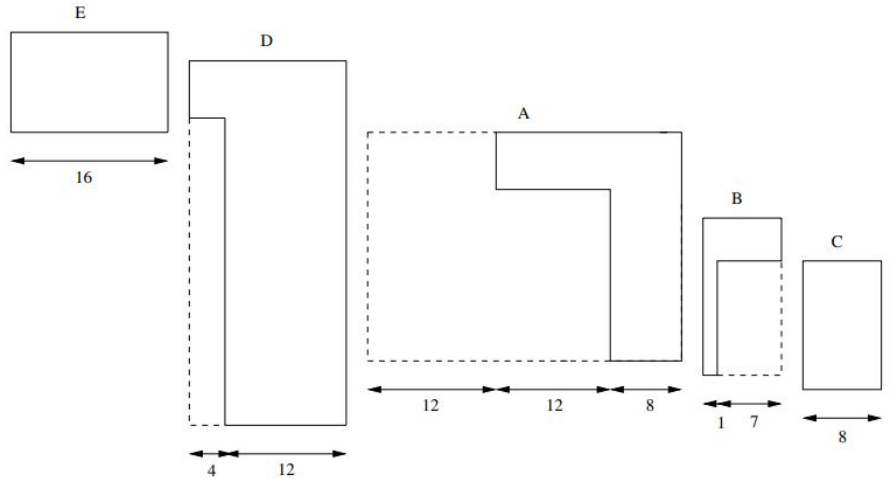
B and A only have 16 bits live after statement 5 while C has all 32 bits live.

Dead Bits and Live Range Width

- Dead bits: Bits that will not be used after a given point
- Live Range Width: Bits that are live at point p in a program.
  - Split into leading, middle and trailing sections
  - Usually due to bit shifts and bitwise operations



Live Range Analysis Example

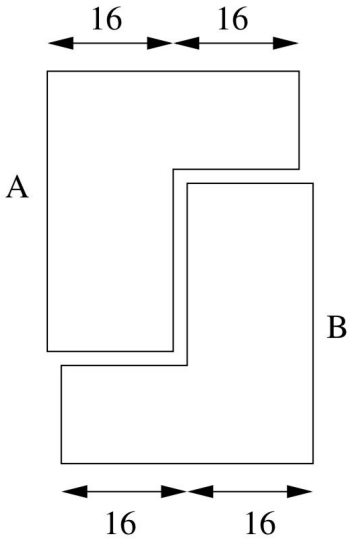
$int\ A;$ – 32 bits $short\ D, E;$ – 16 bits $char\ B, C;$ – 8 bits 1. $E = \ldots$ 2. $D = E + 1$ 3. $D = D >> 4$ 4. $A = (E << 4) \mid 0xf$ $this\ was\ E's\ last\ use.$ 5. $use\ A$ 6. $A = A >> 12$ 7. $B = \ldots$ 8. $C = (B \& 0x7f) + 1$ 9. $last\ use\ of\ A$ 10. $last\ use\ of\ B \& 0x80$ 11. $last\ use\ of\ C$ 12. $last\ use\ of\ D$	Zero Bit Sections 1. E: (0,0) 2. D: (0,0) 3. D: (4,0) 4. A: (12,0) 5. 6. A: (24,0) 7. B: (0,0) 8. C: (0,0) 9. 10. 11. 12.	Dead Bit Sections 0. A:B:C:D:E; $before\ 1$ 1. A:B:C:D 2. A:B:C 3. A:B:C:D(4,0) 4. A(12,0):B:C:D(4,0):E 5. A(12,0):B:C:D(4,0):E 6. A(24,0):B:C:D(4,0):E 7. A(24,0):C:D(4,0):E 8. A(24,0):B(0,7):D(4,0):E 9. A:B(0,7):D(4,0):E 10. A:B:D(4,0):E 11. A:B:C:D(4,0):E 12. A:B:C:D:E	

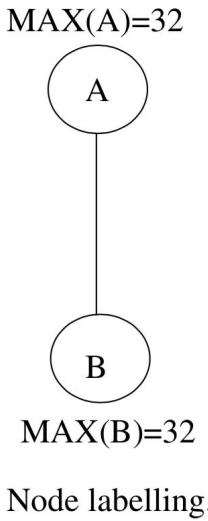
Packing Variables into Registers - Example

1. $E = \ldots$ 2. $D = E + 1$ 3. $D = D >> 4$ 4. $A = (E << 4) \mid 0xf$ $;this\ was\ E's\ last\ use.$ 5. $use\ A$ 6. $A = A >> 12$ 7. $B = \ldots$ 8. $C = (B \& 0x7f) + 1$ 9. $last\ use\ of\ A$ 10. $last\ use\ of\ B \& 0x80$ 11. $last\ use\ of\ C$ 12. $last\ use\ of\ D$	1. $R_{0..15} = \ldots$ 2. $R_{16..31} = R_{0..15} + 1$ 3. $R_{16..27} = R_{20..31}$ $\boxed{R_{20..31} = R_{16..27}}$ 4. $R_{4..19} = R_{0..15}; R_{0..3} = 0xf$ 5. $use\ R_{0..19}$ 6. $R_{0..7} = R_{12..19}$ 7. $R_{8..15} = \ldots$ $\boxed{R_{16} = R_{15}}$ 8. $R_{8..15} = R_{8..14} + 1$ 9. $last\ use\ of\ R_{0..7}$ 10. $last\ use\ of\ R_{16}$ 11. $last\ use\ of\ R_{8..15}$ 12. $last\ use\ of\ R_{20..31}$	1. $R1_{0..15} = \ldots$ 2. $R1_{16..31} = R1_{0..15} + 1$ 3. $R1_{16..27} = R1_{20..31}$ 4. $R2_{4..19} = R1_{0..15}; R2_{0..3} = 0xf$ 5. $use\ R2_{0..19}$ 6. $R2_{0..7} = R2_{12..19}$ 7. $R2_{8..15} = \ldots$ 8. $R2_{16..23} = R2_{8..14} + 1$ 9. $last\ use\ of\ R2_{0..7}$ 10. $last\ use\ of\ R2_{15}$ 11. $last\ use\ of\ R2_{16..23}$ 12. $last\ use\ of\ R1_{16..27}$
Original code.	Code using one register.	Code using two registers.

Packing Variables into Registers - Algorithm

- Shape of live ranges determine if the live ranges can be coalesced
- Node labeling: take maximum bitwidth of the variable in its live range
- Edge labeling: take maximum bitwidth of each node in their interfering range



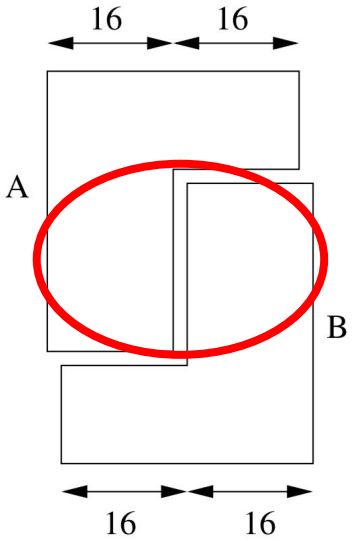


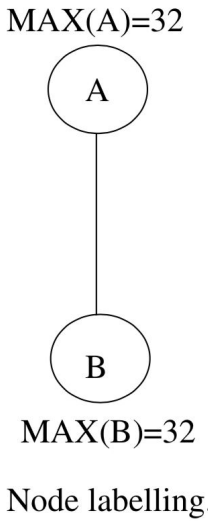
Node labelling.

Edge labelling.

Packing Variables into Registers - Algorithm

- Shape of live ranges determine if the live ranges can be coalesced
- Node labeling: take maximum bitwidth of the variable in its live range
- Edge labeling: take maximum bitwidth of each node in their interfering range



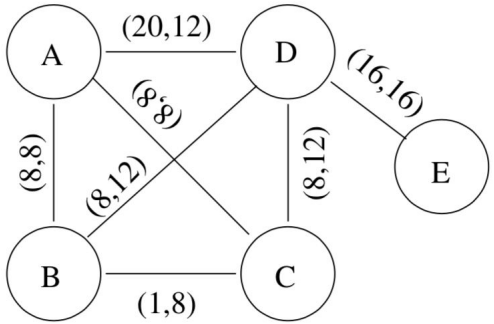


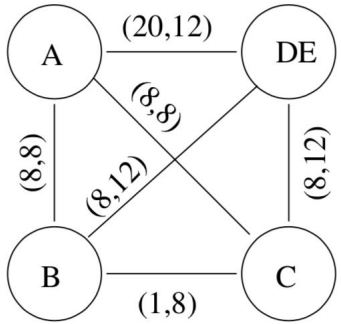
Node labelling.

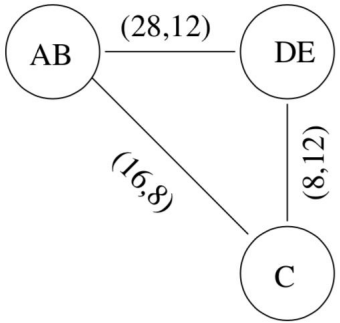
Edge labelling.

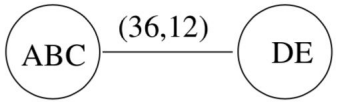
Packing Variables into Registers - Algorithm

- Coalesce sets of nodes in the interference graph
- Priority based - only coalesce if colorability is not worsened
- Trial and error approach for detecting if worsened









Outline

- Motivation
- Implementation
  - Live Range Analysis
  - Packing Variables into Registers
- Results
- Evaluation

Results

Benchmark Function	Number of Nodes	
	Before	After
adpcm_decoder	17	15
adpcm_coder	20	17
g721.update	22	15
g721.fmult	8	7
g721.quantize	8	6
thres.memo	6	4
thres.coalesce	10	5
thres.homogen	12	7
thres.clip	7	5
SoftFloat.mul32To64	14	12
NewLife.main	8	5
MotionTest.main	9	7
Bubblesort.main	15	11
Histogram.main	13	11
crc.main	12	11
dh.encodelastquantum	8	5

Number of live ranges	Live Range Widths (bits)	
	Declared Size	Max Size After BSA
adpcm.decoder		
1	32	4
2	32	1
adpcm.coder		
2	32	1
1	32	8
1	32	3
g721.update		
7	16	16
1	32	3
4	16	5
1	16	1
1	8	1
g721.fmult		
3	16	16
1	16	12
2	16	4

Results - Coalescing

- BU = bitwidth unaware, one variable per register
- NC = naive coalescing, uses declared width of variables
- OC = “our” coalescing, using BSA analysis

Benchmark Function	Registers Used		
	BU	NC	OC
adpcm_decoder	15	15	13
adpcm_coder	18	18	15
g721.update	15	12	10
g721.fmult	4	3	3
g721.quantize	6	5	5
thres.memo	6	6	4
thres.coalesce	10	10	5
thres.homogen	11	11	6
thres.clip	6	6	4
SoftFloat.mul32To64	8	7	7
NewLife.main	7	7	4
MotionTest.main	6	6	5
Bubblesort.main	9	9	7
Histogram.main	7	7	6
crc.main	10	10	9
dh.encodelastquantum	7	7	4

Evaluation

- Seems promising, especially as a conservative approach
  - Could be furthered using value profiling and a speculative approach
- Further optimization possible, as in Nandivada and Barik which allows for optimal register packing in polynomial time
- Focused primarily on embedded processor ISAs which have instructions for directly accessing bit sections
  - For more general ISAs, would have to insert extra packing/unpacking masking code

Conclusion

- Underutilization of registers is common
  - Most variables do not use their declared bitwidth
- We can thus have variables share a register via packing
- This can prevent spills to memory and reduce register pressure