Instruction Folding in a Hardware-Translation Based Java Virtual Machine

Hitoshi Oi

The University of Aizu

May 4, 2006

Computing Frontiers 2006

Outline

- Introduction to the Java Virtual Machine and Hardware Translation
- Instruction Folding of Java Bytecodes
  - Implementation in Sun's Pico Java-II
  - Removal of Uncommon Cases
- Performance Evaluation
- Summary and Future Work

Introduction to the Java Virtual Machine

Features of Java©

- Object-Oriented
- Network
- Security
- Platform Independent

Java Virtual Machine

- Abstract instruction set architecture
- Placed between Java applications and underlying platform
- Stack-based architecture

Implementation: Interpretation

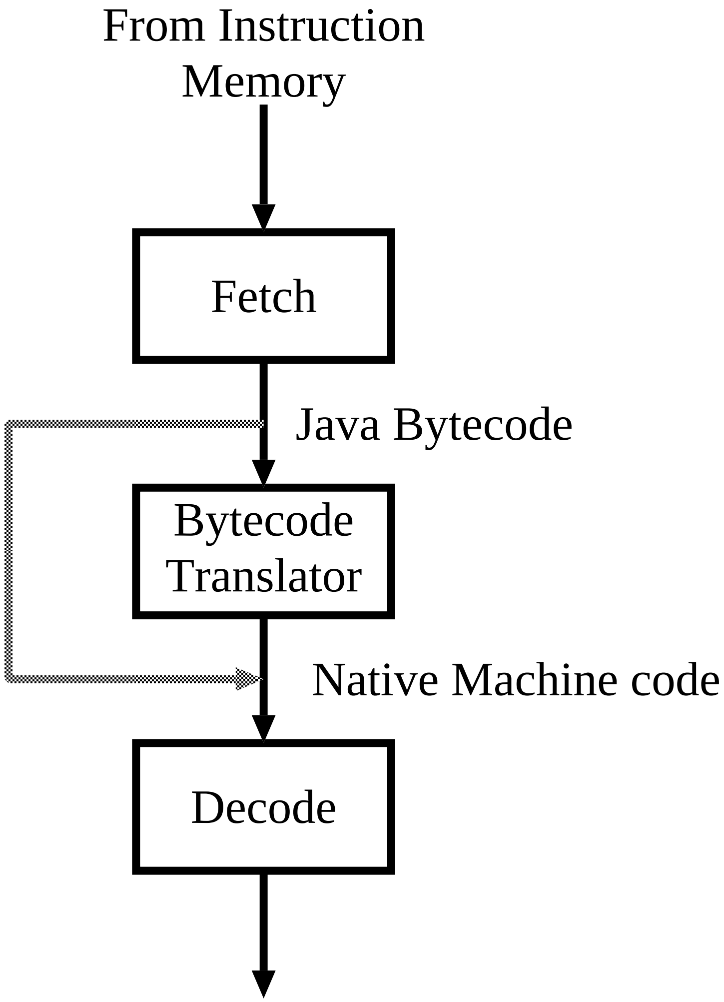
```
switch(*bytecode){
 case ILOAD:
  STACK[SP + 1] = STACK[LV + *(bytecode + 1)];
  SP = SP + 1;
 ..........
}
```

- A software written in native instructions to the platform reads a Java application and interprets its bytecodes.
- Flexible and relatively inexpensive, thus widely adopted (an interpreter is just another program on the platform).
- Slow : Checking a flag takes << 1 clock cycle in hardware but several cycles in software.

Just-In-Time Compilation

- Frequently executed methods (functions) are compiled to native instructions.
- Works well for server side applications but may not be feasible for client side applications (especially those running on portable devices) because:
  - Time and power consumption for compilation
  - Expansion of program size
  - Client side application may not be repeatedly executed and cannot absorb above compilation overhead.

Hardware Translation



- A small translation module between the fetch and decode stages in the pipeline converts simple Java bytecodes into native instruction sequences.
- Complex bytecodes generate branch instructions to emulation routines.
- Small overhead (12K gates in ARM Jazelle) and minimum changes to processor core.

Hardware Translation: Example

Java	Bytecode	ARM Machine Code
b = a + b;	ILOAD_1	LDR R0 [R7, #4]
	ILOAD_2	LDR R1 [R7, #8]
	IADD	ADD R0 R1
	ISTORE_2	STR R0 [R7, #8]

- R0 to R3 hold top four words of operand stack
- R7 points to the local variable 0.
- In the above example, local variables a and b are numbered 1 and 2, respectively.

Redundancies in Hardware-Translation

- Frequent Memory Access for Local Variables:
  - Every local variable access goes to memory
  - A small register file dedicated for local variable storage can eliminate most of memory accesses (see LCTES05 paper).
- Redundant Stack Operations:
  - An arithmetic operation takes four bytecodes (two pushes, arithmetic and one pop)
  - Microprocessors can perform an equivalent operation in a single instruction.
  - Pico Java-II (a dedicated Java processor) removes this redundancy by folding multiple bytecodes into a single operation.

Java Bytecodes Categories in Pico Java-II

LV: A local variable load or load from global register or push constant (e. g. ILOAD)

OP: An operation that uses the top two entries of stack and that produces a one-word result (IADD)

BG2: An operation that uses the top two entries of the stack and breaks the group (IF_ICMPEQ)

BG1: An operation that uses only the topmost entry of the stack and breaks the group (IFEQ)

MEM: A local variable store, global register store, and memory load (ISTORE)

NF: A non-foldable instruction (GOTO)

Foldable Bytecode Sequences in Pico Java-II

Group1: LV LV OP MEM

Group2: LV LV OP

Group3: LV LV BG2

Group4: LV OP MEM

Group5: LV BG2

Group6: LV BG1

Group7: LV OP

Group8: LV MEM

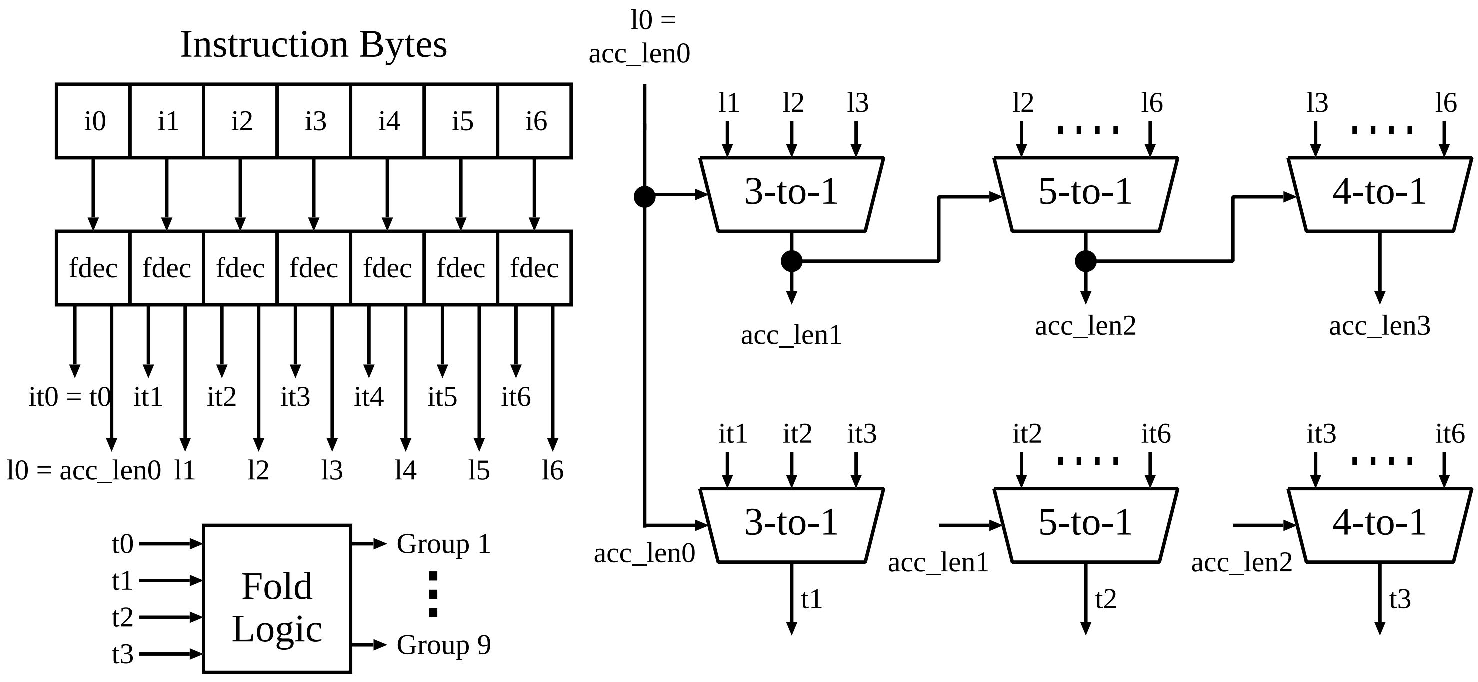
Group9: OP MEM

Example:

Group1: ILOAD_1, ILOAD_2, IADD, ISTORE_2

→ add \$2, \$1, \$2

Foldable Bytecode Detection Logic (Pico Java-II)

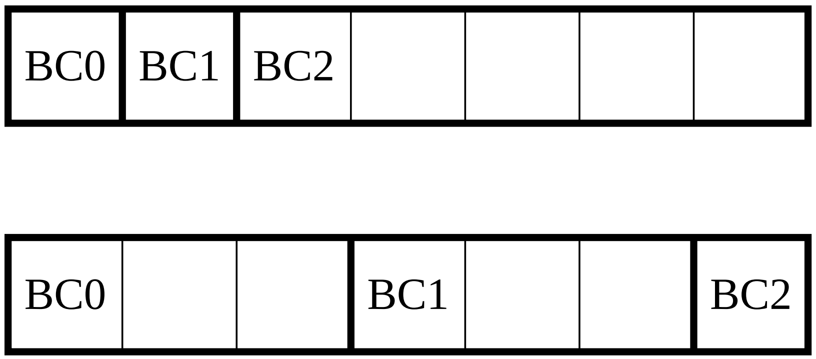


Motivation of This Work

- Pico Java-II is a dedicated Java processor: bytecode decoding begins from the fetch stage (bytecode length decoding).
- A hardware-translation JVM should use the existing RISC processor pipeline as much as possible. Also, the changes should be minimum and localized to the translation module inserted between the fetch and decode stages on the pipeline.
- In this paper, we propose an instruction folding folding scheme with reduced hardware complexity and show that it still achieves the similar performance as Pico Java-II.

Variable Lengths of Bytecodes

- The length of a foldable bytecode ranges from one to three bytes
- This implies that there are three choices for the opcode of the second bytecode.
- Similarly, there are five choices for the opcode of the third bytecode.



Removal of Uncommon Cases

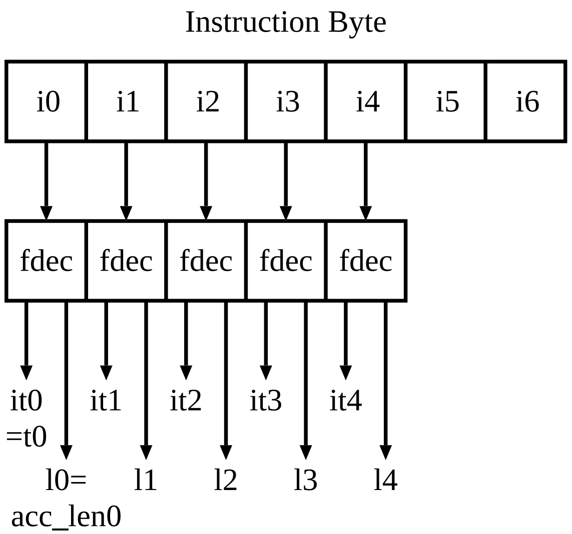
SIPUSH

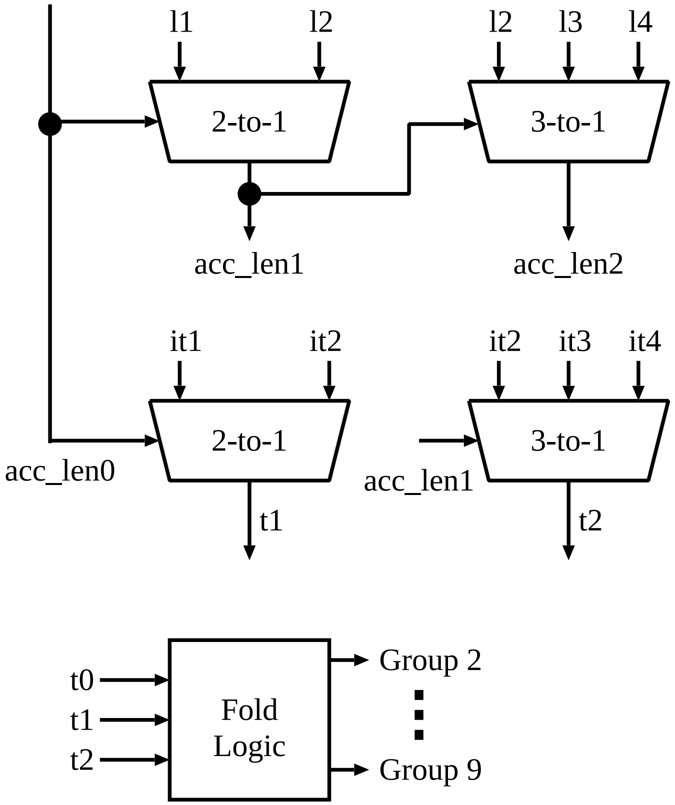
- Only instance of three-byte long LV bytecode.
- Removal of SIPUSH reduces the number of choices for the second and third bytecode opcodes.

Group1 Sequence (LV LV OP MEM)

- Only instance of four bytecode sequence
- Removal of Group 1 reduces the number of stages in the foldable sequence detection logic.

Foldable Bytecode Detection Logic (Simplified)





Latency Reduction: 11%
Area Reduction: 35%

Performance Evaluation

- JVM and JRE: Kaffe version 1.0.7 (interpretation only)
- Compare Pico Java-II, proposed mechanism and Two-Bytecode version of Pico Java-II.
- Show the fractions of folded bytecodes and their breakdown into folding groups (Groups 1 to 9).
- LV_0 to 15 are allocated on the local variable cache (stores always hit, loads hit if previously accessed).
- Abbreviations: **F4** (Pico Java-II), **F3** (Proposed), **F2** (Two-Bytecode version of Pico Java-II).

Benchmark Programs (1)

SAXON Version 6.0 with XSLTMark 1.2.0

chart Generates an HTML chart of some sales data (select, control).

decoy Simple template with decoy patterns to distract the matching process (match).

encrypt Performs a Rot-13 operation on all element names and text nodes (function).

trend Computes trends in the input data (select, functions).

Benchmark Programs (2)

ECM Embedded CaffeineMark
(Sieve, Loop, Logic, Method and Float).

DES DES encryption and decryption of a text file using the Bouncy Castle Crypto package.

PNG Extract PNG image properties (e. g. pixel size, bit depth) using `com.sixlegs.png` .

Execution Summary: SAXON

Bench -mark	Bytecode Types (%)						Exec Len.
	LV*	OP	BG1	BG2	MEM	NF	
chart	44.4 (0.7)	4.3	7.7	12.6	4.1	26.8	11.6
decoy	44.4 (0.2)	2.3	8.3	9.9	4.0	31.1	8.8
encrypt	42.5 (0.1)	4.0	7.4	14.0	3.3	28.8	10.8
trend	39.9 (0.1)	1.6	9.8	5.8	2.6	40.3	4.9

* Numbers in parentheses are fractions of SIPUSH bytecodes

- High fraction of NF bytecodes.
- Short execution lengths.

Execution Summary: ECM, DES and PNG

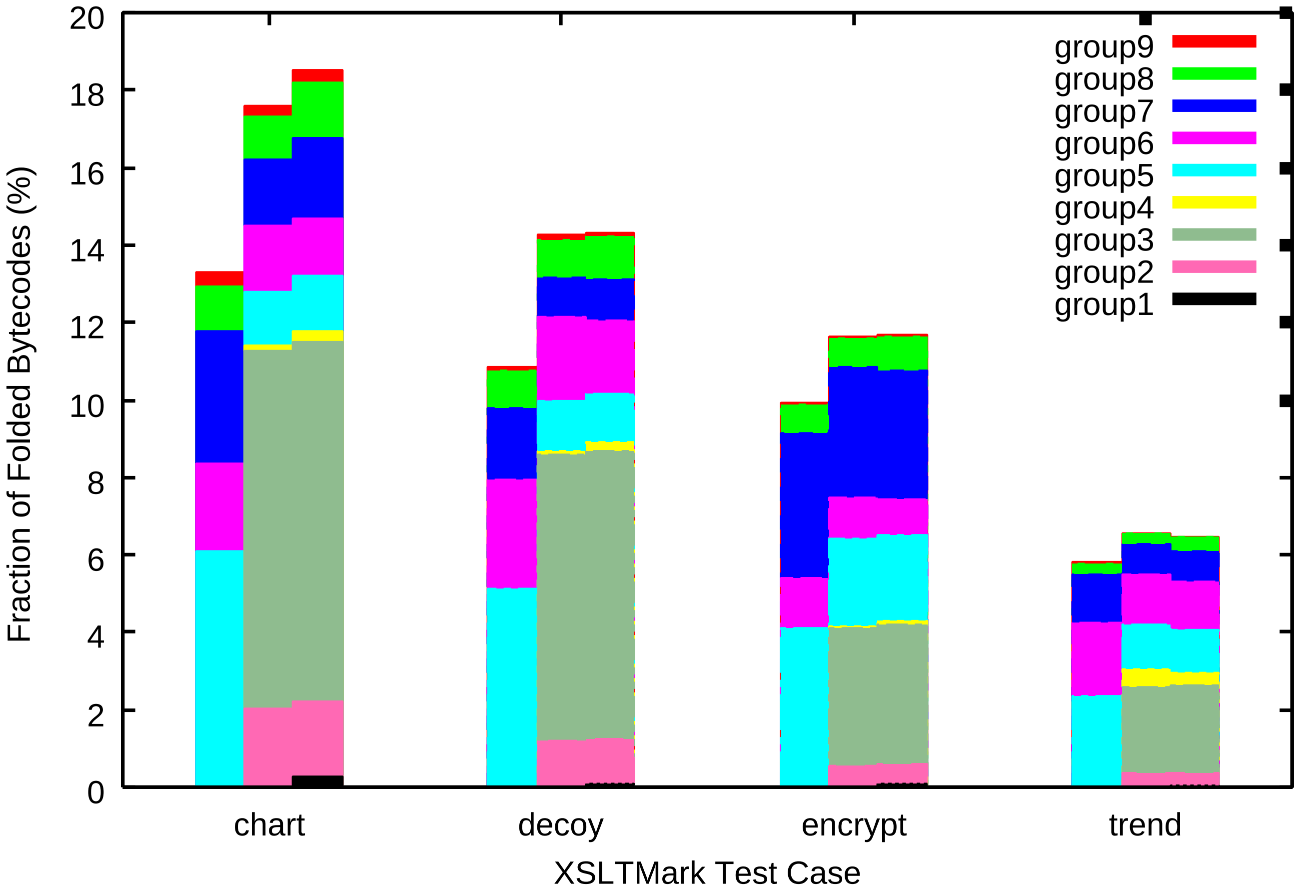
Bench -mark	Bytecode Types (%)						Exec Len.
	LV*	OP	BG1	BG2	MEM	NF	
ECM	45.3 (0.0)	4.7	9.1	14.9	6.7	19.2	90.6
DES	43.8 (0.7)	24.9	1.6	9.8	9.4	10.5	66.1
PNG	42.8 (2.5)	11.0	3.8	13.3	2.9	26.3	24.3

ECM Long uninterrupted execution

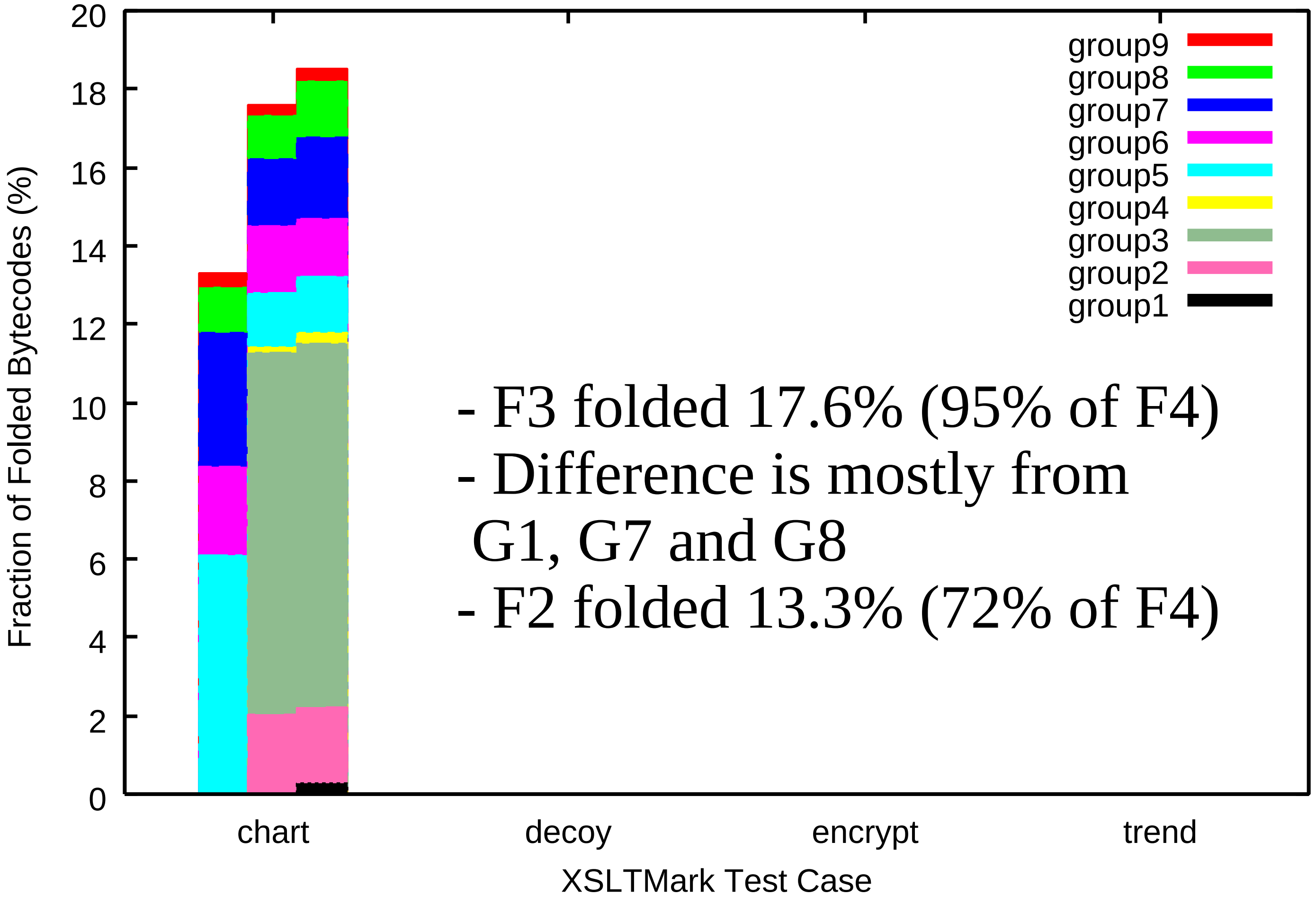
DES Small fraction of NF bytecodes

PNG Large fraction of SIPUSH bytecode

Results: SAXON with XSLTMark Test Cases

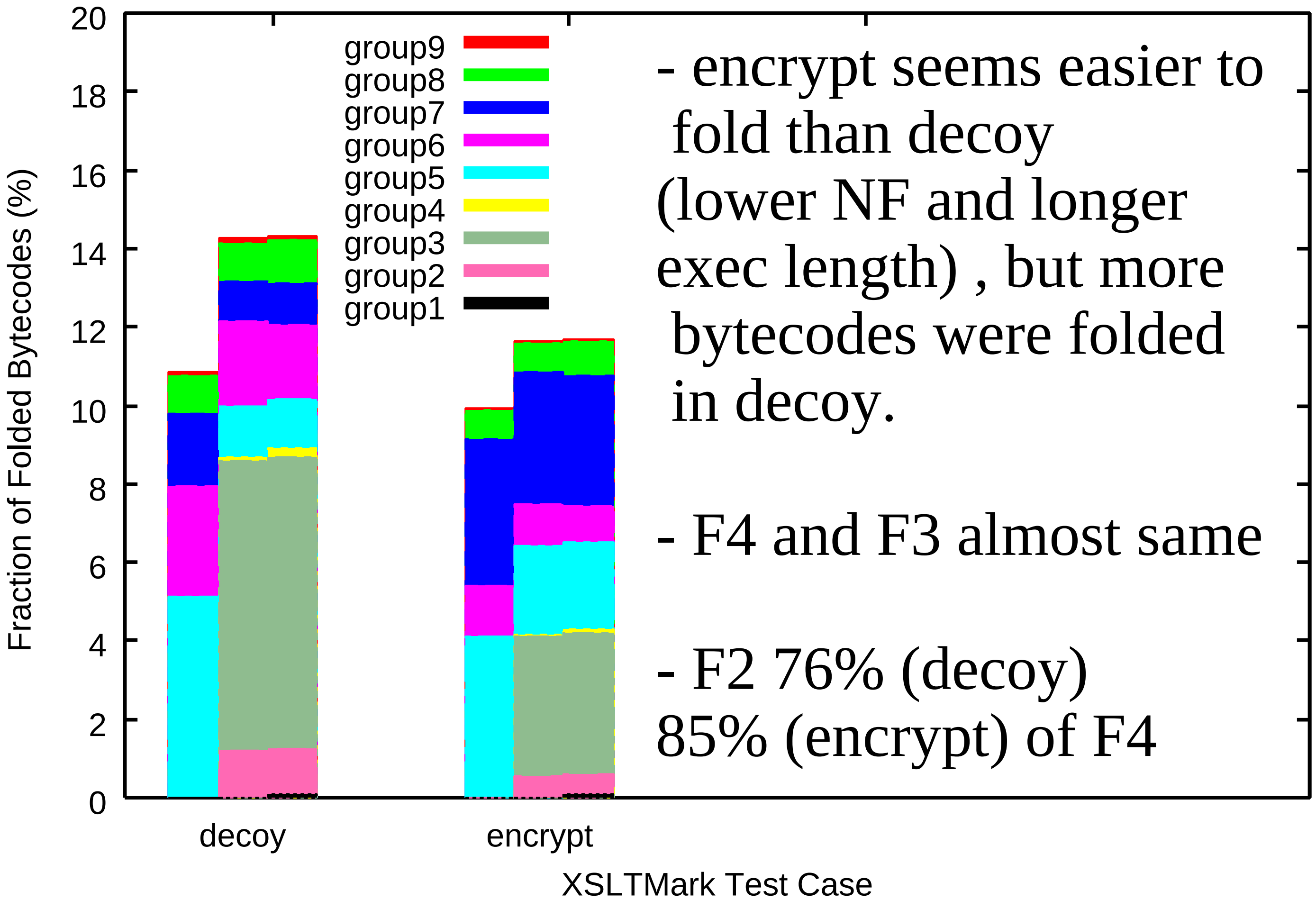


Result: SAXON with Chart

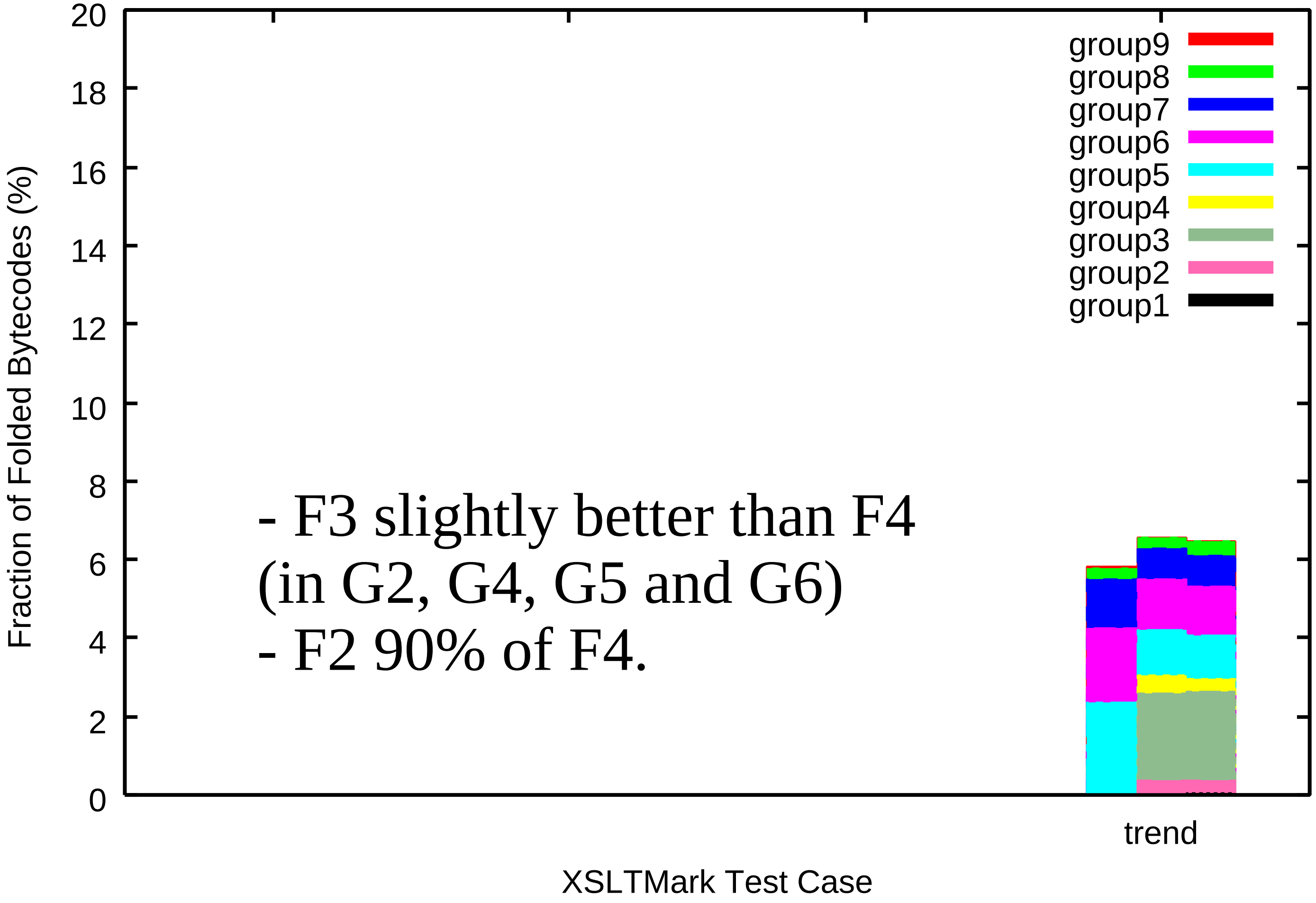


- F3 folded 17.6% (95% of F4)
- Difference is mostly from G1, G7 and G8
- F2 folded 13.3% (72% of F4)

Result: SAXON with Decoy and Encrypt

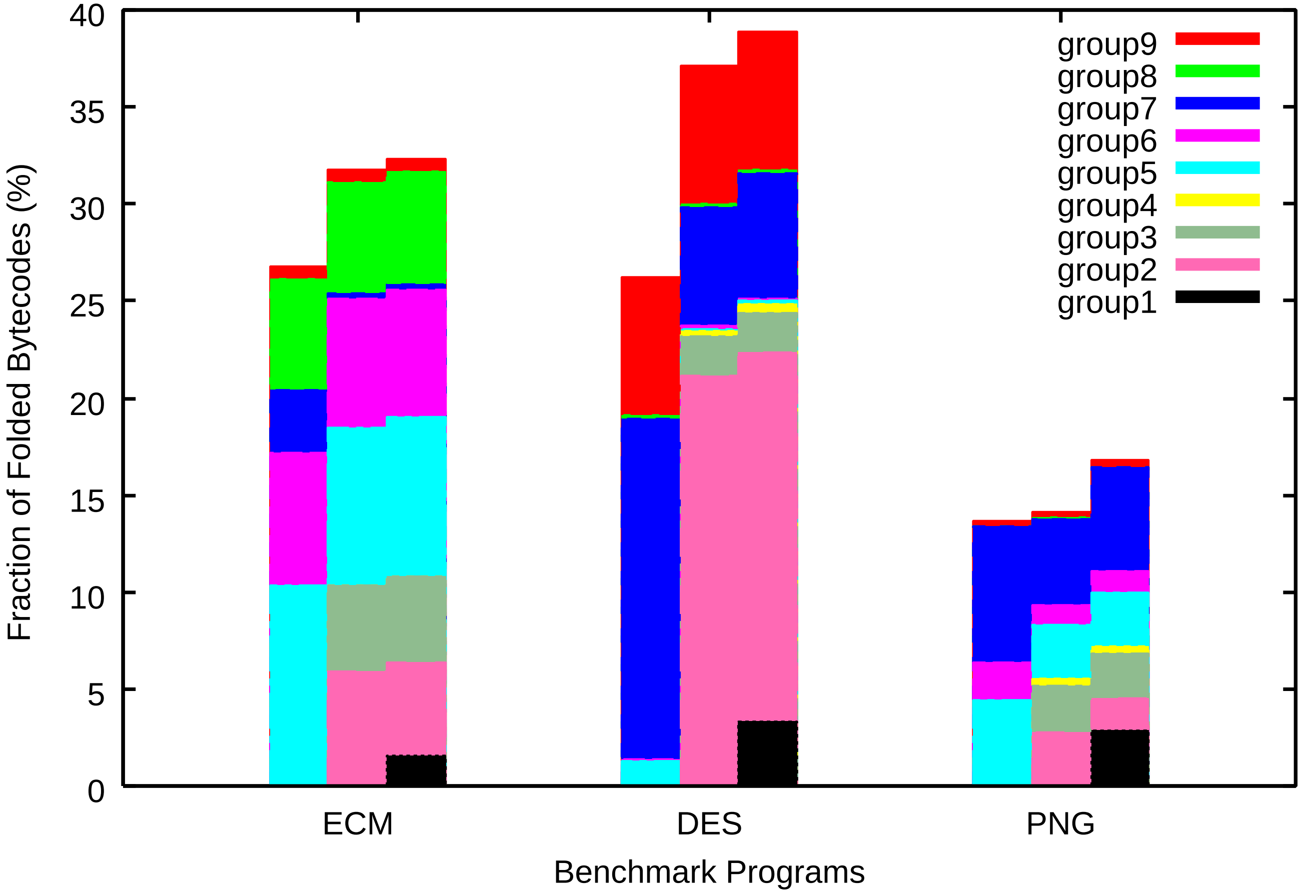


Result: SAXON with Trend

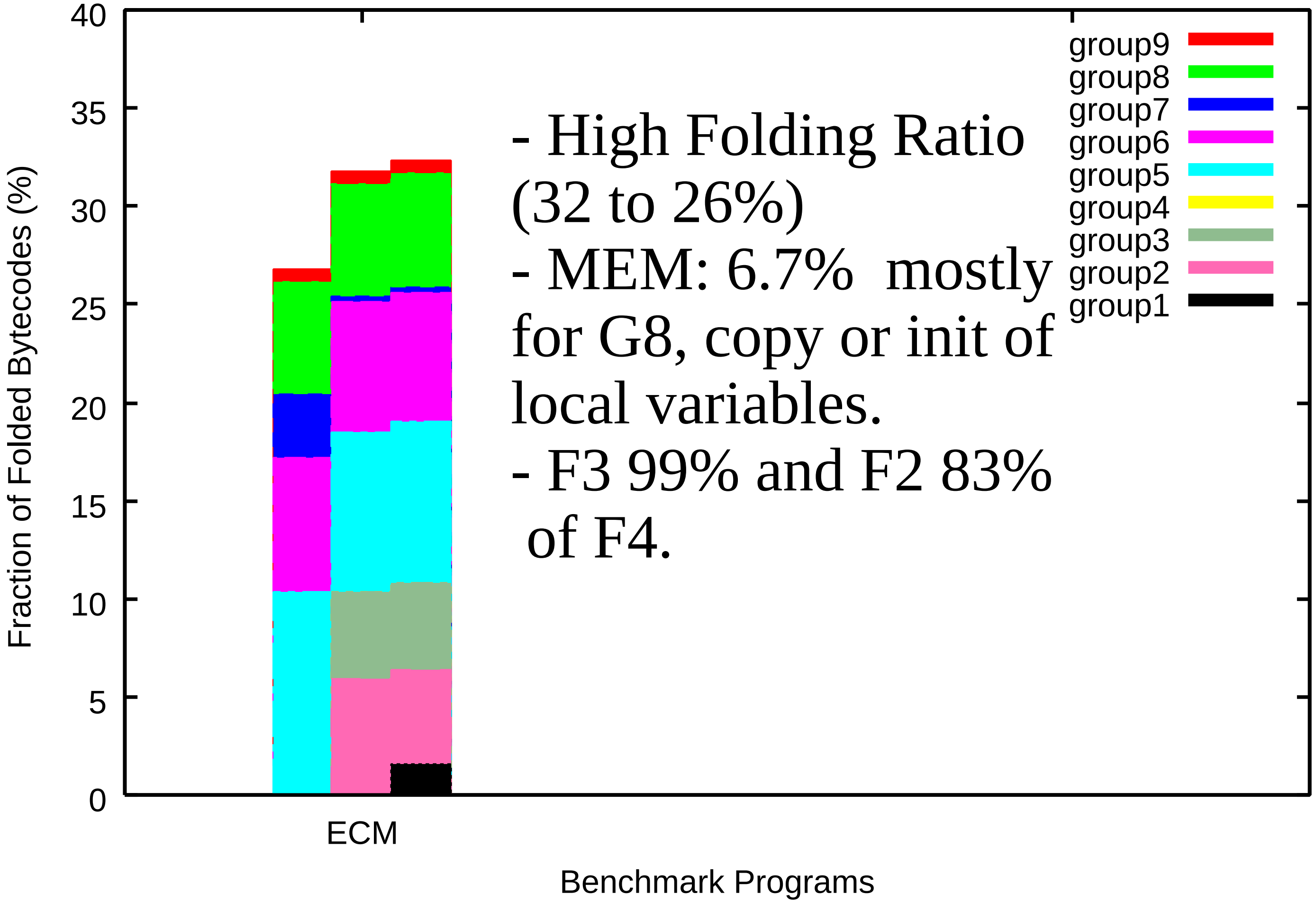


- F3 slightly better than F4
(in G2, G4, G5 and G6)
- F2 90% of F4.

Results: ECM, DES and PNG

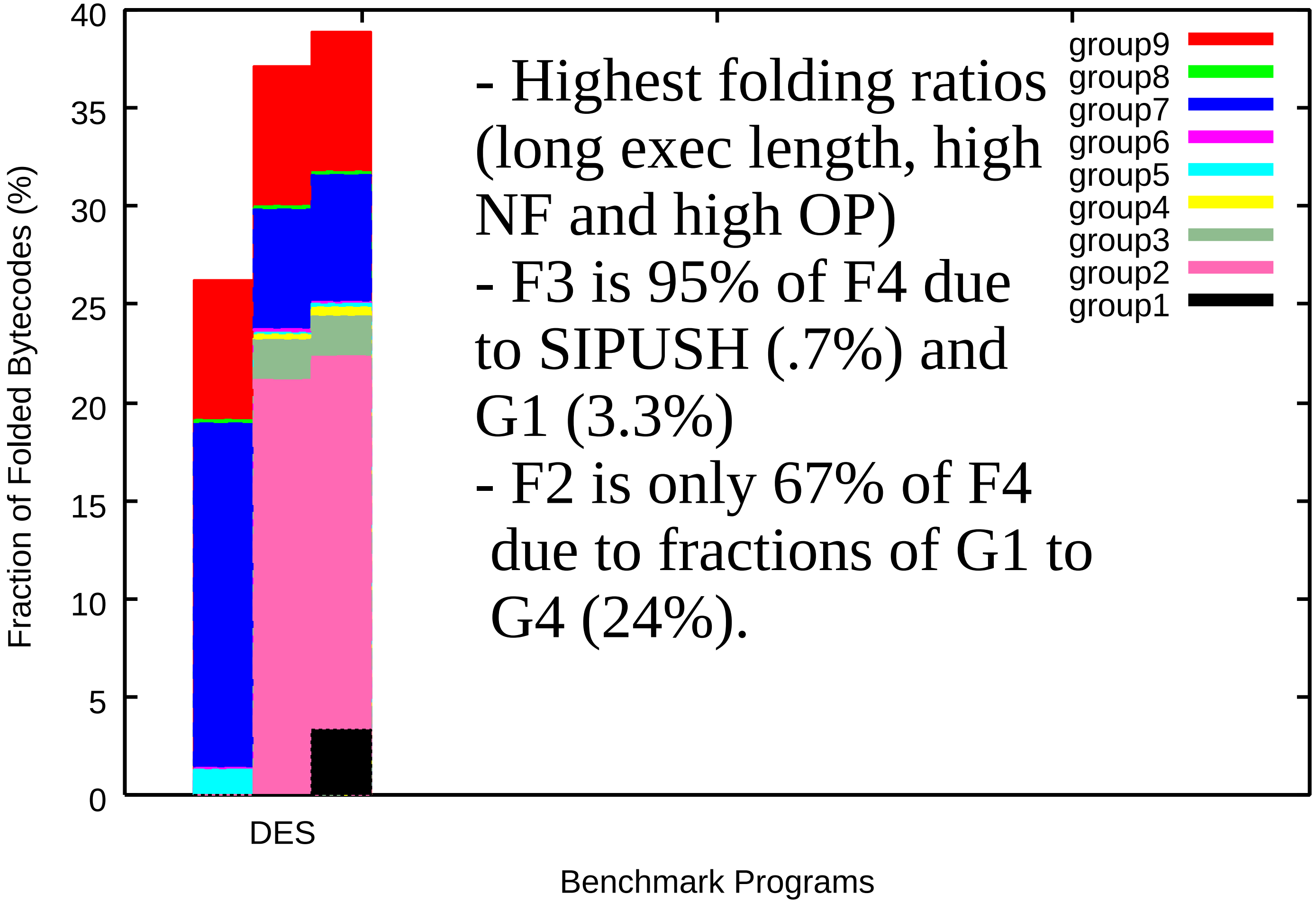


Result: Embedded CaffeineMark



- High Folding Ratio (32 to 26%)
- MEM: 6.7% mostly for G8, copy or init of local variables.
- F3 99% and F2 83% of F4.

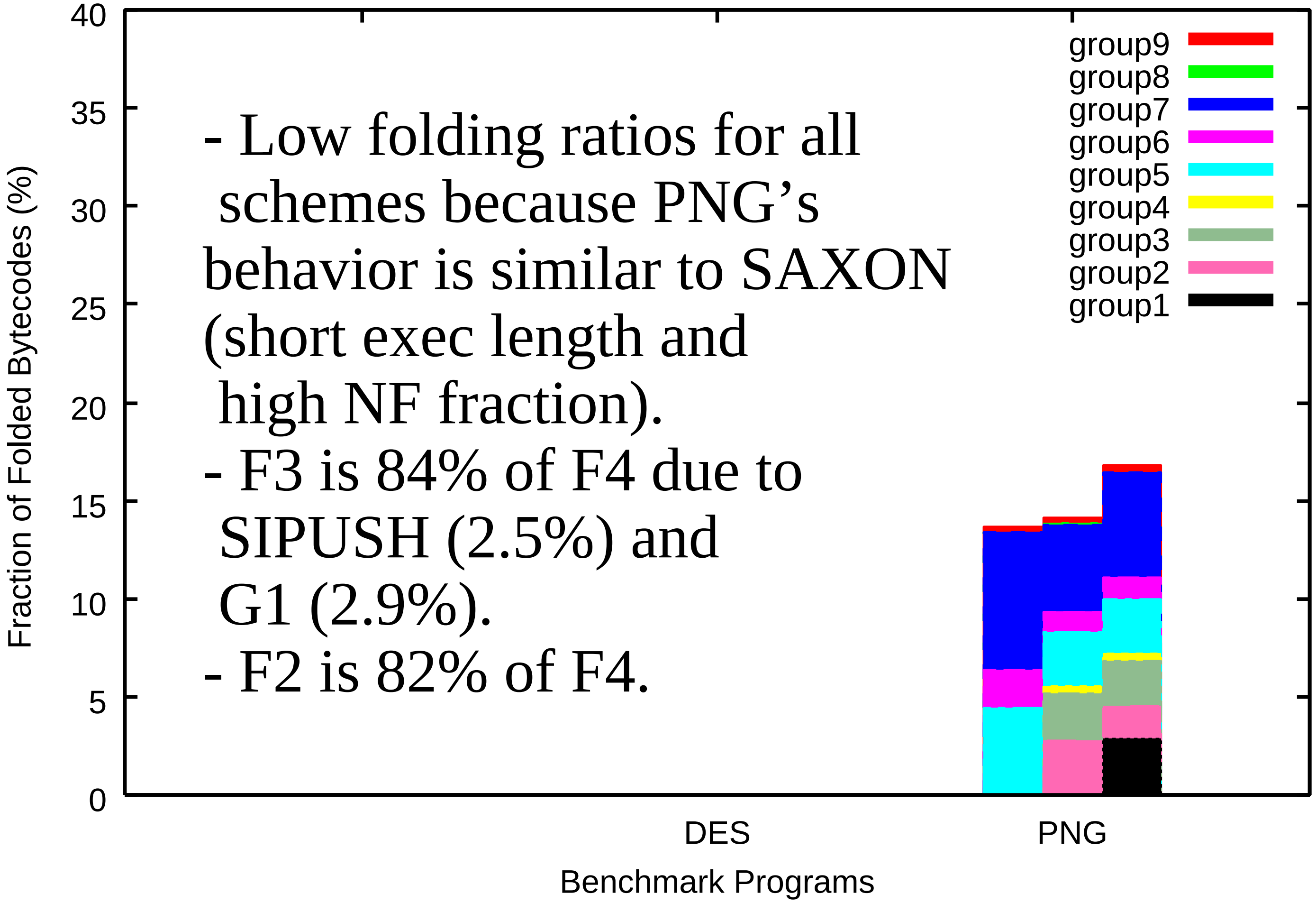
Result: DES Encryption



- Highest folding ratios (long exec length, high NF and high OP)
- F3 is 95% of F4 due to SIPUSH (.7%) and G1 (3.3%)
- F2 is only 67% of F4 due to fractions of G1 to G4 (24%).

Result: PNG Image Property Extraction

- Low folding ratios for all schemes because PNG's behavior is similar to SAXON (short exec length and high NF fraction).
- F3 is 84% of F4 due to SIPUSH (2.5%) and G1 (2.9%).
- F2 is 82% of F4.



Summary and Future Work

- An instruction folding scheme with reduced hardware complexity was proposed.
- The proposed scheme achieved 84.2% (or 95.0% if PNG is excluded) or higher folding ratios with respect to Pico Java-II.
- The folding detection logic was reduced by 11% in latency and by 35% in area (0.35μ rule).
- More complete hardware model (currently, only folding detection logic was used for latency and area estimations)
- More complete workload (not only hardware-translatable bytecodes, but also emulated bytecodes and native methods).

Acknowledgment

- Partial support by the University of Aizu Competitive Research Funding (Grant Number P25).
- Yuichi Okuyama's contribution on the area and delay estimation of the folding logic circuits.
- Helpful comments from anonymous reviewers

Thanks for Your Attention !

Any Question ?