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

Hitoshi Oi

Department of Computer Science
The University of Aizu
Aizu-Wakamatsu, JAPAN
hitoshi@u-aizu.ac.jp

Abstract

Hardware bytecode translation is a technique to improve the performance of the Java Virtual Machine (JVM), especially on the portable devices for which dynamic compilation is infeasible. However, since the translation is done on a single bytecode basis, it is likely to generate frequent memory accesses for local variables which can be a performance bottleneck.

In this paper, we propose to add a small register file to the datapath of the hardware-translation based JVM and use it as a local variable cache. We evaluate the effectiveness of the local variable cache against the size of the local variable cache which determines the chip area overhead and the operating speed. We also discuss the mechanisms for the efficient parameter passing and the on-the-fly profiling.

With two types of exceptions, a 16-entry local variable cache achieved hit ratios of 60 to 98%. The first type of exceptions is represented by the FFT, which accesses more than 16 local variables. In this case, on-the-fly profiling was effective. The hit ratio of 16-entry cache for the FFT was increased from 44 to 83%. The second type of exception is represented by the SAXON XSLT processor for which cold misses were significant. The proposed parameter passing mechanism turned 6.4 to 13.3% of total accesses from miss to hit to the local variable cache.

Categories and Subject Descriptors C.1 [*Computer Systems Organization*]: PROCESSOR ARCHITECTURES

General Terms Performance, Design

Keywords Java Virtual Machine, Hardware-Translation, Memory Hierarchy

1. Introduction

Java Virtual Machine (JVM) is an abstract instruction set architecture that realizes important features of the Java language including platform independence and security model [1]. These features make Java and JVM especially suitable for portable devices for which various platforms are available from the manufacturers who are competing the market shares. The most flexible and hence popular implementation method for the JVM is a software interpretation [2]. An interpreter program written in the native instructions for the platform reads, analyzes and executes Java bytecode (machine instructions for JVM). Since interpreters are merely an application written in the native language of the platform and the specification of the JVM leaves flexibility in the implementation, it has been widely adapted and has become the most standard form of the JVM implementation.

Clearly, the drawback of the interpretation is its performance. For example, to check a single bit in a bytecode may take several native instructions in the software while it takes much shorter than a single clock cycle in the hardware. Various dynamic compilation techniques are proposed and are adapted in mostly for the JVMs on the desktop and server platforms [3]. Its key idea is to detect frequently invoked methods (procedures in the Java language) during the execution and compile them into native instructions of the platform. However, it is not a suitable solution for the JVM on the embedded devices for the following reasons. First, the compilation itself takes execution time and consumes battery power. Second, the code size of the compiled code is increased which is not preferable for the limited memory space of the embedded devices. For a server platform, these compilation overheads are negligible since the compiled code are repeatedly executed many times. On the contrary, it is likely that the application downloaded to a portable device is executed for a few times and it will be replaced by another application quickly. Therefore, due to the compilation overheads mentioned above, some other solution to improve the performance of the JVM on the portable devices is required.

Hardware-translation is a technique to accelerate the execution of Java application by dynamically converting the bytecode into the native instructions. With the low hardware overhead and simple mechanism, it is considered to be suitable for the JVM on embedded processors. In this paper, we investigate the design and performance of the local variable cache which further accelerates the execution of a hardware-translation based JVM by reducing the number of memory accesses.

This paper is organized as follows. In the next section, an overview of the hardware-translation based JVM and the local variable cache are presented. In Section 3, JVM and benchmark programs used for the experiments in this paper are described. Section 4 evaluates the performance of the base design of the local variable cache. Sections 5 and 6 propose and evaluate two ideas to further enhance the performance of the local variable cache. Related work and conclusions are presented in Sections 7 and 8, respectively.

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

2. Hardware-Translation of Java Bytecode

Hardware-translation is a technique to enhance the performance of the JVM by dynamically replacing the bytecodes to native machine instructions [1]. A small translation logic is inserted between the fetch and decode stages of the processor pipeline. When a flag in the processor's status register indicates that the fetched instruction is a Java bytecode, it is converted into native instructions by the translation unit. If the native instruction is fetched, it bypasses the translation logic. Therefore, it is a reconfigurable processor in the instruction set level.

Table 1 shows an example of the bytecode translation. In this example, two local variables which are assigned local variable indicies 3 and 4, are added and the result is written to the local variable 3. First two bytecodes, ILOAD_3 and ILOAD_4 push the values of two local variables onto the stack. Following the ARM Jazelle's specification, R0 to R3 are used to hold the top four words of the operand stack in this example. Therefore, the first two bytecodes are translated into two load word instructions (LDR) using R7 which holds the address of the local variable 0 and corresponding offsets. Next bytecode, IADD, pops and adds two top of stack words and pushes the results onto the stack. This bytecode is translated into a native instruction which adds two registers R0 and R1. The last bytecode, ISTORE_3 pops the top of stack word and writes it to the local variable 3. This bytecode is replaced with a store word instruction (STR).

Bytecode	ARM Inst.
ILOAD_3	LDR R0, [R7, #12]
ILOAD_4	LDR R1 [R7, #16]
IADD	ADD R0, R1
ISTORE_3	STR R0, [R7, #12]

Table 1. Bytecode Hardware-Translation Example

As shown in the above example, the translation unit reads a single bytecode at a time and generates a short sequence of native machine instruction(s). The hardware-translation is limited to the simple bytecodes such as load, store, and arithmetic/logical operations on the stack. Complex bytecodes, such as **new** (create a new object), are emulated by the software. By limiting the complexity of the translation mechanism, the hardware resource overhead and the performance gain are balanced: in the case of Jazelle, it is reported that 8x performance gain was achieved by 12K gates [4].

In a JVM, local variables are allocated on the operand stack. In the literature [9], as well as in our analysis, a significant fraction (35% to more than 80%) of bytecodes are accesses to local variables. Since the translation is done on a single bytecode basis, the same local variables may be repeatedly loaded from and stored to the main memory. These memory accesses for local variables not only degrade the performance by the access latency but also result in a shorter battery life. Moreover, having local variables in the memory makes the instruction folding [18] difficult to implement (this topic will be discussed in Section 8).

Figure 1 shows a sample datapath with the local variable cache. The local variable cache should have a similar structure as the register file used for the general purpose registers (GPR). It has two read ports and one write port. The difference is that each entry is associated with *v* and *m* bits indicating that the entry is valid and modified, respectively.

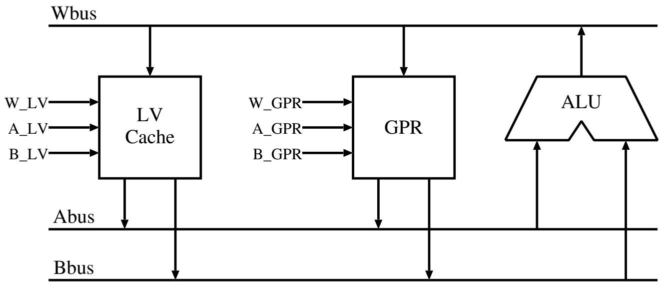

Figure 1. A Datapath with Local Variable Cache

In Figure 2, the operation of the local variable cache is described for representative bytecodes, ILOAD, ISTORE, INVOKEVIRTUAL and IRETURN as example. Note that operations not directly related to the local variable cache, such as handling overflow of top of stack registers, are not included.

On the ILOAD, the local variable index (idx) is checked if it is within the range of cache size (LVC_SIZE). If true, the valid bit of the corresponding entry (v[idx]) is checked. If the entry is valid, the value of the local variable entry is copied to one of top-of-stack word registers which is one of R0 to R3 pointed to by the srp counter. The control sequence for this copy operation should be similar to the MOV instruction with an exception that W_LV (instead of W_GPR) is enabled so that the value will be written to the local variable cache (not GPR). If v[idx] is not set, the local variable is read from the memory by issuing LDR instruction using the local variable pointer (R7) and idx as offset. When the data is returned from the memory, it is written to R[srp] and LV[idx] simultaneously by enabling both W_GPR and W_LV. The valid bit of the corresponding entry is also set. If the index is out of range, ILOAD is replaced with a memory read instruction.

If the index of the ISTORE is within the LVC_SIZE, it is replaced with a copy (MOV) from R[srp] to LV[idx] and the modified bit of the corresponding entry is also set. Otherwise, a memory write instruction (STR) is issued.

On the INVOKEVIRTUAL and IRETURN bytecodes, all valid and modified bits are set to 0. In addition, the modified entries in the local variable cache are written back to the memory for the INVOKEVIRTUAL. Therefore, the local variable cache is caller-saved.

3. Experimental Environment

In this section, the experimental environment for the performance evaluation in later sections is described. For the Java Virtual Machine and runtime environment, we used Kaffe version 1.0.7 [10]. Kaffe is an open-source implementation of the JVM and we compiled it with "–with-engine=intrp" option so that all bytecodes are interpreted.

We use several different types of Java benchmark programs listed in Table 2. The first set of benchmarks are from SciMark 2.0 Java Numerical Benchmark [11], which includes FFT, LU, Monte Carlo, SOR and Sparse Matrix Multiplication. These are computational kernels and chosen to represent the usage of local variables in loop-intensive applications.

The next benchmark is JOrbis, which is a pure Java implementation of Ogg Vorbis audio decoder [14]. For the input to the decoder, startup2.ogg which is a 2-channel bitstream sampled at 44.1KHz included in the JOrbis installation kit was used.

[1] In the court order [8] dated September 30, 2003, ARM's Jazelle and Nazomi's U.S. Patent No. 6,332,215 are distinguished as follows. While Nazomi's patent translates Java bytecodes into native instructions before reaching the decode stage of the CPU, ARM's Jazelle translates bytecodes into controls signals. While this difference may be important for the patent issues, it is not essential for the ideas discussed in this paper. Therefore, the readers of this paper can interchangeably read "native instructions" as "the sequence of control signals corresponding to the native instructions", vice versa.

```
switch(bc){
 case ILOAD:
  if(idx < LVC_SIZE){
   if(v[idx]){
    MOV R[srp], LV[idx];
   }
   else{
    LDR R[srp], [R7, 4*idx];
    MOV LV[idx], R[srp];
    v[idx] = 1;
   }
  }
  else{
   LDR R[srp], [R7, 4*idx];
  }
  break;
 case ISTORE:
  if(idx < LVC_SIZE){
   MOV LV[idx], R[srp];
   v[idx] = 1;
   m[idx] = 1;
  }
  else{
   STR R[srp], [R7, 4*idx];
  }
  break;
 case INVOKEVIRTUAL:
  for(i = 0; i < LVC_SIZE; i++){
   if(m[i]){
    STR LV[i], [R7, 4*i];
    m[i] = 0;
   }
   v[i] = 0;
  }
  break;
 case IRETURN:
  for(i = 0; i < LVC_SIZE; i++){
   v[i] = 0;
   m[i] = 0;
  }
  break;
}
```

Figure 2. Bytecode Translation and Local Variable Cache Operation. bc: bytecode, idx: local variable index, srp: pointer to one of top-of-stack registers (R0 to R3) LVC_SIZE: number of words in the Local Variable Cache, v[]: valid bit, m[]: modified bit.

The third benchmark is SAXON Version 6.3, an XSLT processor [12], driven by four test case XML documents from XSLTMark [13]. We chose four test case documents, chart, decoy, encrypt and trend, based on the average number of bytecode executed for a method invocation and the functional categories defined in the XSLTMark.

The Embedded CaffeineMark consists of five tests, Sieve, Loop, Logic, Method and Float. Each of these tests is basic and tries to measure various aspects of JVM. Due to the license restriction, we only report the composite results of all five tests.

Table 3 shows a brief summary of the benchmark program execution. The number of bytecode executed per method invocation (third column) indicates the length of execution path of a method. The longer the execution path, the more likely to access the same local variable repeatedly. Note that long and double data types are two-word long in Java. Therefore, an access to a local variable of these types is counted as two words in Table 3. All computation-intensive benchmark programs (SciMark 2.0, JOrbis and ECM) have loops which resulted in long average execution paths: except MonteCarlo, they executed more than 200 bytecodes per method invocation. On the other hand, SAXON processing XSLTMark test case documents has short execution paths, which range from 9.77 to 23.5 bytecodes per method invocation. Access frequency distribution over local variable index are shown in Figures 3 and 4.

Some of local variables are used for passing parameters (arguments) to a method, which is the topic of Section 5. The sixth and seventh columns summarize the accesses to arguments.

Benchmark	Bytecode		LV Access		Arg Access	
	Tot	pmi	pmi	pbc	pmi	pbc
FFT	4.50	519.4	455.3	.877	59.5	.115
LU	6.44	294.8	276.6	.938	27.5	.093
MonteCarlo	2.14	54.1	24.50	.453	12.3	.115
SOR	5.73	344.9	319.1	.925	11.9	.035
SparseMatMult	3.91	310.3	316.9	1.02	57.4	.185
chart	2.29	23.5	13.8	.584	5.57	.237
decoy	11.2	17.9	10.0	.562	4.44	.248
encrypt	24.6	21.6	13.2	.610	5.73	.265
trend	13.8	9.77	4.13	.423	2.92	.299
JOrbis	390	238.6	207.1	.868	31.5	.132
ECM	17.2	247.8	153.8	.621	33.1	.114

Table 3. Summary of Benchmark Execution. Total executed bytecodes are in millions (10^6). pbi: per method invocation. pbc: per bytecode.

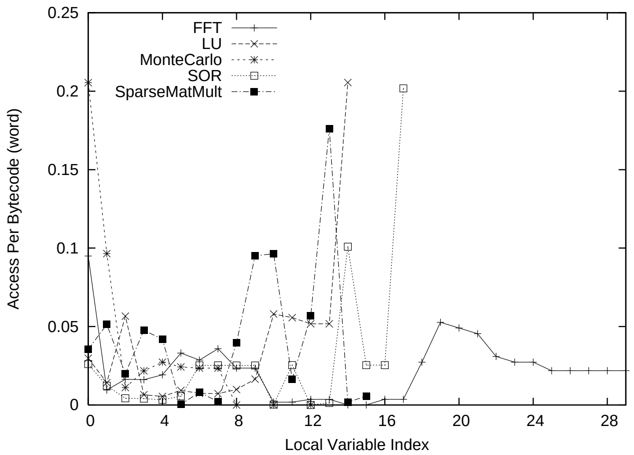

Figure 3. SciMark 2.0 Local Variable Access Distribution

Benchmark	Description
SciMark 2.0 Java Numerical Benchmark	
FFT	FFT on 1024 complex doubles
LU	LU factorization of a 100×100 double dense matrix
MonteCarlo	Monte Carlo integration to compute π
SOR	Successive Over-relaxation on a 100×100 grid.
SparseMatMult	Multiplication of 1000×1000 sparse matrices
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).
JOrbis	Ogg Vorbis audio decoder in Java
ECM	Embedded CaffeineMark (Sieve, Loop, Logic, Method and Float).

Table 2. Benchmark Program Description

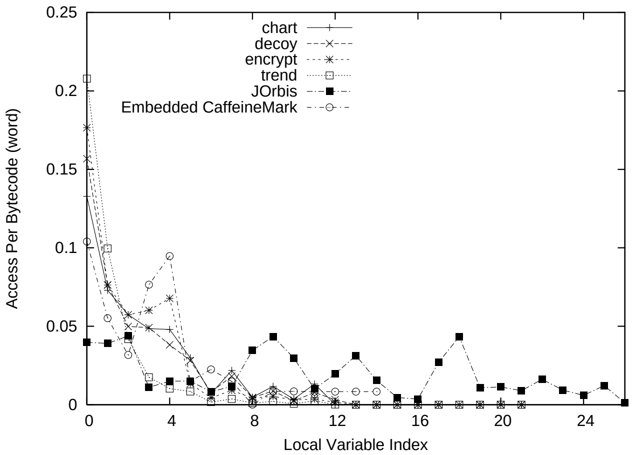

Figure 4. SAXON, JOrbis and Embedded CaffeineMark Local Variable Access Distribution

4. Evaluation of the Base Design

In this section, the performance of the local variable cache presented in Section 2 is evaluated using the benchmark described in the previous section. The size of the local variable cache is varied from 4 to 32 words (depending on the highest local variable index for each benchmark) and the hit ratio to the local variable cache is used as the performance metrics.

Figure 5 shows the hit ratios of the local variable cache for the SciMark 2.0 benchmark programs. These computation-intensive programs are likely to use a large number of local variables. Their access distributions are somewhat irregular (Figure 3). Consequently, to achieve a high hit ratio, the local variable cache needs to be large enough to accommodate all the local variables. However, once all local variables are mapped to the cache, they are repeatedly accessed in the loop during a single invocation of a method. FFT is the most typical case for these trends. While it accesses 30 local variables, local variables 10 to 17 are much less frequently accessed than others. Therefore, the improvement in the hit ratio for the cache size increase that corresponds to these local variables is gradual. With 32 entries, all the local variable are cached and 455 accesses on the average for each method invocation makes the hit ratio as high as 99%.

The counterexample is MonteCarlo, which uses a small number of local variables and average execution path per method invocation is shorter than other benchmarks in SciMark2.0. 8 entries are sufficient to cache most of local variables, but they are not reused many times due to the short average execution path. Consequently, the best hit ratio is only 71%. LU, SOR and SparseMatMult show similar characteristics as FFT with fewer local variables.

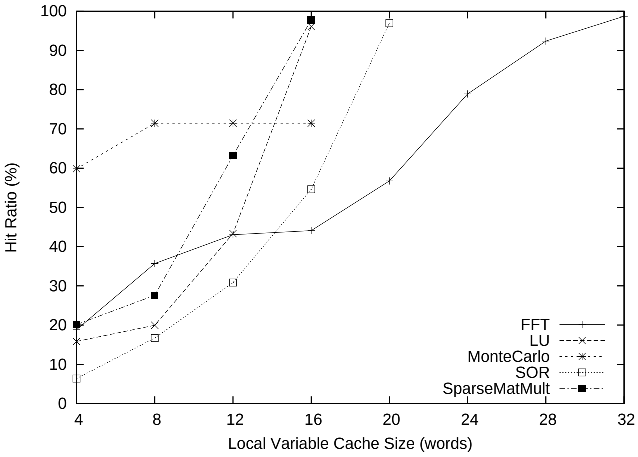

Figure 5. SciMark 2.0 Hit Ratio

Figure 6 shows the hit ratios for JOrbis, Embedded CaffeineMark and SAXON XSLT processor with four XML test documents.

JOrbis is also a computation intensive workload: it uses a large number (27) of local variables and has a long average execution length. Therefore, its performance exhibits a similar trend as the FFT. With a 4-entry local variable cache, the hit ratio is only 22%, but it rapidly goes up to 94% with 28 entries.

The behavior of SAXON is quite different from SciMark 2.0 and JOrbis. In fact, it is a typical behavior of programs written in Java. The average execution length is an order of magnitude shorter than SciMark 2.0 and JOrbis. While it uses up to 22 local variables, 90% of accesses are for variables 0 to 7 and local variables 12 and higher only count 2% of total accesses. Among four test cases, chart, decoy and encrypt show similar curves on their hit ratios: they start at around 40% with 4 entries and increase up to 61 to 66% with 12 entries. The hit ratio of trend is 34% with 4 entries and only increases to 39% with 12 entries. The average execution lengths of trend is only 9.8 bytecode which is less than half of other three test cases. This short execution length emphasizes the effect of cold misses which cannot be reduced by increasing the size of the local variable cache. While the Embedded CaffeineMark does not provide source files, it is considered to have loop structures, which increase chances of repeated accesses to the same local variables. Also, the number of local variables is small (16) and lower eight variables count 86% of the total accesses. Hence, its hit ratio is quite high even with 4 entries (73%) and it increases up to 95% with 16 entries.

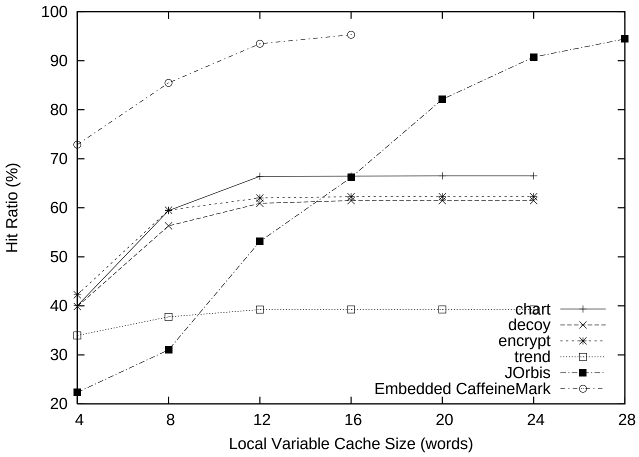

Figure 6. SAXON, JOrbis and Embedded CaffeineMark Hit Ratio

5. Parameter Passing with the Local Variable Cache

In the SPARC architecture, 32 registers are divided into four groups: *Globals* (R0 to R7), *Outs* (R8 to R15), *Locals* (R16 to R23) and *Ins* (R24 to R31) [16]. When a procedure is called, physical registers mapped in Outs in the caller procedure are moved to Ins in the callee procedure. Therefore, the caller procedure write parameters to the callee procedure in the Outs registers and the callee procedure receives them from the Ins registers.

While this sliding register window provides a smooth parameter passing mechanism, it may not be feasible for embedded processors due to the hardware and software overhead [17]. In this section, we propose and evaluate a parameter passing mechanism incorporated into the local variable cache in the hardware-translation based JVM.

In a JVM, parameters to a callee method are pushed onto the operand stack before invocation. After the invocation of the method, these parameters appear as local variables for the callee method. For example, if three words were pushed as input parameters, they appear as local variables 0 to 2 in the callee method. Figure 7 shows the proposed parameter passing mechanism in the hardware-translation based JVM. As explained in Section 2, registers R0 to R3 hold copies of top four words of the operand stack in the hardware translation based JVM. When a method invocation bytecode (such as INVOKEVIRTUAL) is executed, physical registers for R0 to R3 and LV0 to LV3 are swapped. This swapping can be implemented by 2-to-1 multiplexers and a single bit FF which is set and reset on the method invocation bytecodes. Based on the number of parameters pushed on to the stack, the valid bits of the corresponding entries are set. Since parameters can be directly accessed in the callee method, storing the top of stack registers to the memory and loading local variables from the memory are eliminated.

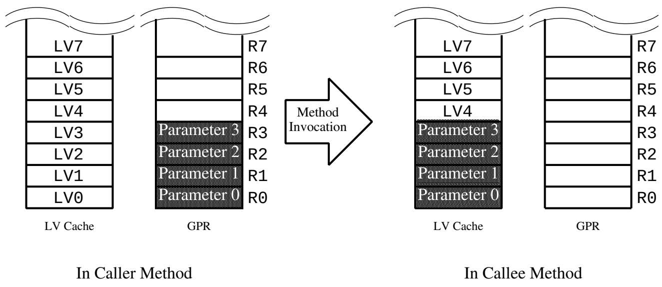

Figure 7. Parameter Passing with Local Variable Cache

As we saw in the previous section, the effect of cold misses is significant in the SAXON XSLT processor. From Table 3, it is shown that the large fractions (40 to 70%) of local variables accesses are for the method parameters in SAXON. The proposed parameter passing mechanism eliminates the cold misses to method parameters. Table 4 shows the increase in the hit ratios by the parameter passing mechanism for the SAXON processing four test cases. Note that, the number of cold misses does not change when the size of local variable cache varies. Therefore, the increase of the cache hit ratio is also a constant. With the proposed mechanism, the hit ratio for a 16-entry local variable cache is increased to 53% (trend) to 75% (chart).

Benchmark programs other than SAXON are loop-oriented for which the effect of cold misses is small. Hence, the improvement on the hit ratios on these benchmarks are negligible (around or much smaller than 1%).

Test Case	Improvement
chart	8.2%
decoy	8.6%
encrypt	6.3%
trend	13.3%

Table 4. Improvement of the SAXON Hit Ratio with Parameter Passing by Local Variable Cache

6. Register Mapping Problem

So far, we have assumed the identity mapping from the local variable index into the cache entry. Therefore, for an n-entry local variable cache, it is assumed that local variables 0 to $n-1$ are cached. However, as shown in Figures 3 and 4, the access frequency of local variable is not a monotonically decreasing function over the index. In the ARM Jazelle, R4 is permanently is assigned to the local variable 0 ('this' pointer) [4]. However, in some benchmarks, other local variables are more frequently accessed than the local variable 0 (e.g. SparseMatMult). For the benchmark programs used in this paper, the largest local variable index was 29 in FFT. Thus, a local variable cache with 30 or more entries can store all the local variables.

On the other hand, a large register file not only occupies a large chip area, but it also slows down the access speed. Considering the fact that the size of register file in most embedded processors is 16, there may be a situation where, rather than a large local variable cache, a smaller cache with optimized mapping scheme is preferable. In this section, we consider the optimization of mapping local variables to cache entries.

The first possibility is to let the programmer, compiler, or off-line profiling utility provide an optimized mapping for each application. However, such information may not be available until the JVM architecture proposed in this paper becomes (nearly) standard. In this section, we consider on-the-fly profiling methods to optimize the register mapping. The assumption is not to analyze the whole structures of the class files before execution, like compilers. Rather, we simply count the number of accesses to each local variable in the course of execution.

The first option (Opt 1 in Table 5) is to count the local variable access separately for each method. When a return bytecode is executed or another method is invoked, the access statistics for that method is stored somewhere (most likely to the memory). The access count for each local variables is averaged over the invocations of the method. Register map is created for each method and is updated with the information up to the previous invocation of the method.

The need of saving and restoring local variable access statistics makes this option infeasible. However, this option captures the local variable access behavior of each method more accurately than other options explained below. Therefore, we use this option as a reference.

The second option (Opt 2) is to use a single counter for each local variable's access, regardless of in which method it is accessed. Since this options does not require saving and restoring access histogram on method invocation and return, it is considered more practical.

In above two options, it is assumed that the number of bits for each counter is long enough to prevent counter overflow. As the third option (Opt 3), we use a fewer number of bits (default 8 bits) for each counter. When a counter is overflown, all the counters are right shifted by one bit. This option is chosen to balance the accuracy of profiling and the hardware overhead. For the options two and three, the register map is updated on the invocation and return bytecode.

The fourth option (Opt 4) is the base design which we have assumed by the previous section, that is, the identical mapping from the local variable index to the cache entry.

Benchmark	Opt 1	Opt 2	Opt 3	Opt 4
FFT	88.3	80.2	82.6	44.1
JOrbis	92.1	78.8	69.2	66.2
SOR	94.1	94.1	94.1	54.6

Table 5. Hit Ratios for a 16-entry Local Variable Cache with Mapping Optimizations

We chose benchmarks that access more than 16 local variables from Table 2 except SAXON for which the fraction of accesses to local variables 16 and higher was quite small. Table 5 shows the hit ratios for a 16-entry local variable cache with the four options mentioned above. Mapping optimization was very effective in SOR: the hit ratio was increased from 54.6% to 94.1%. Also, the difference among options 1 to 3 was negligible.

The effectiveness of the on-the-fly profiling was relatively small and varied among options 1 to 3 in JOrbis. Option 3 only improved the hit ratio by 3% of the total access. Moreover, there were significant differences between options 2 and 3. This indicates the default length of counters (8-bit) is not sufficient for JOrbis.

To further investigate the behavior of JOrbis under these profiling options, we varied the length of the counters in option 3. From the hit ratios shown in Figure 8, it is known that an array of 16-bit counters works similarly to counters with (effectively) unlimited length. It should also be noted that the hit ratio for option 3 with 8-bit counters (80.0%) is worse than that of option 4 (82.1%) for the 20-entry cache size. The difference between these two options are small and hit rations for both cases are high. Moreover, JOrbis was the only case for this phenomena for the benchmarks used in this paper. However, if for any other workload the hit ratio of the proposed on-the-fly is significantly lower than option 4 (i.e. identical mapping), it may be necessary to switch between the identical mapping and the mapping based on the on-the-fly profiling automatically.

The most practical option (option 3) improved the hit ratio of FFT by 1.87 times. Note that, the hit ratio for option 2 is lower than that of option 3, which is an opposite phenomena to JOrbis. It is considered that too long counters prevent the register mapping from adapting to the change of access patterns between methods. The difference on the optimum length of counters between FFT and JOrbis also suggests the necessity of further study in determining the number of bits in the counters.

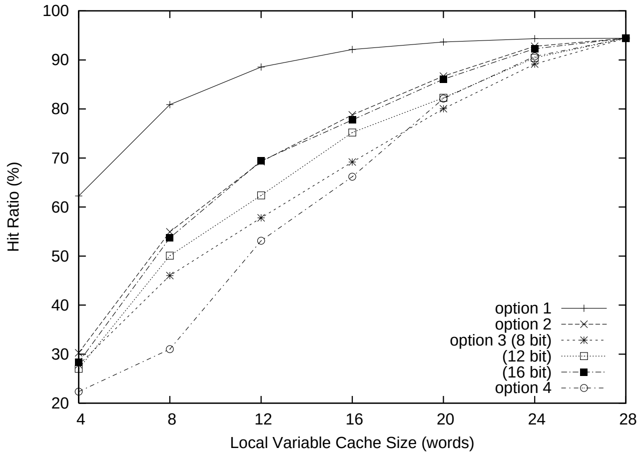

Figure 8. JOrbis Hit Ratio with On-the-fly Profiling Options

7. Related Work

ARM's Jazelle and Nazomi's Jstar are commercial products that incorporate hardware-translation based JVM. In this paper, the based design of the hardware-translation based JVM assumed the information published in ARM's white paper [4]. However, the ideas presented in this paper do not depend on the features specific to ARM or Nazomi's architectures and should be applicable to most embedded RISC microprocessors.

picoJava [18] directly executes Java bytecodes. It has a stack cache of 64 entries (256 bytes) with automatic fill and spill functionalities. Since a pure JVM is not sufficient to build a real system, picoJava's instruction set is extended for running applications written in "legacy" programming languages such as C/C++. Therefore, its design approach takes an opposite direction from the hardware-translation which tries to utilize the existing hardware resources of the standard RISC type microprocessors. When the stack cache of picoJava-II is incorporated into an embedded processor, its control signals and datapath will be an extra overhead as it will not be used for the native instructions.

Delft-Java is another hardware-translation based JVM [19]. One of its design goal is to execute multiple Java threads simultaneously by exploiting the superscalar architecture of the base microprocessor. It uses a set of counters to keep track of the register assignments to the stack operands. However, it does not have a local variable cache and does not reuse the data loaded into registers. In

Delft-Java, parameter passing is performed in a similar way to the mechanism proposed in this paper: the mapping information from registers to the operand stack values are passed to the callee method so that the callee method directly accesses the parameters from the registers. This mapping from the registers to the parameters is part of the run-time register assignment mentioned above. Our solution is simpler than theirs: the registers used for operand stack are fixed and passing parameters is performed by flipping a single bit by the invocation bytecode which swaps the local variable cache entries and the registers assigned to the operand stack.

Radhakrishnan et. al also studied the performance of a hardware-translation based JVM [6]. Their architecture model also did not include any support for local variable caching. One possible reason why local variable cache was not considered in [19, 6] may be they assumed high performance processors as the base of the JVM which normally have 32 or more general purpose registers. This number of GPRs may be practically large enough to map frequently accessed local variables to registers if an elaborate register mapping scheme is used during the translation of Java bytecode. The target of hardware-translation mechanism assumed in this paper is embedded processors which normally have around 16 GPRs, probably not enough to accommodate local variables.

Sethi and Kubiczek proposed to extend the size of register file of their ARC tangent-A4 processor to place the local variables in the register file [7]. Since they chose to handle the bytecode by the software, the execution speed is not accelerated. Also, to manipulate the larger register file, their instruction set must be extended, which may not be a generic solution. The local variable cache proposed in this paper is invisible in both native and JVM modes. Thus, recompilation of applications is not necessary. If the local variable cache and GPRs are merged with an extra write port (to write the value of a local variable to a GPA and an LV entry simultaneously), the proposed local variable cache will have a similar structure as their Unified Register Mapped (URM) stack.

8. Conclusion and Future Work

In this paper, we proposed to add a small register file in the datapath of the hardware-translation based JVM and to use it as a cache for local variables. With few exceptions, a 16-entry local variable cache, which is the size of a typical register file in embedded processors, eliminated 60% to 98% of memory accesses caused by the local variables in the benchmark programs.

Exceptions fell into two types. The first type was the SAXON XML parser in which initial (cold) misses were dominant due to the short execution path. For this type of applications, we proposed to use the local variable cache to pass parameters on the method invocation. This option improved the hit ratio of the local variable cache (16-entry) for SAXON with trend test case from 39% to 53%.

The second type was computation intensive workload represented by the FFT from SciMark 2.0. This type of workload accesses many local variables and their access frequency over the variable index draws a complex curve. While compiler optimization or offline profiler are effective, these options may not alway be available. In this paper, we investigated the on-the-fly profiling of the local variable access behavior using an array of counters. With this option, the hit ratio of FFT was almost doubled (from 44.1% to 82.6%).

The topics of further investigation include the following. To estimate the overhead of chip area and operation speed, it is necessary to design a prototype of the hardware-translation unit including the local variable cache. In this paper, we only considered the bytecode that can be directly executed by the hardware-translation module. By taking into account both bytecodes that are hardware-executable and that require software emulation, we can estimate the performance improvement of the proposed architecture more accurately.

Instruction folding is a technique to combine the operations of multiple bytecodes to accelerate the execution of the JVM [18]. For example, the sequence of `ILOAD_3`, `ILOAD_4`, `IADD`, `ISTORE_3` may be combined and translated into a single native instruction of `ADD [R7, #12], [R7, #16]`. However, this instruction is not possible for most embedded processors which cannot take memory locations as operands for arithmetic and logical operations. Since the proposed local variable cache is placed inside the datapath of the processor and is accessible in a similar manner as general purpose registers, the instruction folding should be applicable to the JVM with a local variable cache. For example, it should be possible to translate the above bytecode sequence into an extended instruction of `ADD LV3, LV4` with the instruction folding. The control signals of this extended instruction can be generated by enabling `A_LV`, `B_LV` and `W_LV` instead of `A_GPR`, `B_GPR` and `W_GPR` in the normal `MOV` instruction. The local variable cache alone reduces the memory access latency (and possibly power consumption). However, when it is combined with the instruction folding, it eliminates the bytecodes accessing the local variables. The complexity of the hardware-translation module is increased since the translation is not done by the single bytecode basis. Rather, it has to detect a sequence of several bytecode for which folding is possible. Therefore, it is necessary to find a set of bytecode sequences for which the complexity of the translation module and the performance gain are balanced.

Acknowledgments

Most of this work was performed when the author was with ASTEM Research Institute Kyoto. I would like to thank Mr. Kenji Yabuuchi, who was formerly a manager of the Software Development Group at ASTEM R. I. Kyoto, for his support. Mr. Yamada personally supported the computer equipment used for the simulations in this paper. The comments from anonymous reviewers were valuable and significantly improved the quality this paper. CaffeineMark is a trademark of Pendragon Software.

References

[1] Tim Lindholm and Frank Yellin, "The Java(TM) Virtual Machine Specification (2nd Edition)", Addison-Wesley Professional, 1999

[2] Douglas Kramer, "The Java. Platform. A White Paper", JAVASOFT, May 1996.

[3] "The Java HotSpot. Virtual Machine. Technical White Paper", Sun Microsystems Inc., May 2001.

[4] Steve Steel, "WHITE PAPER ACCELERATING TO MEET THE CHALLENGE OF EMBEDDED JAVA", ARM Limited, Nov 2001.

[5] "JSTAR - Java Coprocessor: High Performance, Low Cost, Low Power Java Virtual Machine Accelerator", Nazomi Communications, Inc.

[6] Ramesh Radhakrishnan, Ravi Bhargava, Lizy K. John, "Improving Java performance Using Hardware Translation", in *Proceedings of International Conference on Supercomputing*, pp427–439, 2001.

[7] Ashish Sethi and Matt Kubiczek, "Custom processors rev Java execution", `http://www.eetimes.com/story/OEG20020329S0016`, EE Times, April 2001

[8] ORDER GRANTING MOTION FOR PARTIAL SUMMARY JUDGMENT OF NON-INFRINGEMENT , Case No. C 02-02521-JF, UNITED STATES DISTRICT COURT, NORTHERN DISTRICT OF CALIFORNIA, SAN JOSE DIVISION, September 30, 2003.

[9] D. Gregg, J. Power and J. Waldron, "Benchmarking the Java Virtual Architecture - The SPEC JVM98 Benchmark Suite", Java Microarchitectures, pp1–18, Kluwer Academic.

[10] Kaffe.org, `http://www.kaffe.org`.

[11] "SciMark 2.0 Java Numerical Benchmark", `http://math.nist.gov/scimark2/credits.html` .

[12] "SAXON The XSLT and XQuery Processor", http://saxon.sourceforge.net/ .

[13] "DataPower: XSLTMark XSLT Performance Benchmark", http://www.datapower.com/xmldev/xsltmark.html .

[14] "JOrbis – Pure Java Ogg Vorbis Decoder", http://www.jcraft.com/jorbis/, JCraft, Inc.

[15] "The Embedded CaffeineMark", Pendragon Software Corporation, 1997.

[16] "The SPARC Architecture Manual", Version 9, SPARC International, September 2000.

[17] "SPARC-V8 Supplement, SPARC-V8 Embedded (V8E) Architecture Specification", Version 1.0, SPARC International, October 23, 1996

[18] "picoJava-II Processor Core", Sun Microsystems, April 1999.

[19] John Glossner and Stamatis Vassiliadis, "Delft-Java Dynamic Translation", in Proc. of 25th Euromicro Conference, Vol. 1, pp1057–1062, Sep. 1999.