Improving the Energy and Execution Efficiency of a Small Instruction Cache by Using an Instruction Register File

Stephen Hines, Gary Tyson, David Whalley

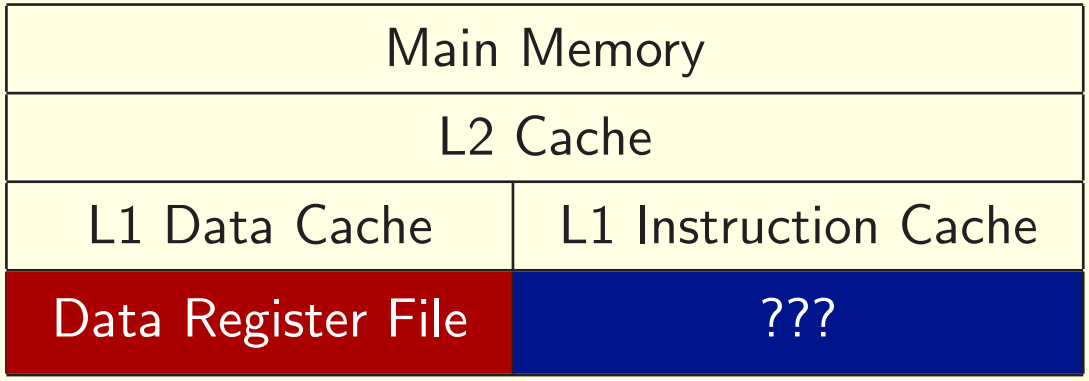
Computer Science Dept.
Florida State University

September 30, 2005

❶ Introduction

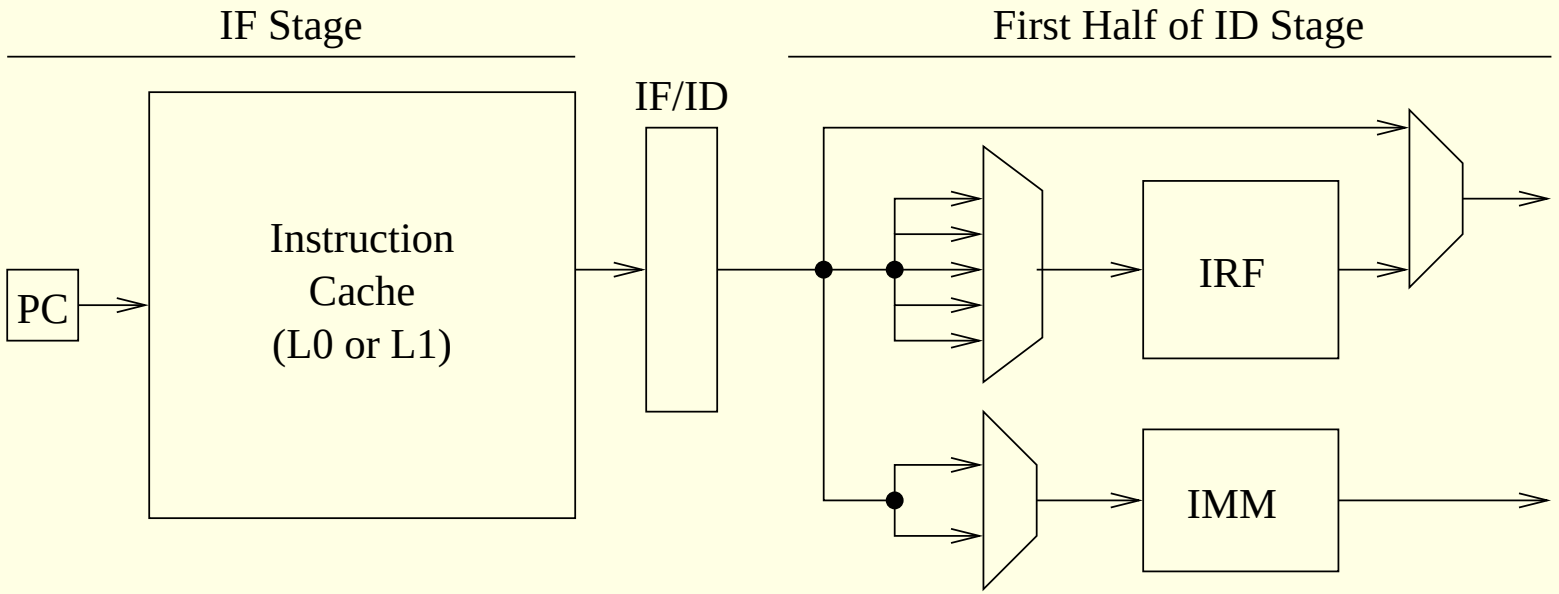
- Embedded Processor Design Constraints
  - **Power Consumption**
  - **Static Code Size**
  - **Execution Time**
- Fetch logic consumes **36%** of total processor power on StrongARM
  - Instruction Cache (IC) and/or ROM — Lower power than a large memory store, but still a fairly large, flat storage 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.

◆ L0 (Filter) Caches

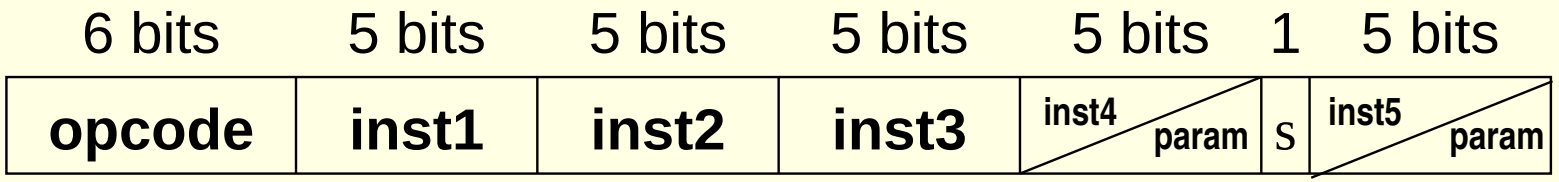
- Small and usually direct-mapped
- Designed to reduce energy consumed during instruction fetch
- Performance penalties due to high miss rate (∼50%)
- Previous studies show 256B L0 cache can reduce fetch energy usage by 68% at the cost of a 46% increase in execution time.

◆ Outline

❷ IRF Overview

- Previous work from ISCA 2005
- 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 with **Parameterization**
- Register windowing hardware for IRF (MICRO 2005)
- Profiled applications are packed using a modified **VPO** compiler.

◆ 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.

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

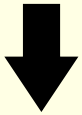
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

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

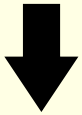
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

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

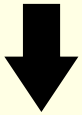
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

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

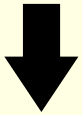
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

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

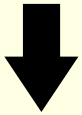
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

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

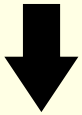
Original Code Sequence

```
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

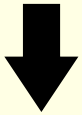
#	Instruction	Default
0	nop	NA
1	addiu r[5], r[3], 1	1
2	beq r[5], r[0], 0	None
3	addu r[5], r[5], r[4]	NA
4	andi r[3], r[3],63	63
...	...	...

Immediate Table

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

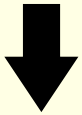
Original Code Sequence

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
...	...

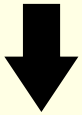
Encoded Packed Sequence

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

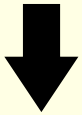
Original Code Sequence

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}

❸ Integrating IRF with L0

- IRF reduces code size, while L0 has no effect.
- Different granularity of fetch energy savings leads to improved energy usage when combining IRF and L0.
- IRF can alleviate **performance penalty** of L0 instruction caches.
  - 1 cycle stall when miss in L0 IC, but hit in L1 IC
  - Overlapped fetch and decreased working set size create this opportunity for IRF to improve instruction fetch.

◆ Overlapping Fetch with an IRF

Cycle	1	2	3	4	5	6	7	8	9
Insn1	IF	ID	EX	M	WB				
Insn2		IF	ID	EX	M	WB			
Insn3			IF		ID	EX	M	WB	
Insn4					IF	ID	EX	M	WB

(a) L0 Cache Miss at Insn3

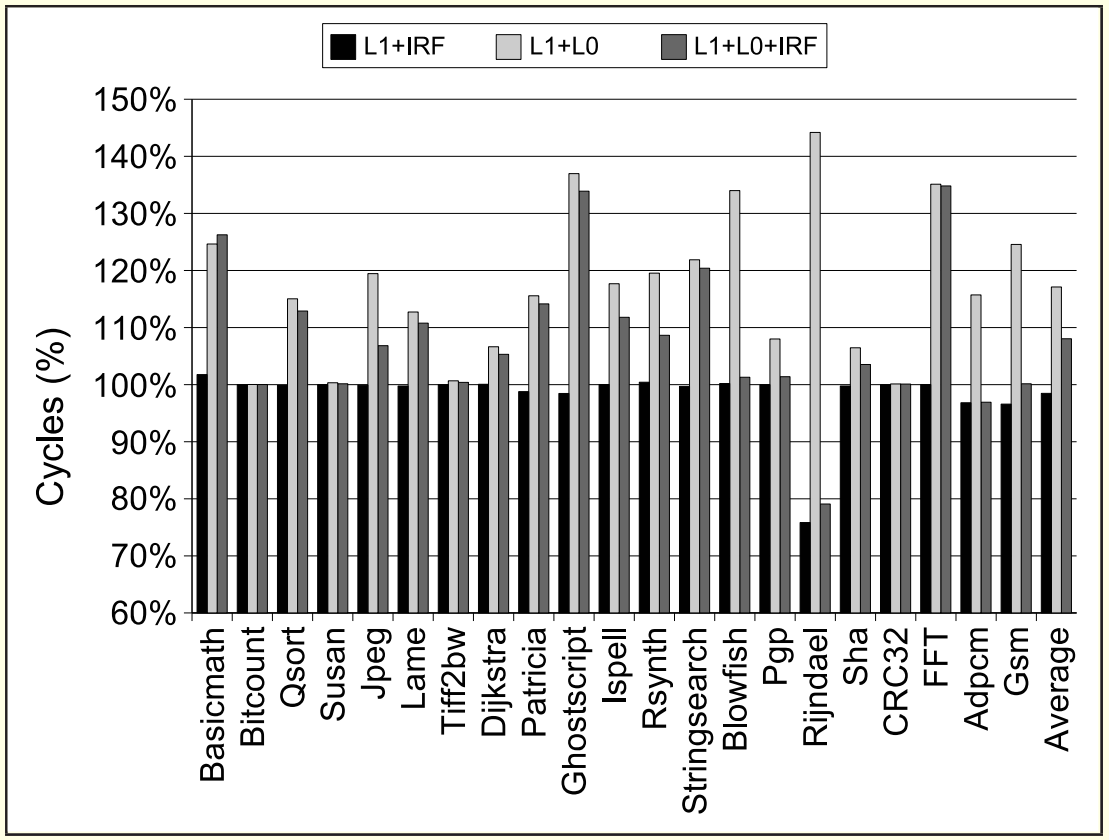
Cycle	1	2	3	4	5	6	7	8	9
Insn1	IF	ID	EX	M	WB				
Pack2a		IF_{ab}	ID_a	EX_a	M_a	WB_a			
Pack2b				ID_b	EX_b	M_b	WB_b		
Insn4			IF		ID	EX	M	WB	

(b) L0 Cache Miss at Insn4 with IRF

❹ Experimental Results

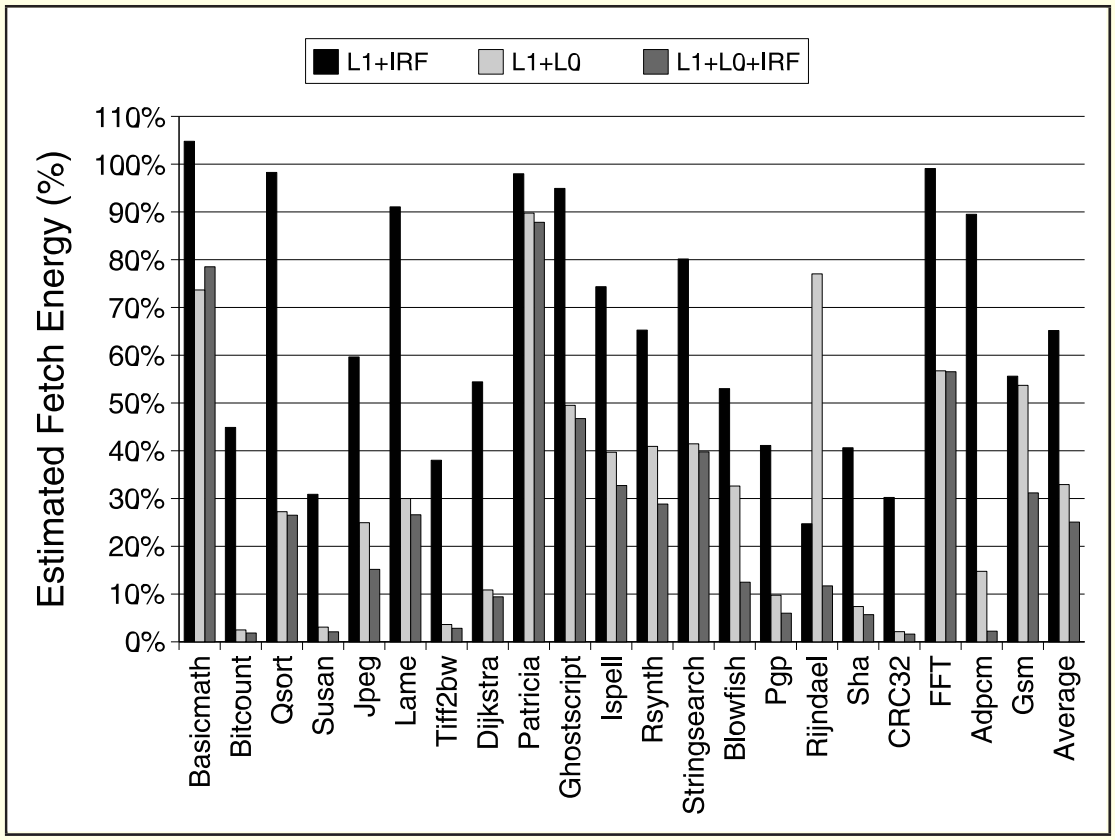
- SimpleScalar PISA
  - Embedded configuration
    - ⋆ In order, 16KB 1-cycle 4-way L1 IC, 256B DM L0 IC
  - High-end configuration
    - ⋆ Out of order, 32KB 2-cycle 4-way L1 IC, 512B DM L0 IC
  - 4-window 32-entry IRF with 32-entry IMM
- Fetch energy estimates constructed based on prior sim-panalyzer results.
- Evaluation with MiBench embedded benchmark suite

◆ Embedded Execution Efficiency



- L1+IRF: **1.52%** improvement
- L1+L0: **17.11%** penalty
- L1+L0+IRF: **8.04%** penalty

◆ Embedded Fetch Energy Efficiency

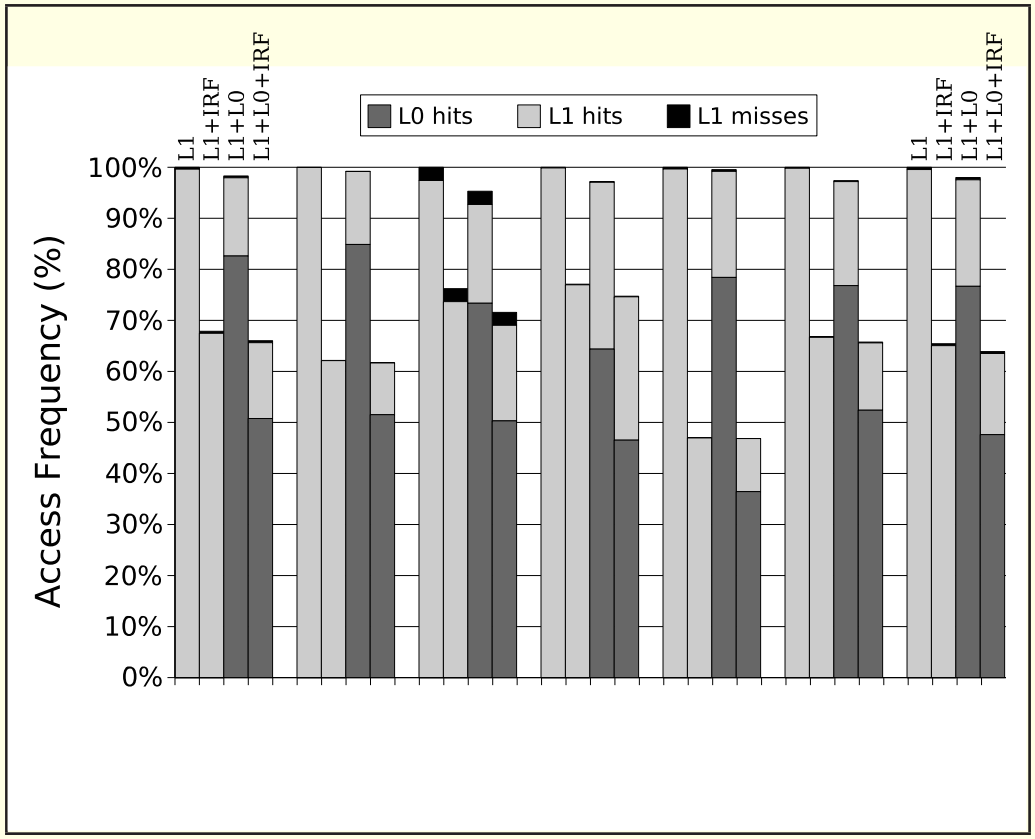


- L1+IRF: 34.83% improvement
- L1+L0: 67.07% improvement
- L1+L0+IRF: **74.93%** improvement

◆ Embedded Total Energy Savings

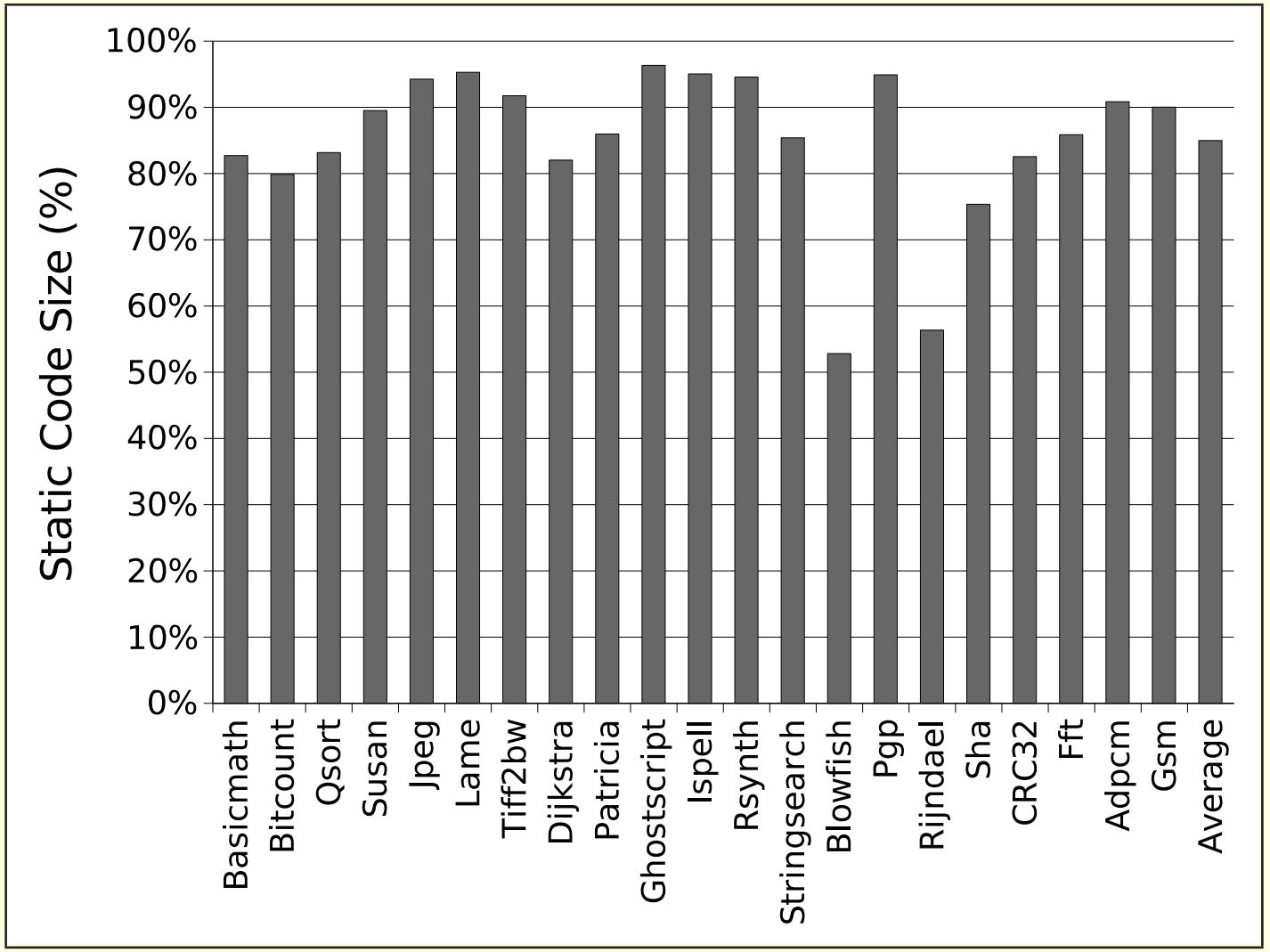
- Assuming that non-fetch energy scales uniformly with execution time
- If fetch energy accounts for 25% of total processor energy:
  - L1+L0: 4% energy savings
  - L1+L0+IRF: **12.7%** energy savings
- If fetch energy accounts for 33% of total processor energy:
  - L1+L0: 10.7% energy savings
  - L1+L0+IRF: **19.3%** energy savings

◆ Embedded Cache Access Frequencies



- IRF eliminates ∼35% of all IC accesses
- IRF + L0 accesses L1 IC only **16.27%** of the time!!!

◆ Reducing Static Code Size



❺ Related Work

- **L-Cache** – separate frequently executed code segments and restructure (Bellas et al.)
- **Loop cache** – detect short backward branches and buffer loops (Lee et al.)
- **Bypassing** L0 using simple prediction (Tang et al.)
- **Zero Overhead Loop Buffer (ZOLB)** – low power execution of an explicitly loaded inner loop (Eyre and Bier)

❻ Future Work

- Improved selection of IRF instructions for areas of code that need to tolerate increased fetch latency.

- Implementation with other techniques that impose a fetch bottleneck:
  - Procedural abstraction and echo factoring
  - Dictionary compression (decompressing into the IC)
  - Encrypted executables (decryption into the IC or of a single IC line)

- Novel architectural designs with asymmetric instruction bandwidth:
  - Reduced fetch width (1-2 instructions) + IRF
  - Additional execution hardware (4+ instructions)

❼ Conclusions

- Instruction packing with an IRF leads to reduced code size, energy consumption and execution time.

- Combined with an L0 IC, an IRF can reduce the miss penalty and further improve energy efficiency in both embedded and aggressively pipelined, high-end processor designs.

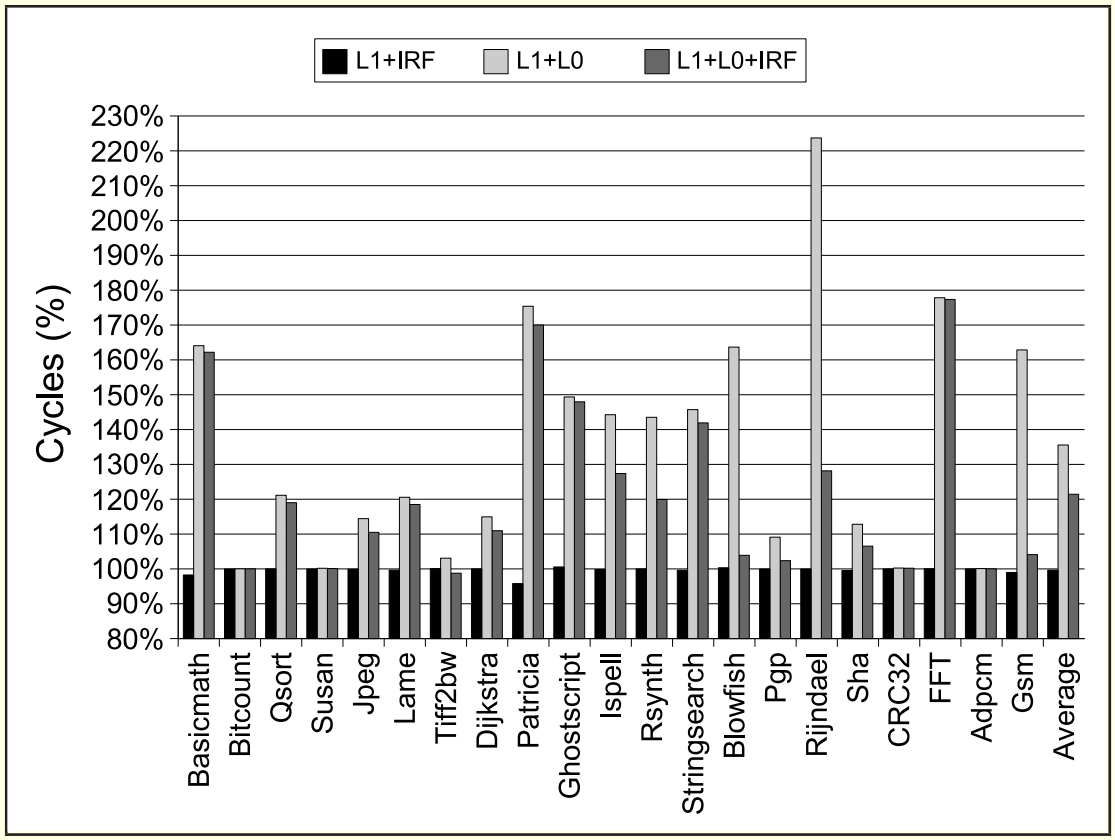
- Lost performance due to fetch bottlenecks can be alleviated since the IRF can essentially fetch and buffer several instructions at a time.

◆ The End

Thank you!

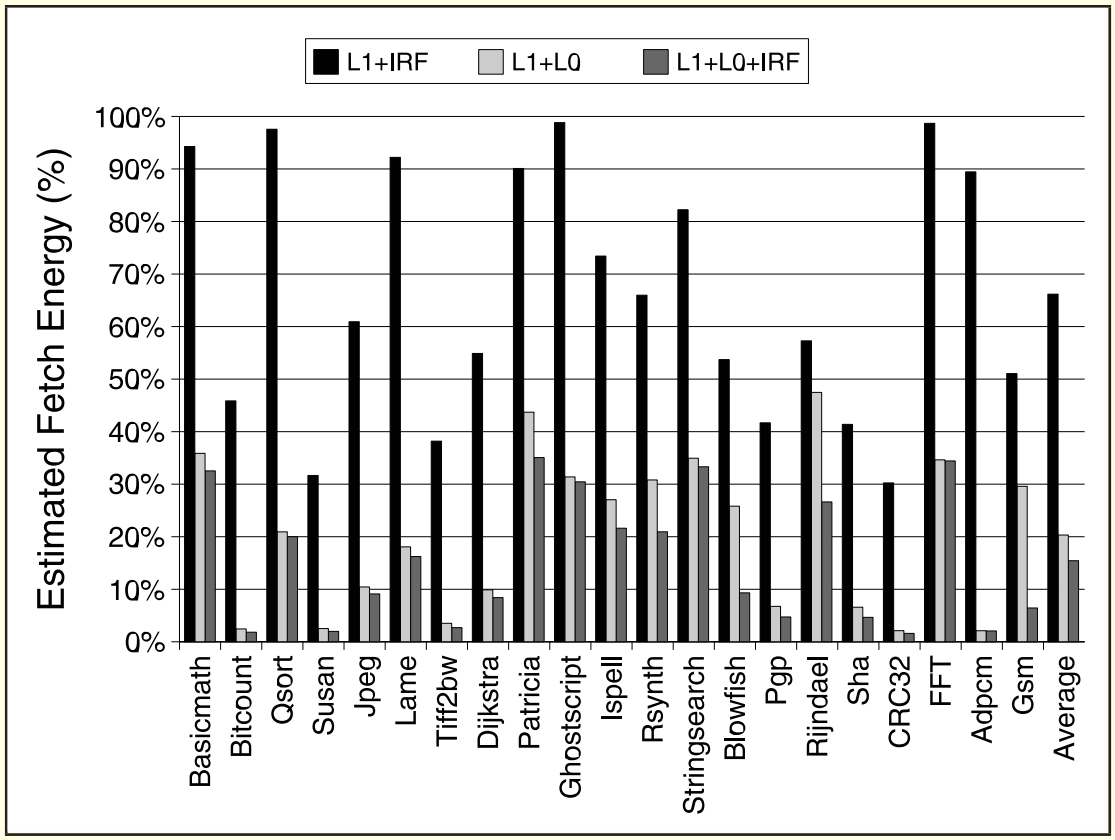
Questions ???

High-end Execution Efficiency



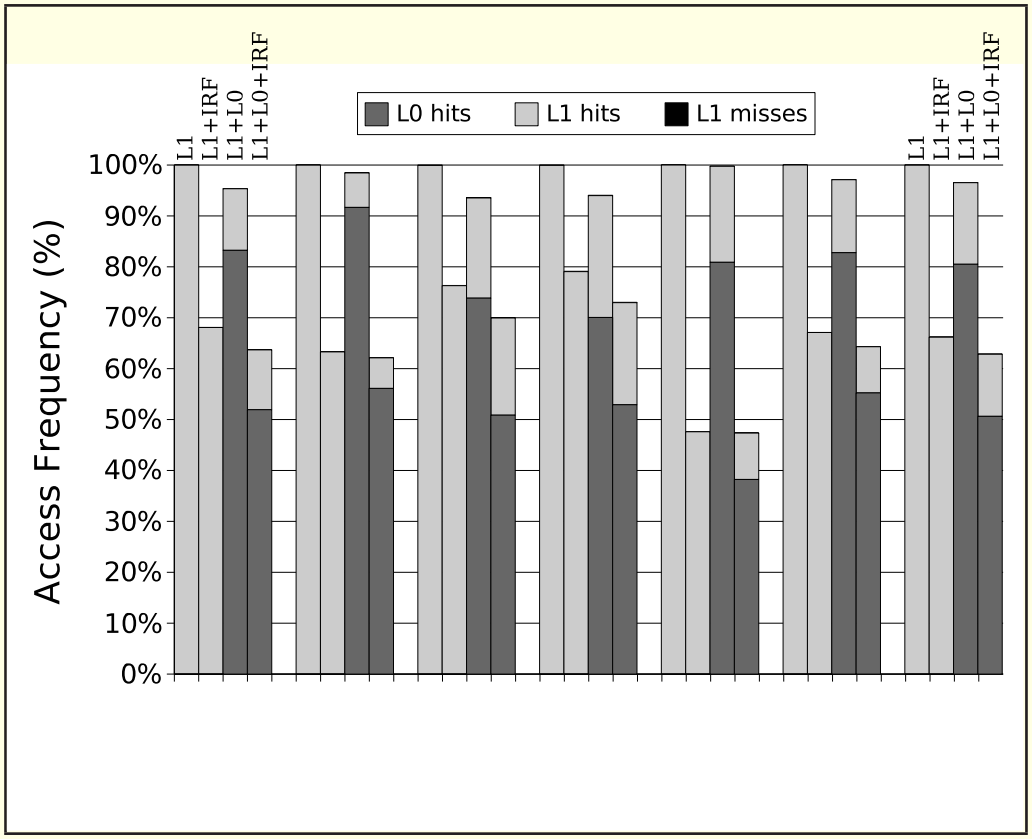
- L1+IRF: **0.35%** improvement
- L1+L0: **35.57%** penalty
- L1+L0+IRF: **21.43%** penalty

High-end Fetch Energy Efficiency



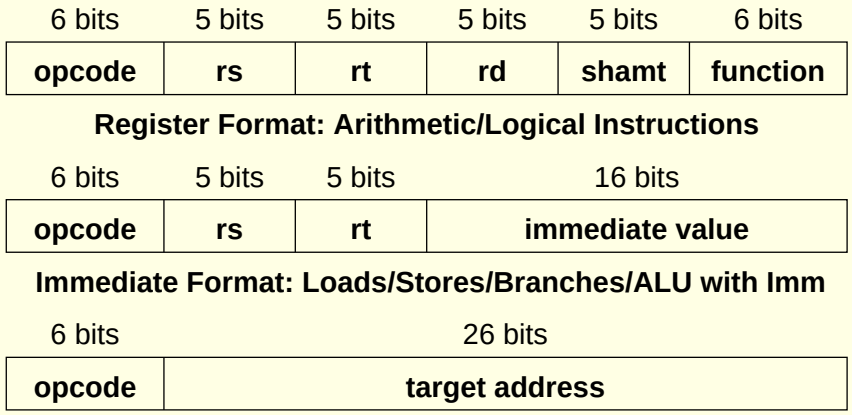
- L1+IRF: 33.83% improvement
- L1+L0: 79.68% improvement
- L1+L0+IRF: **84.57%** improvement

High-end Cache Access Frequencies

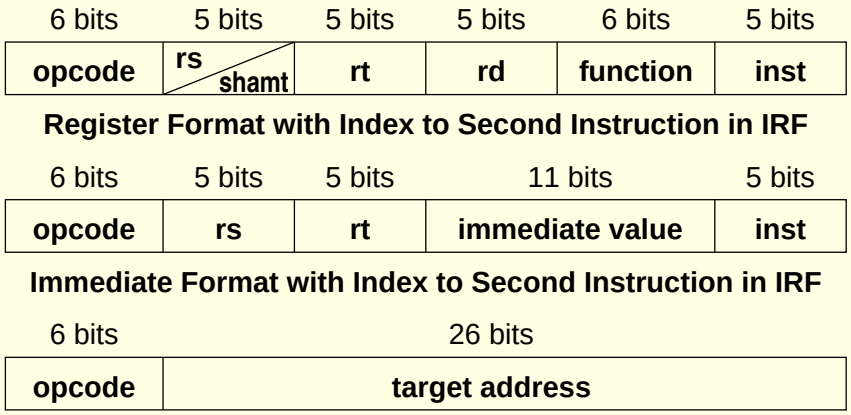


- IRF eliminates ∼33% of all IC accesses
- IRF + L0 accesses L1 IC only **12.22%** of the time!!!

◆ MIPS Instruction Format Modifications



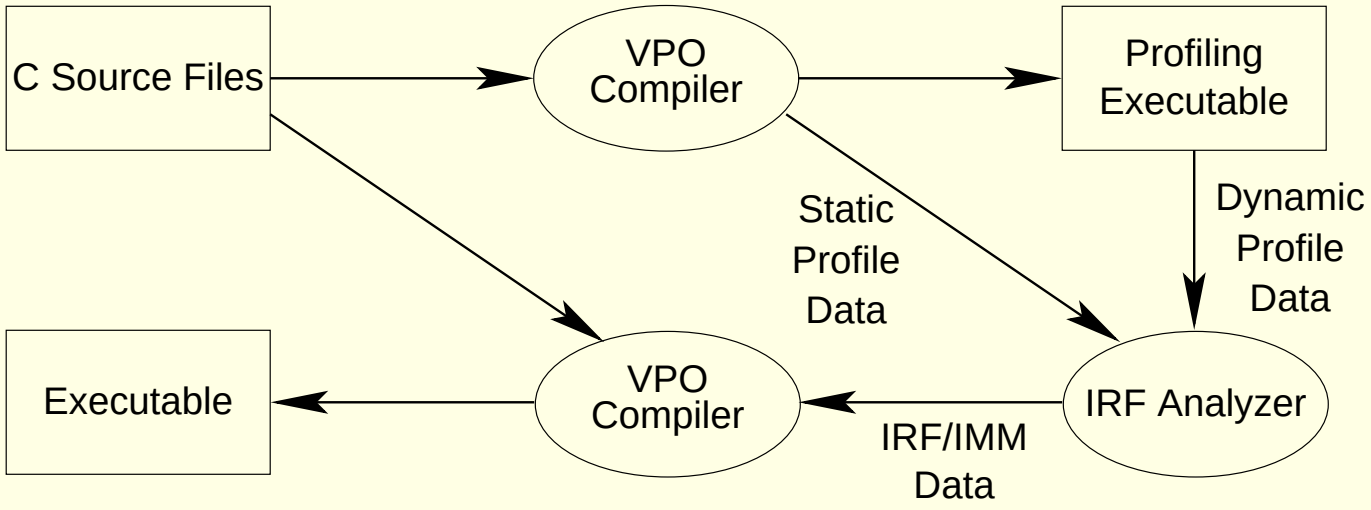
(a) Original MIPS Instruction Formats



(b) Loosely Packed MIPS Instruction Formats

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

◆ 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

◆ 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