Object-Oriented Architectural Support for a Java Processor

N. Vijaykrishnan[1], N. Ranganathan[1], and R. Gadekarla[2⋆]

[1] Center for Microelectronics Research, Department of Computer Science & Engg., University of South Florida, Tampa, FL 33620
[2] Bell South Communications, Birmingham, Alabama.

Abstract. In this paper, we propose architectural support for object manipulation, stack processing and method invocation to enhance the execution speed of Java bytecodes. First, a virtual address object cache that supports efficient manipulation and relocation of objects is presented. The proposed scheme combines the serialized handle and object lookup into a single lookup. Next, the extended folding optimization that combines the execution of a sequence of bytecodes is proposed. This eliminates the redundant loads and stores of local variables associated with stack architectures. Also, three cache-based schemes: hybrid cache, hybrid polymorphic cache and two-level hybrid cache to implement virtual method invocations are presented. These schemes utilize the receiver type locality at call sites and eliminate the need for creating dispatch tables used in current JVM implementations.

1 Introduction

Java's object oriented nature along with its distributed nature makes it a good choice for network computing. The secure, dynamic, multi-threaded and portable nature of Java along with its support for garbage collection meets most of the current software requirements. This along with its compact object code makes it as the language of choice for the burgeoning embedded processor market. Applications in Java are compiled into the byte code format to execute in the Java Virtual Machine (JVM). The core of the JVM implementation is the execution engine that executes the byte codes. This can be implemented in four different ways (Figure 1):

1. An interpreter is a software emulation of the virtual machine. It uses a loop that fetches, decodes and executes the byte codes until the program ends. The Java interpreter due to the software emulation has an additional overhead of executing more instructions than just the byte codes. Also, it suffers from the penalty of inefficient use of micro-architectural features like cache and branch prediction. The software decoder is normally implemented using a large switch statement that affects the locality of the Instruction cache. Both the Java byte codes (instructions) and the associated data become data when the interpreter is executed thereby resulting in more data misses [13].

⋆ This work was done when the author was at University of South Florida

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 330–355, 1998.

2. A Just-in-time (JIT) compiler is an execution model which tries to speed up the execution of interpreted programs. It compiles a Java method into native instructions on the fly and caches the native sequence. On future references to the same method, the cached native method can be executed directly without the need for interpretation. JIT compilers have been released by many vendors like Borland, Symantec [26], Microsoft, and Softway [25]. Compiling during program execution, however, inhibits aggressive optimizations because compilation must only incur a small overhead. This constraint makes the JIT compilers intrinsically slower than direct-native code execution. Further, the quality of the generated code critically depends on the specific features of the target processor. Hence, porting these compilers requires a large amount of work. Another disadvantage of JIT compilers is the two to three times increase in the object code, which becomes critical in memory constrained embedded systems. There are many ongoing projects in developing JIT compilers that aim to achieve C++ like performance, such as CACAO [14]. Recently, performance analysis and tuning environments to improve the performance of JIT compilers on specific-platforms like Intel-based systems have been developed.
3. Off-line bytecode compilers can be classified into two types: those that generate native code and those that generate an intermediate language like C. J2C [27], Jolt [28] Harissa [29], Turbo J [22] and Toba [24] are compilers that generate a C code from byte codes. The choice of C as the target language permits the reuse of extensive compilation technology available in different platforms to generate the native code. In bytecode compilers that generate native code directly like NET [13], Asymetrix SuperCede [23] portability becomes extremely difficult. In general, only applications that operate in a homogeneous environment and those that undergo infrequent changes benefit from this type of execution.
4. A Java processor is an execution model that implements the JVM directly on silicon. It not only avoids the overhead of translation of the byte codes to another processor's native language, but also provides support for Java runtime features. It can be optimized to deliver much better performance than a general purpose processor for Java applications by providing special support for stack processing, multi-threading, garbage collection, object addressing and symbolic resolution. It can also make efficient use of the processor resources like the cache and branch prediction unit unlike the interpreters. Java processors can be cost-effective to design and deploy in a wide range of embedded applications such as telephony and web tops. It is estimated that Java specific processors could capture half the total available micro controller market by 1999 [33].

The various features of Java that make it suitable for software applications also add performance overheads. The use of polymorphic methods, garbage collection, context switching, dynamic loading and symbolic resolution contribute to the slow execution speed of Java code. Implementations of other languages such as Smalltalk and Forth have attempted to reduce some of these overheads.

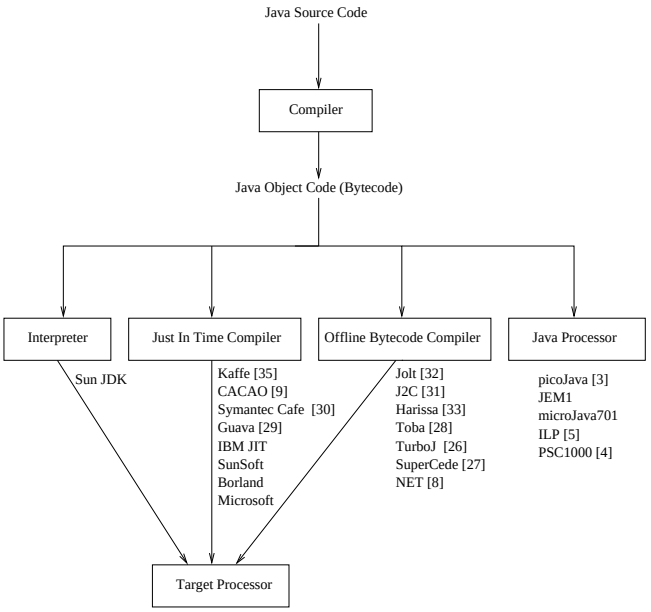

Fig. 1. Executing Java byte codes

Smalltalk systems such as SOAR and Mushroom provide architectural support for efficient object addressing and dynamic dispatches [16], [10]. Further, various Forth machines that are stack based like the JVM include various features to enhance performance of stack operations [30]. The proposed Java architecture is a stack machine executing bytecodes directly on silicon. Sun's microJava, picoJava-I [3] and the Patriot Scientific PSC1000 [4] are other stack-based Java processors. In microJava and picoJava, the JVM instruction set forms the native language of the processor. On the other hand, the PSC1000's stack based instruction set is different but similar to the JVM instruction set. Further, Java processors based on the picoJava core such as Rockwell's JEM1 have been developed. A different approach to Java processors is used in the ILP machine proposed in [5]. Unlike the stack based processors, it is based on an incremental compilation model and involves code expansion and translation overhead. The Java processors have been reported to execute bytecodes much faster than other execution models. picoJava-I was reported to execute bytecodes 15 to 20 times faster than an interpreter running on Intel 486 and five times faster than a JIT compiler running on Pentium [3]. In this paper, we investigate the various aspects of Java execution and identify architectural features that would improve Java's speed of execution. Based on this investigation, a new processor architecture is proposed to execute the JVM instructions with better runtime efficiency. The proposed architecture includes support for object addressing, virtual method invocation and extended folding.

In the next section, the proposed Java processor architecture is described. The processor organization and instruction set are explained in this section. In Section 3, the instruction cache design for the Java processor is investigated. The architectural support for object addressing, extended folding and dynamic dispatch are explained in Sections 4, 5 and 6 respectively. Finally, conclusions are presented in Section 7.

2 Proposed Architecture

The implementation of the JVM on hardware provides the dual benefits of running the byte codes as a native language of the processor and also provides support for its runtime features. In this section, a new processor architecture that executes Java byte codes directly on hardware is described.

2.1 Processor Organization

The Java processor architecture shown in Figure 2 consists of the following functional units: the stack unit, the instruction fetch and program counter(PC) control unit, the variable length instruction decoder, arithmetic unit, memory and I/O interface unit, control unit, and the method inlining unit. The processor includes a 4K direct mapped instruction cache, a 4K direct mapped data cache and an 8K virtually addressed object cache. The processor core consists of a four-stage pipeline: instruction fetch, decode, execute and write-back. The Java bytecodes are initially fetched from the instruction cache by the instruction fetch and PC control unit. These bytecodes are then decoded by the variable length decoding unit. Simple instructions are executed directly on hardware, while the more complex instructions such as object creations are handled by the trap handler. The execution of object manipulation instructions utilizes the virtually addressed object cache. The data is accessed from the object cache using the object reference in the stack unit and the offset field. The offset field is either a part of the instruction or present in the stack unit. The method inlining unit is used to implement the dynamic dispatch associated with virtual method invocations. This unit contains a hybrid cache that caches the method location and receiver types at the call sites. The hybrid cache is indexed using the call site location contained in the PC to obtain the method location for the invoked virtual method. The cached method location is communicated to the instruction fetch and PC control unit when the current receiver type and cached receiver type match. The hybrid cache scheme used to implement the virtual method calls is explained in detail later. The Branch Target Buffer (BTB) is used to predict the branch direction and notify the fetch unit whenever a branch instruction is decoded. This predictor contains a 1K 1-bit prediction history table. The arithmetic and logical instructions are executed by the integer and floating point units. These units communicate with the stack unit to obtain inputs and to write back the outputs. The integer unit consists of a 32-bit ALU and a 32-bit multiplier. The FPU supports both double and floating point arithmetic

operations as well as some of the conversion operations. In order to reduce the complexity of the processor, division is performed using microcode. The memory and I/O interface unit serves as the interface for off-chip communication.

The stack unit forms the core of the processor and contains the stack cache, the stack addressing unit, the stack cache manager and some special purpose registers. The stack cache unit contains two register sets that store the top most elements of the stack frames corresponding to two threads of execution. Each register set contains thirty-two 32-bit registers which can be accessed randomly. Additionally, there are two special registers, the top on stack register (TOS) and next on stack register (NOS) corresponding to each thread. These registers serve as the input for the arithmetic and logical operations. Further, the TOS register is used during object manipulation and method invocation operations. The special purpose registers in the stack unit store the current frame pointer, the local variable base and the stack pointer. These registers are used by the stack cache addressing unit to generate the addresses required to access the data in the stack cache. The random, single cycle access of the local variables in the register set using this addressing support enables the extended folding optimization. This optimization eliminates the requirement to move the data from the local variables to the TOS and NOS register before computation. The stack cache manager prevents underflows and overflows in the stack cache by shuttling data to and from the data cache. The processor also uses a by-pass logic to allow the execute stage to continue without the write back stage having to complete. This is important as most instructions depend on the value of the TOS.

The processor implements the JVM instruction set, the quick format variants and additional instructions to facilitate system calls and resolution operations. The instruction set makes use of a fixed length opcode of 8 bits, but instructions vary in length. The instruction set of the Java processor can be classified into seven classes: load and store, arithmetic and conversion, operand stack management, control transfer, object manipulation, method invocation and system instructions. In the later sections, we will focus more on the object addressing and method invocation instructions.

3 Instruction Cache Behavior

In this section, the instruction cache behavior of the Java bytecodes and the equivalent C code is studied. This analysis is used in designing the instruction cache for the proposed Java processor. First, the experimental model and tools that were used to obtain the traces are explained. Then, the effect of block and cache size variation on the miss rates of the instruction cache are investigated. The interpreter in Java Soft's JDK 1.0.2 version for Solaris running on a Sparc-20 was modified to obtain the traces for performing the Java byte code cache analysis. The byte code addresses were generated by executing the benchmarks using the modified interpreter. The traces obtained were then analyzed using the dineroIII cache simulator [7]. The trace patterns for the C code were obtained

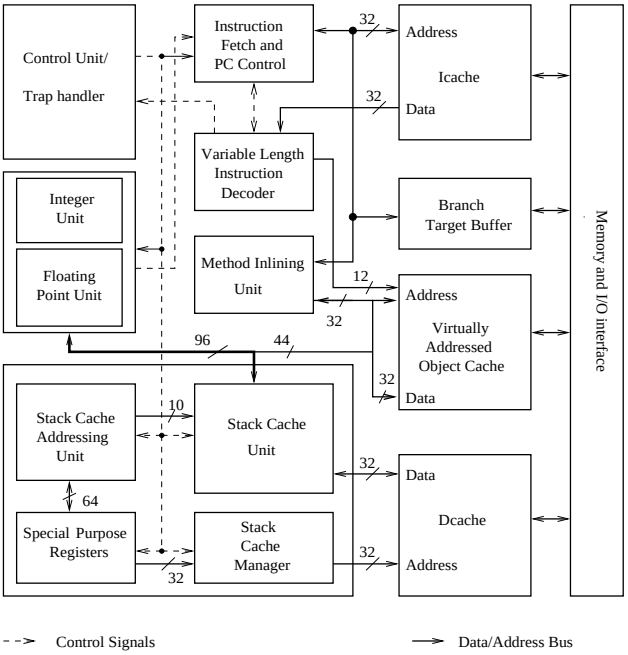

Fig. 2. Processor Architecture

using qpt2, a quick profiler [7] running on a Sparc-20 processor using Solaris. The cache analysis on this trace was again done using the dineroIII cache simulator. The programs selected for comparing C and Java byte codes included some Open Software Foundations Java Test Suite programs [11]: (i) Dhrystone, (ii) RSA Security, Inc. MD5 Message-Digest Algorithm (MDA), (iii) Logical disk simulation (lld), (iv) Hanoi and (v) the Line count utility.

The changes in the instruction miss rate for different block and cache sizes are investigated. For Java byte codes, the effect of block size variation for an 8K direct mapped instruction cache was studied. Figure 3(a) shows that for Hanoi and Dhrystone, when the block size increases from 4 bytes to 8 bytes the instruction cache miss rate decreases. After attaining the best miss rate for an 8-byte or a 16-byte block, the miss rate increases again for larger blocks. Initially, the small blocks fail to capture the locality in full resulting in more misses and hence miss rate decreases with increase in block size. The subsequent increase in miss rates can be attributed to the small locality of the Java byte codes. The frequent invocation of many small methods results in a decrease in locality with an increase in block size. The basic block size of the Java byte code was found to be around 5 (4.9 for Dhrystone) instructions. Smaller the instruction length, more the number of instructions that can fit in a block of fixed size. This results in a smaller locality for the Java byte codes whose average instruction length is around 1.8. Also, it is observed that the lld and MDA benchmarks have low miss

rates for both the Java byte codes and the C code. The lld benchmark makes use of a small loop structure frequently to simulate the accesses to the segments of a disk resulting in low miss rates. In the case of C code, it is observed from Figure 3(b) that the miss rate decreases with increase in block size from 4 to 64 bytes. This indicates a marked variation from our observation for Java byte codes. The C code has more block locality because of fewer method invocations, larger methods and due to optimization done by the compiler like loop unrolling.

The effect of instruction cache size on miss rates was analyzed for direct mapped caches ranging from 512 bytes to 8 Kbytes. Block sizes of 16-bytes and 64-bytes were used for the Java and C Instruction cache analysis respectively. Figures 4 shows that the miss rate decreases as the cache size increases for both Java and C code. Initially, there are large capacity misses, but as cache size approaches 8K, the miss rate becomes very small. It is also observed that the miss rates are larger for C code than for Java byte codes. The smaller miss rates for Java byte codes can be attributed to a reason similar to the one that explains differences in the C and C++ instruction miss rates. It is stated in [2] that the larger text size of the executable content of the C++ code results in the use of a larger cache to achieve the same miss rates as the corresponding C code. Similarly, the smaller executable content of the Java byte code as compared to the C code results in smaller miss rates for Java. This is due to the capacity misses dominating the miss rate for smaller cache sizes and a larger executable size causing capacity misses for larger caches.

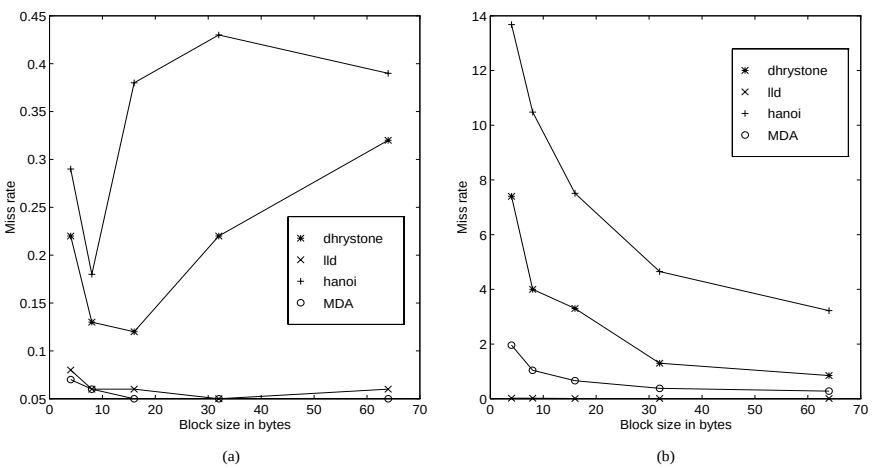

Fig. 3. Effect of varying block size on an 8K Icache for (a) Java byte codes (b) C code

It was observed that the block size of the cache has a significant impact on the performance of a Java processor. It was also noticed that the miss rates of Java byte codes are less than that of C code. Based on this analysis, an instruction

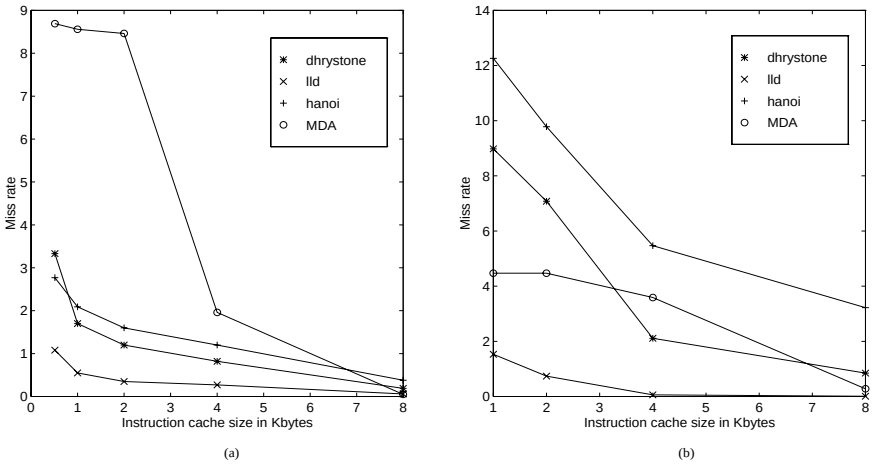

Fig. 4. Effect of varying Icache size for (a) Java byte codes (b) C code

cache of 4Kbytes with a block size of 16 bytes was utilized in the proposed Java processor.

4 Object Addressing Support

Java is an object-oriented language that operates on variable sized contiguous lists of words called objects. Objects, in turn, are comprised of many fields. The fields of the object are accessed and manipulated using the object manipulation instructions of the JVM. The location of the field is determined by adding the base location of the object in which the field is contained with its offset within the object. The number of indirections required to obtain the object's base location varies based on the object representation.

Object representation models used in various JVM implementations can be classified as those that use indirect or direct addressing. In the indirect address object models, the object reference does not point directly to the instance data. Instead, they point to a handle that contains a pointer to the base location of the object. Thus, all references to the object in the code are pointers to the handle. Java Soft's JDK uses an indirect address object model shown in Figure 5. The handle location contains pointers to the object's location in the heap and a pointer to the object's method table. Using this representation, two levels of indirection, one to the handle and another to the object field, are required for each field access. In order to avoid the cost of two levels of indirection for each object access, systems such as NET compiler and CACAO use a direct address object model [14]. In the direct address object model, the object reference directly points to the base location of the object. Though this allows quicker access to data, the relocation of objects during garbage collection and heap compaction lead to time consuming updates. Every reference to the

object needs to be updated after the object is relocated. On the contrary, the use of an additional level of indirection in indirect address models allows the virtual machine implementation to find the up-to-date data after garbage collection and heap compaction efficiently. Only the base pointer location in the object's handle is updated after relocation.

In order to avoid the additional indirection to access an object while maintaining the ability to efficiently relocate the object, the use of a virtually addressed object cache is investigated in this section. This approach is similar to the object cache proposed for a Smalltalk processor [10]. As mentioned in their work, the design of the object cache structure and mapping need to be decided based on the language usage. The focus of this work is on the design of the object cache based on traces obtained from various Java benchmarks. The performance of the various cache configurations is evaluated using trace driven simulations along with an analytical timing model for the cache. The analytical timing model enables to take into account the impact of cache and block size on cache cycle costs unlike the fixed cycle cost used in [10].

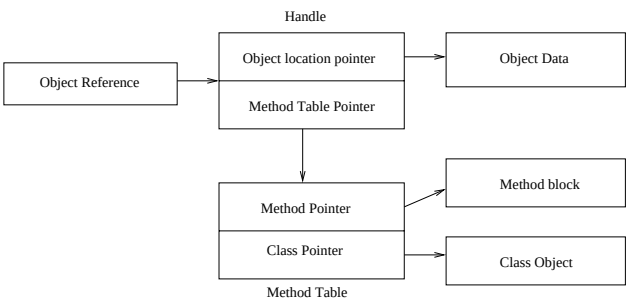

Fig. 5. Sun JVM Object Representation

4.1 Object Manipulation Instructions

The object manipulation instructions of the JVM operate on both class instances and arrays. Distinct sets of instructions are used to manipulate class instances and arrays. However, all these instructions locate the field for manipulation by adding the fields offset with the base location of the object. These instructions constitute around 12% of instructions executed by the JVM. Hence the execution of these instructions has a significant impact on overall performance of the Java code. While the static class objects are manipulated by the *getstatic* and *putstatic* instructions, the object instances are manipulated by the *getfield* and *putfield* instructions of the JVM. The fields of these objects are initially addressed symbolically in these instructions and are converted into a quick format variation after symbolic resolution. The instance variable manipulation instructions are converted into two forms of quick format instructions based on the size

of the field offset. When the offset of the field being accessed is less than 256, the object manipulation instructions are converted to quick format instructions like *getfield_quick* and *getfield2_quick*. In these quick format instructions, the field offset is contained within the instruction format. If the offset is greater than 255, the quick format variation of the instruction contains an index to the constant pool. This index is used to access the offset and type of the field from the constant pool. Due to the small size of the objects used in the benchmark programs, it was found that after resolution about 99% of the object manipulation instructions were converted to a quick format variation, which contains the field offset within the instruction format. Figure 6(a) illustrates the operation of the *getfield_quick* instruction using a handle representation. The instance field is accessed by adding the field's offset present in the instruction with the base location of the object. The base location of the object is obtained from the handle of the object. The object reference on the top of the stack (TOS) is used to locate the object's handle. Array components are accessed using instructions such *iaload* and *iastore*. Both the object reference and the offset are obtained from the stack unit for these instructions.

4.2 Virtual Address Object Cache

In the handle representation, each (object reference,offset) pair maps to a unique memory location. This observation indicates that there would be no aliasing problems in a cache which is virtually addressed using this pair. In the virtually addressed cache, the object reference and the offset are used to access the field directly. This eliminates the additional indirection involved with the indirect object access models. Further, the offset addition operation involved with both the direct and indirect object access models are eliminated. The (object reference,offset) pair, which is used to address the virtually addressed cache, is partitioned to form the block index, block offset and the tag bits. The block index is used to select the appropriate block within the cache which may contain the data, the block offset indicates the displacement into the selected block and the tag bits are used to verify that a hit has occurred.

The virtual address used to index the object cache consists of a 32-bit object reference and a 32-bit field offset. The 64-bit (object reference,offset) pair results in a large overhead for maintaining the tag bits. In order to reduce the tag overhead, the object sizes in various Java benchmarks were analyzed. The profiling results for the various benchmarks indicate that the average object size is around 30 bytes. While an offset of 8 bits would be sufficient for around 99% of all non-array objects, an offset of 12 bits would be useful for array components. Thus, the virtual address to access a field consists of a 32-bit object reference and a 12-bit field. Larger offsets could be handled explicitly using the real address available in the object tables or larger objects could be broken into smaller objects by the compiler. An object table that maintains the handles of the recently used objects is used to handle the virtual address cache misses. This table is used to find the actual address of the object in the memory. Whenever an object is relocated in memory, the corresponding object table entry is updated.

Though the actual address of the object changes, the (object reference, offset) pair associated with the field remains the same. Thus, the entries map to the same location in the virtual address cache after object relocation. The use of the virtually addressed cache avoids additional indirection in accessing objects, while permitting easy relocation of objects.

Cache interference occurs when more than one address maps onto the same cache location. Thus the partitioning of the address to choose the tag bits, block index bits and the block offset bits is critical to the performance of the cache. Poor choice of the block index bits would result in frequent cache interference. Figure 7 shows three ways in which the block offset and block index bits were selected. Schemes 1 and 2, suffer from high cache interference and hence cause high miss rates. In scheme 1, the most significant bits (MSB) of the object references are used to index the block. Since object references were assigned sequentially, most of the references map onto the same block resulting in high miss rates. In scheme 2, a part of the least significant bits(LSB) of the object reference and the most significant bits of the offset is used as block index. Since most objects are small there is a lot of redundancy in the MSB bits of the offset. Hence the miss rates still remain high due to cache interference. In scheme 3, the LSB of the object reference is used as the block index and the LSB of the offset is used as offset. Since there is little redundancy in the block index bits, this scheme provides hit rates that range around 90%. However, the frequent use of large objects results in high cache interference due to the mapping of the various fields of the object onto the same block. Due to the small average size of objects, most of the object accesses hit in the object cache.

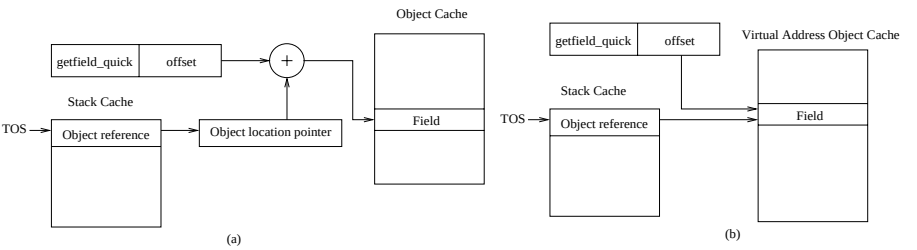

Fig. 6. Field access (a) without a virtual address cache (b) with a virtual address cache

4.3 Performance Analysis

The various configurations of the virtual address object cache were evaluated using the benchmarks described in Table 1. Scheme 3 was used as the mapping strategy to index the object cache and an 1K entry direct mapped object table was used to handle the object cache misses for all the configurations. The average number of cycles required for each object access was studied by varying the cache

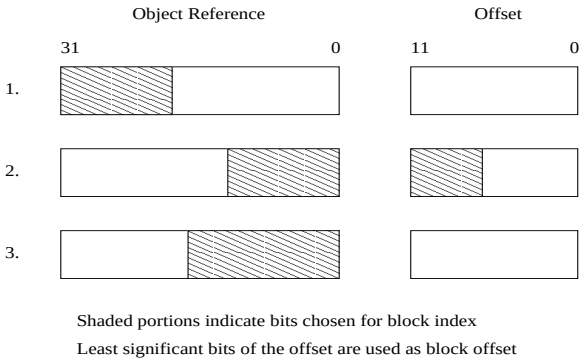

Fig. 7. Virtual Addressing Mapping Strategies

Table 1. Description of Benchmarks and size of dispatch tables

Benchmark	Description	T1	T2
Javac	Java compiler [18]	3932	22488
Javadoc	Documentation generator for Java source [18]	2784	20820
Disasmb	Java class disassembler	988	3380
Sprdsheet	Spreadsheet applet [18]	3304	14396
JLex	Lexical analyzer generator written for Java [20]	1752	10092
Espresso	Java compiler [21]	1836	10224
Lisp	A Lisp interpreter [8]	1120	4788
GUI	A graphical user interface applet [18]	4572	24740
Heap	Garbage collection applet[18]	4336	30496

T1 - DTS size in bytes; T2 - VTBL size in bytes

and block size, associativity and write policies. The access times of the various cache configurations are based on a 0.8μ technology and were obtained using modifications to *cacti* [9]. Further, the number of cycles per object access is based on a 10ns clock used for the processor. The number of cycles per object access also includes the block transfer time, a 70ns latency for each object cache miss, cycles for handling object table misses on an object cache miss and the object cache access time.

Figure 8(a) shows the variation in average number of cycles per object access with change in block size. It was observed that a block size of 8 words resulted in the minimum average number of cycles per object access for all benchmarks except for *spreadsheet*. Since hit rates increase with increase in block size, the average number of cycles decrease when the block size changes from 4 to 8 words. However, the miss penalty also increases for larger blocks. When the block size increases to 16 words, the miss penalty dominates the improvement obtained by higher hit rates. This results in a higher average number of access cycles for 16 word blocks. However for the *spreadsheet* benchmark, the minimum number of

average cycles is obtained for a 16 word block. This is due to the frequent use of large objects that benefit from a larger block.

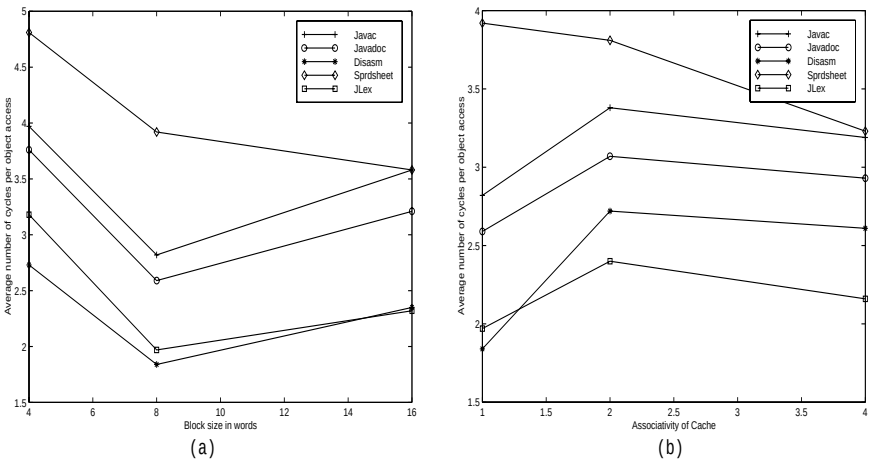

Fig. 8. (a) Variation in average number of cycles per access with block size. A direct mapped, 8K word virtual address object cache that utilizes a write back with dirty replacement scheme is used. (b) Variation in average number of cycles per access with associativity. An 8K word virtual address object cache with an 8 word block that utilizes a write back with dirty replacement scheme is used.

Figure 8(b) shows the effect of the degree of cache associativity on the performance. It was observed that the hit rates are higher for a higher degree of associativity. However, the increase in cache access time with associativity dominates this benefit. Thus, it can be observed that the direct-mapped configuration (associativity = 1) provides faster object access than the two-way and four-way associative mappings. However for the *spreadsheet* benchmark, the improvement in hit rate with increase in associativity dominates the increase in cache access time. Thus, the four-way associative mapping performs the best.

The effect of varying the size of the virtual address object cache is shown in Figure 9(a). It was observed that the access speed improves with increase in cache size due to the reduction in cache interference misses. It was also observed that the rate of improvement reduces as the cache size keeps increasing. The effect of three write policies on the cache performance was also studied. Figure 9(b) shows that the write back with dirty replacement scheme provides the best performance among the three schemes. Among the write through schemes, the write allocate on a write miss performs better than the no allocation on a write miss scheme.

Figure **??** shows the average number of access cycles required for object manipulation instructions with and without the use of a virtual address object cache. These are referred as R2 and R1. Without the use of a virtual address object cache (R1), a handle lookup followed by an offset addition is required

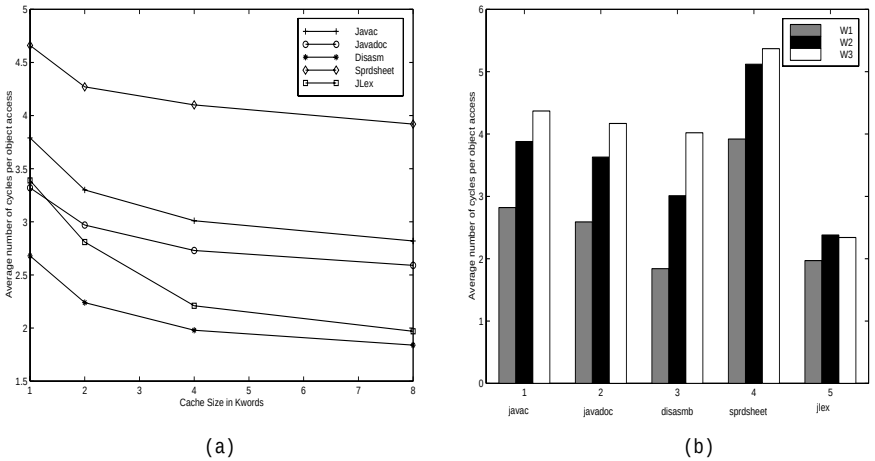

Fig. 9. (a) Variation in average number of cycles per access with cache size. A direct mapped virtual address object cache with 8 word blocks that utilizes a write back with dirty replacement scheme was used. (b) Effect of write schemes on the performance of the Cache. W1 refers to the flagged write back scheme, W2 refers to the write through with write allocate and W3 refers to write through with no write allocate. A direct mapped virtual address object cache of 8K words with 4 word blocks was used for all schemes.

before an object lookup. The virtual address object cache combines the serialized handle lookup, offset addition and object lookup into a single lookup. An 1K word object cache and an 1K word virtually address object cache with 4 word blocks were used for the two schemes, R1 and R2 respectively. Further, the write back policy and LRU replacement strategy are used for the comparison. An 1K entry object table was used to maintain the handles in both the schemes. It was observed that the virtual address cache requires lesser number of average cycles than the indirect object address model for four of the benchmarks. For the *disassembler* benchmark, the direct mapped 4K entry virtual object cache with a block size of 4 words reduces 1.95 cycles for each object access compared to the indirect object access using R1. It was also observed that the high cache interference due to the virtual address mapping scheme results in poor performance for the *spreadsheet* benchmark.

5 Extended Folding

In stack implementations, the operation of bringing local variables of the current frame to the top of the stack and storing the results back consumes a large amount of time. The example expression a = (b - c) + (d - e) translates into the byte codes sequence shown in Figure 11(a). The picoJava processor uses the folding optimization to boost the performance by combining an instruction which

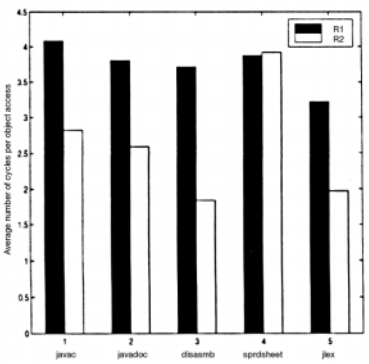

Fig. 10. Average number of cycles per access with and without a virtual address object cache. An 8K word physical address object cache and an 8K word virtual address object cache both with 8 word blocks were used for the two schemes, R1 and R2 respectively.

loads from a stack frame followed by an arithmetic operation or a pushing of a simple constant followed by an arithmetic operation into a single operation. The resulting byte code after folding is given in Figure 11(b). The mapping of the top most elements of the stack onto a register array permit the random access to the local variables. This eliminates the separate step required for loading the local variable onto the top of the stack. However, this optimization is not possible in case the local variable is not present in the register array.

In our processor, we further enhance this folding process to combine other patterns of instructions. Among the various combinations considered, only the (load,load,arithmetic), (load,arithmetic,store), (arithmetic,store) sequences occurred frequently. Other sequences including (pop,pop),(dup,load,arithmetic) were also tried but their frequency was too low to consider implementation. The load includes instructions that load data from a local variable or push a constant. The store operations are those which write back results into the local variables. The extended folding optimization is disabled whenever the local variables are not available in the stack cache. This disabling, however, occurs infrequently due to the small average number of local variables (3.2 for Javac) and the small average size of the maximum operand stack (4.5 for Javac) used by Java methods. This results in the availability of the local variables in the stack cache for most accesses.

The example expression reduces to the code shown in 11(c) after the extended folding optimization Like the folding optimization, the random single cycle access to the registers in the stack cache permits the extended folding optimization. However, the instruction decoder and the frame addressing support required for folding need to be enhanced to support extended folding. The instruction decoder has been modified to identify the new patterns of byte code

that have to be combined using extended folding. Further, the extended folding optimization requires access to an additional local variable in the stack cache over the folding optimization. The frame addressing support has been enhanced to provide the locations of two local variables in the stack cache simultaneously. An adder has been included to add the offset of the local variable to the local variable base pointer in order to access the additional local variable. The additional support required for extended folding over folding has a negligible impact on the cycle time. However, there is a significant reduction in number of execution cycles due to the extended folding optimization. The reduction in number of cycles due to extended folding optimization for the various patterns combined is given in Table 2.

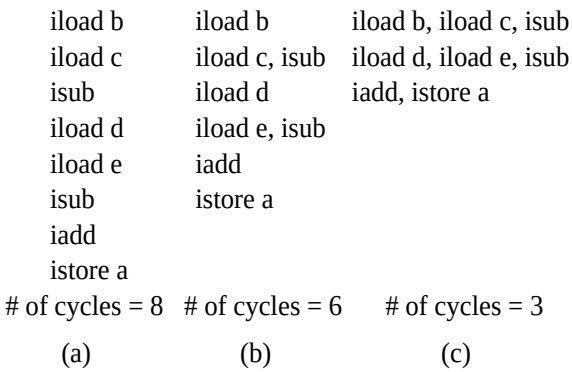

Fig. 11. Extended folding optimization

Table 2. Folding Enhancements Effectiveness

Benchmark	Instructions	P1	P2	P3	P4	I1	I2
Dhrystone	1802073	3429	10564	15352	17021	18781	35802
Lisp Interpreter	798415	1138	3498	59	1643	1197	2840
Ticker Applet	390690	4709	7479	309	2537	5018	7555
Class Disassembler	3776589	28856	33233	8584	216	37440	37656
Linpack	8040994	712186	7681	115	20500	712301	732801
Javac	8428348	181390	154497	11991	69715	193381	263096
Javadoc	6593825	139905	119269	3637	57463	143542	201005
Espresso	6685308	232478	152656	12604	5580	245082	250662
Javalex	754182	512	15487	17	216	529	745

P1 - (load,load,arithmetic) P2 - only (load,arithmetic) P3 - (load arith store) P4 - only (arith store) I1 - Number of three instruction optimizations
I2 - Improvement over picoJava folding

6 Virtual Method Invocation

One of the major objectives of an object-oriented programming language like Java is to allow easy reuse and modification of the existing software components. Unfortunately, the characteristics of this programming style result in large runtime overhead due to the frequent use of dynamic dispatches. The invokevirtual instruction, which is the most frequently used method invocation statement in Java bytecodes, uses dynamic dispatch. The appropriate code for execution at a virtual method call site is determined by the object reference (receiver) on the top of JVM's stack. The most effective way of reducing the dynamic dispatch overhead based on the receiver type would be to eliminate the virtual method invocation and bind the call statically at compile time. Various static analysis techniques can be used to eliminate the virtual method invocations [12]. However, only a portion of the virtual calls can be eliminated due to the difficulty in deciding the call sites that can be safely bound statically. In this work, we focus on handling the calls that cannot be eliminated by these techniques.

Dispatch Table Search (DTS) is the simplest technique to implement the virtual method invocation. When a virtual method is invoked, a search for the invoked method is performed in the class of the current object. If no method is found, the search proceeds in the superclass(es). The DTS mechanism is best option in terms of space cost. However the search penalty renders the mechanism very slow. Table based or cache based schemes can be used to efficiently implement dynamic dispatch. Dispatch table based techniques have been used in various JVM implementations such as Sun's JDK and CACAO [14]. These tables are based on the principle of using a fixed index for each method that a class understands, in such a way that the method will have the same index in all subclasses. This permits table based schemes to provide a constant time dynamic dispatch. However, dispatch table schemes such as VTBL, RD and SC require significantly more data space than DTS because they duplicate information. This is due to each class storing all methods that it understands instead of all methods it defines. Thus, all the inherited methods contribute to the additional entries found in these schemes. Since, many embedded applications are memory constrained the additional data space is a concern. Various compaction techniques like selector coloring and compact tables can be used to reduce the size of these tables [19]. However, most of the compaction schemes have to recompute the dispatch table every time a new class is loaded [6]. This renders the compaction schemes unsuitable in Java's dynamically linked environment.

Cache-based schemes eliminate the need to create and maintain the dispatch tables. Three cache-based schemes: Hybrid Cache (HC), Hybrid Polymorphic Cache (HPC) and Two-level Hybrid Cache (THC) to support virtual method invocation in the Java processor are investigated in this section. The method inlining unit can contain one these three variations based on the performance/cost tradeoffs.

6.1 Hybrid Cache

Each entry in the hybrid cache consists of the class of the object and a method location pointer. Whenever a virtual method instruction is decoded, the n least significant bits of the program counter (the call site address) are used to index into the hybrid cache. The other bits of the program counter (PC) serve as the tag bits for the cache. The comparison of the tag and the comparison of the class of the current object with the cached class entry are performed concurrently. If the tags and class types match, the cached method location entry is used to locate and execute the invoked method. On a cache miss, dispatch table search (DTS) is used as the backup strategy to update the entries of the cache. The case when the tag and receiver hit for the HC scheme is illustrated in Figure 14. The entry (R1,M1) in the HC corresponds to the receiver type R1 and the location of method M1 corresponding to the class of receiver R1.

The HC scheme is similar to other inline cache schemes proposed for Smalltalk systems [15], [16]. The proposed scheme makes use of the receiver type locality at the call sites like the inline cache mechanisms. However, unlike an inline cache mechanism that inserts the receiver type checking the prologue to each method, type checking is performed in hardware in the proposed processor. The sequence of operations involved with the method call shown in Figure 13 are microcoded to implement the invokevirtual instruction of the Java processor. This eliminates the space overhead associated with inserting prologue code. It must also be observed that the proposed scheme is different from the global lookup cache technique [17]. In the global look-up cache, each entry consists of the class of the object, method name and the method location pointer. The method name at call site combined with the class of the object hashes into an index in the cache. If the current class and method name match those in the cached entry, the code at the method location pointer is executed. The computation of the hash function at each dispatch renders the Look-up cache scheme slow. Using HC, the hashing function is eliminated and only the call site location is used to index into the cache.

Unlike an inline cache where misses occur only when the receiver type at the call site changes, the hybrid scheme is also affected by cache interference. Cache interference is the effect of entries of two different call sites mapping onto the same cache location. Thus the mapping strategy used is critical to the performance of the hybrid cache. The results of direct and two-way associative mapping are presented later. Further, the dispatch speed of the hybrid cache mechanism depends on the type locality exhibited at the call-sites by the program. The type locality at the call sites for various Java applications is shown in Figure **??**. The call sites are shown as monomorphic sites (single receiver type), polymorphic sites of degree two and three (two and three receiver types) and call sites with a higher degree of polymorphism. It is observed that the number of different receiver classes actually appearing at a particular call site is very small. The number of monomorphic call sites constitutes more than 40% of the call sites for all the benchmarks. The entry in the hybrid cache always has a hit for such call sites in the absence of cache interference. Further, most of the

polymorphic call sites have a degree of two and less than 15% of call sites have more than two receiver types.

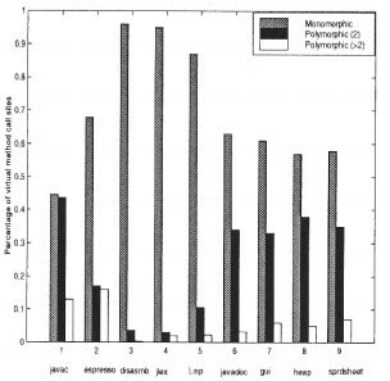

Fig. 12. Type locality at Call Sites

```
Proposed Hybrid Scheme

Call Site Location: invokevirtual object.method
Computation at call site
  entry = cache[PC[1:n]]
  if (entry.class == object->class and cache[entry].tag == PC[32:n+1])
        jump to entry.method_location
  else
     call DTS
  Computation in called method: None
```

Fig. 13. Virtual method invocation using hybrid cache

6.2 Hybrid Polymorphic Cache

The hybrid cache with a single entry performs well for all monomorphic call sites. However, the penalty due to misses at polymorphic call sites increases the average dispatch time. Since most polymorphic call sites have a degree of two as observed in Figure **??**, the use of a per-call-site two entry hybrid polymorphic cache (HPC) to reduce the miss penalty is investigated. Each entry in the HPC contains the class type and method location pointer of two different receiver types. The class of the current object is compared with both the cached classes simultaneously. On a match, the corresponding method location pointer is utilized to invoke

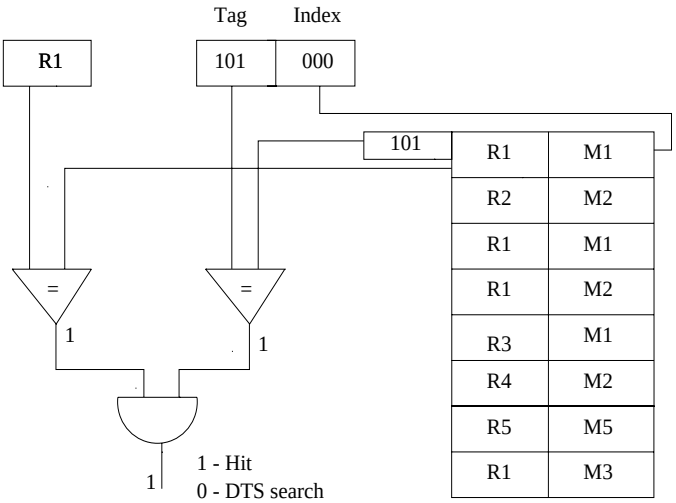

Fig. 14. Hybrid Cache: Tag and receiver hit. Tags are present for all entries but is shown only for the indexed entry

the virtual method. DTS is used to update the least recently used cache entry and perform the dynamic dispatch on a miss. The HPC scheme is similar to the Polymorphic inline cache used as an extension to inline cache mechanism [32].

6.3 Two Level Hybrid Cache

HPC improves the performance at polymorphic call sites by providing two receiver types. However, the large number of monomorphic call sites present in Java code do not utilize the additional receiver type entry associated with the HPC. Hence, the use of a two level single-entry hybrid cache, with a larger first level cache and a smaller second level cache is investigated. The two level scheme enables to provide more than one receiver type for the call sites like the HPC while achieving the space efficiency of the hybrid cache for monomorphic call sites. Polymorphic call sites can benefit from this scheme by allowing two receiver types, one at each cache level. Further, effects of cache interference may reduce as tags are associated with each level of cache. This has an effect similar to increasing the associativity of the hybrid cache. However, the scheme requires extra hardware to compare the tag bits at both the levels simultaneously. On a virtual method invocation, the least significant bits of the program counter index into both levels of the hybrid cache in parallel. If the tag and the receiver of the current object match the cached entry in either level of cache, the corresponding method location is used to perform the method call. If the tag or receiver miss at both the levels, DTS is used to update the entry in the first level cache. Whenever the first level cache entry is updated, its previous contents are copied to the second level cache. Further, the entries of the second level and first level

cache are swapped, when there is a hit in the second level cache. This enables to maintain the most recently used receiver type in the first level cache.

6.4 Performance Analysis

The three cache-based dynamic dispatch techniques were evaluated using the Java applications shown in Table 1. These programs can be considered to be a fair representation of typical Java programming style. The class files generated using the *Javac* compiler were executed using a simulated Java processor. The program counter value of the Java processor associated with the virtual method invocation serves as the call site location used for indexing into the hybrid cache.

The performance of the proposed dispatch schemes is influenced by two types of misses: (i) Misses due to receiver mismatch and (ii) Misses due to cache interference. Figure 15 shows the miss rates corresponding to the two types of misses for the various benchmarks. It was observed that the HPC performance better than the HC due to the substantial reduction in receiver misses. For *Jlex*, the receiver misses decrease from 4.52% to 0.0010%. Correspondingly, the average number of cycles required for dispatch reduces from 9.7 to 5.66. Further, it was observed that the THC performs better than the HPC scheme. While the receiver misses increase in the THC when compared with HPC, the misses due to cache interference reduces significantly for most of the benchmarks. For *Javac*, the receiver misses increase from 1.9% for a HPC to 2.4% for THC. The misses due to cache interference on the other hand reduce from 4.6% to 3.8%.

Figure 16(a) shows the effect of varying HC size and its associativity on the hit rates at the call sites. The figure shows that the hit rates increase with cache size due to the decrease in cache interference. Also, it can be observed that the 2-way associative HC performs better than a direct mapped HC with same number of words. Hit rates range from 86.3% to 96.3% for the various configurations shown in the figure. It was observed that HPC performs similar to HC for variation in associativity and cache size. For Javac, the hit rate improves from 89.8% for a 256 entry direct mapped HPC to 97.4% for an 1K 2-way associative HPC.

Figure 16(b) shows the comparative hit rates for the HC and HPC. Due to the extra receiver type and method location present in the HPC, the HPC should be compared with HC with twice the number of entries present in HPC. It can be observed that the 512 entry 2-way associative HPC has a hit rate of 96.3% for *javac* compared to the hit rate of 93.4% for an 1K entry 2-way associative HC. Thus, HPC was found to have higher hit rates than a HC with equivalent overhead.

Since dispatch speed is a better measure of performance, the average number of cycles required per dispatch was compared. Five cycles are required by the Java processor on a cache hit to perform the dynamic dispatch. It requires two cycles to obtain the current object's class, one cycle to obtain the call site entry from the cache, one cycle to perform the tag and receiver type comparisons, and another cycle to branch to the method location. The miss penalty involves the DTS search. DTS requires 10 cycles for checking each method within a class and

requires 3 cycles for moving to the super class. Thus, the miss penalty would vary based on the number of methods and levels searched before DTS finds the appropriate method. The average number of cycles required for performing the dynamic dispatch using a direct mapped HC and HPC is shown in Figure 17(a). It can be observed that the HPC has a better dispatch speed than HC.

Figure 17(b) shows the average number of cycles required per dispatch for various configurations of the HPC. It can be observed that dispatch speed improves with associativity and cache size. The average number of cycles is a function of both hit rate and the penalty associated with DTS. It was observed that the 2-way associative 1K HPC for the *Javac* and *Espresso* have almost the same hit rates. However, *Javac* requires more number of cycles for effecting a dispatch (9.9 cycles) compared to *Espresso* (6.6 cycles). The deeper hierarchy and more number of methods associated with *Javac* increases the number of cycles required for DTS.

Figure 18(a) shows the effective hit rate using the two-level scheme with a first level 256 entry two-way associative hybrid cache. It can be observed that the increase in associativity for the second level cache does not always improve the performance. For *Javadoc*, the hit rate for a 64 entry direct mapped second level cache is 96.7% whereas it is 96.1% for a 64 entry 2-way associative cache. This is due to the small size of the second-level cache where the misses are mainly due to capacity misses. The capacity misses increase with increase in associativity and hence the observed behavior. Further, it can be observed that the variation in the second-level cache size has no impact on the hit rates of *JLex* due to the very high hit rates for all configurations.

Figure 18(b) compares the dispatch speed of the three schemes with equal overhead. It can be observed that the THC scheme performs the best. This is because a two-level cache can use the second level cache either to reduce cache interference at the first level or improve performance at polymorphic call sites.

7 Conclusions

Architectural support for object addressing, stack processing and method invocation in a Java processor was proposed and evaluated. A virtual address object cache that supports efficient manipulation and relocation of objects was presented. The proposed scheme reduces up to 1.95 cycles per object access compared to the serialized handle and object lookup scheme. Next, the extended folding optimization that combines the execution of a sequence of bytecodes was investigated. This eliminates redundant loads and stores operations that constitute up to 9% of the instructions. Also, three cache-based schemes: hybrid cache, hybrid polymorphic cache and two-level hybrid cache to implement virtual method invocations were presented. These schemes utilize the receiver type locality at call sites and eliminate the need for creating dispatch tables used in current JVM implementations.

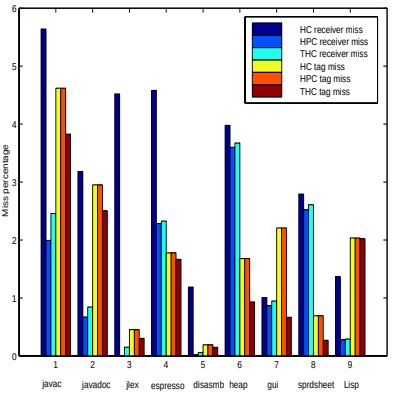

Fig. 15. Receiver and tag misses for HC, HPC and THC; HC and HPC use 256 entry two-way associative cache; THC has a 256 entry two-way associative first level cache and a 128 entry direct mapped second level cache

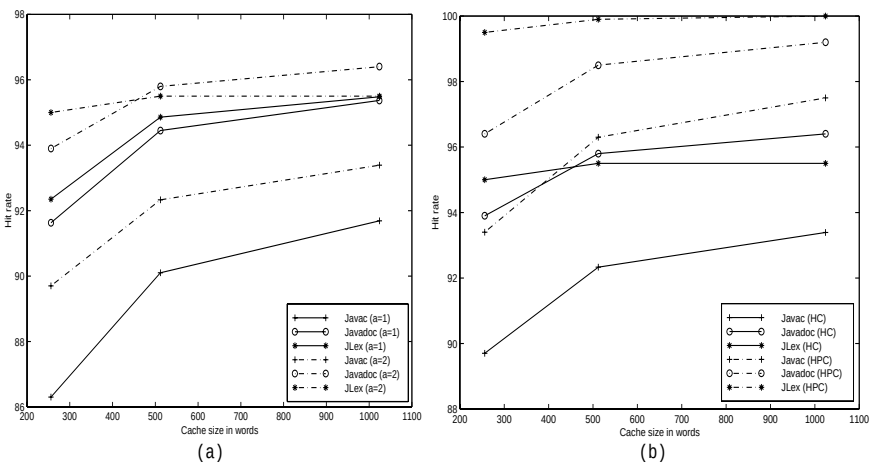

Fig. 16. (a) Variation in hit rate with cache size and associativity in a HC (b) Variation in hit rate for direct mapped HC and HPC

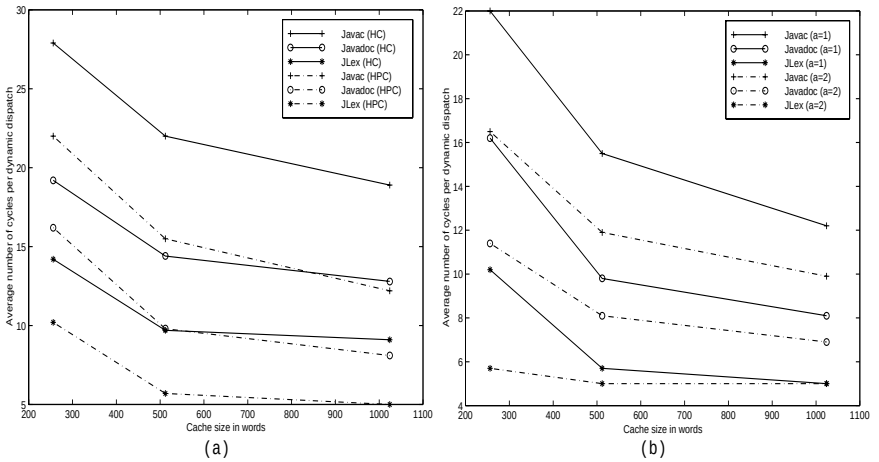

Fig. 17. (a) Average number of cycles per dynamic dispatch for direct mapped HC and HPC (b) Variation in average number of cycles per dynamic dispatch for HPC with associativity and cache size

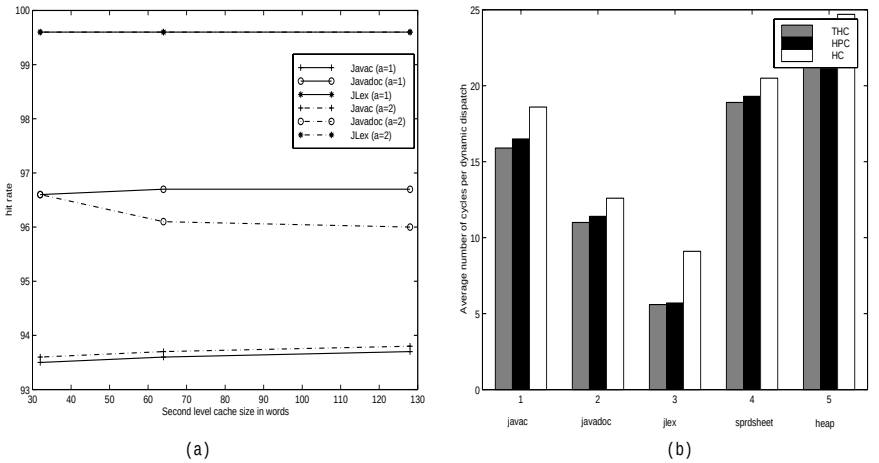

Fig. 18. (a) Variation in THC hit rates with change in second level cache size and associativity (b) Comparison of a 512 entry 2-way associative HC, 256 entry 2-way associative HPC and a 256 entry 2-way associative first level and 128 entry direct mapped second level THC

References

1. T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.
2. B. Calder, D. Grunwald, and B. Zorn, " Quantifying Behavioral Differences Between C and C++ Programs", Journal of Programming Languages, Vol. 2., Num 4, 1994. 336
3. J. Michael O'Connor, M. Tremblay, "picoJava-I: The Java Virtual Machine in Hardware", IEEE *Micro*, March/April 1997, pp. 45-53. 332, 332
4. Patriot Scientific Corporation, the PSC1000 Processor, www.ptsc.com/PSC1000. 332
5. K. Ebcioglu, E. Altman, E. Hokenek, "A JAVA ILP Machine Based on Fast Dynamic Compilation", International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel, January 9-10, 1997 332
6. K. Driesen, U. Hoelzle and J. Vitek, "Message Dispatch on Pipelined Processors", proc. ECOOP' 95, 1995. 346
7. WARTS: The Wisconsin Architectural Research Tools Set, University of Wisconsin, Madison. 334, 335
8. J. R. Jackson, Java by example, SunSoft Press, 1997. 341
9. N. P. Jouppi and S. J. E. Wilton, "An Enhanced Access and Cycle Time Model for On-Chip Caches", DEC- WRL Technical Report, 93.5, July 1994. 341
10. I. W. Williams, Object-Based Memory Architecture, PhD thesis, Department of Computer Science, University of Manchester, May 1989. 332, 338, 338
11. Open Group Research Institute Java Test Suite, http://www.gr.osf.org/java/testsuite/html/. 335
12. J. A. Dean, Whole-Program Optimization of Object-Oriented Languages, Ph.D Thesis, University of Washington, 1996. 345
13. C. A. Hsieh, et. al., "Optimizing NET Compilers for Improved Java Performance", Computer, June 1997, pp. 67-75. 330, 331
14. A. Krall and R. Grafl, "CACAO - A 64 bit JavaVM Just-in-Time Compiler", PPoPP '97 JAVA workshop. 331, 337, 346
15. Deutsch and Schiffman, " Efficient implementation of the Smalltalk-80 system", proc. 11th ACM Symposium on the Principles of Programming Languages, Jan 1984, pp. 297-302. 347
16. D. Ungar, The Design and Evaluation of a High-Performance Smalltalk System, MIT Press, Cambridge, MA, 1987. 332, 347
17. A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation, Second Edition, Addision-Wesley, reading, MA, 1985. 347
18. JavaSoft Home page, http:/www.javasoft.com/ 341, 341, 341, 341, 341
19. J. Vitek, "Compact Dispatch Tables for Dynamically Typed Programming Languages", Object Applications,ed. D. Tzichritzis, Univ. of Geneva, Centre Universitaire d'Informatique, Aug. 1996. 346
20. JLex, http://www.cs.princeton.edu/ appel/modern/java/JLex/. 341
21. EspressoGrinder, wwwipd.ira.uka.de/ espresso 341
22. Turbo J Compiler, http://www.osf.org/www/java/turbo/. 331
23. Asymetrix SuperCede, www.asymetrix.com/products/supercede/. 331
24. Toba: A Java-to-C Translator, www.cs.arizona.edu/sumatra/toba/. 331
25. Guava JIT compiler, guava.softway.com.au/index.html. 331
26. Symantec Cafe, www.symantec.com/cafe/index_product.html. 331
27. J2C, www.webcity.co.jp/info/andoh/java/j2c.html. 331

28. Jolt: Converting bytecode to C, www.blackdown.org/kbs/jolt.html. 331
29. G. Muller, B. Moura, F. Bellard and C. Consel, "Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code", proc. COOTS '97. 331
30. A. Arvindam, The design and behavioral model of an enhanced FORTH architecture, Masters Thesis, Univ. of S. Florida, 1995. 332
31. KAFFE, www.kaffe.org
32. U. Hoelzle, C. Chambers and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches", proc. ECOOP'91, July 1991. 349
33. "The Unprecedented Opportunity for Java Systems", Whitepaper 96-043, Sun Microelectronics. 331