USENIX Association

Proceedings of the 2nd Java^{TM} Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1-2, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

sEc: A Portable Interpreter Optimizing Technique for Embedded Java Virtual Machine

Venugopal K S
venuks@india.hp.com

Geetha Manjunath

Venkatesh Krishnan

Hewlett-Packard Laboratories, Palo-Alto

Abstract

This paper describes a radical approach to aggressively optimize an embedded Java virtual machine interpretation in a portable way. We call this technique Semantically Enriched Code (***sEc***). The sEc technique can improve the speed of a JVM by orders of magnitude. The sEc technique adapts an embedded Java virtual machine to the demands of a Java application by automatically generating an enhanced virtual machine for every application. The bytecode set of the virtual machine is augmented with new application-specific opcodes, enabling the application to achieve greater performance. Aggressive static or offline optimizations are done to ensure tight coupling between the Java application, Java virtual machine and the underlying hardware. sEc makes an embedded Java virtual machine become *a domain specific Java virtual machine* – a versatility not possible with the hardware.

KEY WORDS: embedded JVM, Java virtual machine, optimization, Interpreter, performance, semantically enriched code, JIT

1 Introduction

An entirely new breed of high-technology embedded products, such as Personal Digital Assistants and E-mail enabled cellular phones, have recently emerged. Manufacturers are deploying embedded Java runtime environments on these devices to enable portability and interoperability with software components. Although embedded processors are inferior in speed and code density, performance expectations are still high. This demand forces all the applications on the embedded environment, including the Java runtime environment, to be *squeezed for performance*. Conventional wisdom holds that the Java virtual machine in an embedded Java runtime environment is a bottleneck. Added to this, the rapid inclusion of new embedded appliances to the market demands rapid availability of *portable* and *efficient* embedded Java environments on these platforms.

A striking characteristic of an embedded appliance is its deployment for a *dedicated* or *mission-specific* purpose. This implies that an application that runs on an embedded environment is relatively static and does not vary as much as on a generic computing environment. This is an opportunity for optimizing the performance of an embedded Java environment for the application.

This paper describes a radical approach to aggressively optimize an embedded Java virtual machine interpretation in a portable way. We call this technique Semantically Enriched Code (***sEc***). The sEc technique can improve the speed of a JVM by orders of magnitude. The sEc technique adapts an embedded Java virtual machine to the demands of a Java application by automatically generating an enhanced virtual machine for every application. The instruction set of the virtual machine is augmented with new application-specific opcodes, enabling the application to achieve greater performance. Aggressive static or offline optimizations are done to ensure tight coupling between the Java application, Java virtual machine, and the underlying hardware. The sEc technique makes an embedded Java virtual machine become *a domain- specific Java virtual machine*.

A goal of the sEc technique is have neither a runtime optimization overhead (as with *just-in-time compilation* (JIT) or *dynamic code generation)*, nor to perform exhaustive optimization by ahead of time compilation, thereby losing the dynamic loading capabilities of an application. Our technique provides an intermediary solution. To summarize, the sEc technique makes an embedded Java virtual machine become *a domain specific Java virtual machine* – a versatility that a virtual machine enjoys over hardware processors.

Java[5], JVM[4] and JavaSoft are all trademarks of Sun Microsystems. In the rest of the report, the words JVM, application, Java runtime environment, unless specifically stated otherwise, are assumed to be an embedded JVM, embedded application, or embedded Java runtime environment respectively. Section 2 gives the genesis of the technique and section 3 gives an overview. The full technique is detailed in section 4 followed by implementation and results in section 5. Related work is presented in section 6 followed by conclusion, section 7.

2 The Genesis of sEc:

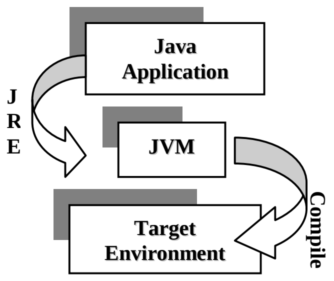


Figure 1: Java Embedded App

The sEc technique is motivated by the salient characteristics exhibited by the three building blocks of a Java application environment: the Java application, the JVM , the target environment, and the two couplings that exist between them as show in the figure-1. We now enumerate the characteristics of these three building blocks.

Java Applications: First, due to their object-oriented nature, applications are method-call intensive. Second, garbage collection frees the program from memory related problems like pointer chasing and dangling pointers. Third, the dynamic behavior of the applications can be characterized by the 80-20 rule of thumb, i.e. 80 percent of the execution time is spent in 20 percent of the application.

Embedded Java Virtual Machine: First, the JVM is a stack-based virtual machine (VM), where the stack is emulated using a heap. The JVM operations are dominated by stack operations. Second, the JVM byte codes have high semantic content compared to the target machine instructions. Third, the JVM is executed on a real machine and the byte code of the application is interpreted and spends 80% of its execution time in the *interpreter* loop. Fourth, the JVM provides the semantics of *dynamic loading* of classes as language feature. This is manifested in the *late binding* model for runtime entities.

Embedded target environment: First, recent embedded processors are register based RISC or CISC machines. Second, an embedded environment is created for a dedicated purpose, which implies applications are relatively static. Third, to keep pace with a fast product cycle environment, it must be both portable or retargetable and efficient.

At the application execution time the above-mentioned traits give rise to two couplings: the *Java runtime environment* (JRE) and *compilation* couplings, as shown by arrows in figure 1. These couplings have the following positive and negative characteristics.

2.1 ***JRE coupling***: This coupling exists between a Java application and the JVM at interpretation time and hence is dynamic in nature. The previously mentioned characteristics of a Java application are manifested by *Pointer-intensive and Call- intensive* nature in this coupling. Indirection of pointers has become the fundamental unifying model of all Object-Oriented runtime environments to support polymorphism and runtime type-check systems [13][14][1], e.g. the dispatch table mechanism used in the implementation of the "invokevirtual" opcode. JVM sub-modules, namely garbage collection and object management also make this coupling pointer- intensive. On the other hand the high semantic content of JVM bytecode causes the implementation of their JVM interpreter action to introduces, on average, one or more function calls per opcode. This makes the JRE coupling exhibit *call-intensive* characteristics.

2.2 ***The Compilation coupling***: This coupling is between the JVM and the target environment and is created during compilation of the JVM source and therefore it is static. Treating a JVM as just another program by the compiler has several disadvantages. First, the JVM features mentioned above introduce imprecise information to the compiler and thereby greatly hamper the optimizations by the compiler. The pointer-intensive characteristics of the JVM source induce data dependencies, leading to imprecision in the compiler analysis required to perform optimization transformations on the JVM code. Second, the pointer-intensive and call-intensive nature of the JVM also introduces control dependencies and hampers inter-procedural analysis. Modern target machines are more efficient when jumping to a constant address than when indirectly fetching an address from a table, such as a dispatch-table, which stalls the instruction-issue and execution pipeline for several cycles[8]. Third, the compiler used to (cross) compile a JVM source is ignorant of the JVM high level semantic constructs and architecture, and therefore fails to exploit precise information (e.g. the semantics of the stack operations) to improve the efficacy of the optimization. Fourth, the late binding model of the JVM also hampers optimizations by deferring the binding to the runtime. For example, the precise information about the binding address of symbols and the branch targets will increase the efficacy of optimizations of the JVM interpreter action code.

The driving philosophies of the sEc technique are (1) to make the application drive the semantic content of the

JVM by creating new opcodes (sEc-opcodes), resulting in an adaptive opcode set for JVM -- creating a domain specific Java virtual machine, (2) to do offline aggressive optimization for speed of the frequent case or frequently executed traces, while exploiting information on runtime constants and (3) to make implementations of the stack-based JVM tightly coupled to the real target (register-based) machines.

3 The sEc solution – an overview

This section gives an overview of the sEc technique, with sections giving details. Functionally, the sEc technique has three major phases namely, (a) sEc detection, (b) sEc Code Generation and (c) sEc embedding, as seen in figure 2.

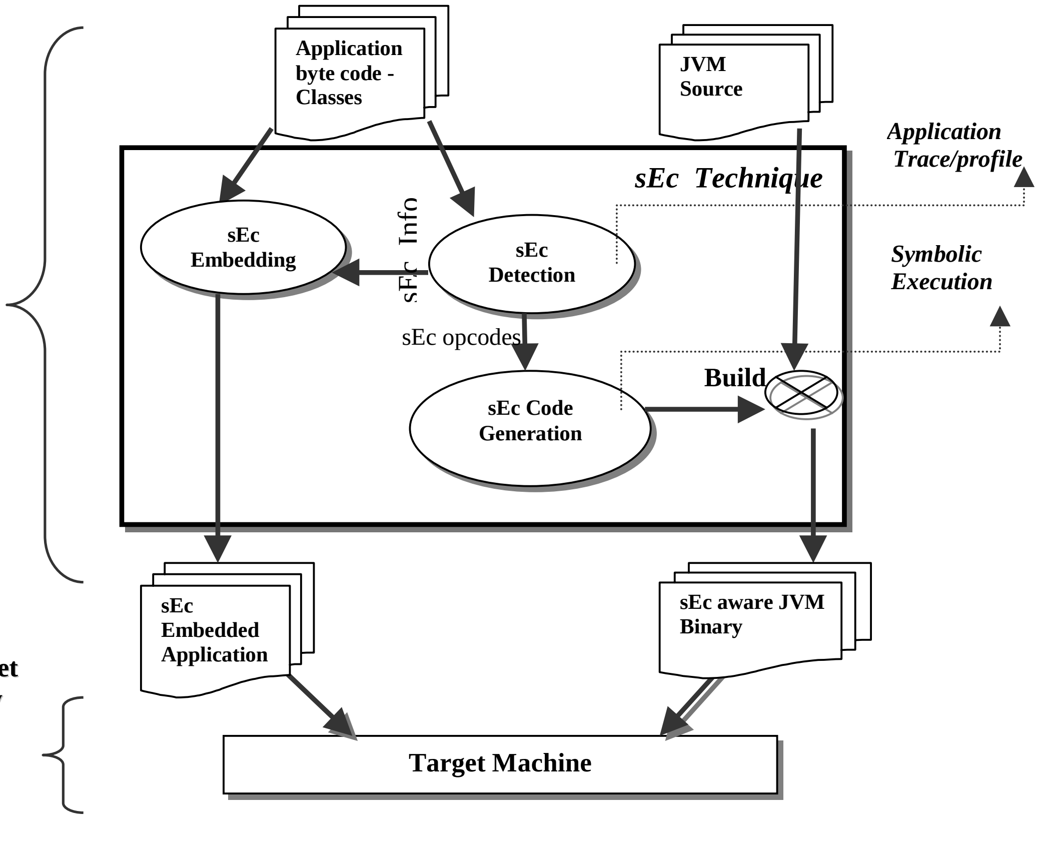


Figure 2: Three phases of the sEc Technique.

(a) *sEc detection phase*: The core of the JVM is the interpreter module, which interprets the bytecodes of the JVM. Interpretation is based on the fundamental pattern of *fetch-decode-execute* and *loop back*. This phase deduces the sEc-opcodes - new JVM bytecodes - from the execution semantics of the Java application. The new sEc-opcodes are synthesized from the stream of an application bytecodes using the application trace / profile or the probabilistic expectations of the execution. In our experiments, we use application profile information.

(b) *sEc code generation*: This phase takes the sEc-opcodes as input and generates efficient 'C' native code, which is smaller and faster than the equivalent JVM 'C' action code for the bytcode sequence represented by the sEc-opcode. This phase not only eliminates the per-opcode overhead of interpreting the sEc-opcodes - fetching and decoding each bytecode - but it also optimizes based on the JVM semantic content of the sEc-opcode. A novel technique called *sEc Symbolic Execution* is devised for sEc-opcode code generation. The generated code will be the JVM interpreter action for the sEc-opcode.

(c) *The sEc Embedding* makes the generic JVM aware of the synthesized sEc-opcodes and embeds the sEc-opcode appropriately in the place of the (Java) bytecode sequence in the application. This can be done either dynamically or statically, at runtime or offline, respectively. Finally, the sEc embedded application is executed using the *sEc-aware JVM* on the target machine for faster execution of the application.

4 The sEc Technique

This section details the design alternatives and phases of the sEc technique. An *sEc-opcode* is defined as a flow-sensitive maximal sequence of computation of Java bytecodes with a candidate based on execution speed using the properties of a dynamic execution. In this definition the basic block (BB), extended basic block and fragment (similar to the BB but with backward branches allowed) in the application trace are candidates for an sEc-opcode. Further more, the definition of equality of sEc-opcodes is based on the equality of the corresponding bytecode sequence. sEc-opcode equality is of the following types:

A. Exact match equality – The opcodes of the two bytecode sequences match and corresponding bytecodes have corresponding matching attributes.
B. Template match - This is similar to the exact match equality above but matching corresponding opcodes alone will be sufficient for a match.

4.1 *sEc Detection*

This phase deduces effective sEc-opcodes that will speedup the Java runtime of a given application from the stream of the Java bytecodes. The sEc detection can further be classified as (a) static sEc detection or (b) dynamic sEc detection.

In static sEc detection the given application is parsed for the most repetitive longest sequence of Java bytecodes, and sEc-opcodes are selected from these based on cost and control flow criteria. This method fails to capture the dynamic semantics of the application, as these patterns may not account for the dominant dynamic behavior of the application. This alternative is better suited for the bytecode compression of the application than for sEc-opcodes synthesis.

Dynamic sEc detection uses the profile of an application generated using representative input as the *best approximation* of the dynamic semantics of the application. Finding the optimal sequences of bytecode to select as an sEc-opcode with respect to speed and space criteria is combinatorially difficult [12]. Hence, the following heuristics based on greedy and non-greedy approaches are used.

A. Greedily select the bytecode sequence which captures the longest repetitive computational sequence of the dynamic bytecodes stream. The disadvantage of this method is that it is insensitive to control flow into the sequence. The overhead of guaranteeing correctness in sequences with multiple branch-targets in sEc-opcode, and Java's precise exceptions cut into the anticipated gain.

B. Non-greedy bytecode sequence selection will deduce the sEc-opcode bytecode sequence under structural constraints like control and data flow. The structural constraints could be the basic block, extended basic block or fragment. These constraints improve the efficacy of sEc-opcode optimization compared to multiple branch targets in the sEc-opcode.

4.2 *sEc optimization and code generation*:

This phase maps the bytecode sequence in the sEc-opcode onto optimized portable native 'C' code for the target machine. This code is the JVM interpreter action code for the sEc-opcode. A unique technique called *sEc Symbolic execution* – an integrated optimizer and code generator - optimizes the sEc-opcode with respect to the JVM semantic domain as well as the target architecture semantic domain, and generates the JVM action code in 'C'. This optimization is effective not only because bytecode dispatch overhead is eliminated for the bytecode in the sEc-opcode, but also because stack operands are folded, redundant local variable accesses are eliminated, and more precise information is available for the offline 'C' compiler optimizer. The resulting 'C' code undergoes further optimization – optimization with respect to the target architecture semantics - by a global optimizing compiler like GCC[10]. This yields efficient sEc-opcode execution resulting in higher coupling of a JVM to the underlying target machine semantics in the resulting sEc-aware JVM.

4.2.1 *sEc-opcode optimization*:

Since the stack based JVM opcodes are emulated over the register-based target machine instructions, the sEc technique allows the optimization of sEc-opcode as listed below.

1. ***Virtual Machine architecture dependent optimizations*:** These optimization techniques are dependent on the virtual machine architecture and its emulation aspects. In summary these optimizations provide efficient storage and access for the sEc-opcode operands and local variables.

- *Java stack access operands are subsumed:* Within the context of an sEc-opcode, stack operand access is intrinsically subsumed by the efficient 'C' code, eliminating store and load operations on the Java stack frame.
- *Elimination of redundant Java Local Variable Accesses*: Accesses to the redundant local variable on the Java frame are eliminated by reusing the value in the generated 'C' code. This optimization keeps track of *read-after-read* data dependencies with respect to the JVM architecture across the Java bytecodes that are within the sEc-opcode, and propagates the value.
- *Elimination of JVM operand stack manipulation operations*: Our studies have shown that using dynamic instruction distributions, 40% of the instructions are related to moving the data between the Java operand stack and local variables, duplicating values on the stack, and constants. For example, consider the bytecodes pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x1, dup_x2, and swap. This technique keeps track of the stack state in the sEc-opcode and propagates the value to the target operation, thus eliminating the respective bytecode action or the need to generate 'C' code to perform the operation.
- *Java bytecode is semantically rich*: Within an sEc-opcode, the Java bytecode semantics can be treated as composed of fine-grained sub-operations or predicates. These can be the class, method and field attribute checks. For example, isNativeMethod (method) is a predicate in the bytcode "invokestatic". Runtime checks like exception checks (null value check, array boundary check etc. are also handled). This optimization eliminates these redundant sub-operations within the sEc-opcode. We note here that a common sub-expression in the sEc-opcode is not necessarily possible so in the underlying compiler for the target machine (i.e. pointer aliasing problems arise because of emulating the JVM stack using the heap memory)

2. ***Virtual Machine architecture independent optimizations***: These optimizations are independent of the VM architecture and its implementation, but are limited to the semantic domain of the virtual machine architecture. Some examples of these are common sub-expression elimination within the bytecode sequence of the sEc-opcode, deducing *polymorphic* call points as *monomorphic* call points (method-pointers are compile time constants) and so on.

3. ***Virtual Machine Runtime Bindings***: The *late binding* or *dynamic loading* feature of the Java VM gives rise to new kinds of optimization opportunities based on runtime constants after the late binding process within an sEc-opcode. We term this novel optimization technique ***sEc-rewriting.*** sEc-rewriting is the dynamic self-redirection of the sEc-opcode to an efficient runtime implementation based on runtime constants and bindings. In case of the JVM, the real machine address bound to symbolic information, (e.g. field or branch offset), is constant after it is resolved, or bound, at runtime. Similarly, some predicate or attribute checks outcomes are constant once resolved. The sEc-opcode is aggressively optimized offline for this specialized runtime variant. During sEc-opcode interpretation, the call-point in the method code is rewritten to jump to the specialized implementation in the very first interpretation, taking runtime values as parameters. Every sEc-opcode potentially has a specialized sEc-rewriting opcode variant, and the virtual machine developer has the option of fine-grained control to select them.

4. ***Target architecture dependent and independent optimizations***: These optimizations are in the purview of the underlying real machine semantics or architecture[7]. In the case of RISC architectures, optimizations dependent on the register architecture, like instruction selection, instruction scheduling, register allocation and register assignment are target architecture dependent optimizations. On the other hand, optimizations independent of the register based architecture but constrained by the semantics of the underlying register architecture are termed target architecture independent optimizations. Common sub-expression elimination, copy propagation, and loop invariant code motion, are but a few from the cornucopia of possible optimizations[7]. The sEc technique depends on a global optimizing compiler like GCC to do the optimizations under this category, but provides more precise information to aid the global optimizations.

4.2.2 *sEc Code Generation*

The sEc code generator's aim is to generate efficient retargetable 'C' code for the sEc-opcodes. Clearly, a naive sEc code generator could just concatenate the corresponding JVM interpreter action code of all the individual bytecodes in the given sEc-opcode. This could be visualized *as interpreter switch unrolling* for the respective sEc-opcode, which only eliminates the per-bytecode dispatch overhead. However our code generation technique - sEc Symbolic execution - exploits the optimization techniques explained above. The sEc code generation has to abide by structural constraints that the Java bytecode guarantees, namely, (a) *Stack Invariance of Java Bytecode*: each bytecode must only be executed with the appropriate type and number of arguments on the operand stack or in local variables, regardless of the execution path that leads to its invocation, and (b) *Variant data type of Java stack frame*: At different execution points of the same method, the local variable slot in a Java invocation frame can hold different data types.

1. 'C' local variable allocation of the Java operand and local variable[s] under the sEc code generation constraints enumerated earlier.
2. tracking the state of the JVM to eliminate or subsume sub-actions of the bytecode.
3. ordering the sub-operations like type check, null check etc.
4. ensuring the state of JVM is consistent when control flows into and out of sEc-opcodes.

sE-opcode sequence	***Symbol created***	***sEc Symbolic state*** $\longrightarrow$	***sEc Local Var Table***					***Code Generated in 'C'***
			1	**2**	**3**	**4**	**5**	
Prologue		//well formed sEc-opcode, no code generated						// No code generated
ALOAD_1	O1	O1	O1					O1 = JLV(1)
ILOAD_3	O2	O1,O2			O2			O2 = JLV(3)
IALOAD	O3	O3						O3 = Macr(O1, O2)
ALOAD_2	O4	O3,O4		O4				O4 = JLV(2)
ILOAD_3	O5	O3,O4,O5			O5			O5 = O2
IALOAD	O6	O3,O6						O6 = Macr(O4, O5)
IMUL	O7	O7						O7 = O3 * O6
ILOAD 5	O8	O7,O8					O8	O8 = f(JLV(1))
IADD	O9	O9						O9 = f(O7,O8))
ISTORE 5							O9^D	// No code generated
IINC 3 1					O5^D			O5^D = f(O5,1)
Epilogue		//No stack update //consolidated JVM pc update						JLV(3) = O5 JLV(5) = O9 PC = PC + 15

JVM_word *Ptr_lvp = Jvm_stack_base + lvp // Pointer to current Java invocation Frame //	
#define JLV(x) *(Ptr_lvp + x)	***// Reference to Java Local Variable on a Java Frame***
Macr(O1, O2)	***// Macro to fetch O2th element for O1 array object*** ***// O1 and O2 are macro parameter.***

Figure 3: Illustration of sEc Symbolic Execution on the sEc-opcode

4.2.3 *sEc Symbolic Execution*

This section defines some terms and explains the code generation constraints and sEc symbolic execution in detail. During the symbolic execution, the bytecodes logically making up the sEc-opcode sequence are symbolically *interpreted* for the purpose of integrated code generation across the sEc-opcode and optimization within the sEc-opcode. This process has knowledge of the JVM stack as well as the target machine architecture, and therefore yields better optimization results. This results in faster execution of the sEc-opcode when interpreted by the sEc-aware Java virtual machine. The following are the essential issues handled in the process:

4.2.4 ***An example of the sEc Symbolic Execution***

Consider the bytecode of the source statement, ***sum = a[i] * b[i] + sum,*** represented as a sEc-opcode. The corresponding sequence of bytecodes is ***[ALOAD_1, ILOAD_3,IALOAD, ALOAD_2, ILOAD_3, IALOAD, IMUL, ILOAD #5 IADD, ISTORE #5, IINC 3 1].*** The snap shot of the trace of the sEc symbolic execution is shown in figure 3. In this particular sEc-opcode, there is no 'C' code generated for epilogue because the sEc opcode has all the stack operands of the bytecode within the sEc-opcode (well-formed sEc-opcode). We note the following points about the example:

1. The suffix of O's is maintained by the sEc code generator. The string-symbol is generated and pushed on to the symbol stack (Column 3).

2. Although there is a scope to replace O3 by O1 for the "iaload" bytecode, the translator generates the new symbol O3, since replacing O3 by O1 will not add to the ability of the code generator to perform optimizations. However, the GCC compiler easily optimizes the code by eliminating O3.

3. The JLV(n) is a macro which accesses the nth Java local variable from the current Java frame.

4. For certain bytecodes in the sEc-opcode, a new symbol is created with an appropriate suffix and pushed on to the symbolic stack, as shown in figure-3.

5. Every local variable load and store is tracked for redundant usage by using the symbolic local variable table, as shown in the figure. A write to a local variable is tracked using the dirty status in the local variable status.

6. Redundant access to the Java local variable 3 is subsumed by reusing the symbol O2

7. The bytecode "Istore 5" does not cause any code to be generated because it is subsumed by the symbolic transformation in the symbolic local variable table.

8. At end of the sEc-opcode – the epilogue – the JVM runtime locals that are dirty will be written back.

4.3 *The sEc Embedding*

The sEc embedding is a process that embeds the sEc-opcode into the Java application and modifies the generic JVM with the new sEc-opcode interpreter action. This phase is divided into 2 parts (a.) modifying the JVM and (b.) embedding the sEc-opcode into the Java application. The sEc embedding can be performed offline or online.

4.3.1 The JVM modification:

The JVM interpreter loop is modified to detect and execute the new sEc-opcode. In the offline model, modifications related to the JVM source (mainly the interpreter) is done in the host environment of the embedded target and cross built to get the sEc aware JVM. In the online model, a generic JVM is modified to have a stub, whose function is to automatically load the new sEc-opcode when the main interpreter loop traps for the new sEc-opcode. The disadvantage of the later method is the need to dynamically load of module in the runtime environment.

4.3.2 ***Embedding the sEc-opcode into the Java application:***

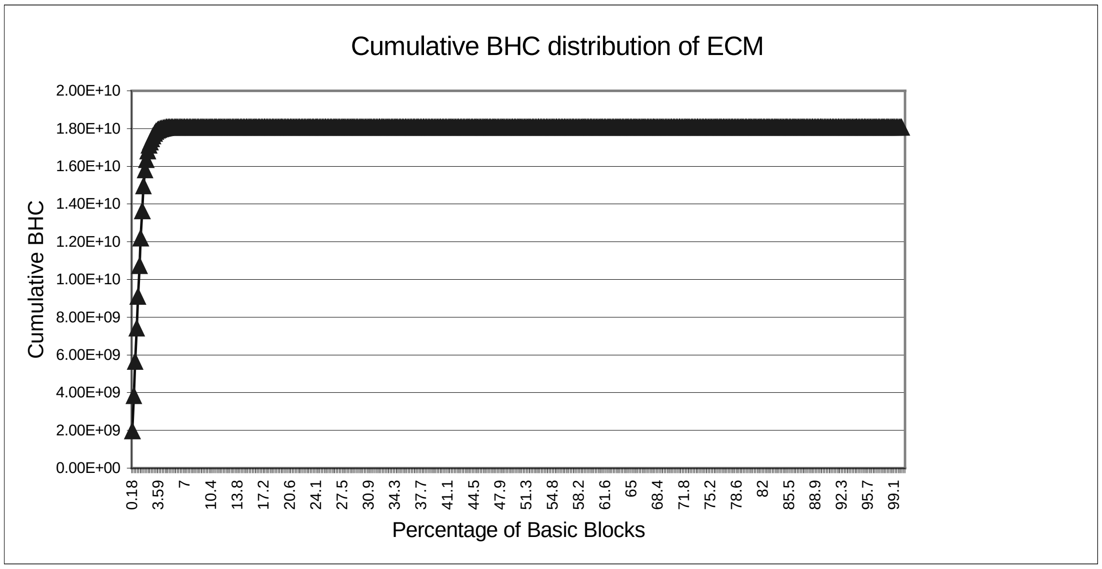
The new sEc-opcode is embedded into the given Java application by modifying the method *code attributes*. This process can be performed either offline or online. The sEc detection phase gives the *sEc-hook* information, which has the location of the new sEc-opcode – class, method and offset - and the size of the replaced bytecode sequence. In the offline model, all of the class methods are rewritten with the new sEc-opcode by bytecode rewriting tools [16]. In the online model, the Java bytecode sequence of the new sEc-opcode is replaced at the runtime of the application using a *special class loader*. The special class loader will track the first call of methods to be sEc-opcode embedded – using sEc-hook information – and the method code attributes of the method are rewritten with the corresponding sEc-opcode. Thereafter execution resumes the normal path of interpretation. The sEc-hook will help the special class loader to pinpoint the method, location and size of the bytecode stream to be rewritten with the corresponding sEc-opcode.

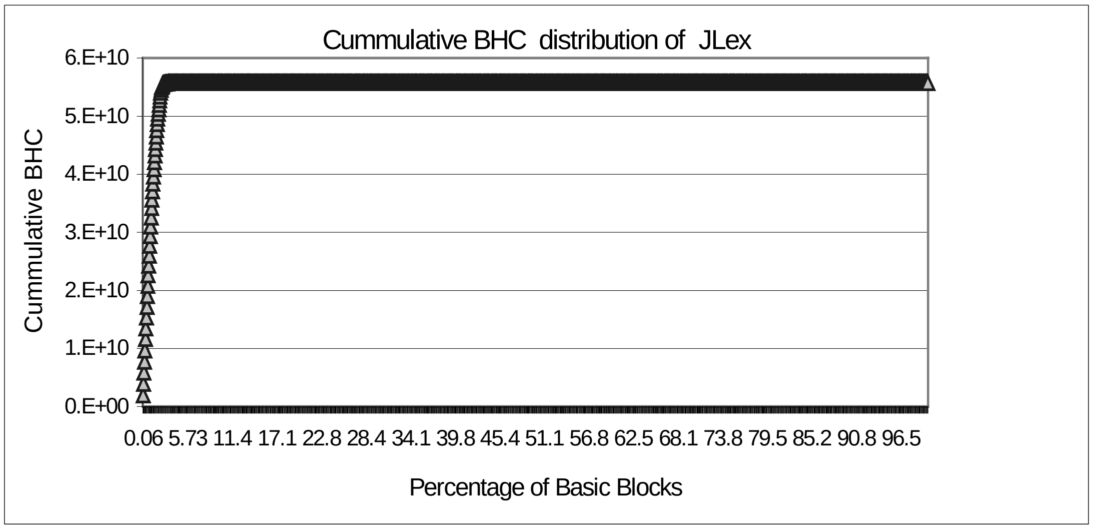
SEC_hook **[Dot, loopit, ()V < offset:31, NUM_INS:9, SIZE:12>**	**SEC_hook** **[Dot, loopit, ()V <offset:54, NUM_INS:11, SIZE:15>**
Begin Basic Block ALOAD_1 ILOAD_3 ALOAD_2 ILOAD_3 BIPUSH DUP_X2 IASTORE IASTORE IINC ***End Basic Block***	***Begin Basic Block*** ALOAD_1 ILOAD_3 IALOAD ALOAD_2 ILOAD_3 IALOAD IMUL ILOAD IADD ISTORE IINC ***End Basic Block***
[1800000]: ***Bytecode Hit Count*** **sEc opcode: sEcopcode_235**	***[2200000]: Bytecode Hit Coun*** **sEc opcode: sEcopcode_236**

Figure 4: Bytecode sequence of the sEc-opcodes.

5 Implementation and results

This section gives some preliminary results of the sEc technique. An in-house research JVM [1] was used for the sEc technique and the instrumentation. The host environment was HP-UX 10.20 and the embedded target environment is an NS486 based custom embedded board. The GCC compiler was used for cross compilation.





Dynamic sEc detection with non-greedy heuristics and the BB structural constraint - bytecode patterns are limited to basic blocks - was applied to the benchmarks below. Further selection of the bytecode sequence for the sEc-opcode was based on the *BHC* metric (Bytecode Hit Count) a product of the execution frequency of the BB and the number of the bytecode in the BB. The JVM is instrumented to obtain the BHC and sEc-hook information for every BB.

Fine grained space and speed tradeoff of the JVM:
The Benchmarks ECM (Embedded Caffine Mark) and Jlex (Java lexical analyzer) were used to study the effect of the sEc-opcode on the dynamic semantics of applications and their impact on the speed and space of the applications. Measurements of these are given below. The cumulative BHC chart of ECM shows 6% of BBs accounts for 99.97% of the total BHC for ECM. This amounts to 34 BBs and the probable sEc-opcodes for the ECM application. Similarly, 2.5% of BBs cover 98% of the BHC in Jlex. This amounts to 40 BBs and these are the most probable candidates for sEc-opcodes. A similar graph of JVM code size for every addition of a sEcopcode can be done. This gives the embedded JVM developer the flexibility to do a fine-grained quantitative tradeoff between the application speed and the target space constraints. We note that the number of extended BBs and the fragments that account for the most execution time will be less than the number of BBs using the basic-block structural criteria.

JVM Speedup – Some performance results*:*
The dot product benchmark was used for the study of the speedup of the JVM. The application was subjected to sEc-detection as detailed earlier. We only considered the two basic block bytecode sequences shown in figure

4 as sEc-opcodes, namely sEc-opcode_235 and sEc-opcode_236 – these sequences had the highest BHC. The sEc-opcode_235 and sEc-opcode_236 are subjected to the sEc code generation algorithm - sEc symbolic execution – performed manually with the JVM dependent optimizations. The resulting 'C' code was used to augment the generic JVM.

The JVM was modified offline to be aware of the new sEc-opcodes. The action 'C' code, was embedded into the JVM interpreter loop and JVM specific changes were made to recognize the new sEc-opcodes. Then, the JVM was built using the GCC compiler with highest level of optimization enabled.

Online sEc-opcode embedding was adopted to replace the sequence of bytecode comprising the sEc-opcode with the sEc-opcode proper. A new class loader was introduced into the JVM to read the sEc-hook information and track all of the loaded classes for embedding the sEc-opcodes at the sEc-hook specified locations. This modification resulted in a speedup of *300%*. We note that only 2 sEc-opcodes were considered for experimentation purposes. Also the bytecode sequences in the sEc-opcodes are less rich in semantics compared to semantically rich opcodes like those for object management and method invocation etc (which would enable more scope to optimize).

6 Related Work

Optimizing a Java application for speed has become an active research area. Broadly, this research can be classified as follows:

Interpreter Techniques: Bringing traditional compiler techniques to the runtime environment will not be a viable solution for resource constrained embedded Java deployment. Studies have been done to improve interpreter techniques as in [19][20]. All of these approaches exploit a variation of threading to improve the per-opcode overhead by removing the opcode dispatch overhead. The sEc technique is a portable, static-optimization interpreter technique. To the best of our knowledge, the sEc technique is the first JVM interpreter optimizing technique of its kind. Compared to the above interpreter techniques, the sEc technique not only removes the per-opcode overhead in the sEc-opcode but also does aggressive optimization in the sEc code generation phase. Unique to the sEc technique, it divides the sEc-opcode optimizations between Java virtual machine dependent and independent optimizations, and JVM runtime binding optimizations and the target machine dependent and independent optimizations. These phases optimize the sEc-opcode by knowing the semantics of the JVM bytecode in the sEc-opcode. They also improve the efficacy of state-of-the-art target optimizations by feeding precise information to the optimizer, resulting in an efficient coupling of the sEc-opcode to the target machine. Along with these optimizations, the sEc-rewriting technique can switch the sEc-opcode to a more efficient implementation after runtime binding – exploiting runtime binding constants.

BrouHaHa[19], Objective Caml and Interpretation of C[12] are some implementations that make use of "macro" opcodes similar in concept to sEc-opcodes The sEc approach differs from these in the following ways:

(1) The sEc-opcode is produced offline by taking into account the dynamic characteristic, control flow and data flow of the embedded Java application.
(2) The cost of dispatch for a RISC-like opcode set is relatively greater (compared to the cost of executing the bytecode) than it is for Java bytecode. On the other hand the Java bytecode is semantically rich – most of the bytecodes can be decomposed into sub-operations and optimized. This demands a complex analysis to optimize sEc-opcodes and is done offline instead of at runtime.
(3) The sEc-opcode optimization exploits runtime binding constants as discussed in the sEc-rewriting section.

Superoperators[12] for ANSI C interpreters are a technique for specializing a bytecoded C interpreter according to the program that it is to execute. Superoperators make use of an lcc tree-based IR to synthesize superoperators. The sEc-opcode differs with respect to superoperators as follows. The semantic content of the lcc IR is very much the same as real machine instructions [9] – it consists of expression trees over a simple 109-operator language – hence folding of instruction using tree pattern[11] matching works every well. On the other hand, lcc's tree structure limits the effectiveness of this system. Putting it in another way, the lcc IR is the sequence of trees *at the semantic level of the target machine*. In contrast, the Java bytecodes are semantically rich– each bytecode is composed of many sub-operations - because of the high level of abstraction. The larger the sEc-opcode context, the more JVM dependent and independent optimization can be done – e.g. elimination of stack operands, redundant local variable access, elimination of redundant sub-operation in sEc-code bytecode sequence like method attribute check, null value check etc. After the JVM related optimizations, target machine optimizations are done by a compiler like GCC (-o). In fact gcc's RTL, (Register Transfer Language [10], the intermediate form used for most of the optimizations and for code generation in GCC can be considered as a counterpart of the lcc IR. The Superoperator technique is integrated into the lcc compilation and uses the lcc backend, which does not do advanced global optimization.

Just In Time compilation (JIT): The principle of the JIT compilation is, in general, to dynamically compile a method to native code – after some threshold number of calls to the method has occurred - pausing the application execution. Extensions and refinements of the JIT principle have spun-off many techniques under the following constraints:

1. Generating efficient native code: Optimized native code is generated on the fly by adapting traditional optimization techniques to run-time code generation.

2. Efficient code generation techniques: Optimize the JIT techniques themselves for size and speed, executing efficient code generation and register allocation algorithms. Some of recent studies in this area are Hotspot[6] CACAO[34][35] from DEC, Jalapeno[28] (now called the "Jikes RVM") from IBM, Annotated JVM[31] and Hybrid JIT [3]

Native Java Compilers: Unlike with JIT compilers Java source and binary (classes) are statically compiled to native code. This method does not have constraint 2 of JITs stated above, and hence better native code can be generated by employing state-of-art optimizations. Generally these methods disallow dynamic class loading. Toba[29], Harissa[30] and commercial products like TowerJ™and Cygnus (GNU) Java native compiler fall into this category.

7 Conclusions

Semantically enriching the Java bytecode using the sEc technique is an innovative JVM optimization technique. The sEc Technique avoids the problem of efficiently integrating traditional compilation phases into the Java runtime environment, which is a widespread problem for JIT techniques.

We have presented an abstract model for a portable way of determining the sEc-opcodes and an efficient code generation. The sEc-opcode optimization is classified into a 5-phase process, a JVM dependent, JVM independent, sEc-rewriting, target dependent and target independent. This model shows that a traditional compilation optimization problem exists with equal complexity in the JVM dependent and independent optimization phases of sEc-opcodes. The five phases of the optimization make the sEc-opcode implementation efficient by tightly coupling it to the target machine. This has shown that more than opcode dispatch optimization can be achieved by employing JVM independent and dependent optimizations, sEc-rewriting and the aggressive target machine related optimizations.

We have shown in our preliminary exploration of the implementation that the sEc technique can increase the speed of JVM interpretation by orders of magnitude for some embedded Java applications. The sEc technique employs 'C' as its intermediate language, and hence is portable to many embedded platforms, which reduces porting and retargeting cost and time. We believe that the speedup gain from the sEc technique in its aggressive form will be comparable to the JIT technique. We have also shown that the sEc technique provides fine-grained tradeoff of speed and space in an embedded JVM. More information about the sEc Technique can be found in the Hewlett-Packard laboratories Technical Report[2].

As next steps, we intend to evaluate quantitatively the different optimizations discussed in this paper. We also want to apply the sEc technique to dynamically loaded modules by '*probabilistic based sEc-opcode deduction*' for statically determinable dynamic classes. We are also interested in quantitatively evaluating the foot print overhead of sEc-technique.

8 Acknowledgments

Mr Sam Midkiff deserves a special thanks for helping us to improve the readability of this paper. We specially acknowledge Nagarajan K and Joseph Koshy for the technical discussions and encouragement we got from them. We also thank members of our research JVM team Devaraj Das and Venkata Ramani.

We thank Subrahmanyam V. S., Jeff Morgan and Ananthraman P. N. for their enduring support through out the execution of the project. Finally, we express our appreciation to the anonymous reviewers for their constructive comments and suggestions.

9 References

[1] *Coorg: Design of a Virtual Machine for Java on Embedded Systems, Hewlett-Packard Laboratories – Technical Report* - HPL-2001-68

[2] *sEc: An Interpreter Optimization technique for Embedded Java Virtual Machine, Hewlett-*

Packard Laboratories Technical Report - HPL-2001-69

[3] *A Hybrid Just*-In-Time Compiler That Consumes Minimal Resources, Geetha Manjunath, Hewlett-Packard, US patent number 6332216, issued on 18-Dec-01.

[4] Java ^{TM} Virtual Machine Specification, Tim Lindholm and Frank Yellin, *The Java Series, Addison Wesley Publications, ISBN 0-201-63452-X.*

[5] The Java Language Specification, James Gosling, Bill Joy and Guy Steele, *Addison Wesley Publications.*

[6] The Java Hotspot™Performance Engine Architecture. *Javasoft*

[7] Compilers - Principles, Techniques, and Tools, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, - *Addison Wesley Publications, ISBN 0-201-10194-7*

[8] Computer Architecture A Quantitative Approach, David A. Patterson, John L. Hennesy – Morgan Kaufmann Publishers, Inc. ISBN 1-55860-069-8

[9] A Code Generation Interface for ANSI C, Christopher W. Fraser, David R. Hanson, *Software – Practice And Experience, Vol. 21(9), 963-988 (September 1991)*

[10] Using and Porting GNU CC, Richard M. Stallman, *Free Software Foundation, Cambridge, MA, 1990.*

[11] Code Generation Using Tree Matching and Dynamic Programming, A. V. Aho, M. Ganapathi, S. W. K. Tjiang –*ACM Transaction on Programming Languages and Systems, October 1989.*

[12] Optimizing an ANSI C Interpreter with Superoperators, Todd A. Proebsting, *Proc POPL'95 pages 322-332*

[13] The Annotated C++ Reference Manual, Marget A Ellis and Bjarne Stroustrup, *Addison Wesley Publications.*

[14] Inside the C++ Object Model, Stanley B. Lippman, *Addison-Wesley Publishing*.

[15] Simple and Effective Analysis of Statically-Typed Object-Oriented Programs, Amer Diwan, J. Eliot B. Moss, Kathryn S. McKinley, OOPSLA 96: *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages , and Appliccations*.

[16] Byte Code Engineering with the JavaClass API – Markus Dahm, Berlin, Technical Report B-17-98,

[17] JVM Subsetting for an Embedded Application, Devaraj Das, Geetha Manjunath, Internal Technical Report – HPLabs

[18] Optimizing direct threaded code by selective inlining, Ian Piumarta and Fabio Riccardi,, *ACM SIGPLAN 1998*

[19] BrouHaha – A Portable Smalltalk Interpreter - Eliot Miranda, Proc. OOPSLA 1987. *Published as SIGPLAN Notices 22(12):354-365.*

[20] A Portable Forth Engine, M. Anton Ertl, *Proc, euroForth 1993,*

[21] Compilers and Computer Architecture - William A. Wulf, Carnegie-Mellon University, IEEE Computer Journal, July 1981.

[22] A Tree-Based Alternative to Java Byte-Codes – Thomas Kistler and Michael Franz, *University of California, Irvine.*

[23] Abstract Interpretation : A Unified Lattice Model For Static Analysis Of Programs by Construction Or Approximation Of Fixpoints – Patrick Cousot and Radhia Cousot, Fourth ACM symposium on Principles of Programming Languages, 1977.

[24] Reducing garbage in Java – C. E. McDowell, http://www.cse.ucsc.edu/research/embedded/pubs/gc/index.html

[25] The Stack Allocation Optimization – Real Time Java discussion group, http://www.nist.gov/itl/div896/emaildir/rt-j/threads.html

[26] Garbage collection can be faster than stack allocation - Andrew W. Appel. *Information Processing Letters 25(4):275-279, 17 June 1987.*

[27] Optimal code generation for expression trees – A. V. Aho and S. C. Johson, *Journal of the ACM 23(3): 488 – 501, July 1976.*

[28] The Jalapeno Dynamic Optimizing Compiler for Java – Michael G. Burke, Vivek Sarkar, J. Cho, M. J. Serrano, S. Fink, V. C. Sreedhar, David Grove, Michael Hind, H. Srinivasan. – *JAVA'99 ACM.*

[29] Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler – Todd. A. Proebsting, J. H. Hartman, G. Townsend, Tim Newsham, Patrick Bridges, S. A. Watterson. – *The University of Arizona*

[30] Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code – Gilles Muller, Barbara Moura, Fabrice Bellard, Charles Consel – *IRIA / INRIA – University of Rennes.*

[31] An Annotation-aware Java Virtual Machine Implementation – Ana Azevedo, Alex Nicoloau, - University of California, Irvine Joe Hummel – University of Illinois, Chicago.

[32] DashO-Pro – preEmtive solutions http://www.preemptive.com

[33] A Comparison of TowerJ™ and HotSpot™ Implications for Enterprise Java Deployment[White paper] – Tower Technology Corporation – http://www.towerj.com/products/whitepapertjhs.shtml.

[34] Efficient Java VM Just-in-Time Compilation – Andreas Krall, PACT'98

[35] CACAO – A 64 bit JavaVM Just-in-Time Compiler – Andreas Krall and Reinhard Grafl.