Exploiting Value Locality in Physical Register Files

Saisanthosh Balakrishnan Guri Sohi
University of Wisconsin-Madison

36th Annual International Symposium on Microarchitecture

Motivation

More in-flight instructions (ILP, SMT)

Need **more** physical registers

Increase in area, access time, power

Optimized Design

Access locality: Hierarchical and register caches

Communication locality: Banked and clustered design

Optimized Storage

Late allocation: Virtual-physical registers

Value locality: Physical register reuse

Reduce storage requirements:

1. Exploit register value locality
2. Simplify for common values

Outline

- **The property:** Value locality in register file
- Optimized storage schemes
- Results
- Conclusion

Value Locality

Locality of the **results** produced by **all** dynamic instructions

1. Identify the most common results
2. Locality in the results produced (register writes)
3. Duplication in register file

Value Locality

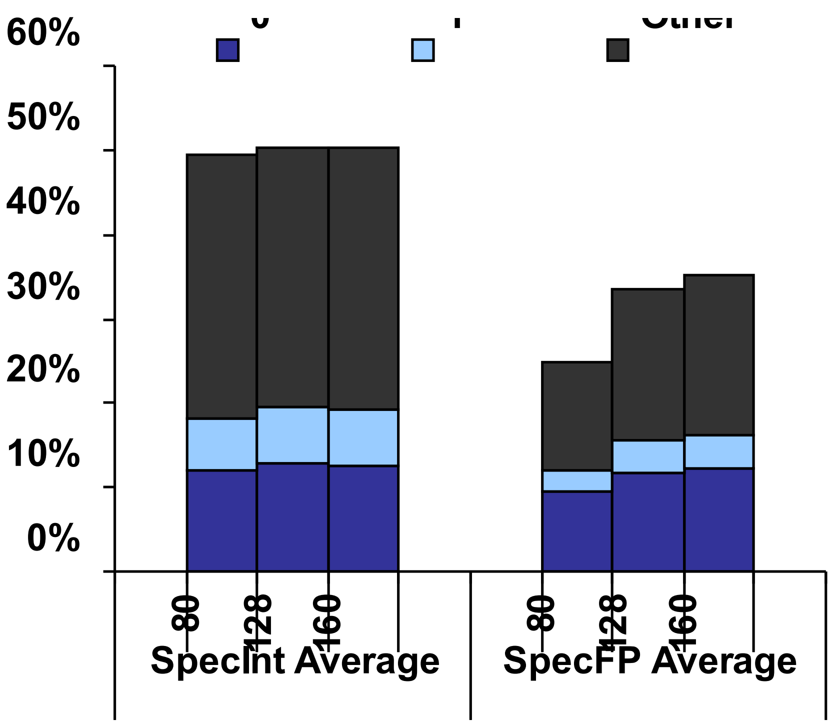
10 most common values in some SPEC CPU2000 benchmarks

bzip2	crafty	gap	gcc	gzip	mcf	ammp	art
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
4831843632	7	2	4	4	5368778784	5368710000	2
5368712376	3	81	4831839248	32	5	2560	4.59187e+18
62914560	5369648560	5	3	2	4831863760	1884672376	4.58549e+18
65536	8	5369767936	52	3	10	3584	3
5368712432	2	8	-1	5368758224	32	6656	5370448344
32	5369777344	3	59	16	2	5632	32
2	6	4	7	-1	49	48	7
3	5368862128	16	5	8	-1	14848	10

0 and 1 are most common results on all benchmarks

Value Locality

Locality in all results produced (register writes)

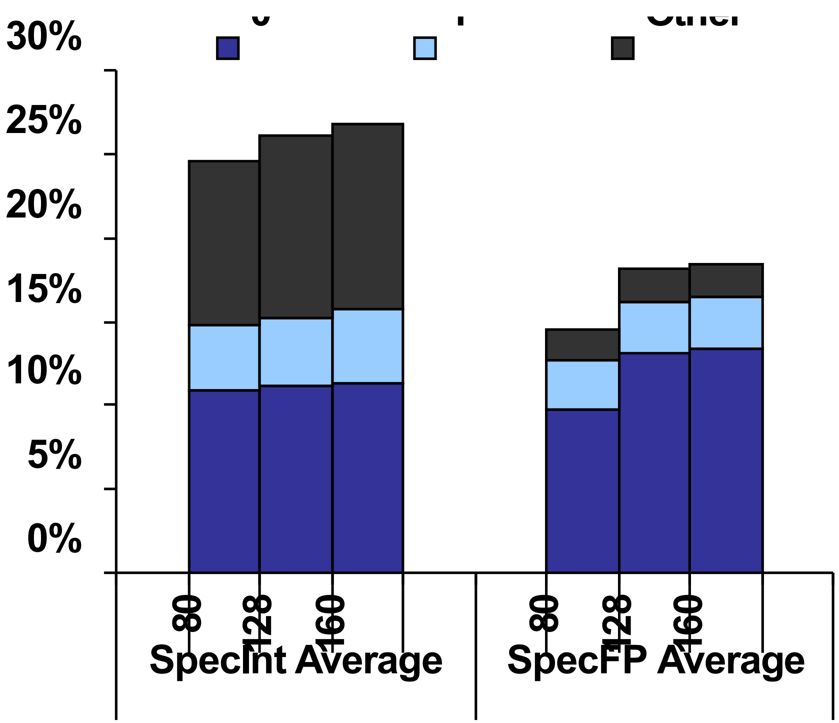


- Consider valid registers only
- Many registers could hold the same value
- Duplication in register file

Percentage of values written to registers
already present in the physical register file (80, 128, 160 regs.)

Value Locality

Duplication of values in physical register file



- Value being held in **`n`** registers → **`(n-1)`** duplicate registers
- Number of duplicate registers depends on register lifetime
- **`60%`** to **`85%`** of duplication because of 0 and 1

# of duplicate registers =
# of valid physical registers – # of unique values in the register file

Observations

1. Many physical registers hold same value

Reuse physical registers

Instructions with same result mapped to one physical register

First proposed by [Jourdan et al. MICRO-31]

Reduced storage requirements

2. 0's and 1's are most common results

Optimized storage for common values

Registerless Storage

Extended Registers and Tags

Simplified micro-architectural changes

Outline

- Value locality in register file
- **Exploiting it:** Optimized storage schemes
  - Physical Register Reuse
  - Registerless Storage
  - Extended Registers and Tags
- Results
- Conclusion

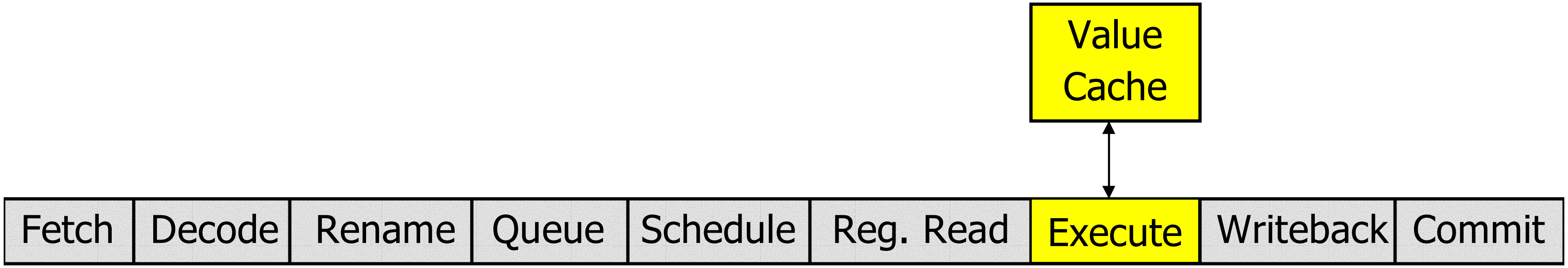
Physical Register Reuse

Many instructions with same result map to one physical register

1. Conventional renaming of destination register
2. On execution, detect if result present in register file
3. Free assigned physical register
4. Remap instruction’s destination register
5. Handle dependent instructions

Register file with unique values → More free registers

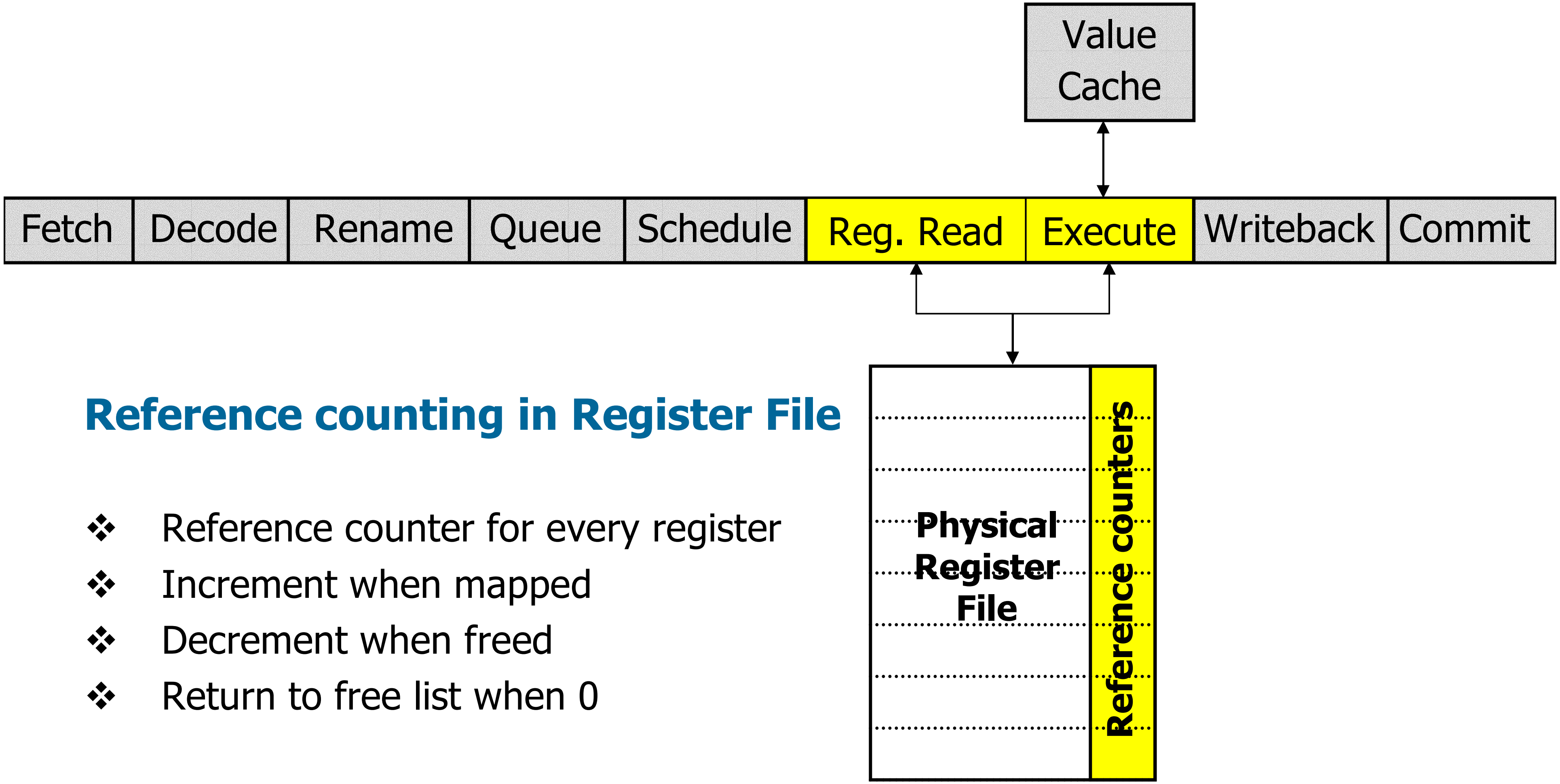
Physical Register Reuse



Value cache – to detect duplicate results

- Maps physical register tags to values
- CAM structure, returns tag for a value
- Actions to invalidate / add entries

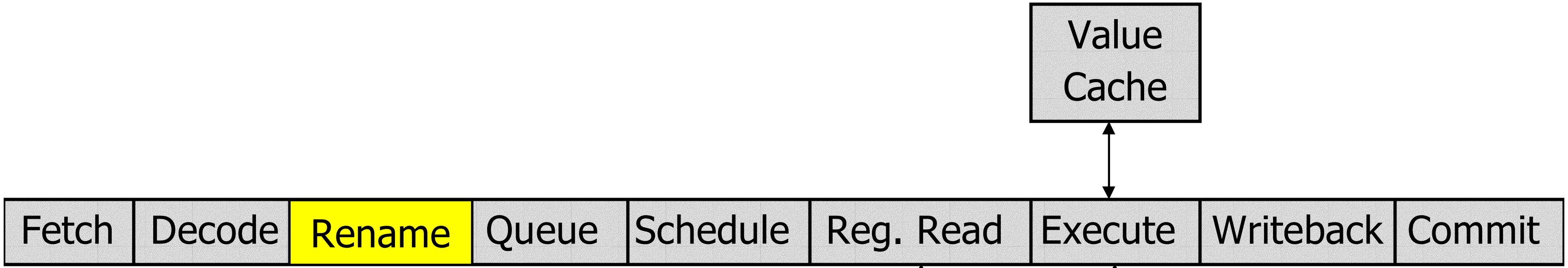
Physical Register Reuse



Reference counting in Register File

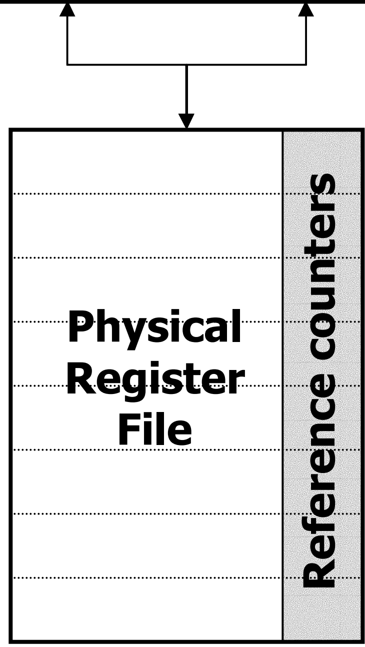
- Reference counter for every register
- Increment when mapped
- Decrement when freed
- Return to free list when 0

Physical Register Reuse

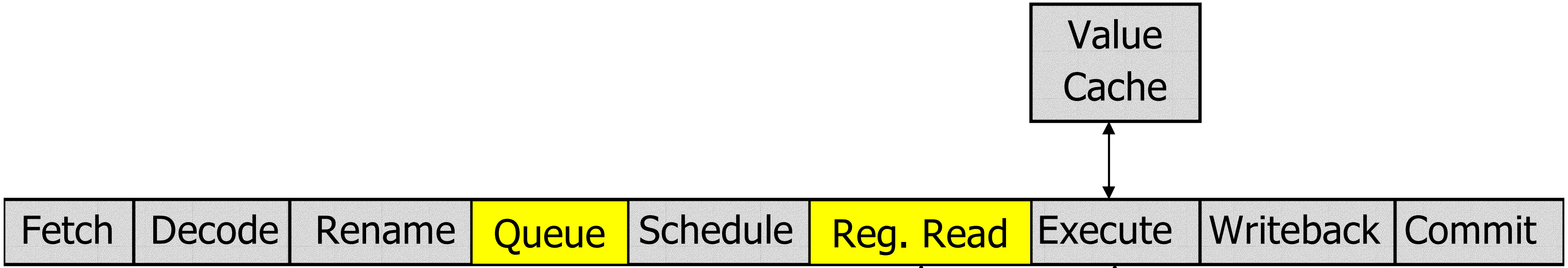


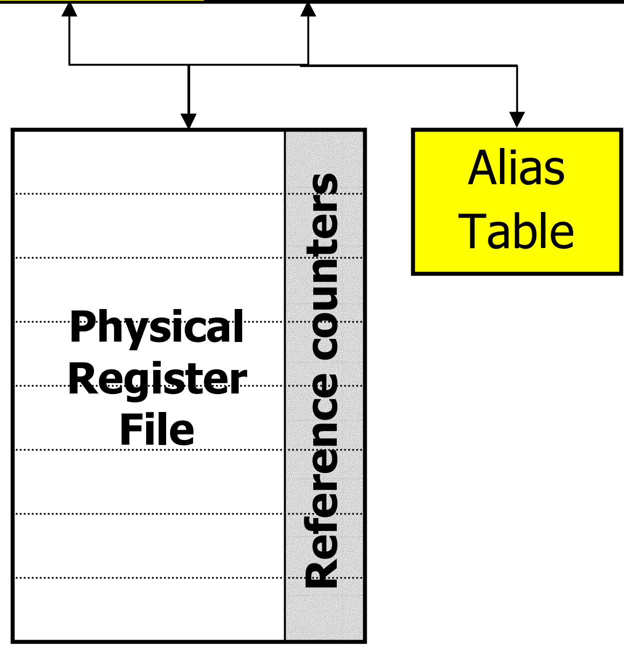
Handling dependent instructions

- Update rename map. Remap instruction's destination register



Physical Register Reuse





Handling dependent instructions

- Re-broadcast new destination register tag
- Re-broadcast **`<invalid_src_tag>`**. Lookup **alias table** on register read

Physical Register Reuse

- Reduced register requirements
- Avoids register write of duplicate values
- Non-trivial micro-architectural changes
  - Value Cache lookup, Alias Table indirection, Reference counts
  - Recovering from exceptions
- Remapping of destination register requires re-broadcast

Outline

- Value locality in register file
- Optimized storage schemes
  - Physical register reuse
  - Registerless Storage (0’s & 1’s)
  - Extended Registers and Tags (0’s & 1’s)
- Results
- Conclusion

Registerless Storage

Exploit common value locality – **state bits for 0 and 1**

1. Conventional renaming of destination register
2. On execution, detect if result is 0 or 1
3. Free assigned physical register
4. Remap instruction's destination register to reserved tags
5. Communicate value directly to dependent instructions

Register file without 0s and 1s ➔ More free registers

Registerless Storage

- Simplified micro-architectural changes
  - No Value Cache, Alias Table, Reference counts
- No registers for 0 and 1: Reduced register requirements
- Eliminates register reads and writes of 0 and 1
- Remapping of destination register requires re-broadcast

Optimize storage for common values.
But, **avoid remapping** destination register tag

Extended Registers and Tags

- Associate physical register with 2-bit extension
  - V: Valid and D: data = {0, 1}

- Rename: Assign physical register with its extension (if available)
- Execute: If result is 0 or 1
  - Use extension, if available. Free physical register.
  - Physical register can be assigned to some other instruction

Most 0's and 1's in register extensions

Extended Register and Tags

Extended tagging scheme eliminates remapping

Tag management

- Increase tag (**N**-bits) namespace by **1**-bit (MSB)
- Unmodified free list
- To assign a tag:
  1. Get tag from free list
  2. Get MSB from the corresponding extension's valid bit
- MSB = 0 ➔ {register, extension}
- MSB = 1 ➔ {register}

Extended Registers and Tags

Extended Registers and Tags

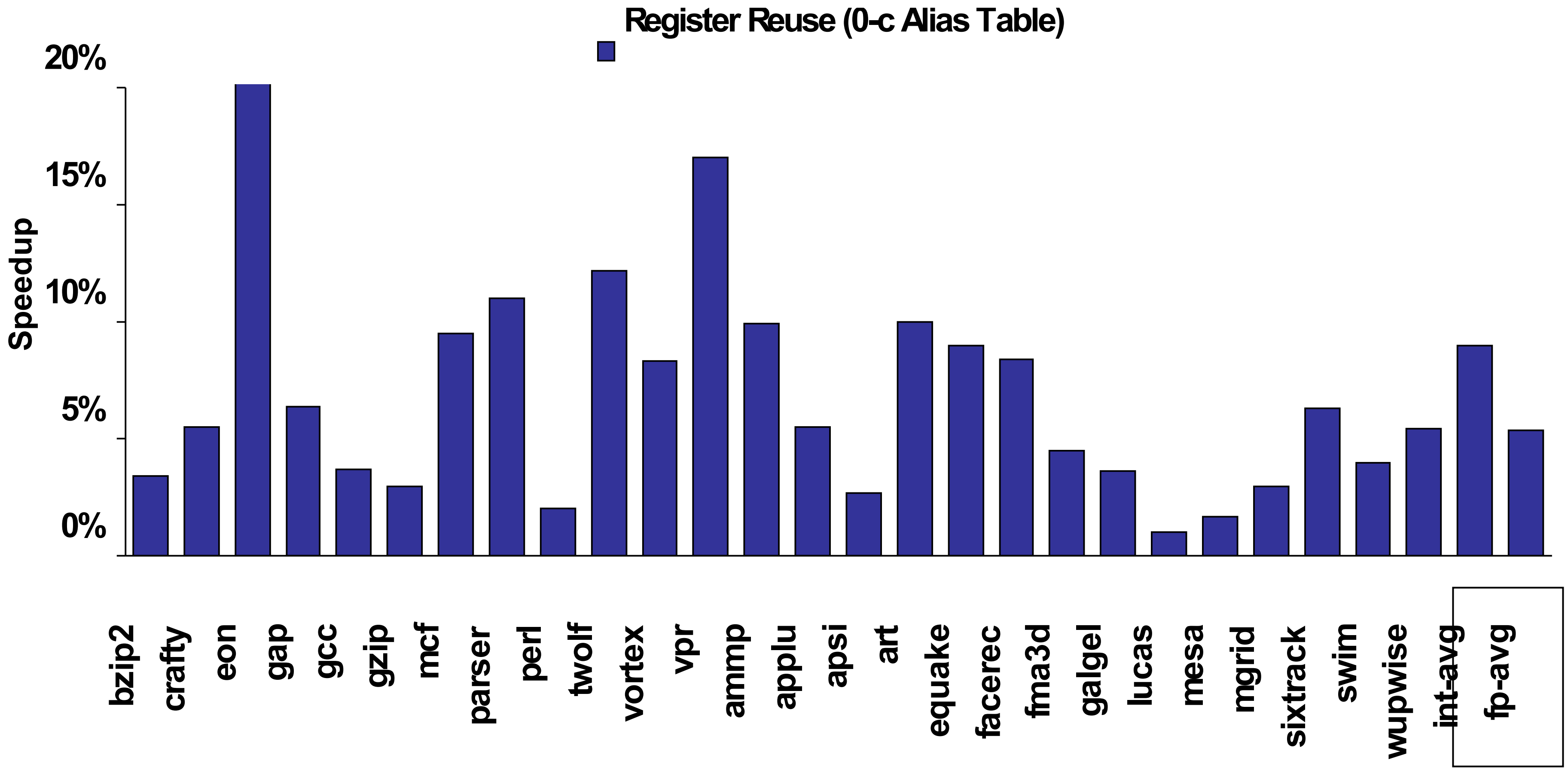
- Simplified micro-architectural changes
- Extended registers hold 0’s and 1’s
- Better design
- Some common values use physical registers

Outline

- Value locality
- Optimized storage schemes
- **Results**
  - Performance – more in-flight instructions
  - Benefit – smaller register file
  - Reduced register traffic
- Conclusion

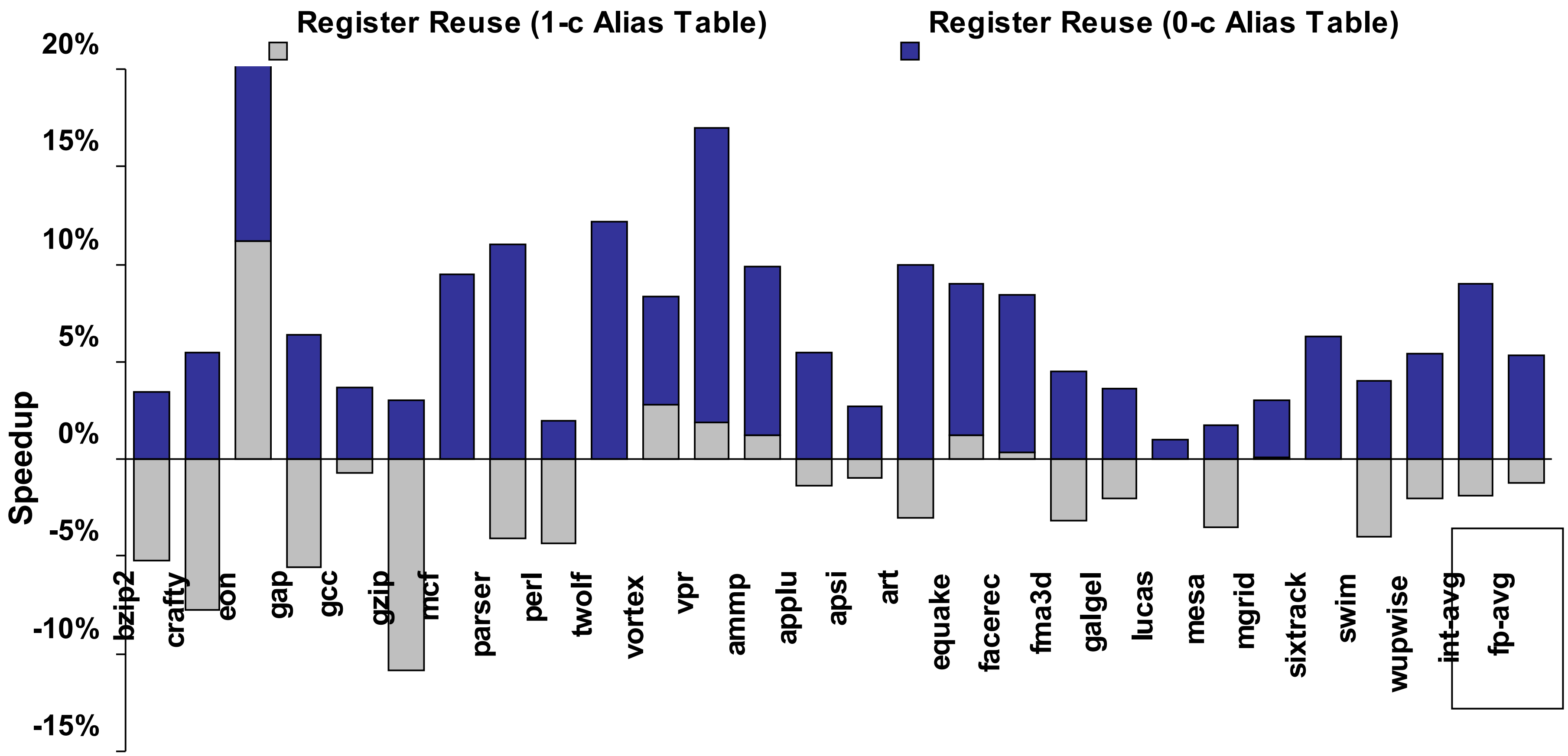
Performance

Relative performance improvement with increased instruction window



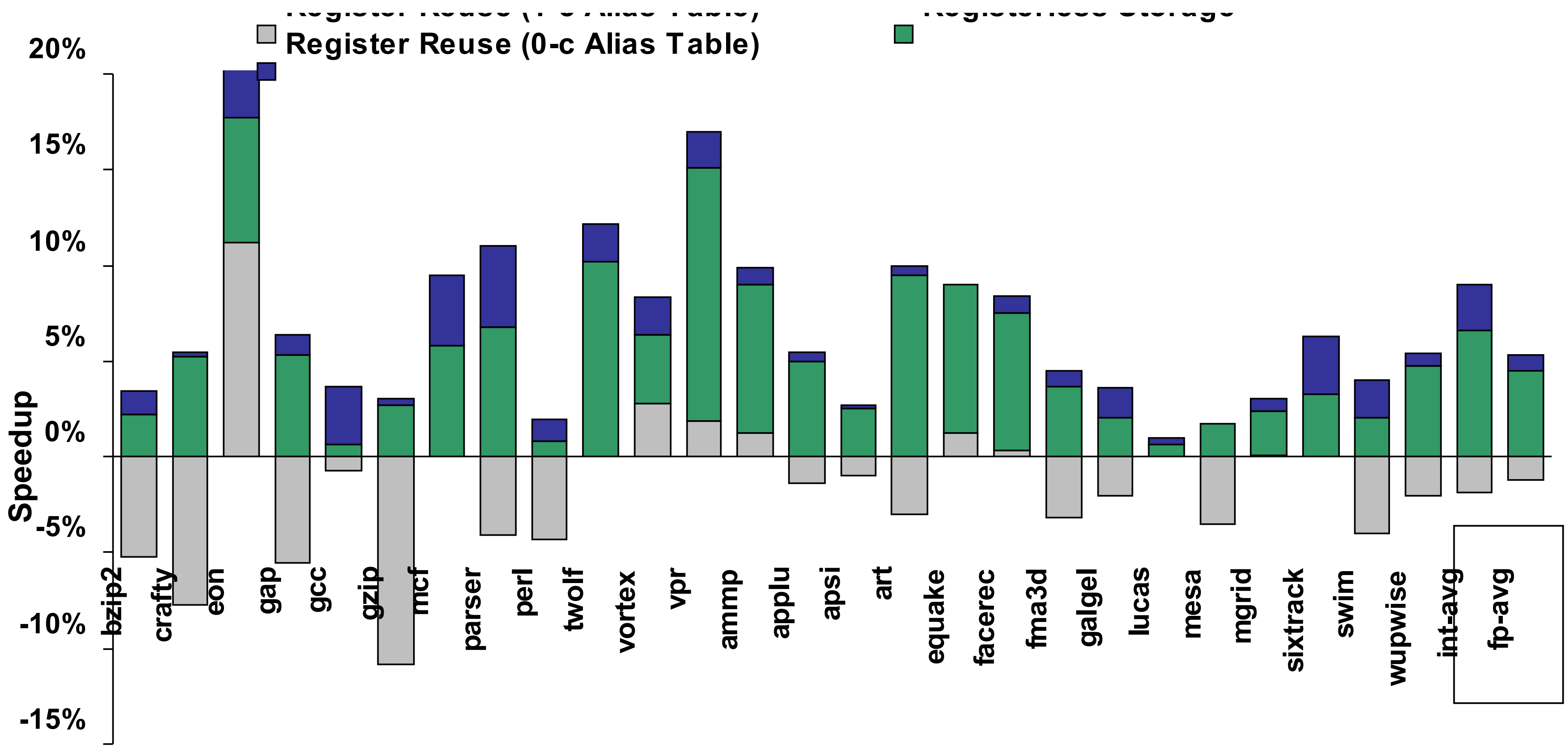
Performance

Relative performance improvement with increased instruction window



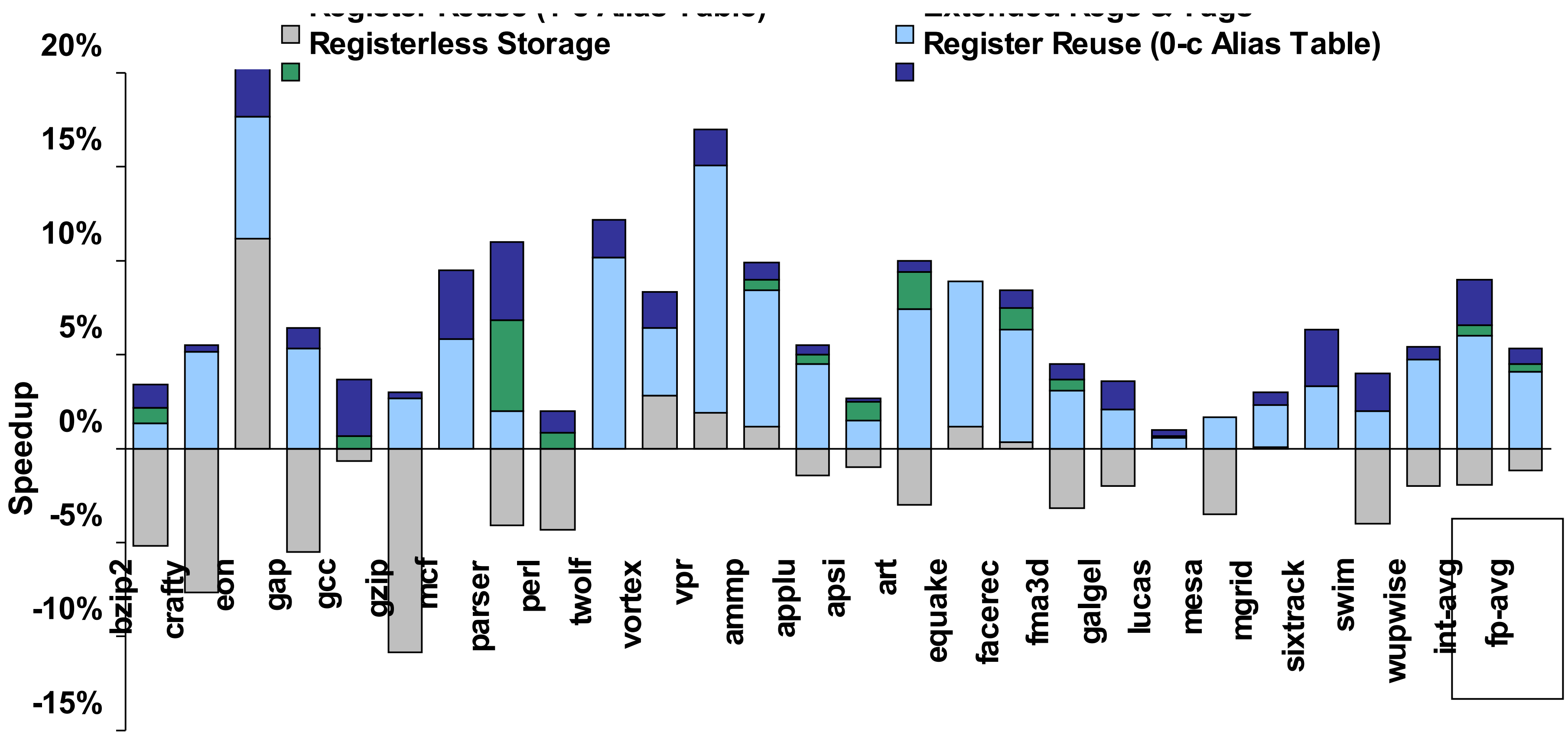
Performance

Relative performance improvement with increased instruction window



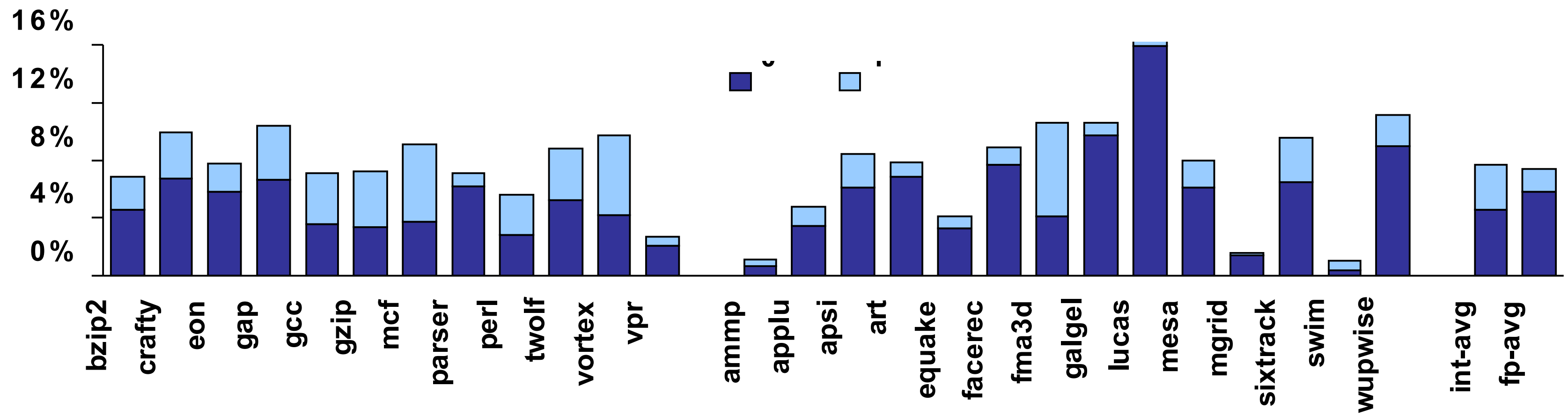
Performance

Relative performance improvement with increased instruction window

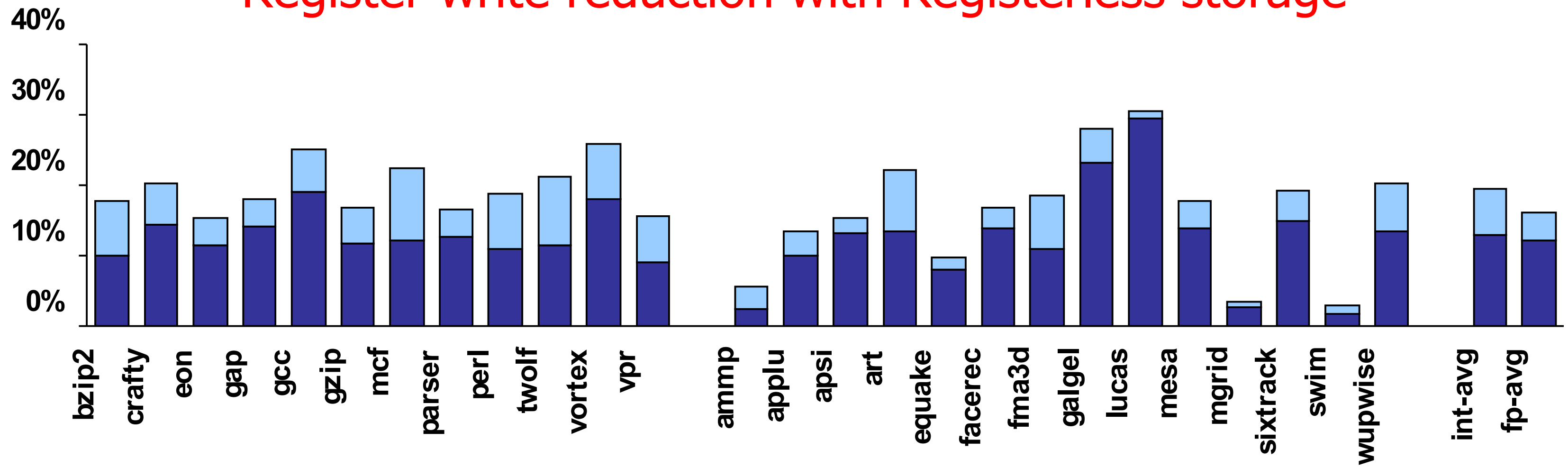


Register Traffic

Register read reduction with Registerless storage



Register write reduction with Registerless storage



Conclusions

- Value locality
  - Observe duplication in physical register file
  - Significant common value locality
- Optimization schemes
  - Physical Register Reuse
  - 0's and 1's: Registerless Storage and Extended Registers and Tags
- Benefits
  - Reduced register requirements
  - Reduced register traffic
  - Power savings, better design

Questions

Comparison

	Physical Reg. Reuse	Reg.less storage	ERT
Detect	Value cache	Identify {0, 1}	Identify {0, 1}
Exploit	Reuse register holding the same value. Free reg.	Free register	Write 0 or 1 to extension, if available. Free register
Handle dep instns	Update rename map. Alias table or Re-broadcast	Update rename map. Re-broadcast {0, 1}	
Handle exception	Recover ref. counts, alias table, value cache		
Outcome	Reg. File with unique values	Reg. File without 0, 1	0, 1 in extensions

Simulation Methodology

- Alpha ISA. SimpleScalar based.
  - Nops removed at the front-end
  - Register r31 not included in value locality measurements
- Detailed out-of-order simulator
  - 4-wide machine, 14-stage pipeline, 1-cycle register file access
  - Base case: 128 physical registers, 256 entry instruction window
  - Instruction and data caches
    - 64KB, 2-way set associative, 64-byte line size, 3-cycle access
    - L2 is 2MB unified with 128-byte line size, 6-cycle access
- Benchmarks
  - SPEC CPU2000 suite (12 int and 14 fp benchmarks)
  - Reference inputs run to completion

Performance

Relative performance improvement with increased instruction window