Improving Program Efficiency by Packing Instructions into Registers

Stephen Hines, Joshua Green, Gary Tyson, David Whalley

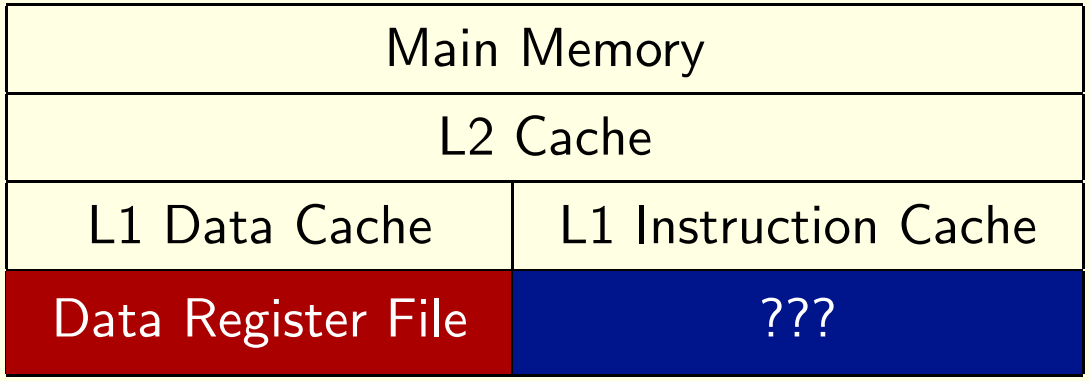
Computer Science Dept.
Florida State University

June 7, 2005

◆ Introduction

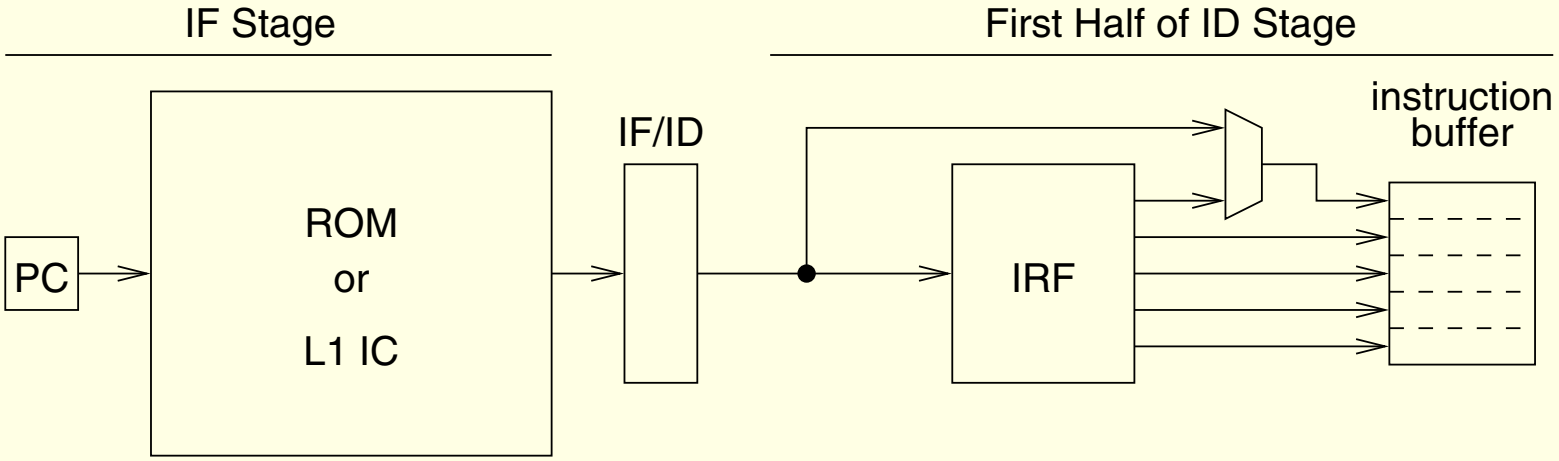
- Embedded Processor Design Constraints
  - **Power Consumption**
  - **Static Code Size**
  - **Execution Time**
- Fetch logic consumes **36%** of total processor power on StrongARM
  - Instruction Cache (IC) and/or ROM — Lower power than a large memory store, but still a fairly large, flat storage method
- Instruction encodings can be **wasteful** with bits
  - Nowhere near theoretical compression limits
  - Maximize functionality, but simplify decoding (fixed length)
  - Most applications only apply a subset of available instructions

◆ Access of Data & Instructions



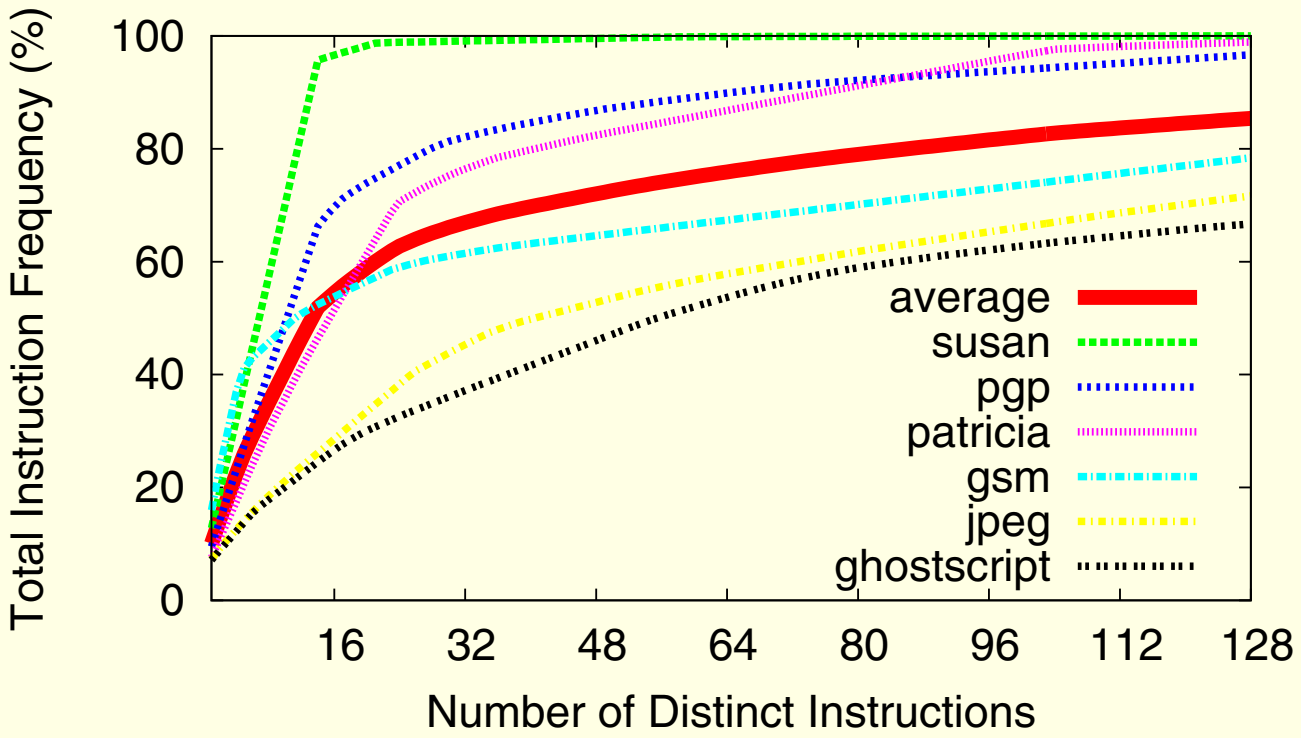
- Each lower layer is designed to improve accessibility of current/frequent items, albeit at a reduction in number of available items
- Caching is beneficial, but compilers can do better for the "most frequently" accessed data items (e.g. **Register Allocation**)
- Instructions have no analogue to the **Data Register File** (RF)

◆ Instruction Register File — IRF



- Stores frequently occurring instructions as specified by the compiler (potentially in a partially decoded state)
- Allows multiple instruction fetch with packed instructions

Dynamic Instruction Redundancy

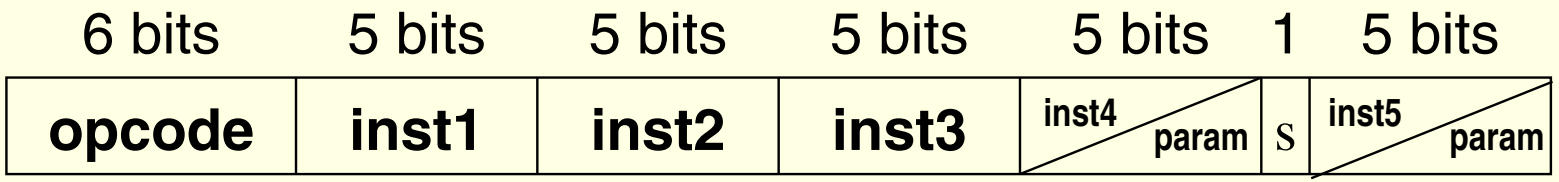


- Profiling the largest benchmark in each category of MiBench
- 32-entry IRF can capture 66.51% of all dynamic instructions executed on average

◆ ISA Modifications

- MIPS ISA — commonly known and provides simple encoding
- **RISA** (Register ISA) — instructions available via IRF access
- **MISA** (Memory ISA) — instructions available in memory
  - Create new instruction formats that can reference multiple RISA instructions — **Tightly Packed**
  - Modify original instructions to be able to pack an additional RISA instruction reference — **Loosely Packed**
- Increase packing abilities
  - **Parameterization**
  - **Positional Register Specifiers**

◆ Tightly Packed Instruction Format



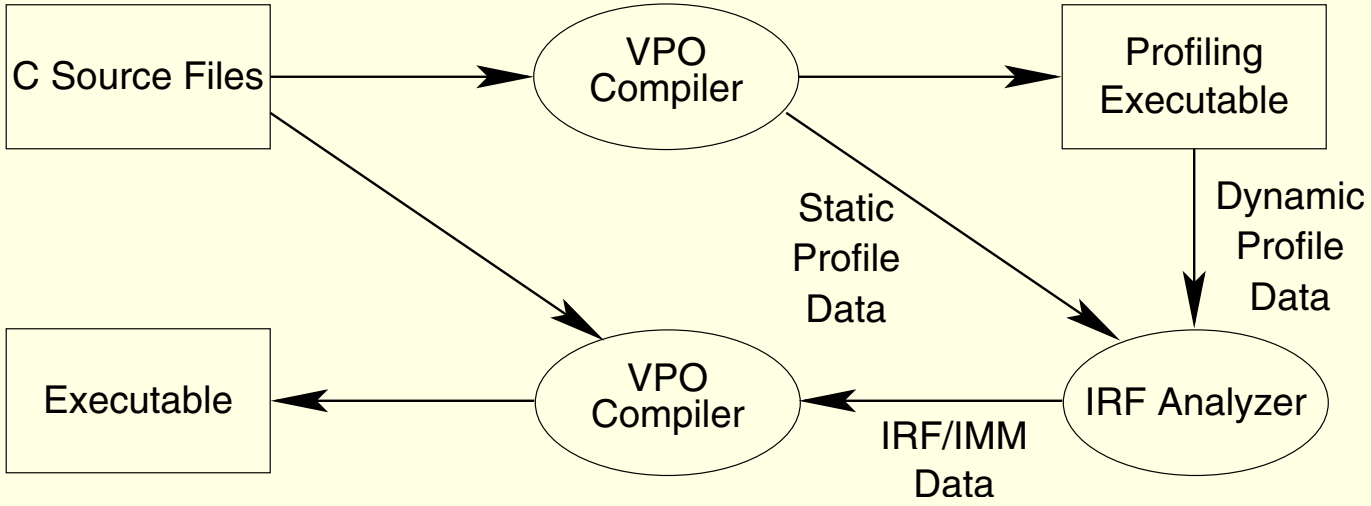
- New opcodes for this T-format of MISA instructions
- Supports sequential execution of up to 5 RISA instructions from the IRF
  - Unnecessary fields are padded with *nop*
- Supports up to 2 parameters replacing instruction slots
  - Parameters can come from 32-entry **Immediate Table** (IMM)
  - Each IRF entry retains a default immediate value as well
  - Branches use these 5-bits for displacements

◆ Positional Register Specifiers

#	RTL	RTL (positional)
1	`r[2]=R[r[29]+4];`	`r[2]=R[r[29]+4];`
2	`r[2]=r[2]+r[5];`	**`s[0]=s[0]`**`+r[5];`
3	`R[r[29]+4]=r[2];`	`R[`**`u[2]`**`+4]=`**`s[0]`**`;`
	...	...
4	`r[3]=R[r[29]+4];`	`r[3]=R[r[29]+4];`
5	`r[3]=r[3]+r[5];`	**`s[0]=s[0]`**`+r[5];`
6	`R[r[29]+4]=r[3];`	`R[`**`u[2]`**`+4]=`**`s[0]`**`;`

- Abstract out common register usage patterns (e.g. load/add/store)
- Increases code redundancy, so greater opportunity for compression
- Positional register values can be obtained via modifications to standard pipeline register forwarding logic

◆ Compiler Modifications



- **VPO** — Very Portable Optimizer targeted for SimpleScalar MIPS/Pisa
- IRF-resident instructions are selected by a greedy algorithm using profile data including parameterization/positional hints
- Iterative packing process using a sliding window to allow branch displacements to slip into (5-bit) range

Instruction Register File

#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

Original Code Sequence

```
lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8
```

Instruction Register File

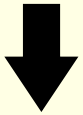
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

Original Code Sequence

```
lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8
```



Marked IRF Sequence

```
lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)
```

Instruction Register File

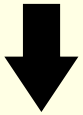
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

Original Code Sequence

lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8



Marked IRF Sequence

lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)

Instruction Register File

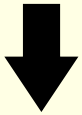
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

Original Code Sequence

lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8



Marked IRF Sequence

lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)

Instruction Register File

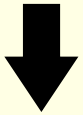
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

Original Code Sequence

lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8



Marked IRF Sequence

lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)

Instruction Register File

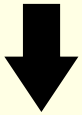
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

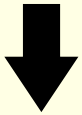
Original Code Sequence

```
lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8
```



Marked IRF Sequence

```
lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)
```



Packed Code Sequence

```
lw r[3], 8(r[29]) {4}
```

Instruction Register File

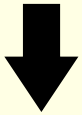
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

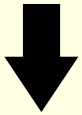
Original Code Sequence

lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8



Marked IRF Sequence

lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)



Packed Code Sequence

lw r[3], 8(r[29]) {4}
param3_AC {1,3,2} {3,–5}

Instruction Register File

#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

#	Value
...	...
3	32
4	63
...	...

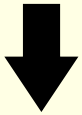
Encoded Packed Sequence

opcode	rs	rt	immediate	irf
lw	29	3	8	4

opcode	inst1	inst2	inst3	param	s	param
param3_AC	1	3	2	3	1	–5

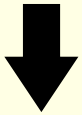
Original Code Sequence

lw r[3], 8(r[29])
andi r[3], r[3], 63
addiu r[5], r[3], 32
addu r[5], r[5], r[4]
beq r[5], r[0], –8



Marked IRF Sequence

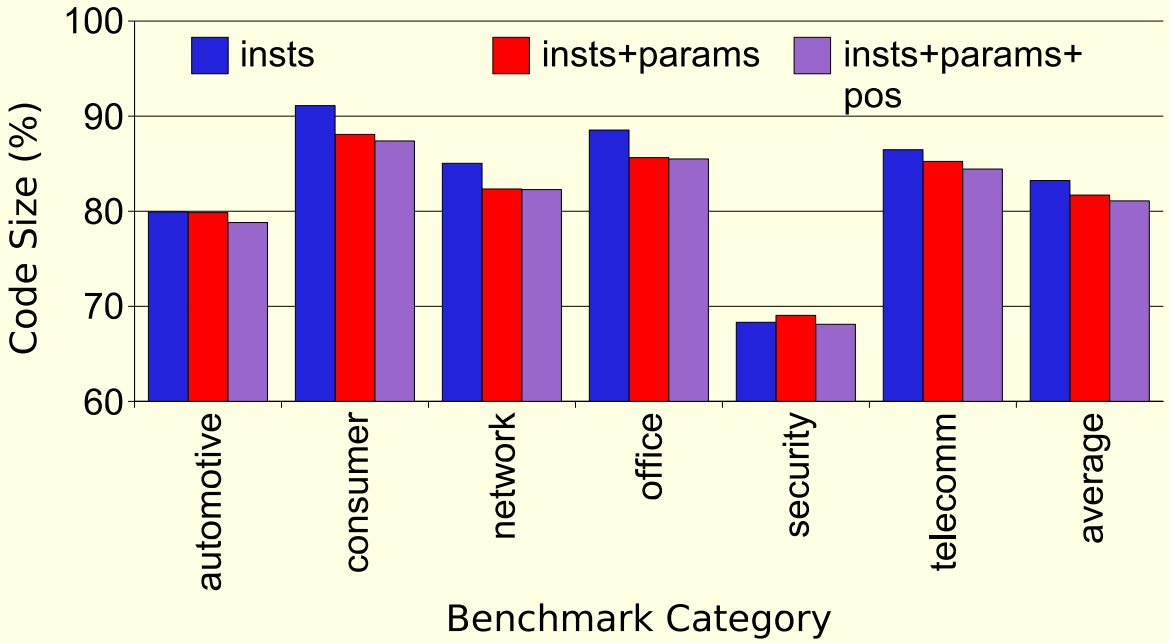
lw r[3], 8(r[29])
IRF[4], default (4)
IRF[1], param (3)
IRF[3]
IRF[2], param (branch –8)



Packed Code Sequence

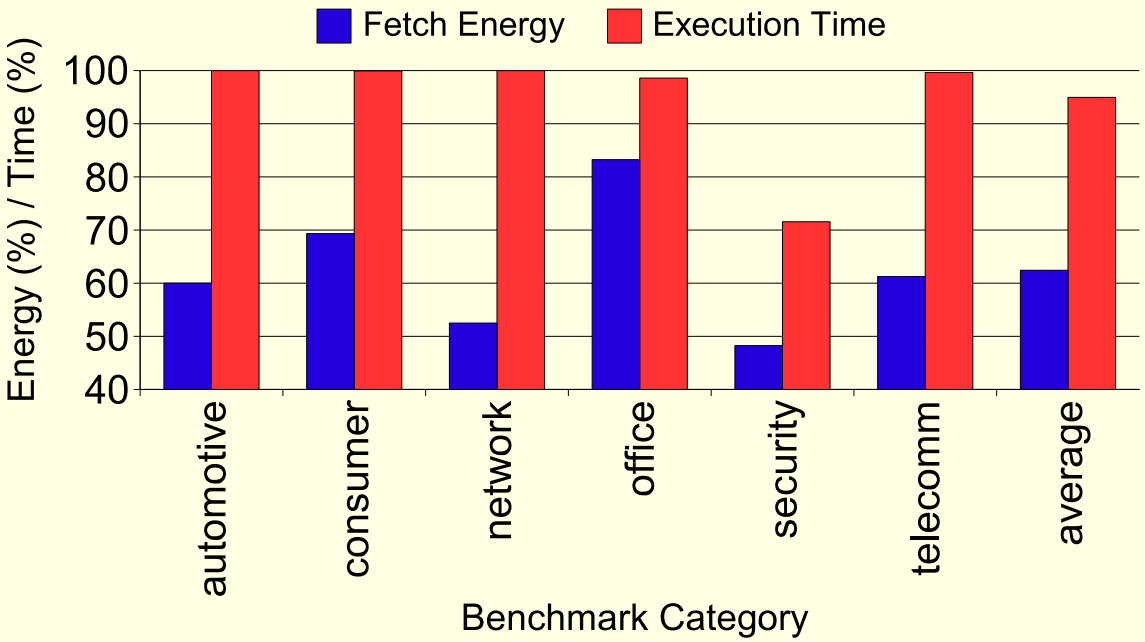
lw r[3], 8(r[29]) {4}
param3_AC {1,3,2} {3,–5}

◆ Reducing Static Code Size



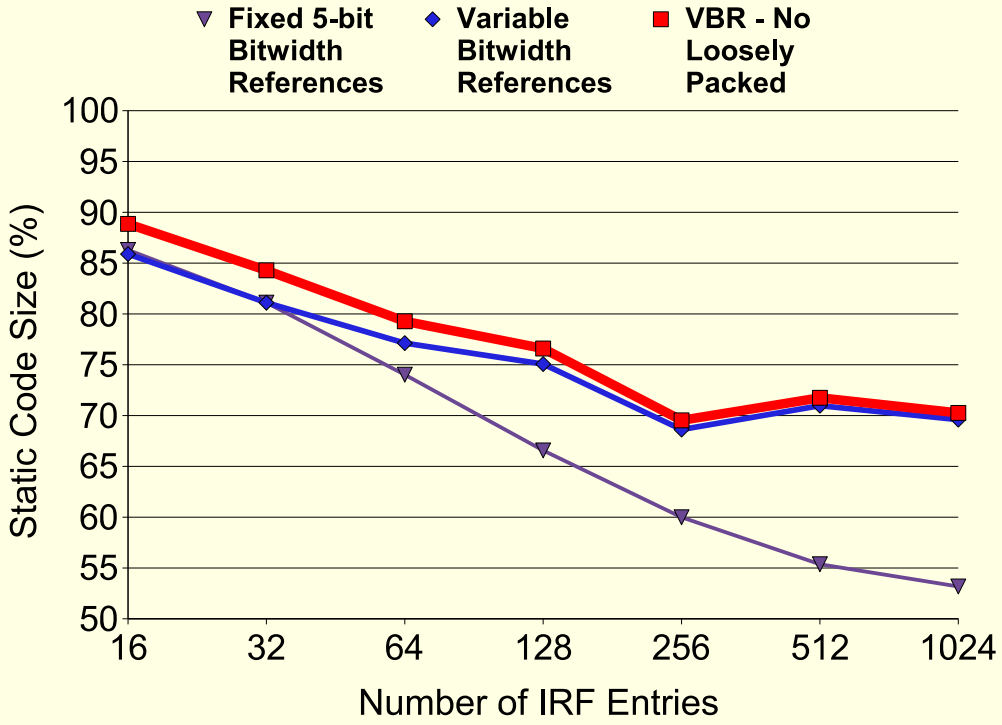
- 32-entry IRF Impact on Code Size
  - 83.23% ← Packing instructions alone
  - 81.70% ← Packing instructions with params
  - 81.09% ← Packing instructions with params and positional registers

◆ Reducing Fetch Energy & Exec. Time



- Sim-panalyzer used to gather energy data alongside SimpleScalar
- IC access is > 100 times as costly as IRF access
- 55% of instructions fetched from 32-entry IRF $\sim$ 37% reduction in energy
- Fewer cycles due to improved cache effects and fetch rate

IRF Static Code Size Sensitivity



- Pack sizes can differ with IRF size (e.g. *tight5* & *param4* not available for $>$ 32 entries; *tight4* & *param3* not available for $>$ 64 entries; . . .)

- Static code size decreases when packing with a larger IRF until reduced pack sizes overwhelm the benefit of greater entries

◆ Crosscutting Issues

- Context switching — Must preserve IRF, IMM and positional registers as part of process state
  - Pointer to routine for loading IRF for each particular process
  - Only restore IRF/IMM, never save; positional registers need to be saved/restored
- Exceptions — How to restart execution of a packed instruction?
  - Keep track of how many RISA instructions have completed already
  - Store a bitmask of completed instructions for improved restart

◆ Related Work

Technique	Code Size Reduction	Power Savings	Speed	Hardware Complexity
Proc. Abs.	+	−	−	Minimal
L0	0	++	− −	Minimal
Echo	++	−	+/−	Easy
ZOLB/Loop Cache	0	+	+	Easy
IRF	**++**	**++**	**+**	**Easy**
Codewords	++	−	?/− −	Moderate
Arm/Thumb	++	−	− −	Moderate
Arm/Thumb/AX	++	?/−	− −	Moderate
Heads and Tails	++	?/−	?/−	Moderate
DISE	++	?/+	+	Difficult
Mini-graphs	++	?/−	+	Difficult

Legend: + means that improvement is $< 10\%$. ++ means that improvement is $\geq 10\%$. 0 means that there is very little to no effect. ? means that results are speculative since they are not presented or explained in detail. − means that penalty is $< 10\%$. − − means that penalty is $\geq 10\%$. Hardware complexity is scaled from easy (no changes) to difficult (complete redesign).

◆ Related Work

Technique	Code Size Reduction	Power Savings	Speed	Hardware Complexity
Proc. Abs.	+	−	−	**Minimal**
L0	0	++	− −	Minimal
Echo	++	−	+/−	**Easy**
ZOLB/Loop Cache	0	+	+	Easy
IRF	++	++	+	**Easy**
Codewords	++	−	?/− −	Moderate
Arm/Thumb	++	−	− −	Moderate
Arm/Thumb/AX	++	?/−	− −	Moderate
Heads and Tails	++	?/−	?/−	Moderate
DISE	++	?/+	+	Difficult
Mini-graphs	++	?/−	+	Difficult

Legend: + means that improvement is $< 10\%$. ++ means that improvement is $\geq 10\%$. 0 means that there is very little to no effect. ? means that results are speculative since they are not presented or explained in detail. − means that penalty is $< 10\%$. − − means that penalty is $\geq 10\%$. Hardware complexity is scaled from easy (no changes) to difficult (complete redesign).

◆ Related Work

Technique	Code Size Reduction	Power Savings	Speed	Hardware Complexity
Proc. Abs.	+	−	−	Minimal
L0	0	++	− −	Minimal
Echo	++	−	+/−	Easy
ZOLB/Loop Cache	0	+	+	Easy
IRF	**++**	**++**	**+**	**Easy**
Codewords	++	−	?/− −	Moderate
Arm/Thumb	**++**	**−**	**− −**	**Moderate**
Arm/Thumb/AX	**++**	**?/−**	**− −**	**Moderate**
Heads and Tails	**++**	**?/−**	**?/−**	**Moderate**
DISE	++	?/+	+	Difficult
Mini-graphs	++	?/−	+	Difficult

Legend: + means that improvement is $< 10\%$. ++ means that improvement is $\geq 10\%$. 0 means that there is very little to no effect. ? means that results are speculative since they are not presented or explained in detail. − means that penalty is $< 10\%$. − − means that penalty is $\geq 10\%$. Hardware complexity is scaled from easy (no changes) to difficult (complete redesign).

◆ Related Work

Technique	Code Size Reduction	Power Savings	Speed	Hardware Complexity
Proc. Abs.	+	–	–	Minimal
L0	**0**	**++**	**– –**	**Minimal**
Echo	++	–	+/–	Easy
ZOLB/Loop Cache	**0**	**+**	**+**	**Easy**
IRF	**++**	**++**	**+**	**Easy**
Codewords	++	–	?/– –	Moderate
Arm/Thumb	++	–	– –	Moderate
Arm/Thumb/AX	++	?/–	– –	Moderate
Heads and Tails	++	?/–	?/–	Moderate
DISE	++	?/+	+	Difficult
Mini-graphs	++	?/–	+	Difficult

Legend: + means that improvement is $< 10\%$. ++ means that improvement is $\geq 10\%$. 0 means that there is very little to no effect. ? means that results are speculative since they are not presented or explained in detail. – means that penalty is $< 10\%$. – – means that penalty is $\geq 10\%$. Hardware complexity is scaled from easy (no changes) to difficult (complete redesign).

◆ Related Work

Technique	Code Size Reduction	Power Savings	Speed	Hardware Complexity
Proc. Abs.	+	−	−	Minimal
L0	0	++	− −	Minimal
Echo	++	−	+/−	Easy
ZOLB/Loop Cache	0	+	+	Easy
IRF	**++**	**++**	**+**	**Easy**
Codewords	**++**	**−**	**?/− −**	**Moderate**
Arm/Thumb	++	−	− −	Moderate
Arm/Thumb/AX	++	?/−	− −	Moderate
Heads and Tails	++	?/−	?/−	Moderate
DISE	++	?/+	+	Difficult
Mini-graphs	++	?/−	+	Difficult

Legend: + means that improvement is $< 10\%$. ++ means that improvement is $\geq 10\%$. 0 means that there is very little to no effect. ? means that results are speculative since they are not presented or explained in detail. − means that penalty is $< 10\%$. − − means that penalty is $\geq 10\%$. Hardware complexity is scaled from easy (no changes) to difficult (complete redesign).

◆ Future Work

- Compiler Enhancements
  - Dynamic loading of IRF entries (or windows similar to SPARC RF)
  - Improved packing algorithms
  - Predication support
- Hardware Enhancements
  - Split compression of opcodes and operands in RISA
  - Decouple MISA and RISA by developing a split ISA
    - ⋆ MISA facilitating code size reduction with traditional compression
    - ⋆ RISA focusing on improved execution time and energy usage

◆ Conclusions

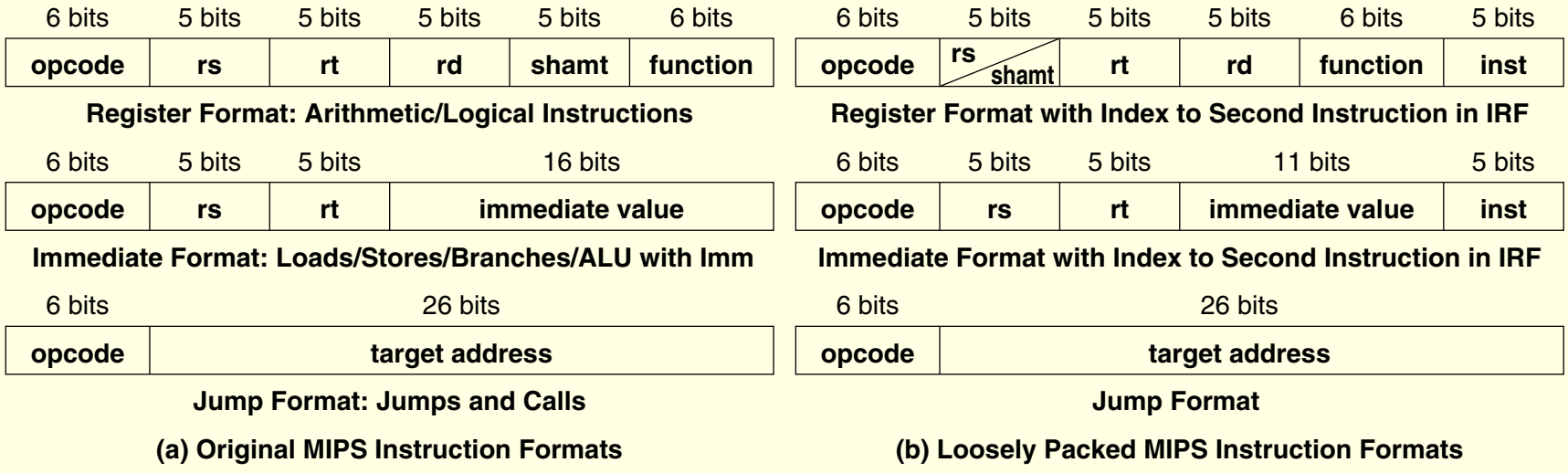
- **Instruction Register File** provides an improved fetch mechanism
- Focus is on common/frequently accessed instructions, similar to RF, enabling the compiler to promote instructions
- Rare combination compiler/hardware optimization that can yield improvements in all 3 performance metrics
  - Static code size reductions of $\sim 20\%$
  - Fetch energy reduced 37% (total energy $\sim 15\%$)
  - Execution time reduced 5% due to better IC behavior

◆ The End

Thank you!

Questions ???

◆ MIPS Instruction Format Modifications



- Creating Loosely Packed Instructions
  - R-type: Removed *shamt* field and merged with *rs*
  - I-type: Shortened immediate values (16-bit $\rightarrow$ 11-bit)
    - $\star$ *Lui* now uses 21-bit immediate value, hence no loose packing
  - J-type: Unchanged

◆ Selecting IRF-Resident Instructions


```
Read in instruction profile (static or dynamic);
Calculate the top 32 immediate values for I-type instructions;
Coalesce all I-type instructions that match based on parameterized immediates;
Construct positional and regular form lists from the instruction profile, along with conflict information;
IRF[0] ← nop;
foreach i ∈ [1..31] do
    Sort both lists by instruction frequency;
    IRF[i] ← highest freq instruction remaining in the two lists;
    foreach conflict of IRF[i] do
        Decrease the conflict instruction frequencies by the specified amounts;
```

- Greedy heuristic for selecting instructions to reside in IRF
- Can mix static and dynamic profiles together now to obtain good compression and good local packing

◆ Coalescing Similar Instructions

Opcode	rs	rt	immed	prs	prt	Freq
addiu	r[3]	r[5]	**1**	s[0]	NA	400
addiu	r[3]	r[5]	**4**	s[0]	NA	300
addiu	r[7]	r[5]	1	s[0]	NA	200
...						
⇓ Coalescing Immediate Values ⇓						
addiu	**r[3]**	r[5]	1	s[0]	NA	700
addiu	**r[7]**	r[5]	1	s[0]	NA	200
...						
⇓ Grouping by Positional Form ⇓						
addiu	NA	r[5]	1	s[0]	NA	900
...						
⇓ Actual RTL ⇓						
r[5]=s[0]+1						900

- Semantically equivalent and commutative instructions are converted into single recognizable forms to aid in detecting code redundancy

◆ Packing Instructions

Name	Description
tight5	5 IRF instructions (no parameters)
tight4	4 IRF instructions (no parameters)
param4	4 IRF instructions (1 parameter)
tight3	3 IRF instructions (no parameters)
param3	3 IRF instructions (1 or 2 parameters)
tight2	2 IRF instructions (no parameters)
param2	2 IRF instructions (1 or 2 parameters)
loose	Loosely packed format
none	Not packed (or loose with nop)

- Instructions are packed only within a basic block
- A sliding window of instructions is examined to determine which packing (if any) to apply
- Branches can move into range (5-bits) due to packing, so we repack iteratively in an attempt to obtain greater packing density