On the Design of the Local Variable Cache in a Hardware Translation-Based Java Virtual Machine

Hitoshi Oi

The University of Aizu

June 16, 2005

Languages, Compilers, and Tools for Embedded Systems (LCTES'05)

Outline

- Introduction to the Java Virtual Machine
- Implementation: Software Interpretation, Just-in-time Compilation, Hardware Translation
- Local Variables Cache
- Base Design Performance
- On-the-fly Profiling of Local Variable Access
- Parameter Passing with Local Variable Cache
- 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

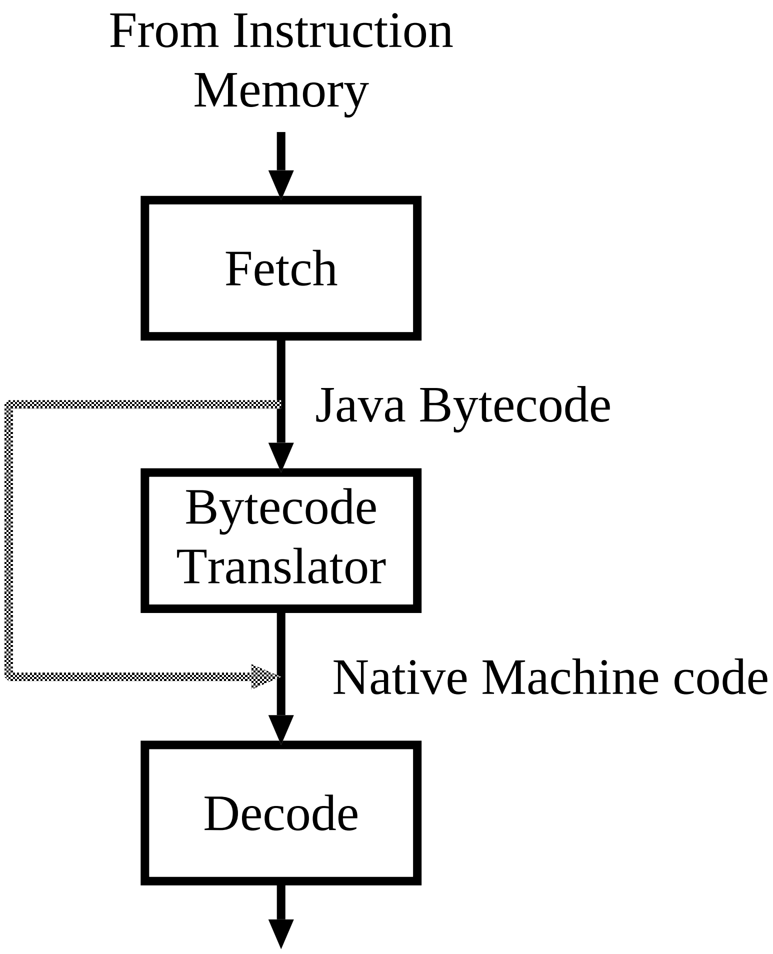
```
switch(*bytecode){
 case ILOAD:
  STACK[SP + 1] = STACK[LV + *(bytecode + 1)];
  SP = SP + 1;
 ..........
}
```

- 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]

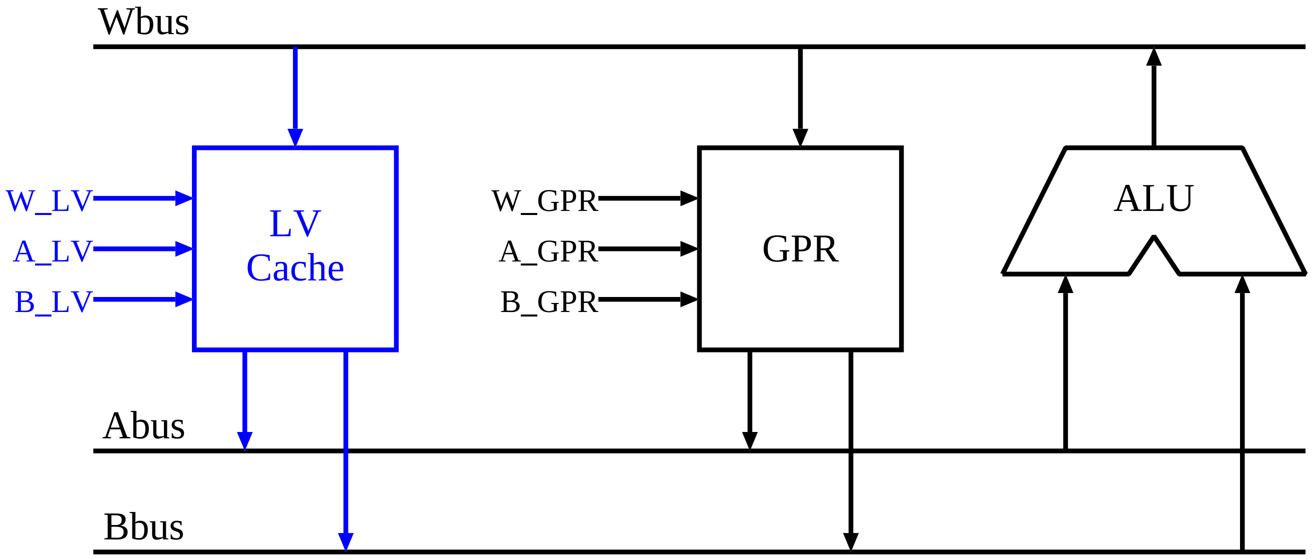
- R0 to R3 hold top four words of operand stack
- R7 points to Local Variable 0.
- In the above example, local variables a and b are numbered 1 and 2, respectively.
- Translation is on the single bytecode basis and causes frequent memory accesses.

Objectives

- Investigate the local variable access behavior of Java applications.
- Add a small register file to the datapath of the hardware translation-based JVM and use it as a local variable cache.
- Evaluate the effectiveness of the local variable cache by changing its size.
- Options to further improve the effectiveness of the local variable cache.

Local Variable Cache

A small register file with two status bits (**valid** and **modified**) is added to the datapath. Utilize existing control and data signals as much as possible.



ILOAD: Push the content of a local variable

```
if(idx < LVC_SIZE){ index is within LVC
 if(v[idx]){ check valid bit
  MOV R[srp], LV[idx]; copy from LVC to t-o-s register
 }
 else{ entry not valid
  LDR R[srp], [R7, 4*idx]; load from memory to t-o-s register
  MOV LV[idx], R[srp]; also copy to LVC
  v[idx] = 1; set valid bit
 }
}
else{ index is outside LVC
 LDR R[srp], [R7, 4*idx]; load from memory to t-o-s- register
}
```

ISTORE: Pop t-o-s word and write it back to a LV

```
if(idx < LVC_SIZE){ index is within LVC
  MOV LV[idx], R[rsp];  copy from t-o-s regisger to LVC
  v[idx] = 1; m[idx] = 1; set valid and modified bits
}
else{ index is outside LVC
  STR R[rsp], [R7, 4*idx]; copy from t-o-s register to memory
}
```

INVOKEVIRTUAL: Invoke another method

```
for(i = 0; i < LVC_SIZE; i++){ For all LVC entries
  if(m[i]){ if the entry is modified
    STR LV[i], [R7, 4*i]; write back to memory
    m[i] = 0; clear modified bit
  }
  v[i] = 0; clear all valid bits
}
```

IRETURN: Return from current method

```
for(i = 0; i < LVC_SIZE; i++){
  v[i] = 0; m[i] = 0;
} clear all valid and modified bits
```

Performance Evaluation Environment

- JVM and Runtime Environment
  - Kaffe 1.1.4, an open-source JVM/JRE.
  - Compiled with interpretation-only option.
- Change the size of local variable cache from 4 to benchmark's maximum
- Performance metrics: hit ratio to the local variable cache

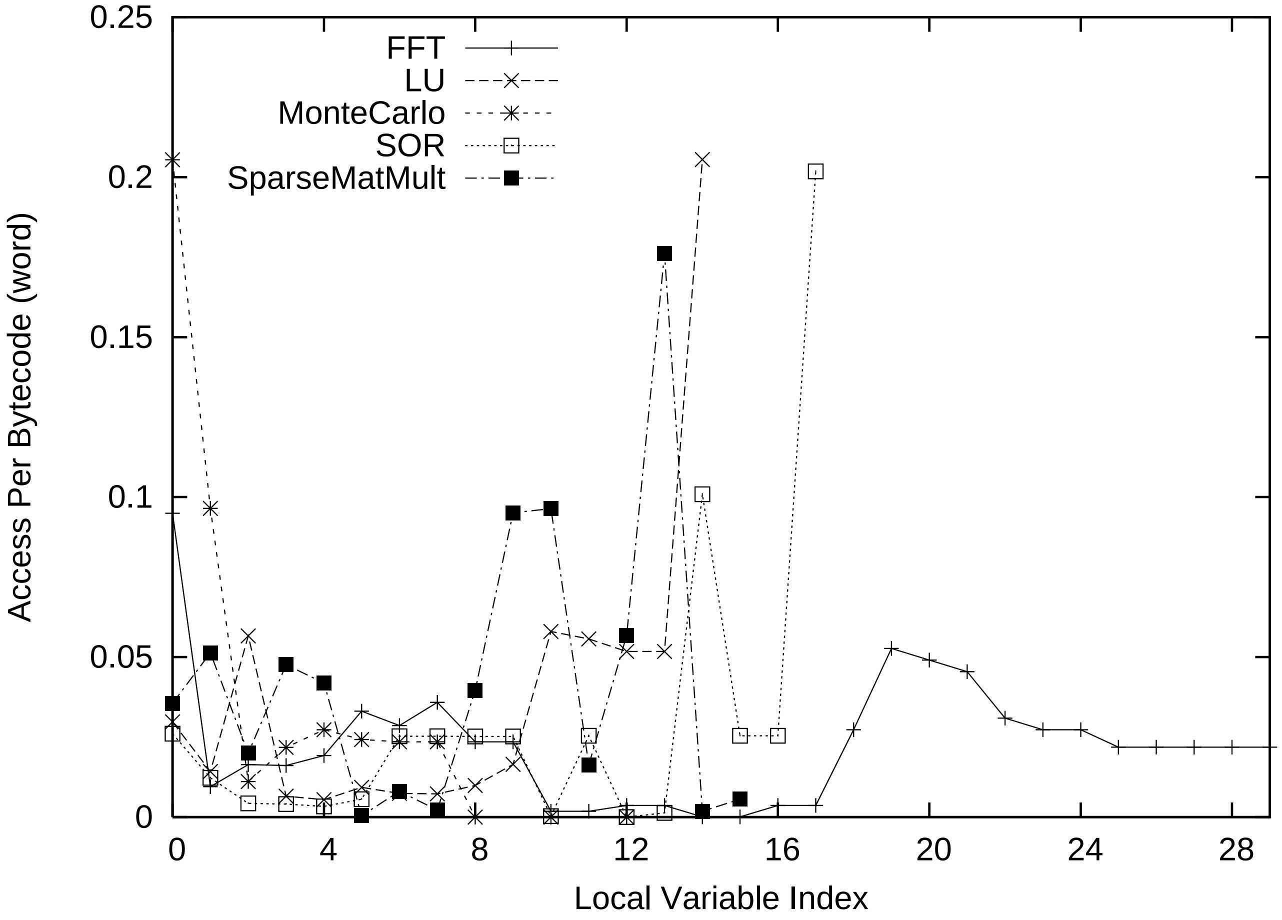
Benchmark Programs (1)

- SciMark2.0 (FFT, LU, MonteCarlo SOR, SparseMatMult)
  - Computation intensive kernels
  - Long loop iterations per invocation
  - Many local variables
- JOrbis
  - Ogg Vorbis Audio Decoder in Java
  - Similar characteristics to SciMark2.0

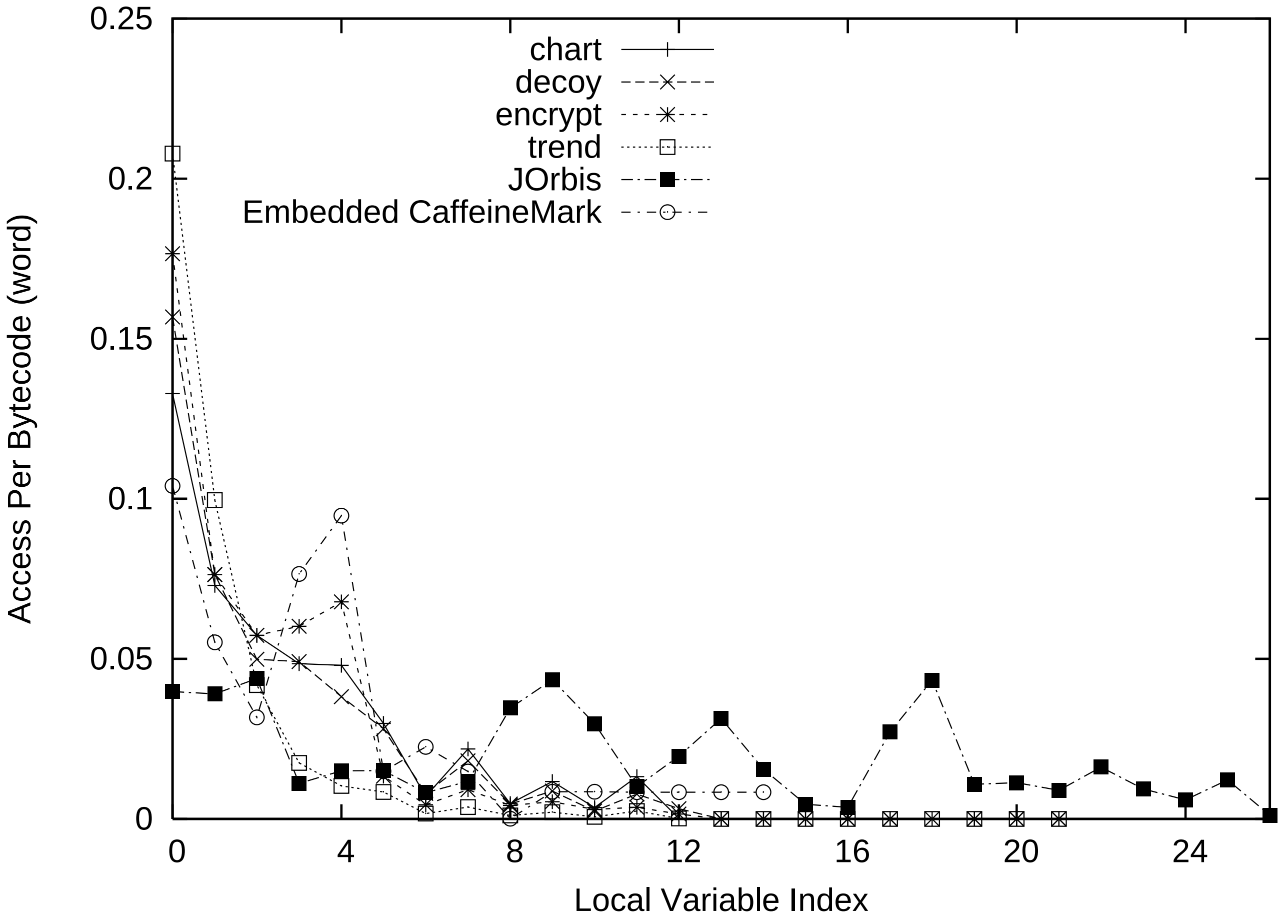
Benchmark Programs (2)

- SAXON XSLT processor
  - XML parser with four test documents from XSLTMark
  - Most accesses are to lower index local variables
  - Short execution per invocation
- Embedded CaffeineMark
  - Composite results of Sieve, Loop, Logic, Method and Float
  - Long execution per invocation
  - Most accesses are to lower index local variables

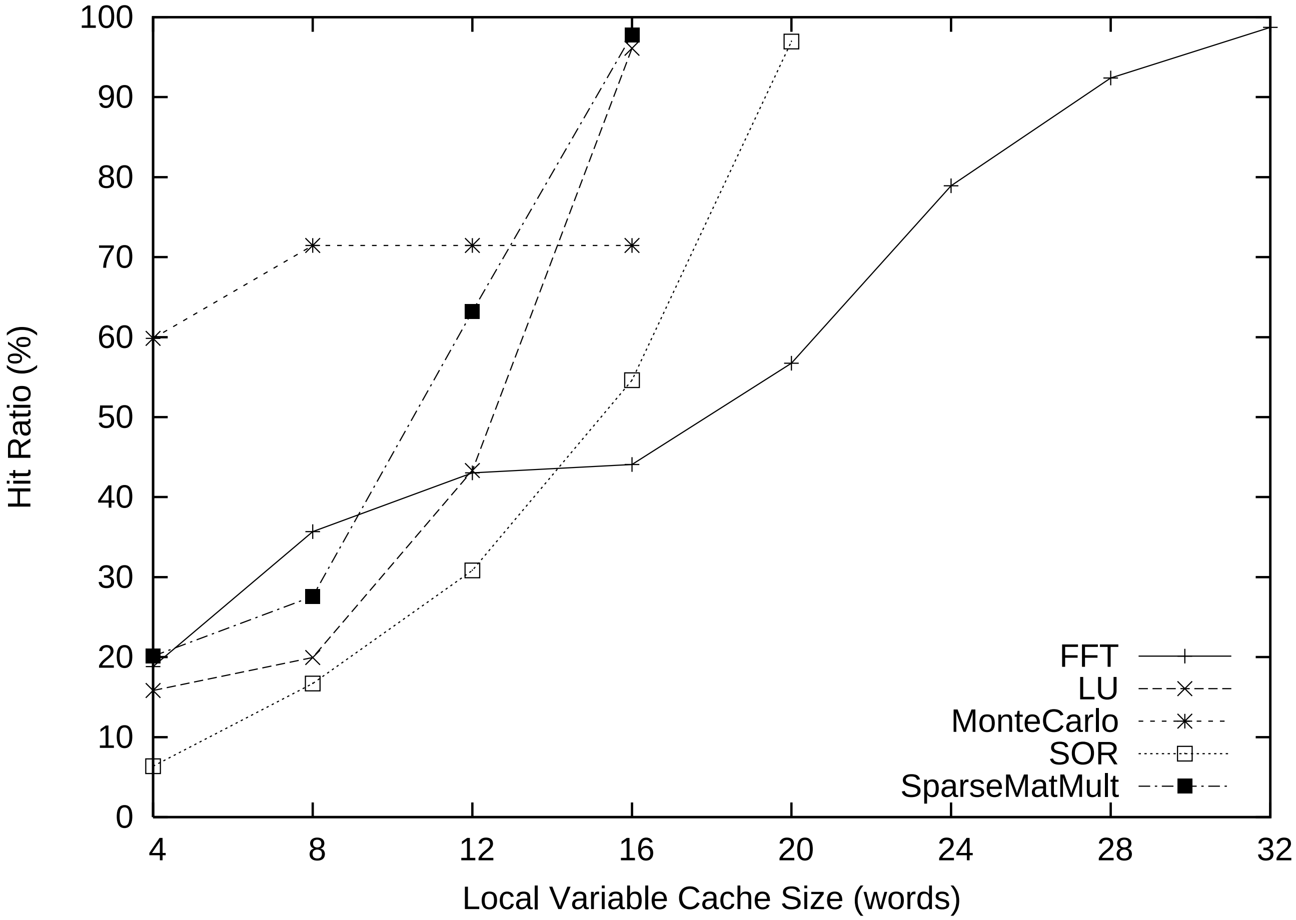
SciMark 2.0 Local Variable Access Distribution



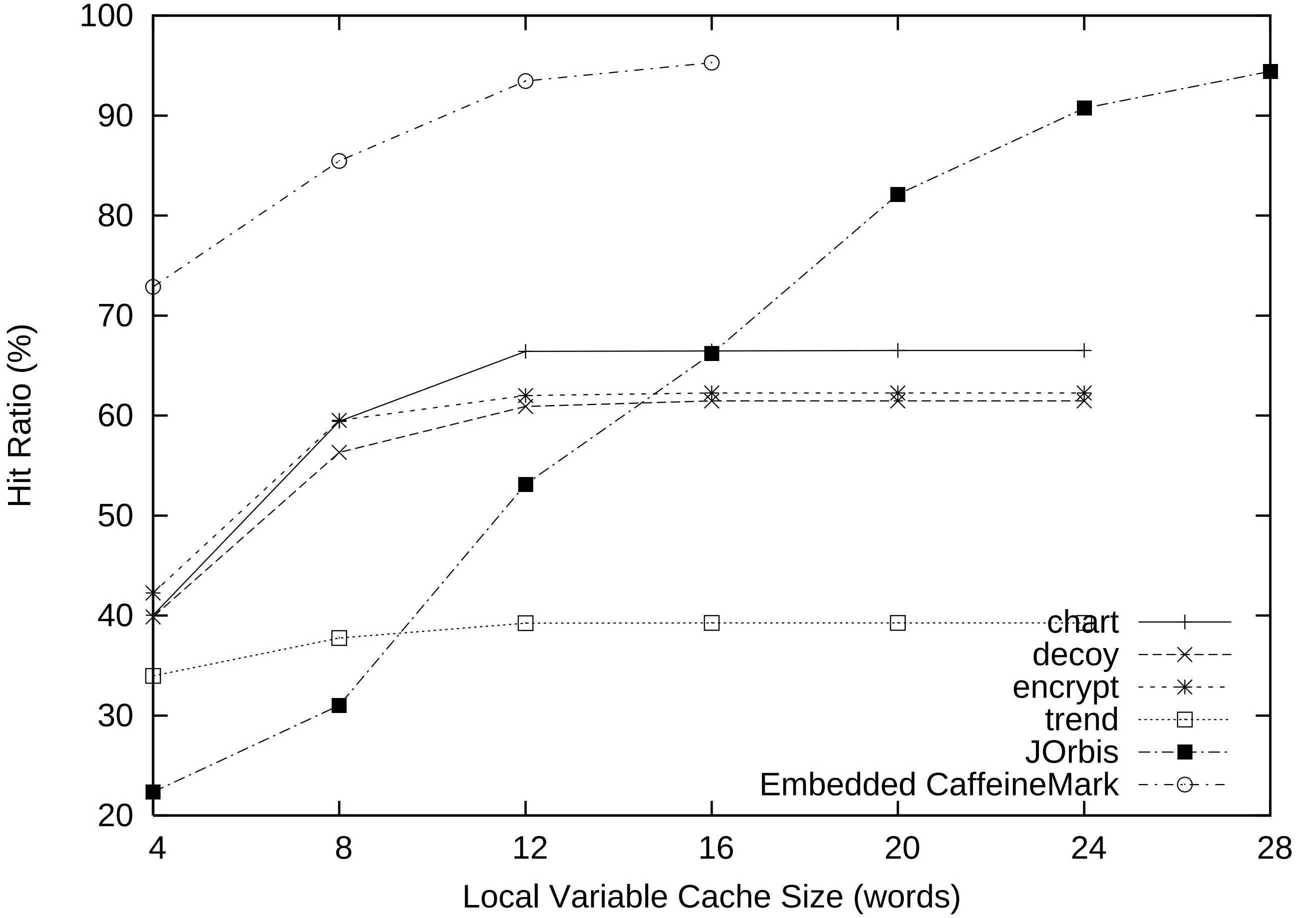
SAXON, JOrbis and Embedded CaffeineMark



SciMark 2.0 Hit Ratio



SAXON, JOrbis and Embedded CaffeineMark

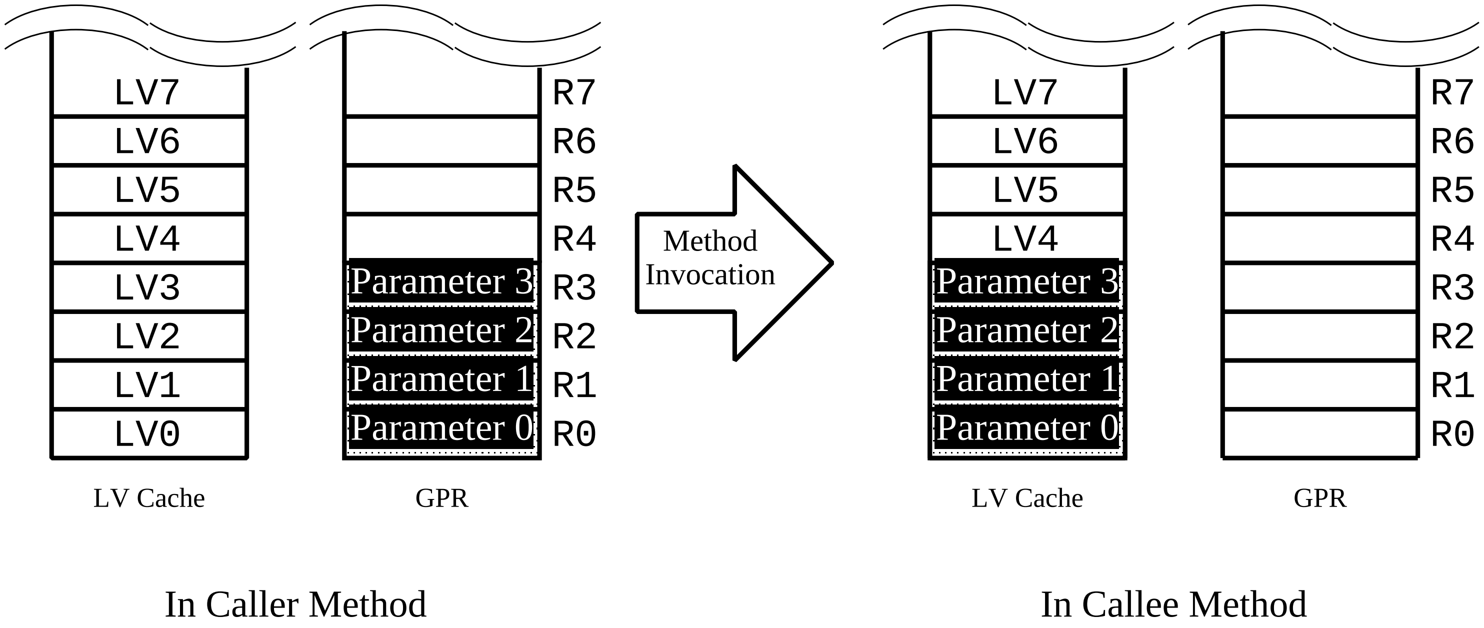


Parameter Passing in JVM

1. The caller method pushes parameters
2. The caller method invokes another method
3. These parameters appear as local variables for the callee method

- In a hardware translation-based JVM, Step 1 is equal to loading (up to four) t-o-s GPRs with parameters.
- By "swapping" t-o-s GPRs and local variables 0 to 3, the callee method can directly access local variables from the local variable cache.

Parameter Passing in JVM (cont.)



Parameter Passing in JVM (cont.)

- This parameter passing method eliminates initial loading of the local variable cache corresponding to the parameters.
- Effective for SAXON XSLT, but not for loop-oriented benchmarks (SciMark 2.0, JOrbis, ECM).

	Hit Ratio (%)		
Test Case	w/o	w/	Δ
chart	66.5	74.7	8.2
decoy	61.5	70.1	8.6
encrypt	62.3	68.5	6.3
trend	39.3	52.6	13.3

Changes in SAXON XSLT Hit Ratio

Optimizing Local Variable Mapping

- So far identity mapping from LV to LVC has been assumed (i. e. local variables 0 to $n - 1$ are mapped to an n-entry LVC)
- Local variable access distributions draw very complex curves (e. g. in FFT, accesses to local variables 10 to 16 are only 2% of total).
- By storing most frequently accessed local variables, a small local variable cache can perform close to a larger one.

On-the-fly Access Profiling

Assumptions:

- Do not analyze the entire method (or even classfile)
- Just count the local variable access for each bytecode as it is processed by the JVM
- Create a mapping from local variables to cache entries based on the access statistics.

Effectiveness of profiling scheme is evaluated by the benchmarks that access many local variables (FFT, SOR and JOrbis).

Profiling Options

Opt1: • Count accesses for each method separately.

- LV mapping is updated on invocation/return and next invocation is executed with the updated LV map.
- Infinite length of counters (no overflow)
- **advantage:** Captures local variable access behavior which differs from method to method.
- **drawback:** Statistics for each method must be saved and restored on each invocation and return.

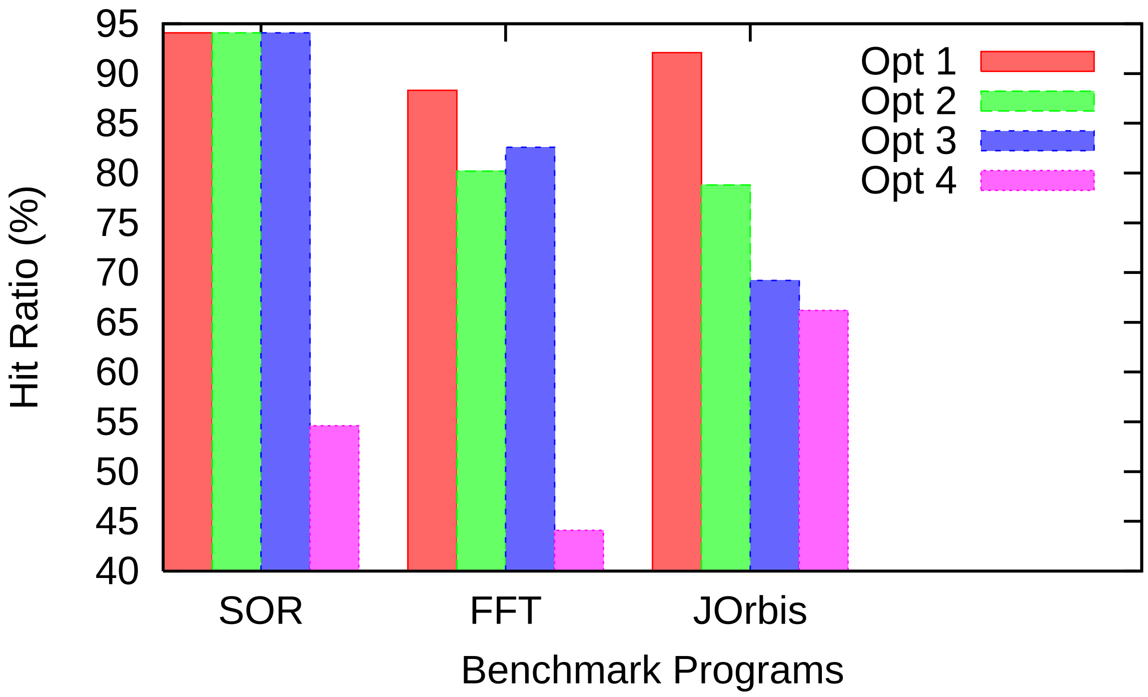
Opt2: A single set of counters for all method.

Opt3: Finite length counter (default 8-bit), shift right all counters by 1-bit on overflow.

Opt4: Identity mapping

Hit Ratios and Profiling Options

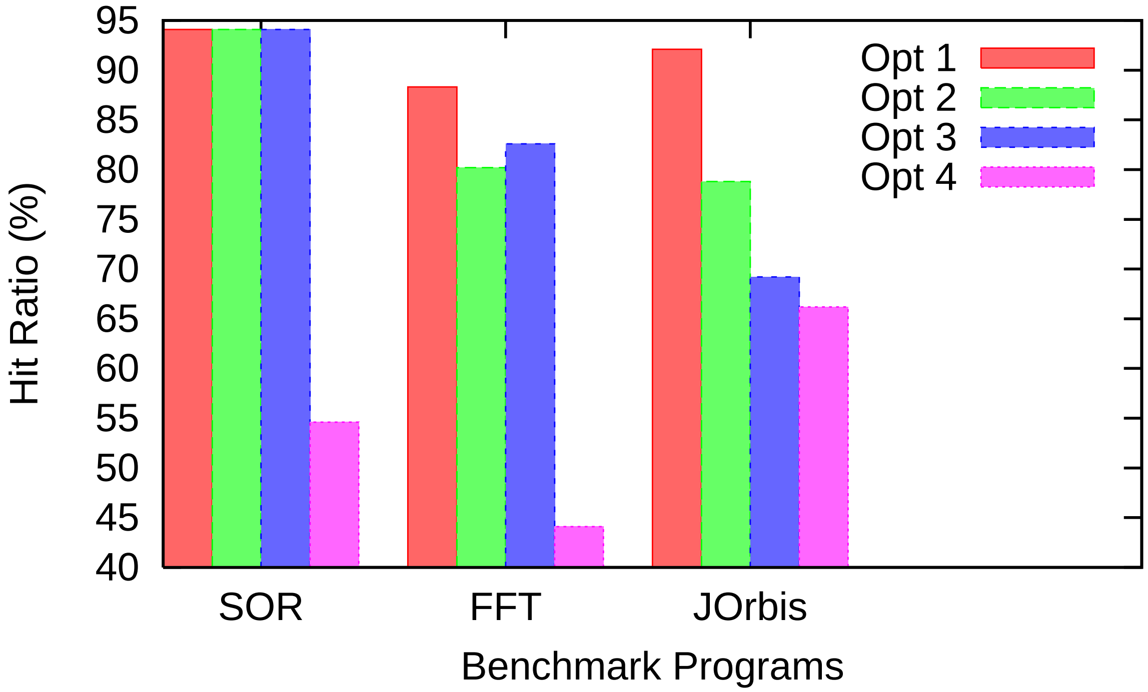
Hit Ratios for a 16-entry Local Variable Cache



For SOR, a single set of 8-bit counters sufficiently improve the hit ratio.

Hit Ratios and Profiling Options

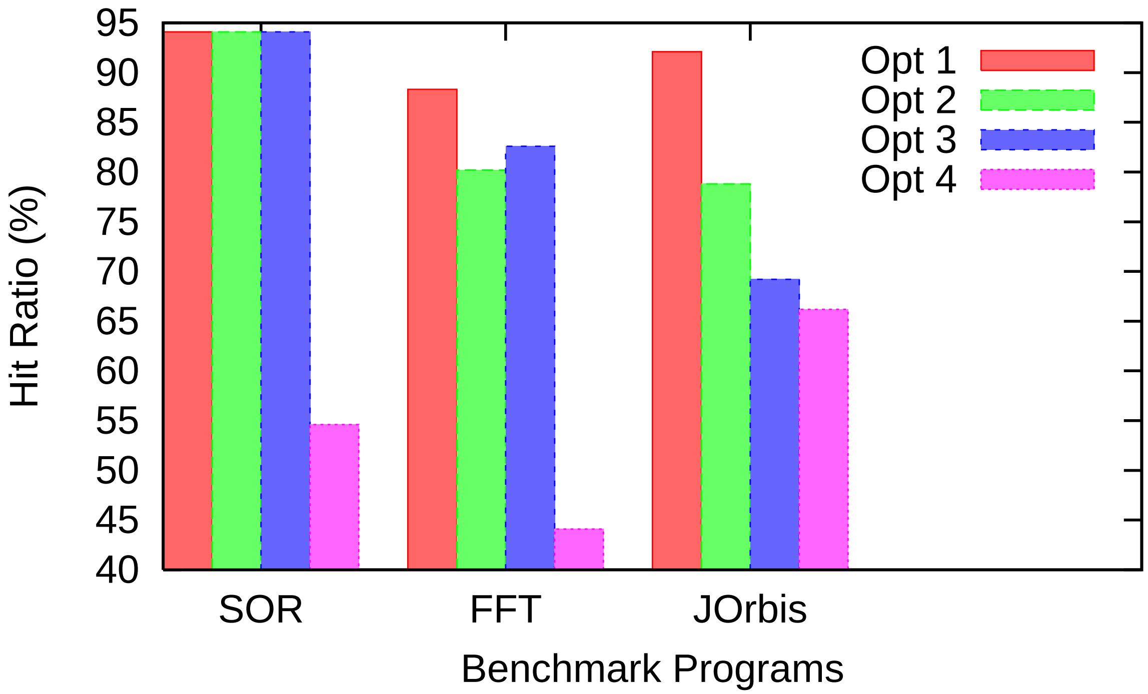
Hit Ratios for a 16-entry Local Variable Cache



FFT's hit ratio is improved by 1.87 times, but Opt 2 is worse than Opt 3 (need more investigation).

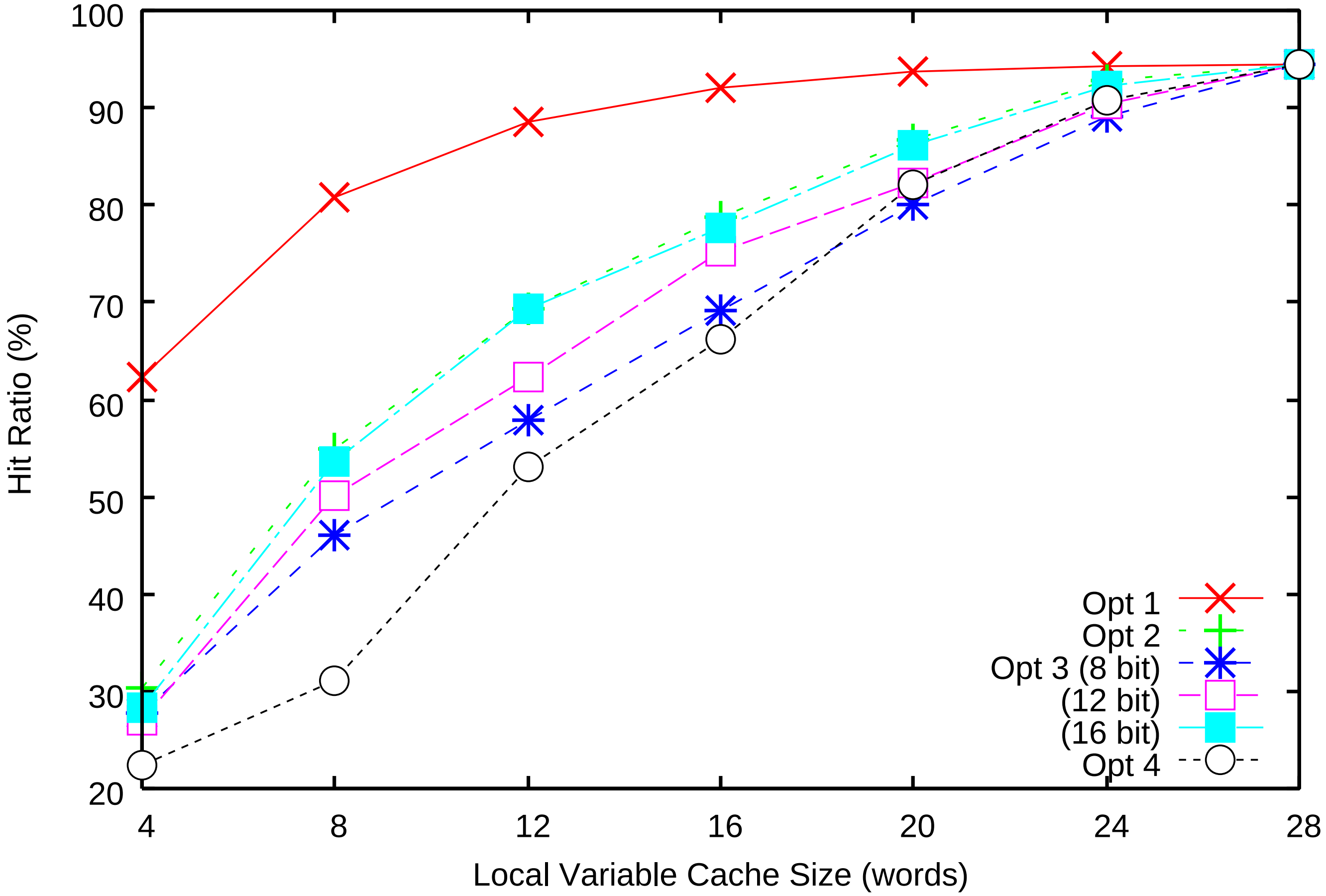
Hit Ratios and Profiling Options

Hit Ratios for a 16-entry Local Variable Cache



JOrbis suffers from both of finite length and single set of counters.

JOrbis Hit Ratio with On-the-fly Profiling



Summary

- Proposed to add a small register file to the datapath of a hardware translation-based JVM to be used as the local variable cache.
- With two exceptions, 60% to 98% of memory accesses for local variables can be eliminated by a 16-entry local variable cache for the various benchmark programs tested.
- For SAXON XSLT processor, parameter passing using LVC was effective: 6.3% to 13.3% of accesses can be turned from miss to hit.
- For applications with more than 16 local variables, on-the-fly profiling by a set of counters was effective: 8-bit counters improved the hit ratios by 4 to 87%.

Future Work (or limitations of this work)

- Instruction folding using local variable cache (not possible for most RISC processors if local variables are in memory).
- Design a prototype JVM with the local variable cache to estimate the hardware resource overhead and operation speed.
- Performance evaluation including more complex bytecode (software emulated code) with more various benchmark programs.