

---

---

# Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processor

Sergei Y. Larin  
Thomas M. Conte

North Carolina State University

<http://www.tinker.ncsu.edu>

---

---

## Talk Outline

- Introduction
- Problem statement
  - ◆ Compression and Custom ISA for embedded systems
  - ◆ Compression Strategy
- Proposed solution
  - ◆ Compression Implementation
  - ◆ Instruction Fetch architecture
- Conclusion and Future work

---

---

## Introduction

- **Embedded microprocessors**
  - ◆ Used to be- low performance designs with non-challenging design constraints
- **Recent requirements for embedded systems are more aggressive, conflicting goals:**
  - ◆ Low Power
  - ◆ High Performance
  - ◆ Small Size
  - ◆ Extremely Small Code Size

---

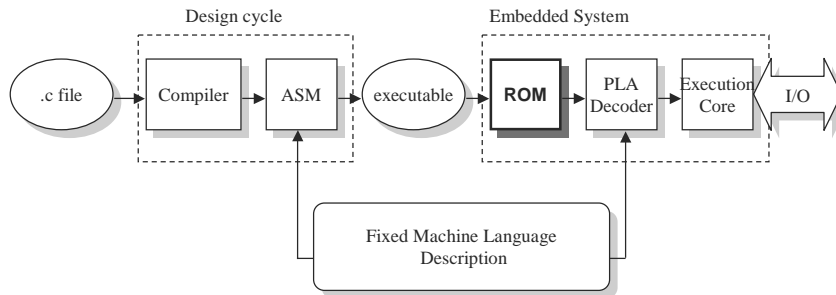
---

## Introduction

- **Use of VLIW architecture is a fast growing trend in embedded world**
  - ◆ TI's 320C6xx, Sun's MAJC, Philips TriMedia TM-1000, etc
- **The reasons are:**
  - ◆ High performance (ILP) without as high a hardware cost
  - ◆ Less market pressure for cross-generation binary compatibility
  - ◆ Often highly accurate profile information is available for high quality static code scheduling
- **Familiar problems remain**
  - ◆ Static code size
  - ◆ Sophisticated compiler technology

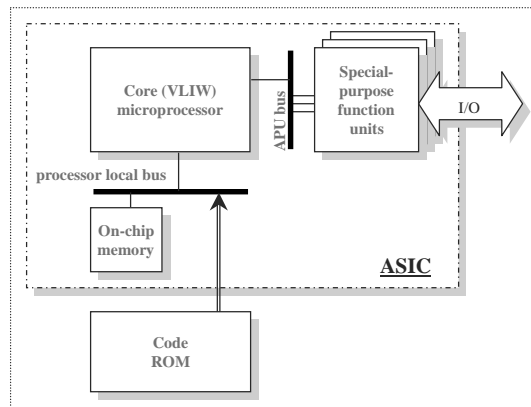
## Current Design process

- One approach to the embedded systems design -- ASIC includes core processor with code placed in ROM
- If fixed machine language is used the design cycle could be briefly outlined in following way:



## Embedded Systems architecture

- As a result we will have the following structure:
  - ◆ Very flexible design
  - ◆ Short design cycle



---

---

## Embedded Systems architecture

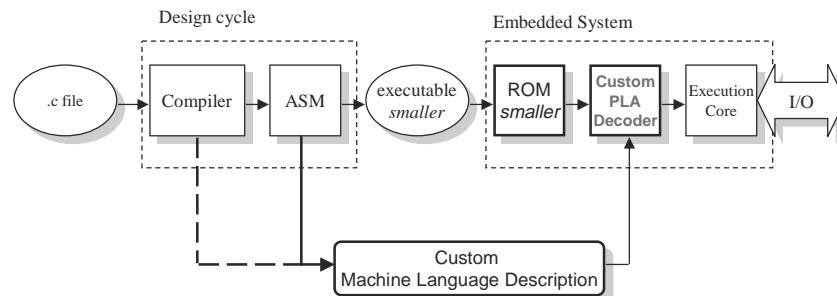
- But there are some problems with this solution
  - ◆ For somewhat complicated design the size of the ROM is large
  - ◆ As a result ROM could not be placed ON in same package as core
- Power consumption increase
- Performance suffers

---

---

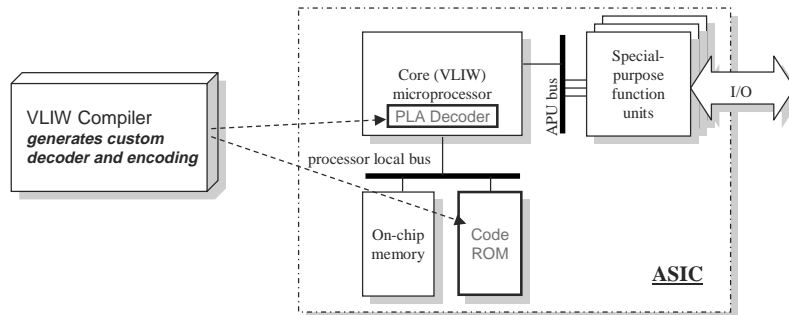
## Proposed alternative design process

- Since a PLA is used for decoding instructions, we can picture the following system:

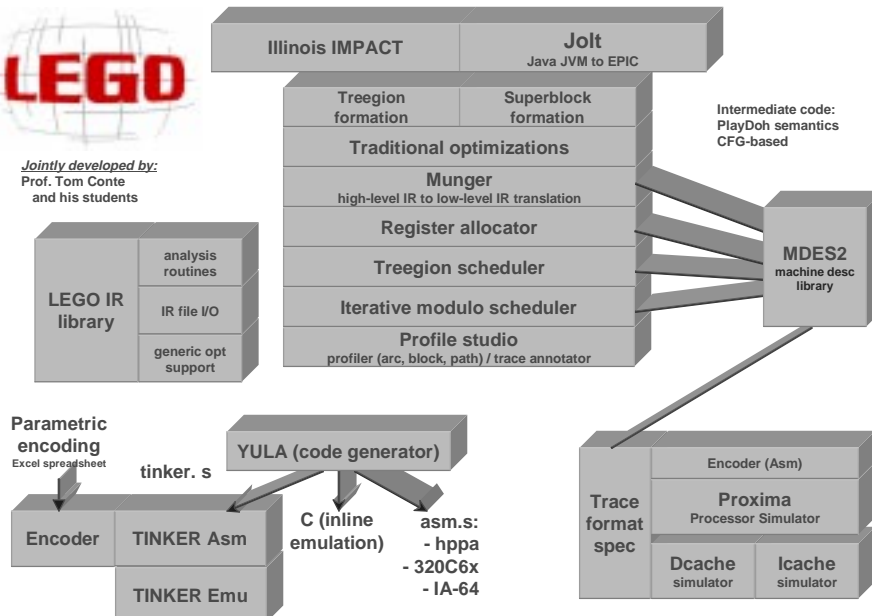


# Embedded systems architecture

- Can automate the decoder customization using the VLIW compiler



Jointly developed by:  
Prof. Tom Conte  
and his students



---

---

## TINKER ISA for Compressed Operation Encoding

- TINKER architecture and ISA:
  - ◆ Based on HP Labs PlayDoh architecture specification [Kathail/Schlansker/Rau 94], a major influence for IA-64
  - ◆ The TINKER ISA is a superset of the HP PlayDoh specification
  - ◆ 40 bit encoding EPIC style extension for this study
- TINKER ISA is a *memory efficient encoding*
  - ◆ it does not hold NOPs (zero-NOP encoding)
- Tinker ISA is expandable and backward compatible
  - ◆ well suited for future hardware expansion, direct binary rescheduling [Conte/Sathaye-95] and dynamic optimization [Sathaye-98]

---

---

## TINKER ISA for Compressed Operation Encoding

- Operation (OP) structure
  - ◆ Separate Op is always viewed as a part of a VLIW Multi Op (MOP)
  - ◆ Designed for easy compression/expansion
  - ◆ ... and 'non-aligned' memory layout
- MultiOp (MOP) structure
  - ◆ MOP consist of several conventional Ops according to the target machine specification
  - ◆ All Ops in a MOP are independent and *must be issued* in same cycle (VLIW requirement)

<u>Integer ALU Operation</u>													
1	1	2	5	5	5	2	8	5	1	5			
T	S	OPT	OPCODE	Src1	Src2	BHWX	Literal extension	Dest	L1	PREDICATE			
<u>Integer Compare-to-Predicate Operation</u>													
1	1	2	5	5	5	2	3	5	5	1	5		
T	S	OPT	OPCODE	Src1	Src2	BHWX	D1	Literal Extension	Dest	L1	PREDICATE		
<u>Integer Load Immediate Operation</u>													
1	1	2	5	20				5	1	5			
T	S	OPT	OPCODE	Src1 (literal)				Dest	L1	PREDICATE			
<u>Floating Point Operation</u>													
1	1	2	5	5	5	1	6	3	5	1	5		
T	S	OPT	OPCODE	Src1	Src2	S/D	Literal Extension	tssL/U	Dest	L1	PREDICATE		
<u>Load Operation</u>													
1	1	2	5	5	2	2	1	2	3	5	5	1	5
T	S	OPT	OPCODE	Src1	BHWX	SCS	Res	TCS	Res	Lat	Dest	Res	PREDICATE
<u>Store Operation</u>													
1	1	2	5	5	5	2	2	11			1	5	
T	S	OPT	OPCODE	Src1	Src2	BHWX	TCS	Displacement/Short Addr			L1	PREDICATE	
<u>Branch Operation</u>													
1	1	2	5	5	16				5				
T	S	OPT	OPCODE	Src1	Counter	Branch offset				PREDICATE			
											39		

Tom Conte, NC State University    [conte@ncsu.edu](mailto:conte@ncsu.edu)    <http://www.tinker.ncsu.edu>    CCODE-13

## TINKER ISA for Zero-NOP operation encoding

- Example of a MOP layout in memory
  - ◆ for an 8-wide VLIW machine

**Traditional encoding**

0	nop	nop	Op_A_2	nop	Op_A_4	nop	nop	nop
8	Op_B_0	nop	Op_B_2	nop	nop	Op_B_5	Op_B_6	nop
16	Op_C_0	nop	Op_C_2	nop	Op_C_4	nop	nop	Op_C_7

Total 960 bit, 560 bit are in NOPs

**TINKER Zero-NOP encoding**

0	0	Op_A_2	1	Op_A_4	OpType			
2	0	Op_B_0	0	Op_B_2	0	Op_B_5	1	Op_B_6
6	0	Op_C_0	0	Op_C_2	0	Op_C_4	1	Op_C_7

Total 400 bit, 0 bit are in NOPs

Tom Conte, NC State University    [conte@ncsu.edu](mailto:conte@ncsu.edu)    <http://www.tinker.ncsu.edu>    CCODE-14

---

---

## Compressed Encoding

- **Zero-NOP encoding alone cannot solve the ROM size problem**
  - ◆ Only reduces VLIW overhead
- **Several researchers have proposed various ways to compress code for RISC-based systems**
  - ◆ Code is uncompressed prior to execution
- **This work proposes a combined approach for VLIW embedded systems**
  - ◆ Exploit the compiler's role
  - ◆ Allow compressed code to be cached
  - ◆ Not only reduce ROM size, but increase in lfetch effectiveness as well

---

---

## Previous Research

- **Wolfe [Wolfe et al. 92,94] proposed ways to execute compressed program on an embedded RISC architecture**
  - ◆ One common histogram was used for all benchmarks
  - ◆ No caching of compressed code
- **Several Industrial solutions include:**
  - ◆ **IBM CodePack** - uses adopted Huffman algorithm but in rather different way from the current work
    - ◇ no caching of compressed code
  - ◆ **Both ARM Thumb and SGI MIPS16** provide a special subset of the original ISA which in turn reduces their strength and ultimately increases the Op count

---

---

## Previous Research

- **Cooper and McIntosh [Cooper, McIntosh 99] reorganized code at the assembly level via suffix-tree code compression**
  - ◆ Only modest gains achieved ~20%
- **Similar to that, a number of studies [Fraser 97,99] consider elaborate compression algorithms on assembly level**
  - ◆ Goal is smaller image size
- **Work by Liao [Liao et al. 95] also on assembly level achieves significant code size reduction by increasing number of branches and slight increase in Op count**
  - ◆ It is in fact complementary to our work

---

---

## Present work

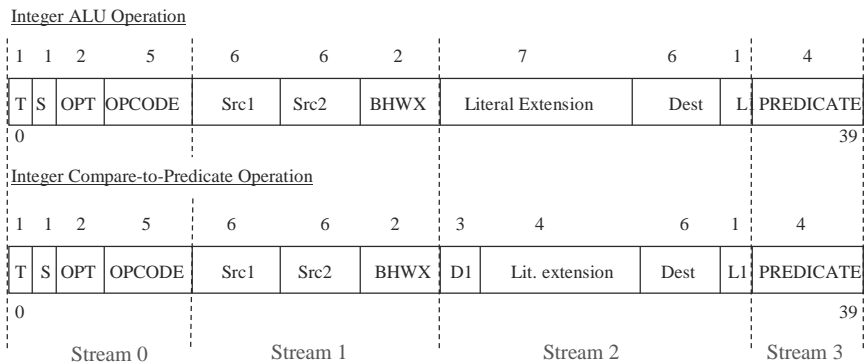
- **Given an individual operation there are two ways to reduce its size**
  - ◆ Compress it (e.g, Huffman compression) based on frequency of this instruction or its elements in the *static* code
  - ◆ Create unique tailored encoding for this operation based on amount of resources used by it
  - ◆ Both done on the lowest level: ISA and combine software and hardware decisions
  - ◆ Custom approach to each application results in near optimal results
- **Code is kept compressed in the ICache**
- **Integrated Branch Prediction with branch target resolution**
- **Modified VLIW Icache architectures**
- **The use of PLA to implement customized decoder**
- **Branch targets change**
  - ◆ Solution: special hardware is added to remap targets

# Compression

- Compression can be performed in number ways
- In this study we used various adaptations of the Huffman encoding algorithm
  - ◆ Byte Based - single byte is considered as an alphabet entry
  - ◆ Stream Based - variable length chunks of ops are compressed separately
  - ◆ Full Op Based - the whole Op is an alphabet entry
- Since the whole code segment is statically available, Huffman gives near theoretically optimal results

# Compression

- Byte based Huffman is the very traditional approach
- Stream based approach exploits higher repetitiveness of certain fields in operations:



## Tailored encoding

- Tailored encoding does not COMPRESS Ops, but rather creates *new* encoding for them
  - ◆ To enhance its effectiveness, the compiler forces long literals into separate 'Load Immediate' ops

Original ADD Operation and its Custom Encoded equivalent

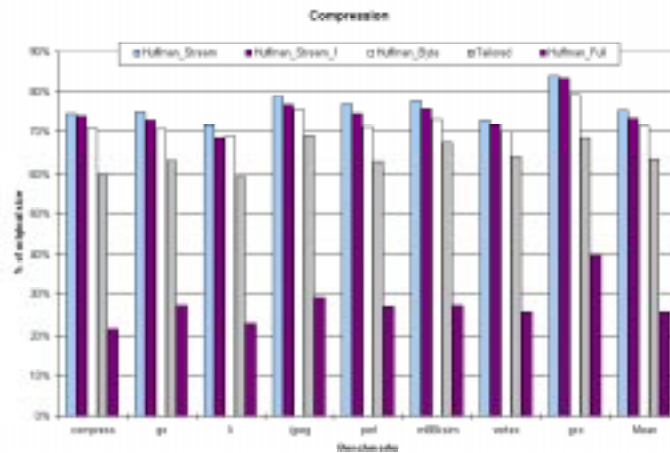
1	1	2	5	6	6	2	7	6	1	4
T	S	OPT	OPCODE	Src1	Src 2	BHWX	Literal Extension	Dest	L1	PREDICATE
0	1	2	5	6	6	2	5	1		39
T	OPT	OPCODE	Src1	Src 2	BHWX	Dest	L1			
0										27

Original PBRA Operation and its Custom Encoded version

1	1	2	5	6	4	5	6	6	1	4
T	S	OPT	OPCODE	Src1	Src 2	Br Delay	Service	Dest	L1	PREDICATE
0	1	2	5	4	1	1	2			39
T	OPT	OPCODE	Src1	Src 2	Br Delay	Dest				
0										15

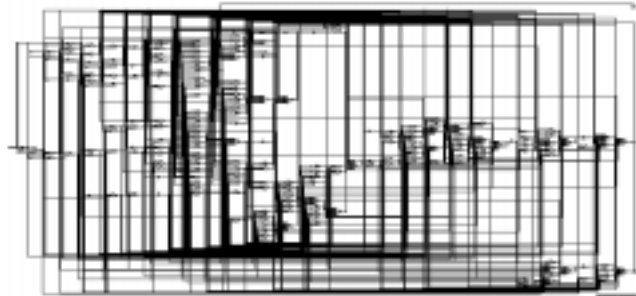
## Compression Results

- Degree of compression:
  - ◆ without branch remapping table overhead



## Decoder Generation

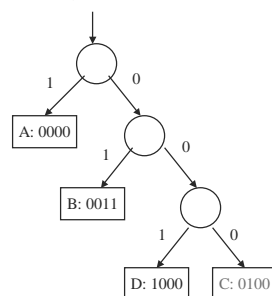
- The Compiler automatically generates:
  - ◆ Suitable compression method ...
  - ◆ ... and hardware description of the decoder in Verilog
- The Verilog code is then synthesized with Synopsys:



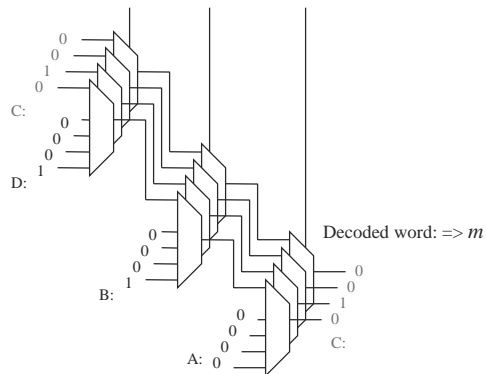
## Decoder Complexity Estimation

- To *estimate* Decoder complexity we do not have to synthesize all possible configurations
  - ◆ If Muxes are interpreted as TG logic (4/2 Trans/Mux) then the *worst case* number of elements is:  $2m(2^n - 1) + 4m(2^n - 2^{(n-1)} - 1) + 2n$

Original Tree:



Huffman Code: 0 0 0  $\Rightarrow n$  bits



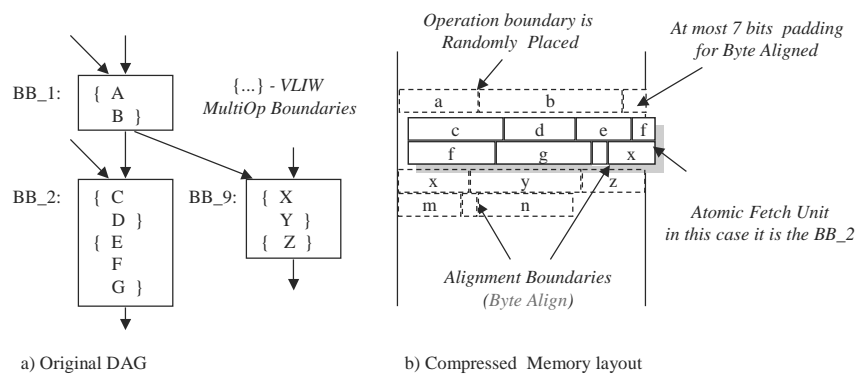


# Integral design of Instruction Fetch

- Code size reduction methods must also consider the microarchitecture of the system
  - ◆ the critical step for executing compressed Ops is quick decompression
  - ◆ Since ICache is kept compressed, it must be re-designed
  - ◆ The address space conversion between the Compressed and the Uncompressed spaces is done by special hardware which is also used for branch prediction

# ICache design

- An atomic IFetch unit – a Basic Block :
  - ◆ The beginning of the block is byte aligned in memory
  - ◆ A procedure call is a basic block boundary
  - ◆ Entire block is executed once first op is fetched



---

---

## ICache design (cont.)

- The use of a larger region (e.g, hyperblocks or treeregions) is also possible
  - ◆ It is only matter of performance, not correctness
- Intelligent code layout can be performed to further increase performance
  - ◆ Reserved as future work
- If both are done, fewer points of address conversion will be needed

---

---

## ICache design – The ATB

- To solve the branch target mapping problem, introduce a special hardware buffer (*Address Translation Buffer - ATB*), similar to TLB, which maps compressed address space into the uncompressed one
  - ◆ Used also by CodePack and by Wolfe
- Each ATB entry holds a pair of addresses for each branch target: *original (uncompressed)* and its *new location in compressed space*
  - ◆ Also holds additional information that helps decompression
  - ◆ ATB has a small number of entries (e.g., 32) and holds current working set of entries
  - ◆ Full list of entries (one for each BB in the code) is stored in *Address Translation Table (ATT)* in compressed form along with the rest of the code

## ATB

- Since the atomic unit of cache transfer is a BB, the number of entries for ATT is tolerable
- The ATB miss ratio is very low (<1%)
- Besides the main function of address translation, the ATB assists in decompression and branch prediction
  - ◆ Since last operation of a BB by definition is a branch, a prediction hardware is associated with each ATB line
    - ◇ More complex predictors are possible
  - ◆ A two-bit counter is used to predict branch outcome
    - ◇ Call-return stacks, etc., could also be included

## Compression including ATT

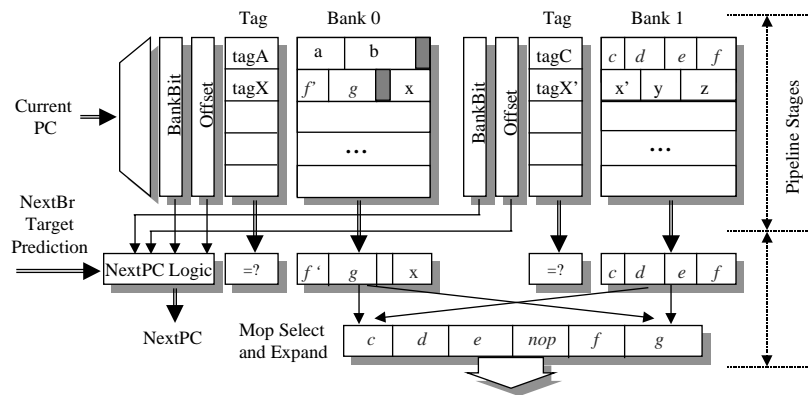
- When the size of ATT is added to the code we have the following:





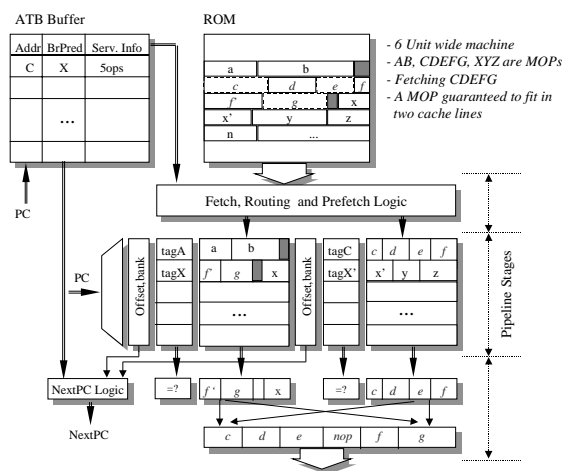
## ICache (cont.)

- Banked Cache for Zero-Nop encoded TINKER architecture [Conte, et al., MICRO-29]



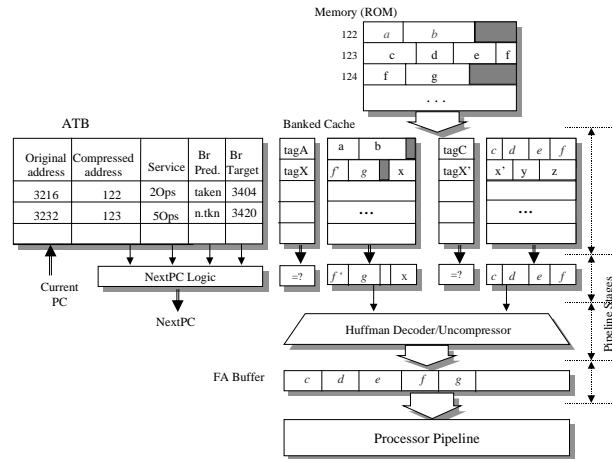
## ICache (cont.)

- For Tailor encoding – ATB added to NextPC logic



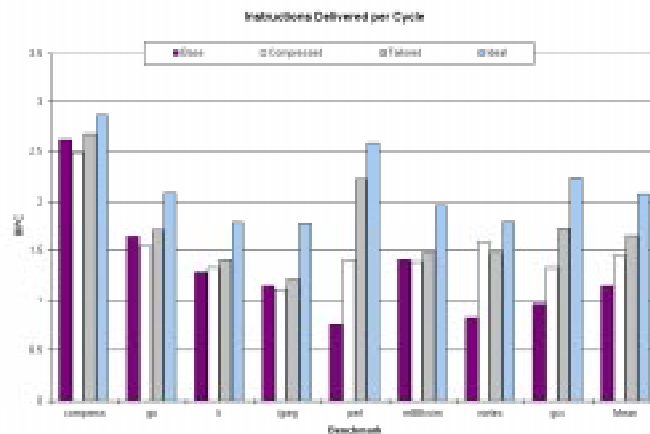
## ICache (cont.)

- For compression, ICache with a fully-associative uncompression buffer and ATB



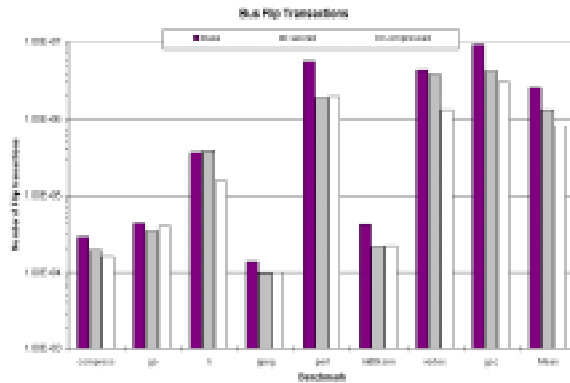
## IFetch Performance

- The Instructions-Delivered-Per-Cycle metric
  - ◆ 6 OP wide MOP, Icache is 16KB, 2-way SA
  - ◆ *Base*- Uncompressed, *Compressed*- Huffman-Full, *Ideal*- perfect Icache



## Bus Transaction Density

- Change in bus power due to ICache miss
  - ◆ Fewer misses and transfer of compressed blocks reduce the number of bit flips on bus, but entropy is higher than uncompressed



## Conclusions

- Instruction fetch must be factored into the decision
- Compiler can play a significant role
- Comparing schemes
  - ◆ Tailored-
    - ◇ Simplicity of decoding results in better ifetch performance
    - ◇ Smaller hardware for decoding
  - ◆ Compressed-
    - ◇ Huffman-Full – Best compression, unreasonably large decoder (but a synthesized decoder may be tolerable)
    - ◇ Huffman-Streams – Poor ifetch performance, large decoder size
    - ◇ Huffman-Byte – Small compression ratio, modest decoder size
- Future work:
  - ◆ Compiler optimizations impact on compression (e.g., [Liao et al. 95])
  - ◆ Different atomic units for ATT/ATB