Method Call Acceleration in Embedded Java Virtual Machines

M. Debbabi[1,2], M. Erhioui[2], L. Ketari[2], N. Tawbi[2], H. Yahyaoui[2], and S. Zhioua[2]

[1] Panasonic Information and Networking Technologies Laboratory
Princeton, NJ, USA
debbabim@research.panasonic.com
[2] Computer Science Department, Laval University, Quebec, Canada
{erhioui,lamia,tawbi,hamdi,zhioua}@ift.ulaval.ca

Abstract. Object oriented languages, in particular Java, use a frequent dynamic dispatch mechanism to search for the definition of an invoked method. A method could be defined in more than one class. The search for the appropriate method definition is performed dynamically. This induces an execution time overhead that is significant. Many static and dynamic techniques have been proposed to minimize the cost of such an overhead. Generally, these techniques are not adequate for embedded Java platforms with resource constraints because they require a relatively big memory space. The paper proposes a dynamic, flexible and efficient technique for accelerating method calls mechanism in embedded systems. This acceleration technique spans over 3 aspects of the method call: (1) lookup, (2) caching, and (3) synchronized methods.

1 Motivations

The main intent of this research is to accelerate Java-based embedded systems. Nowadays, there is growing interest and potential for embedded systems to make a contribution in scientific computing. For instance, a Java-based sensor that collects (often massive) data for scientific experiments needs to be accelerated for a shorter pre-processing time. This acceleration is highly relevant especially in the context of Java-based distributed systems supporting HPC activities. An increased performance will be perceived as both useful and interesting by members of the community.

In this paper, we focus on accelerating embedded Java virtual machines. More accurately, we put the emphasis on one particular aspect that is the method call. Method invocation is a crucial aspect in the design and implementation of the runtime of any object oriented language. This aspect is even more important when it comes to dynamically typed languages such as Java. In fact, Java offers a dynamic dispatch mechanism that allows to invoke the appropriate method depending on the runtime class of the receiving object. Usually, this process is inevitable because the actual invoked method can not be known statically. In fact, the method could be defined in more than one class. Hence, to compute the

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 750–759, 2003.

invoked method, a dynamic search (or a dynamic dispatch) procedure should be performed. This search procedure starts from the call receiver class. It spans over the hierarchy to know if a superclass of the call receiver class defines a method having the same signature as the statically computed method for this call. Unfortunately, method invocations involving dynamic dispatch are slower in an interpreter and incur an additional overhead which is significant. So the frequent use of this mechanism hurts badly the execution performance of Java applications. Many static and dynamic techniques have been proposed to minimize the cost of such an overhead. Generally, these techniques are not adequate for embedded Java platforms with resource constraints because they require an important memory space. Consequently, lightweight and tiny optimizations, with minimum space and time strategy, are required to accelerate Java calls in embedded virtual machines.

The remainder of this paper is organized as follows. Section 2 presents the Related Work. Section 3 outlines our approach. Section 4 presents the results. Finally, Sect. 5 concludes the paper.

2 Related Work

Object-oriented languages support inheritance and polymorphism to allow the development of flexible and reusable software. The type of a specific object would usually be determined at run time. This feature is called the dynamic binding. In this context, the selection of the appropriate method to execute is based on a lookup mechanism which means that the actual method to be executed after an invocation is determined dynamically based on the type of the method's receiver, the class hierarchy and the method inheritance or overloading schema. The lookup mechanism consists of determining the actual method to be executed when an invocation occurs. If this class implements a method that has the same signature as the called one, the found method will be executed. Otherwise, the parent classes will be checked recursively until the searched method is found. If no method is found, an error is signaled (*MsgNotUnderstood*). Unfortunately, this operation is too frequent and is very expensive. The principal dynamic binding algorithm is called the Dispatch Table Search [1] (DTS). It proceeds as mentioned above. The DTS is good in terms of memory cost, however the search overhead makes the mechanism too slow. Many techniques have been proposed to minimize the overhead associated to DTS: static techniques which pre-compute a part of the lookup and dynamic techniques which cache the results of previous lookup, thus avoiding other lookups. In the following, we present one of these techniques: the Selector Table Indexing technique (STI) [2]. Given a class hierarchy of C classes and S selectors, a two-dimensional array of $C * S$ entries is built. Classes and selectors are given consecutive numbers and the array is filled by pre-computing the lookup for each class and selector. An array entry contains a reference to the corresponding method or to an error routine. These tables are computed for a complete system. The main drawback of STI is that space requirements are huge for a large system. Hence, many dis-

patch table compression techniques have been proposed (Selector coloring [7], Row displacement [3], etc) to minimize space overhead. Another drawback of this technique is that the compiled code is very sensitive to changes in the class hierarchy. However, this technique delivers fast and constant time lookup.
Devirtualization technique with code patching mechanism [4] is another optimization technique that converts a virtual method call to a direct call. Given a dynamic site call, the current class hierarchy is analyzed by the compiler to determine if the call can be devirtualized. If it is true and if the method size is small, the compiler generates the inlined code of the method with the backup code of making the dynamic call. When devirtualization becomes invalid, the compiler performs code patching to make the backup code executed later. Otherwise (devirtualization is valid), only the inlined code is actually executed. The main drawback of this technique is that it relies on heavy analysis (flow-sensitive type analysis, preexistence analysis, dynamic class hierarchy analysis, etc.) so it is not convenient for embedded systems due to the fact that it can be too expensive in both time and space.
A static method call resolution technique [8] has been proposed to solve dynamic method calls. A variable-type analysis and a declared-type analysis use the whole class hierarchy program to compute a set of a method call receivers. This technique is limited because it does not deal with the dynamic class loading problem. In fact, the class hierarchy could change while the program is executing. This could change a method call receivers set and make the static performed optimizations inaccurate.
Dynamic techniques consist in caching results of previous lookups. Cache-based techniques eliminate the requirements to create huge dispatch tables, so memory overhead and table creation time are reduced. There are two main approaches to caching: global caches [1] and small inline caches [5]. Global cache technique stores the previous lookup results. In the global cache table, each entry consists of triplets (receiver class, selector and method address). The receiver class and the selector are used to compute an index in the cache. If the current class and the method name match those in the cached entry at the computed index, the code at the method address is executed. Hence, method lookup is avoided. Otherwise, a default dispatching technique (usually DTS) is used and at the end of this search, a new triplet is added to the cache table and control is transferred to the found method. The run-time memory required by this algorithm is small, usually a fixed amount of the cache and the overhead of the DTS technique. The main disadvantage of this technique is that a frequent change of the receiver class slows the execution. The inline cache technique consists in caching the result of the previous lookup (method address) in the code itself at each call site. Inline cache changes the call instruction by overwriting it with a direct invocation of the method found by the default method lookup. Inline cache assumes that the receiver's class changes infrequently otherwise, the inline cache technique delivers slow execution time. Polymorphic inline cache [9] is an extension of the inline cache technique. The compiler generates a call to a special stub routine. Each call site jumps to a specific stub function. The function is initially a call

to a method lookup. Each time the method lookup is called, the stub function is extended. This technique has the cost of a test and a direct jump in the best case. Moreover, the executable code could expand dramatically when the class hierarchy is huge and the receiver class changes frequently.

3 Approach

We propose a dynamic, flexible and efficient technique for accelerating method calls in embedded systems. This acceleration technique spans over 3 aspects of the method call: (1) lookup, (2) caching, and (3) synchronized methods.

3.1 Lookup

The lookup is accelerated by the application of a direct access to the method tables. This is achieved by using an appropriate hashing technique. Actually, we build a hash table for each virtual method table. Each index of the hash table is a hashing result of the method signature. The size of the hash table should be carefully chosen so as to have a low footprint while minimizing the collisions between method signatures. By doing so, we get a more efficient and flexible lookup mechanism. Efficiency stems from the fact that we have direct access to the method tables. Flexibility stems from the fact that we can tune the size of the hash table so as to have the best ratio for speed versus footprint.
In what follows we explain how this lookup acceleration could be implemented within a conventional embedded Java virtual machine such as KVM [6] (Kilobyte Virtual Machine). The method lookup mechanism in KVM is linear i.e. it uses a sequential access. A hash-based lookup will definitely yield a better performance. The implementation of such a mechanism will affect two components of the virtual machine: the loader and the interpreter. The loader is modified to construct hashed method tables for each loaded class. The interpreter is modified to take advantage of the new method tables to perform fast and direct-access lookups. During the loading process of a class, a hash method table is built. A hash is computed from the method signature. Each entry in the hash table consists of two components. The first component is a flag indicating whether the class contains a method with such a definition. The second component is a pointer to the method definition. In the case of a collision, this second component is a pointer to a list of method definitions. The method hash table construction algorithm is depicted in Fig. 1.

The original lookup algorithm is linear. It tests in each method table, by iteration over its elements, of a class if it has a method that has the same signature as that invoked. In Fig. 2 we give the original lookup mechanism.

The new lookup algorithm uses the hash obtained from the method signature to access the corresponding entry in the hash table. If the flag associated with this entry is ON, it accesses the method definition thanks to the second component of the entry. If the flag is OFF, this means that the class does not implement

```
HashTableBuild() {
   for each method of the class method table {
    h = compute_hash(method);
    element = get_element(hashTable, h);
    if (element->flag == ON) {
      allocate_space(method);
      register_method_in_collision_list();
    }
    else {
      register_method_in_hashTable();
      method->flag = ON;
    }
   }
}
```

Fig. 1. Method hash table construction algorithm

```
lookupMethod(class, key) {
    while (class) {
       table = get_method_table(class);
       for each method of table {
         if (method->signature == key) {
           return method;
         }
         class = class->superclass;
       }
    }
}
```

Fig. 2. Original lookup algorithm

```
lookupMethod(class, key) {
   while(class) {
     hashTable = get_hash_table(class);
     h  = compute_method_hash(key);
     entry = get_element(hashTable, h);
     if (entry->flag == ON) { /*Test the flag field of the hashing table entry*/
       for each element in collision list {
         if (key == element->signature)
             return element;
       }
     }
    class = class->superclass;
   }
}
```

Fig. 3. New lookup algorithm

such a method and the search is directed to the super-class. In Fig. 3, we give the new lookup algorithm.

The new lookup algorithm performs fewer iterations than the original one. In fact, in the worst case, the whole collision list has to be visited. The lookup method acceleration depends on the hash table size. In fact, a big size requires a high memory space but it minimizes the collisions. On the other hand, it might induce an additional cost in terms of memory management (allocation, garbage collection, compaction, etc.).

3.2 Caching

Another factor in the acceleration of a method call is the caching approach. We claim in this paper that the inline cache technique could achieve a significant speed-up of a program execution by slightly modifying the cache layout. Actually, the modification consists in adding a pointer to the receiver object in the cache entry. We explain hereafter why such a simple modification will result in a significant speed-up. In the conventional inline cache technique (such as the one implemented in KVM), only a pointer to the method definition is stored in the cache structure. When a method is invoked, the class of the receiver is compared to the class of the invoked method. If there is an equality between these two classes, the method definition is retrieved thanks to the cache entry. If there is no equality, a dynamic lookup is performed to search for the method definition in the class hierarchy. This inline cache technique could be significantly improved by avoiding many of the dynamic lookups when there is a mismatch between the class of the receiver and the class of the invoked method. Actually, when there is such a mismatch, if we can detect that the receiver has not changed, we can retrieve the method definition from the cache. This is done by:

- Adding a pointer to the receiver in the cache structure,
- Modifying the condition that guards cache retrieval. Actually, when a method is invoked, the condition to get a method definition from the cache is:
  - The class of the receiver is equal to the class of the invoked method, or,
  - The current receiver is equal to the cached receiver (the receiver has not changed).

Here is an illustration when this inline cache technique yields a significant speed-up. Assume that we have a class B that inherits a non-static method m from a class A. Assume also that we have a loop that will be executed very often in which we have the following method invocation: the method m is invoked on an object say o_B that is instance of the class B. In our inline caching technique the object o_B is going to be cached after the first invocation of the method m. In the subsequent invocations of the method m, since the class of the receiver (B) is different from the class of the invoked method (A), the behavior of the conventional inline cache technique will be very different from the one of the proposed inline technique. The conventional technique will perform a dynamic

lookup for each subsequent invocation of the method m resulting in a significant overhead. The inline technique with the modified cache structure will simply test if the current receiver equals the one that is stored in the cache. Accordingly, the method definition will be retrieved from the cache for all subsequent invocations of the method m resulting in a significant speed-up.

3.3 Synchronization

Before executing a synchronized method, a thread must acquire the lock associated with the receiver object. This is necessary if two threads are trying to execute this method on the same object. Locking an object slows the execution.

The proposed technique is related to the acceleration of multithreaded applications. It improves the synchronization technique used in many virtual machine interpreters. The synchronization technique used in conventional embedded Java virtual machines (especially those based on the KVM) associate to an object one of four states: unlocked, simple locked, extended locked and associated to a monitor.

The unlocked state is for an object that is not synchronized yet. The object passes form the unlocked state to the simple lock when a thread tries to lock it for the first time. The object state passes from the simple lock state to the extended lock state when a thread tries to lock the object for the second time. The extended lock state will be the state of the object until a second thread tries to lock it. From any of these states, a creation of a monitor can happen when a second thread tries to lock an object while another is the owner of the lock. A transition from any state to the monitor state could happen. Exiting a synchronized method triggers the transition from a monitor state or extended lock to a simple lock or an unlocked state. Note that a transition to a bad state can occurs when an exception is raised. We present in the following the automaton that represents the object state. Figure 4 depicts the original automaton where the states are:

- A: unlocked.
- B: simple lock state.
- C: extended state.
- D: monitor state.
- E: exception state (bad state).

And actions are:

- u: set_unlocked.
- s: set_simple_lock.
- e: set_extended_lock.
- m: set_monitor.
- x: raise_exception

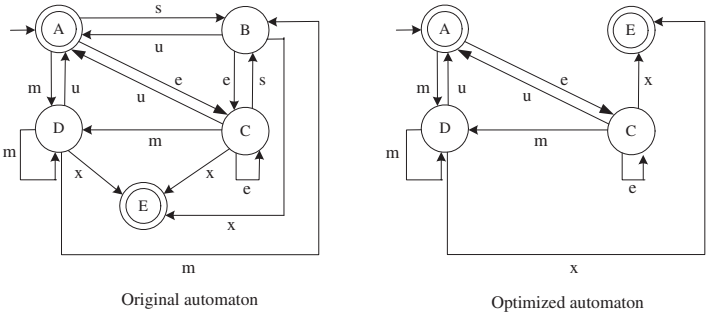

Fig. 4. Original and optimized automaton

By removing the state B, we can avoid transitions from and to the simple lock state, therefore going directly from the unlocked state to the extended lock state. Figure 4 depicts the optimized automaton.

An object passes from the unlocked state to the simple lock when a thread tries to lock it for the first time. In this case, the depth (number of time the object is locked) is equal to one. Since, an object passes from the simple lock state to the extended lock state when a thread tries to lock the object for the second time incrementing the depth, we can remove the simple lock state. In fact, it can be considered as an extended lock state which depth can be greater or equal to one. The elimination of the simple lock state improves thread performance because the execution of some portion of code is avoided. In fact, the transition from the simple lock to the extended lock is avoided.

4 Results

We implemented our technique in the original KVM 1.0.3. We did some testing on the embedded *CaffeineMark* benchmark without floats. Figure 6 presents a performance histogram that illustrates the comparison between the original and the optimized VMs. The comparison proves a reasonable speedup. For some typical examples (e.g. Java programs that frequently call inherited methods), our technique is capable to reach a speedup of more than 27%. Figure 5 shows a typical example that contains a class hierarchy and an invocation of an inherited method *m*. Figure 7 shows the execution time acceleration for this example. This time is given with respect to the hash table size.

5 Conclusion and Future Work

In this paper, we addressed the acceleration of method calls in an embedded Java virtual machine such as the KVM. We have proposed techniques for the acceleration of dynamic lookups, caching and synchronized method calls. The

```
public class A {
 public void m() {};
}

public class B extends A {
 public void m() {};
}

public class C extends B { }

public class D extends C {
    public static void main(String args[]) {
      A o;
      o = new D();
      int i = 0;
      while (i < 1000000) {
          o.m();
          i++;
      }
    }
}
```

Fig. 5. Typical example for our optimization technique

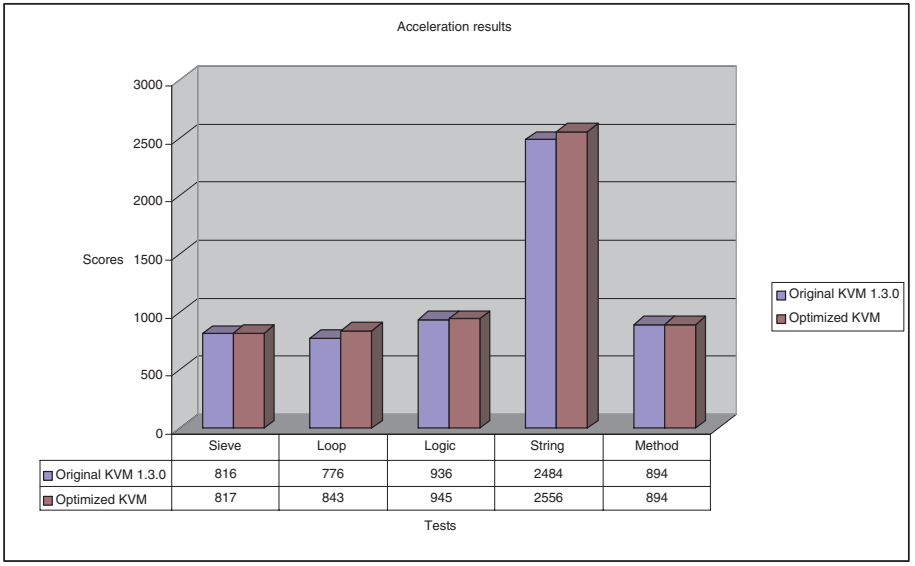

Fig. 6. Acceleration results for the embedded *CaffeineMark* benchmark

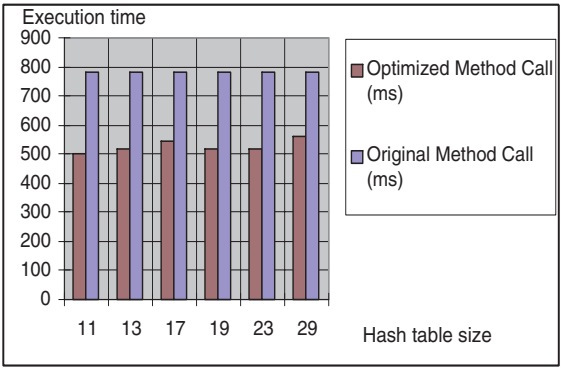

Fig. 7. Execution time acceleration for a typical example

results show that our optimizations are efficient and not expensive from the footprint standpoint. Moreover, these techniques are very generic and could be successfully applied to other virtual machines.

References

1. Goldberg A. and Robson D. *Smalltalk-80: The Language and its Implementation*. Addison–Wesley, 1985.
2. Cox B. *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley, 1987.
3. Driesen K. Selector Table Indexing and Sparse Arrays. In *Proceedings of OOPSLA'93*, Washington, DC, 1993.
4. Ishizaki K., Yasue T., Kawahito M., and Komatsu H. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the ACM-SIGPLAN Conference on Object-Oriented Programmimg Systems, Languages and Applications*, pages 294–310, Minneapolis, Minnesota, USA, October 2000.
5. Deutsch L.P. and Schiffman A. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Symposium on Principles of Programming Languages*, Salt Lake City, UT, 1984.
6. Sun MicroSystems. KVM Porting Guide. Technical report, Sun MicroSystems, California, USA, September 2001.
7. Dencker P., Durre K., and Heuft J. Optimization of Parser Tables for Portable Compilers. *ACM TOPLAS*, 6(4):546–572, 1984.
8. Vijay S., Laurie H., Chrislain R.and Raja V., Patrick L., Etienne G., and Charles G. Practical Virtual Method Call Resolution for Java. In *Proceedings of the ACM-SIGPLAN Conference on Object-Oriented Programmimg Systems, Languages and Applications*, pages 264–280, Minneapolis, Minnesota, USA, October 2000.
9. Holzle U., Chambers C., and Ungar D. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of ECOOP'93*, 1993.