Reducing Instruction Fetch Cost by Packing Instructions into Register Windows

Stephen Hines, Gary Tyson, David Whalley

Computer Science Dept.
Florida State University

November 14, 2005

❶ Introduction

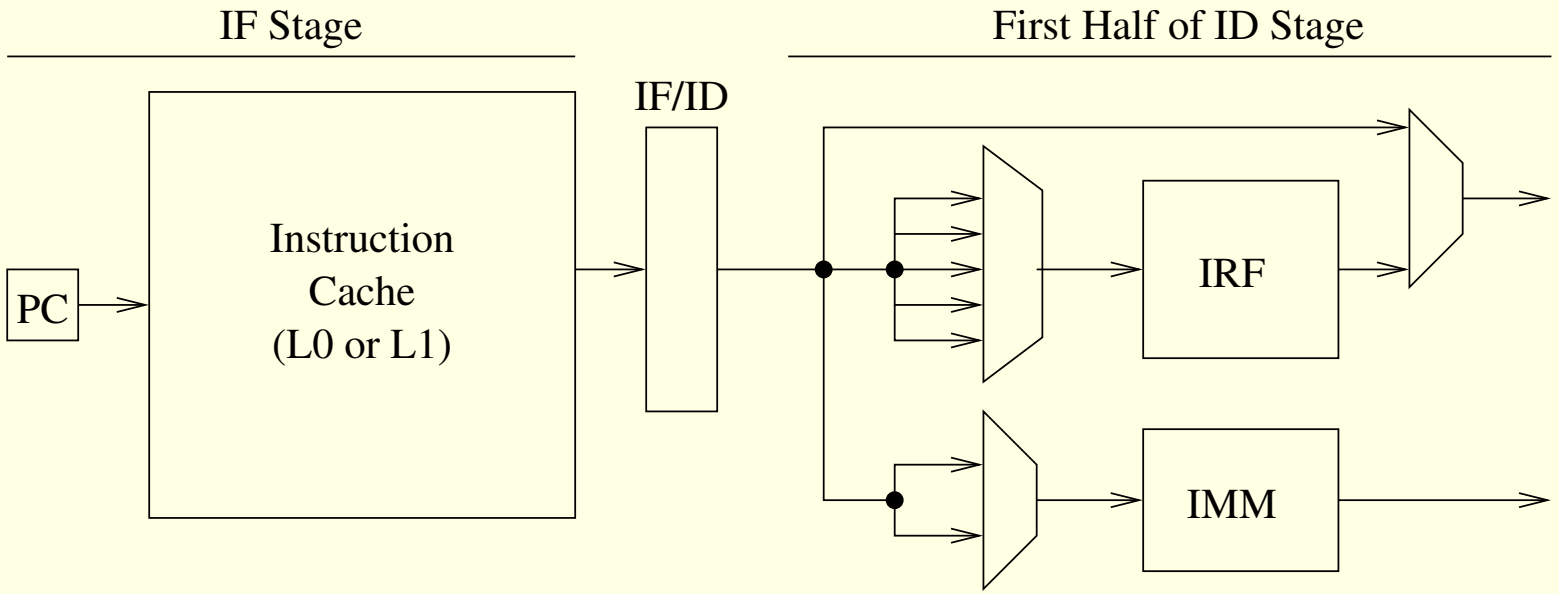
- Reducing fetch energy consumption is important for embedded devices
  - Fetch accounts for 1/3 of total processor power on a StrongARM
  - Existing techniques provide tradeoffs with execution time (L0 caches), or can only target certain innermost loops (loop caches, ZOLB)
- Instruction Packing with an **Instruction Register File** (**IRF**)
  - Targets fetch energy, code size, and execution time for improvement
  - Place frequently accessed instructions into a small register file for easy, lower-power access
  - Original ISCA 2005 version limited to 32 instructions/registers locked in at program load

◆ Reducing Fetch Energy Consumption

- Can be improved by fetching even more instructions from the IRF
- SPARC uses register windows to reduce overhead of register saves/restores on function calls
- Windowing is also better than just increasing the size of the IRF, as the larger IRF would require a greater number of bits to address each entry
- Windowing eliminates the need to modify our original proposed IRF instruction formats
- Allow the compiler to make decisions about which instructions should be promoted to the IRF for each particular function/phase of execution

◆ Outline

❷ IRF Overview



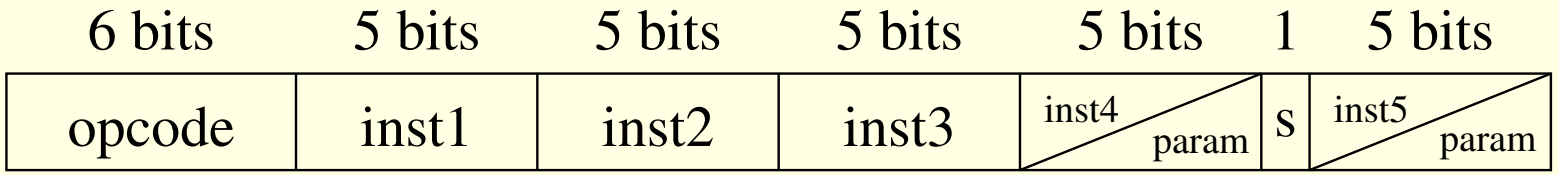
- Stores frequently occurring instructions as specified by the compiler (potentially in a partially decoded state).
- Allows multiple instruction fetch with packed instructions.

◆ 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 with **Parameterization**

◆ 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.

◆ Experimental Setup

- **SimpleScalar PISA** and **VPO** targeted for MIPS with IRF
- Dynamically profiled applications + **irfprof** for selecting IRF entries
- Library code is not packed and thus not evaluated
- 21 **MiBench** embedded benchmarks
- Power analysis validated by sim-panalyzer and Cacti approximations:

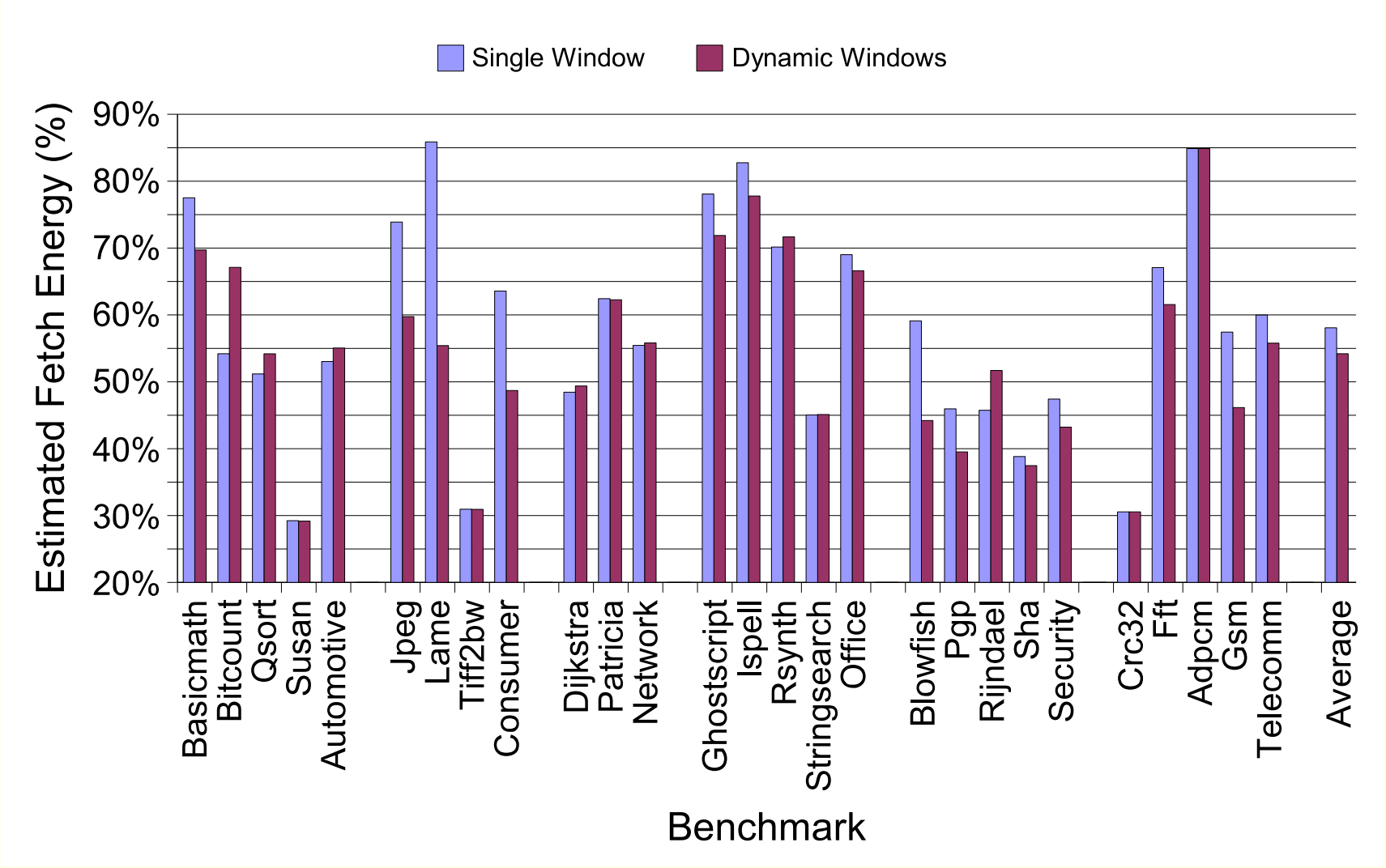
$$E_{fetch} = Cost_{IC} \times Accesses_{IC} + Cost_{IRF} \times Accesses_{IRF}$$

- $Cost_{IC}$ hit is approximately 2 orders of magnitude greater than $Cost_{IRF}$ for an 8KB IC and 32-entry IRF

❸ Software Windowing

- Improve utilization by replacing entries in the IRF on a per-function basis
- **load_irf** – Compiler-generated instruction to replace IRF entries
- Greedy partitioning algorithm selects instructions from similar functions to share space in a window
  - Depending on benefit/cost of splitting, choose whether to merge function profile with an existing partition, or create a new partition
  - Each function only placed once, so the algorithm is guaranteed to terminate
- Results
  - Fetch Energy – Standard IRF 58.08% $\rightarrow$ SW windows 54.40%
  - Windows from 1 – 32 with median of 4 and mean of 8.33
  - Approximately 24.54 different IRF entries between partitions

◆ Software Partitioning – Results



❹ Hardware Windowing

- Overhead in switching software windows hides some of the additional benefit when working with smaller, less diverse programs

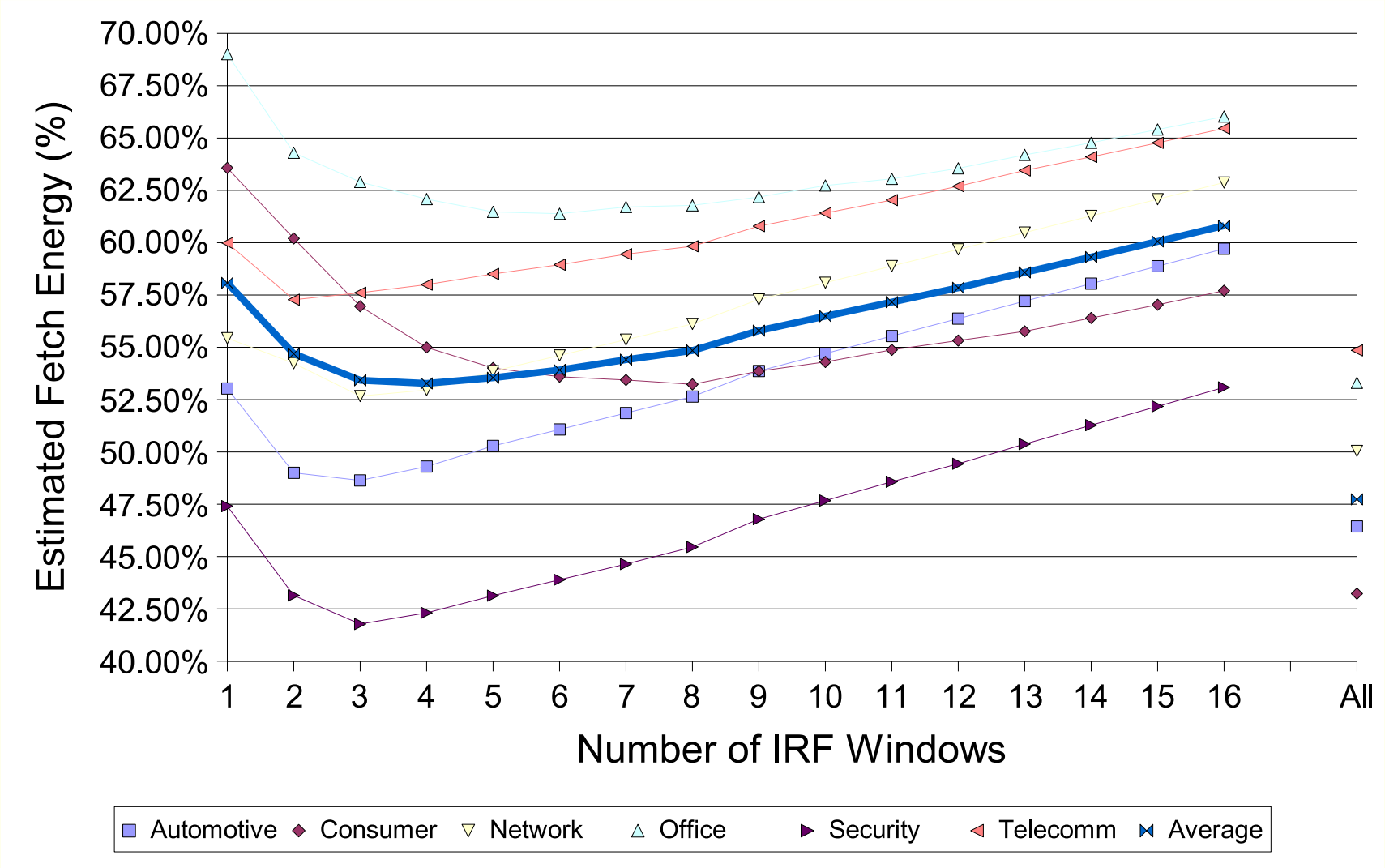
- Similar to SPARC data register windows, IRF can support multiple hardware windows, although they are not handled in a LIFO manner
  - Function addresses are modified to include an instruction register window pointer (IRWP)
  - Calls transfer control to the specified address and set the new IRWP
  - When saving the return address, the current IRWP is also saved
  - Returns also restore the proper window based on the saved IRWP

- Windows can be purely physical or managed through parallel register copying

◆ Hardware Window Partitioning

- Greedy algorithm operates similar to previous software partitioning, but no need to estimate overhead costs
- First build up the N partitions by choosing the function with the greatest minimum increase in **cost** for adding to the existing partitions
- For each remaining function, choose to allocate it to the partition that yields the lowest overall cost
- Fetch Energy – Standard IRF 58.08% $\rightarrow$ 4 windows 53.28%
- Results can be further improved if inactive partitions are kept in a drowsy low-power state

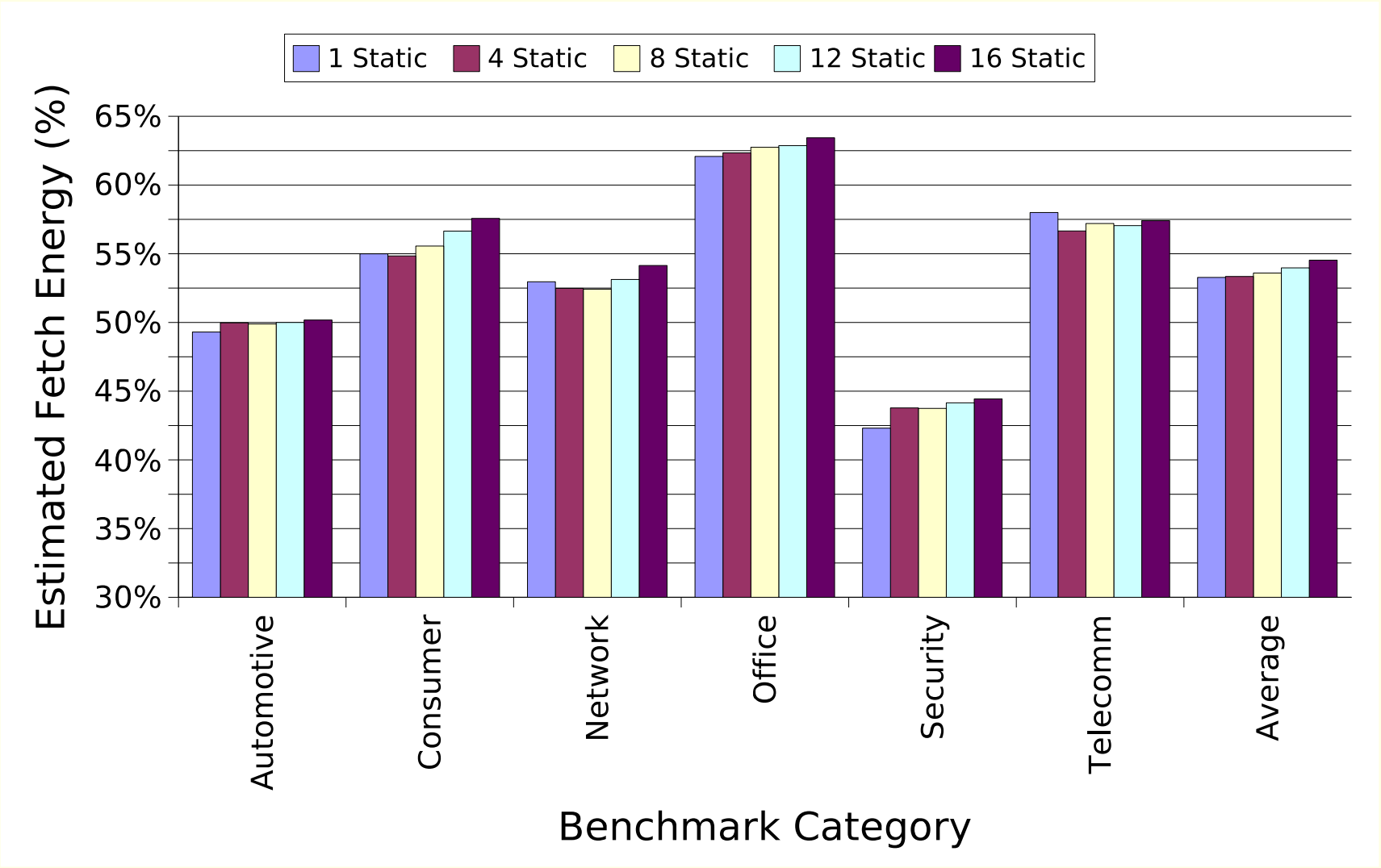
Hardware Partitioning – Results



❺ Static IRF Portions

- Goal is to minimize area requirements of IRF windowing while still providing improved fetch energy consumption
- Similar to SPARC global registers remaining the same on window switches, we noticed instructions are often duplicated in multiple windows
- Selection algorithm chooses the M shared entries and then proceeds with the standard selection algorithm for 4 windows, considering that some instructions are available in each IRF window
- Fetch Energy – 4 windows IRF 53.28% $\rightarrow$ 4 shared entries 53.35%
- Reduced leakage energy for smaller IRF area may be more important for future design processes

◆ Static IRF Portions – Results



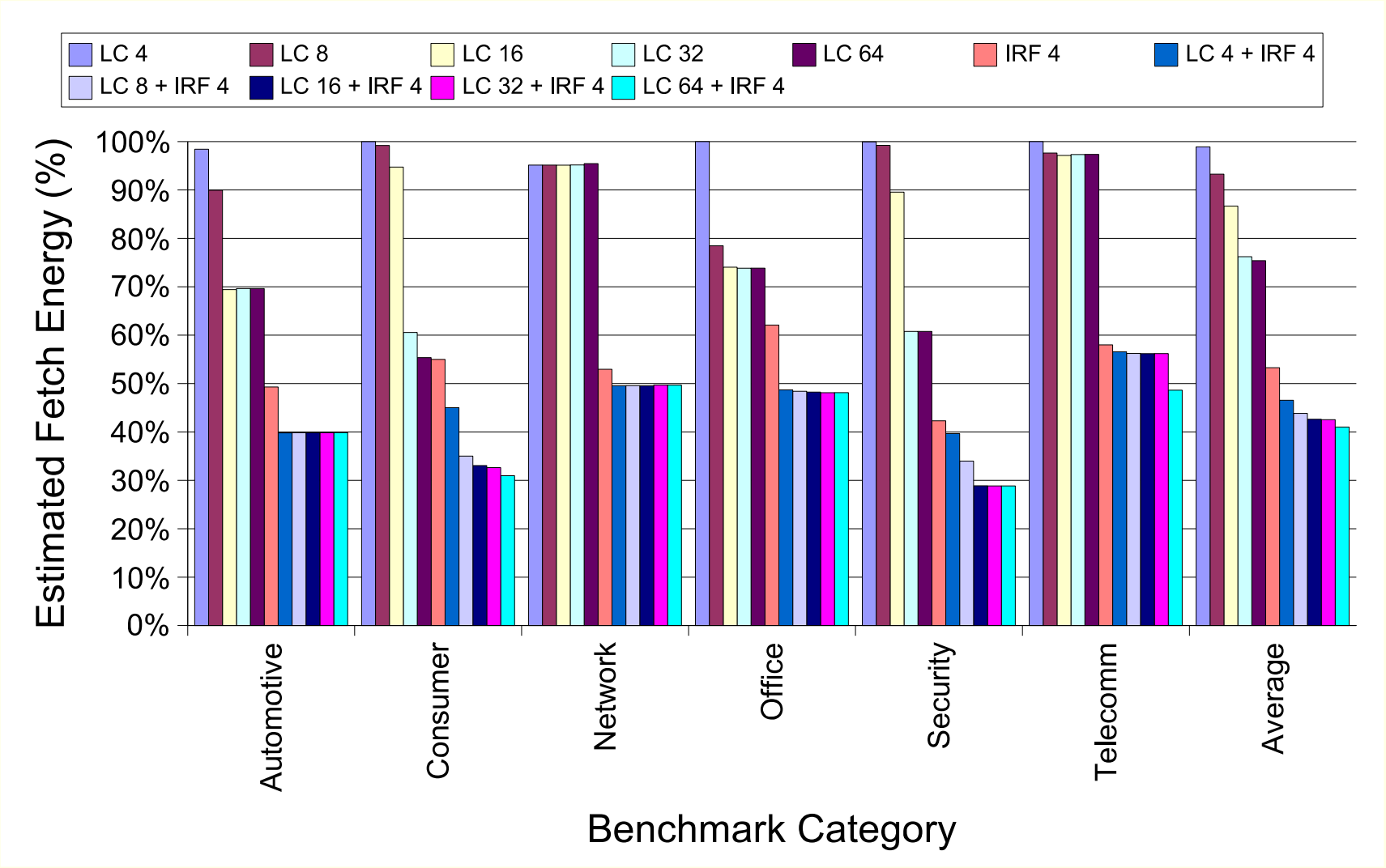
❻ Using an IRF with a Loop Cache

- Loop Cache – automatically places inner loop instructions into a small fetch buffer (reducing energy consumption)
- Three modes
  - **Inactive** – waiting to detect a short backward branch (**sbb**)
  - **Fill** – after sbb detection, fill the buffer with instructions until another taken sbb or a different jump (canceling the fill)
  - **Active** – after filling back to sbb, all fetches can be handled by the circular buffer, until the sbb is not taken or another branch is
- Limitations
  - Can only capture innermost loops
  - No additional transfers of control
  - Difficult to handle long loops

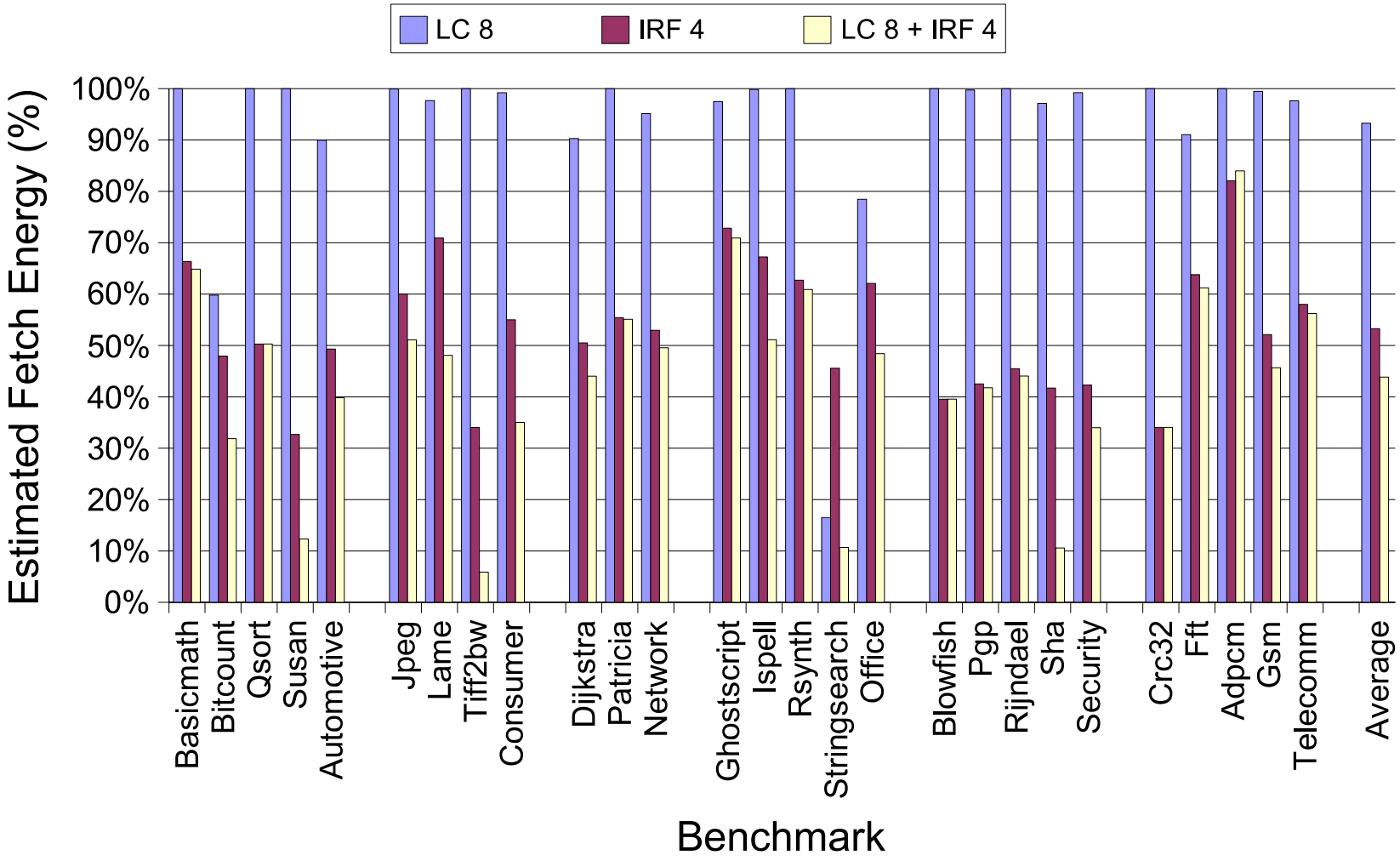
◆ IRF and Loop Cache Synergy

- Allow loop cache to handle more instructions (packing reduces code size)
- Better than just lengthening the loop cache buffer, since IRF acts as a filter against instructions that also are not effective with the loop cache
  - Calls are not packable (and not able to be in the loop cache)
  - Branches terminate packs and thus do not condense as well as straight-line code
  - Densely encoded inner loops are formed by the IRF, and then able to be detected/fetched easily by a loop cache
- IRF normally has to fetch from the IC at least once every five instructions (for a new MISA packed instruction), but the loop cache replaces the expensive IC fetch with a reduced cost fetch as well
- Fetch Energy – 8 entry LC 93.26% $\rightarrow$ 4 windows IRF 53.28% $\rightarrow$ 8 entry LC + 4 windows IRF 43.84%

IRF with Loop Cache – Results



◆ IRF with Loop Cache – Details



❼ Future Work

- Combine software partitioning and hardware partitioning to allow for a greater number of available physical IRF entries
  - Use **load_irf** instructions in spots where behavior changes
  - Virtualize the IRF and let windows be cached/loaded as necessary
- Combine IRF with existing techniques that have fetch bottlenecks
  - Code compression and encryption can impose serious penalties on instruction fetch due to extra work decompressing/decrypting
  - Similarly, L0 (filter) caches can have performance penalties due to low hit rates
  - IRF can reduce the latency since more instructions are executed than are fetched from the IC (essentially masking the pipeline stalls)

❽ Conclusions

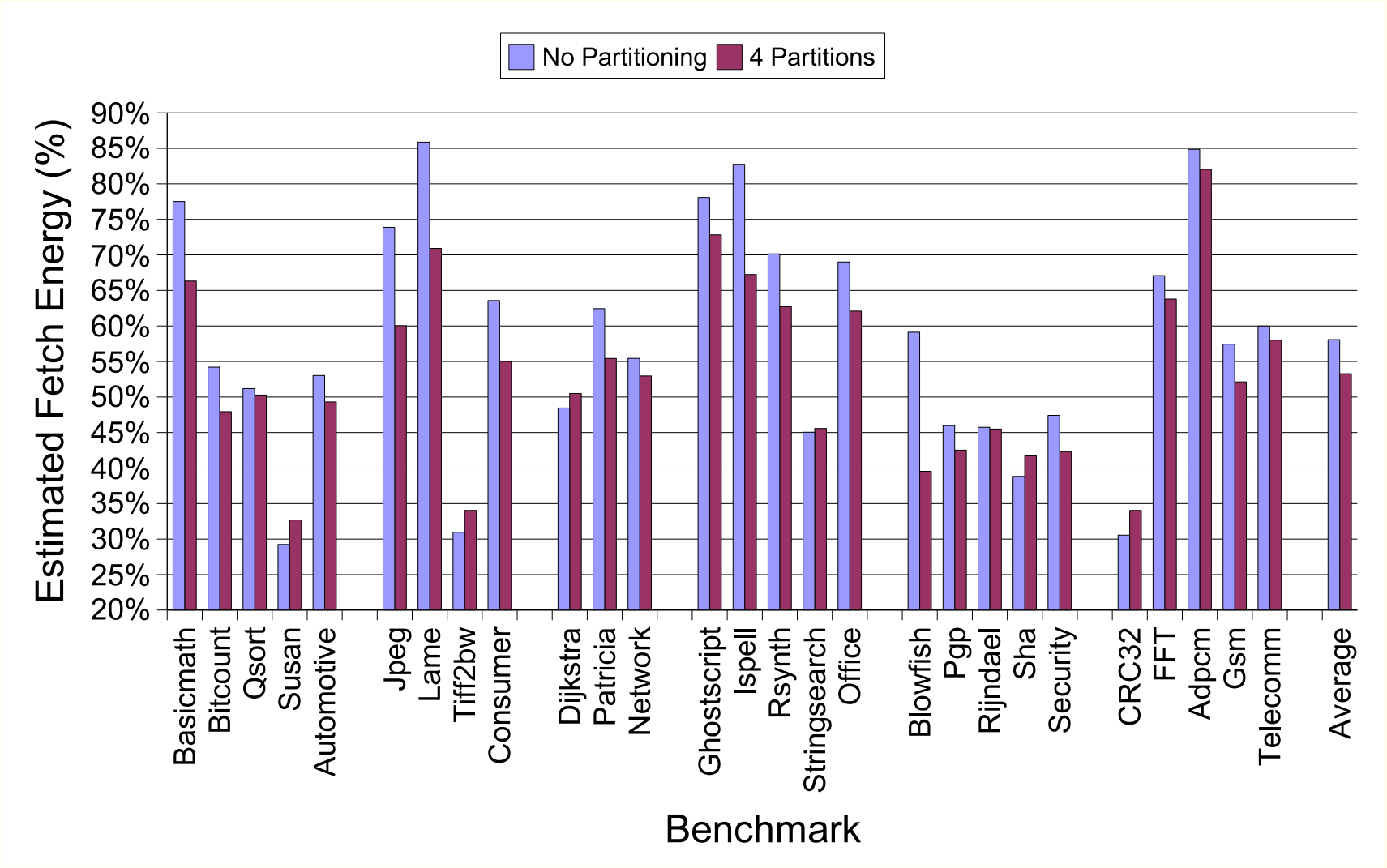
- Improve IRF packing for varied function/phase behavior by windowing the register file (software or hardware)
- Can reserve a portion to be statically shared for reduced area overhead
- Nearly **half** of the fetch energy can be eliminated using a windowed IRF
- Synergistic relationship between IRF and Loop Cache, since each operate at a different granularity
- IRF provides significant fetch energy savings with reduced code size and slightly improved execution behavior
- Can be added to an existing architecture with just a few spare opcodes, providing a rich extension to the traditional ISA via the RISA

◆ The End

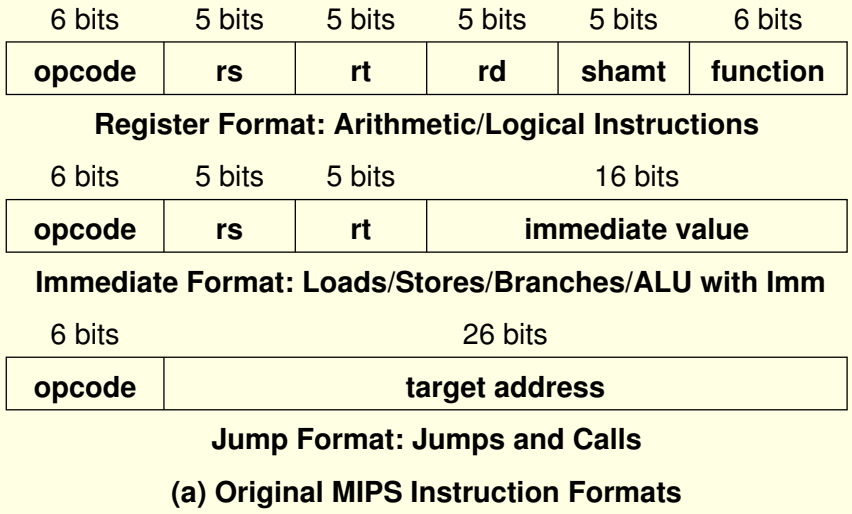
Thank you!

Questions ???

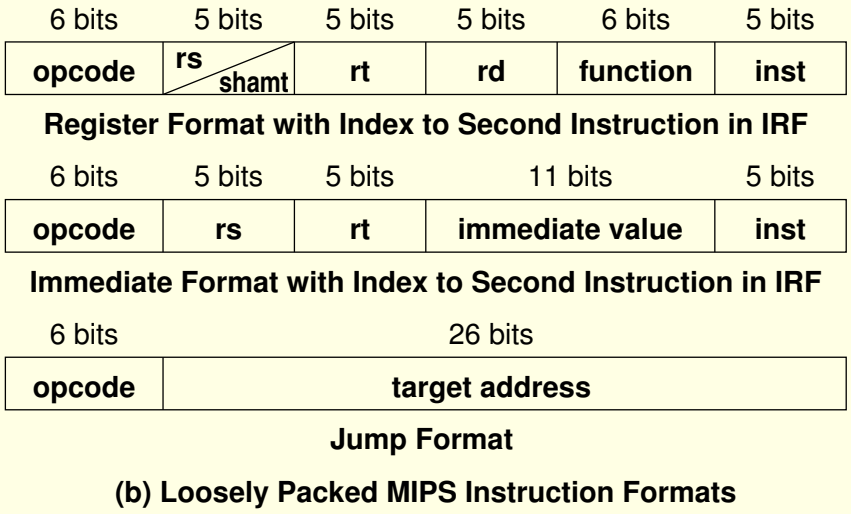
Hardware Partioning – Details



◆ MIPS Instruction Format Modifications



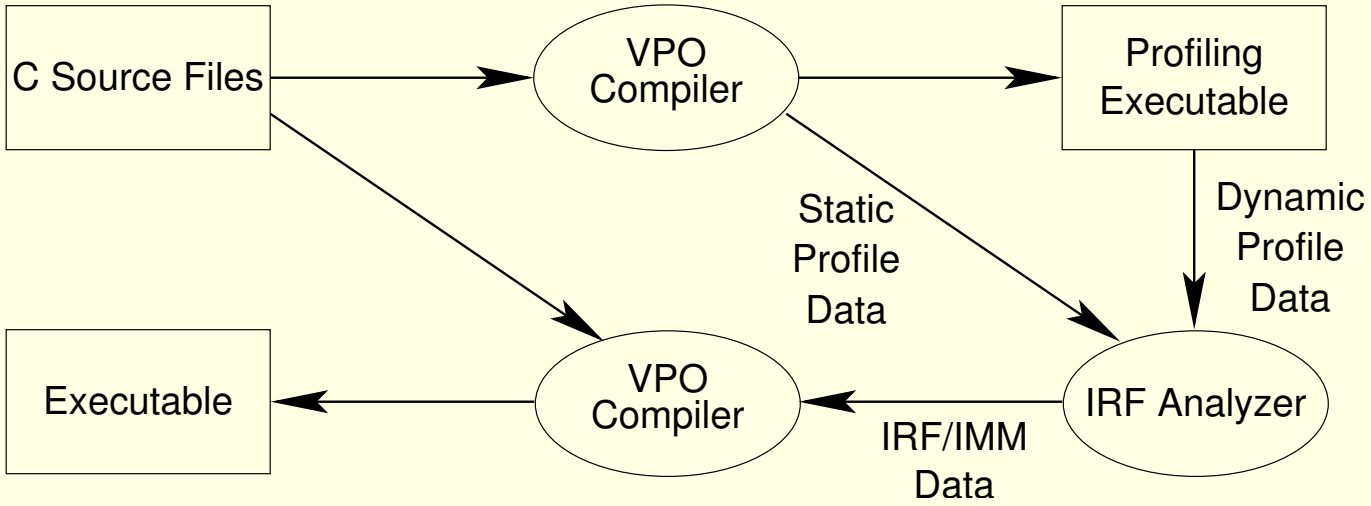
(a) Original MIPS Instruction Formats



(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)
    - $\star$ *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