Cache Sensitive Code Arrangement for Virtual Machine

Chun-Chieh Lin and Chuen-Liang Chen

Department of Computer Science and Information Engineering,
National Taiwan University, Taipei,
10764, Taiwan
{d93020, clchen}@csie.ntu.edu.tw

Abstract. This paper proposes a systematic approach to optimize the code layout of a Java ME virtual machine for an embedded system with a cache-sensitive architecture. A practice example is to run JVM directly (execution-in-place) in NAND flash memory, for which cache miss penalty is too high to endure. The refined virtual machine generated cache misses 96% less than the original version. We developed a mathematical approach helping to predict the flow of the interpreter inside the virtual machine. This approach analyzed both the static control flow graph and the pattern of bytecode instruction streams, since we found the input sequence drives the program flow of the virtual machine interpreter. Then we proposed a rule to model the execution flows of Java instructions of real applications. Furthermore, we used a graph partition algorithm as a tool to deal with the mathematical model, and this finding helped the relocation process to move program blocks to proper memory pages. The refinement approach dramatically improved the locality of the virtual machine thus reduced cache miss rates. Our technique can help Java ME-enabled devices to run faster and extend longer battery life. The approach also brings potential for designers to integrate the XIP function into System-on-Chip thanks to lower demand for cache memory.

Keywords: cache sensitive, cache miss, NAND flash memory, code arrangement, Java virtual machine, interpreter, embedded system.

1 Introduction

Java platform extensively exists in all kinds of embedded and mobile devices. The Java™ Platform, Micro Edition (Java ME) [1] is no doubt a de facto standard platform of smart phone. The Java virtual machine (it is KVM in Java ME) is a key component that affects performance and power consumptions.

NAND flash memory comes with serial bus interface. It does not allow random access, and the CPU must read out the whole page at a time, which is a slow operation compared to RAM. This property leads a processor hardly to execute programs stored in NAND flash memory using the "execute-in-place" (XIP) technique. In the

We acknowledge the support for this study through grants from National Science Council of Taiwan (NSC 95-2221-E-002 -137).

meanwhile, NAND flash memory offers fast write access time, and the most important of all, the technology has advantages in offering higher capacity than NOR flash technology does. As the applications of embedded devices become large and complicated, more mainstream devices adopt NAND flash memory to replace NOR-flash memory.

In this paper, we tried to offer an answer to the question: can we speed up an embedded device using NAND flash memory to store programs? "Page-based" storage media, like NAND flash memory, have higher access penalty than RAM does. Reducing the page miss becomes a critical issue. Thus, we set forth to find way to reduce the page miss rate generated by the KVM. Due to the unique structure of the KVM interpreter, we found a special way to exploit the dynamic locality of the KVM that is to trace the patterns of executed bytecode instructions instead of the internal flow of the KVM. It turned out to be a combinatorial optimization problem because the code layout must fulfill certain code size constraints. Our approach achieved the effect of static page preloading by properly arranging program blocks. In the experiment, we implemented a post-processing program to modify the intermediate files generated by the C compiler. The post-processing program refined machine code placement of the KVM based on the mathematical model. Finally, the obtained tuned KVMs dramatically reduced page accesses to NAND flash memories. The outcome of this study helps embedded systems to boost performance and extend battery life as well.

2 Related Works

Park *et al*., in [2], proposed a hardware module to allow direct code execution from NAND flash memory. In this approach, program codes stored in NAND flash pages will be loaded into RAM cache on-demand instead of moving entire contents into RAM. Their work is a universal hardware-based solution without considering application-specific characteristics.

Samsung Electronics offers a commercial product called "OneNAND" [3] based on the same. It is a single chip with a standard NOR flash interface. Actually, it contains a NAND flash memory array for storage. The vendor intent was to provide a cost-effective alternative to NOR flash memory used in existing designs. The internal structure of OneNAND comprises a NAND flash memory, control logic, hardware ECC, and 5KB buffer RAM. The 5KB buffer RAM is comprised of three buffers: 1KB for boot RAM, and a pair of 2KB buffers used for bi-directional data buffers. Our approach is suitable for systems using this type of flash memories.

Park *et al*., in [4], proposed yet another pure software approach to achieve execute-in-place by using a customized compiler that inserts NAND flash reading operations into program code at proper place. Their compiler determines insertion points by summing up sizes of basic blocks along the calling tree. Special hardware is no longer required, but in contrast to earlier work [2], there is still a need for tailor-made compiler.

Typical studies of refining code placement to minimize cache misses can apply to NAND flash cache system. Parameswaran *et al*., in [5], used the bin-packing

approach. It reorders the program codes by examining the execution frequency of basic blocks. Code segments with higher execution frequency are placed next to each other within the cache. Janapsatya *et al.*, in [6], proposed a pure software heuristic approach to reduce number of cache misses by relocating program sections in the main memory. Their approach was to analyze program flow graph, identify and pack basic blocks within the same loop. They have also created relations between cache miss and energy consumption. Although their approach can identify loops within a program, breaking the interpreter of a virtual machine into individual circuits is hard because all the loops share the same starting point.

There are researches in improving program locality and optimizing code placement for either cache or virtual memory environment. Pettis [7] proposed a systematic approach using dynamic call graph to position procedures. They tried to place two procedures as close as possible if one of the procedure calls another frequently. The first step of Pettis' approach uses the profiling information to create weighted call graph. The second step iteratively merges vertices connected by heaviest weight edges. The process repeats until the whole graph composed of one or more individual vertex without edges.

However, the approach to collect profiling information and their accuracy is yet another issue. For example, Young and Smith in [8] developed techniques to extract effective branch profile information from a limited depth of branch history. Ball and Larus in [9] described an algorithm for inserting monitoring code to trace programs. Our approach is very different by nature. Previous studies all focused in the flow of program codes, but we tried to model the profile by input data.

This research project created a post-processor to optimize the code arrangements. It is analogous to "Diablo linker" [10]. They utilized symbolic information in the object files to generate optimized executable files. However, our approach will generate feedback intermediate files for the compiler, and invoke the compiler to generate optimized machine code.

3 Background

3.1 XIP with NAND Flash

NOR flash memory is popular as code memory because of the XIP feature. There are several approaches designed for using NAND flash memory as an alternative to NOR flash memory. Because NAND flash memory interface cannot connect to the CPU host bus, there has to be a memory interface controller to move data from NAND flash memory to RAM.

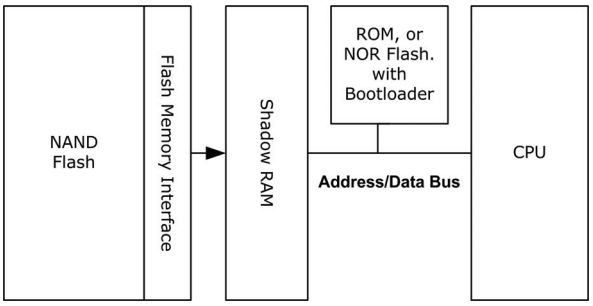


Fig. 1. Access NAND flash through shadow RAM

In system-level view, Figure 1 shows a straightforward design which uses RAM as the shadow copy of NAND flash. The system treats NAND flash memory as secondary storage device [11]. There should be a boot loader or RTOS resided in ROM or NOR flash memory. It copies program codes from NAND flash to RAM, then the processor executes program codes in RAM [12]. This approach offers best execution speed because the processor operates with RAM. The downside of this approach is it needs huge amount of RAM to mirror NAND flash. In embedded devices, RAM is a precious resource. For example, the Sony Ericsson T610 mobile phone [13] reserved 256KB RAM for Java heap. In contrast to using 256MB for mirroring NAND flash memory, all designers should agree that they would prefer to retain RAM for Java applets rather than for mirroring. The second pitfall is the implementation takes longer time to boot because the system must copy contents to RAM prior to execution.

Figure 2 shows a demand paging approach uses limited amount of RAM as the cache of NAND flash. The "romized" program codes stay in NAND flash memory, and a MMU loads only portions of program codes which is about to be executed from NAND into the cache. The major advantage of this approach is it consumes less RAM. Several kilobytes of RAM are enough to mirror NAND flash memory. Using less RAM means integrating CPU, MMU and cache into a single chip (the shadowed part in Figure 2) can be easier. The startup latency is shorter since the CPU is ready to run soon after the first NAND flash page is loaded into the cache. The component cost is lower than in the previous approach. The realization of the MMU might be either hardware or software approach, which is not covered in this paper.

However, performance is the major drawback of this approach. The penalty of each cache miss is high, because loading contents from a NAND flash page is nearly 200 times slower than doing the same operation with RAM. Therefore reducing cache misses becomes a critical issue for such configurations.

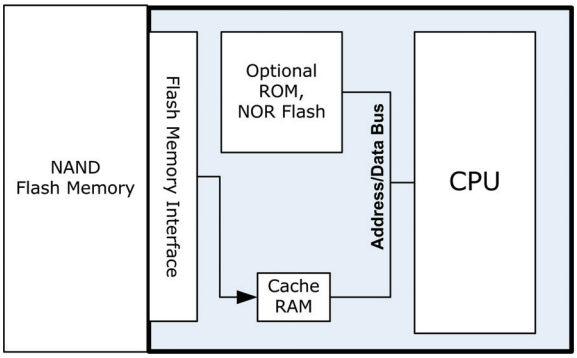


Fig. 2. Using cache unit to access NAND flash

3.2 KVM Internals

Source Level. In respect of functionality, the KVM can be broken down into several parts: startup, class files loading, constant pool resolving, interpreter, garbage collection, and KVM cleanup. Lafond *et al*., in [14], have measured the energy consumptions of each part in the KVM. Their study showed that the interpreter consumed more than 50% of total energy. In our experiments running Embedded Caffeine Benchmark [15], the interpreter contributed 96% of total memory accesses. These evidences lead to the conclusion that the interpreter is the performance bottleneck of the KVM, and they motivated us to focus on reducing the cache misses generated by the interpreter.

Figure 3 shows the program structure of the interpreter. It is a loop enclosing a large switch-case dispatcher. The loop fetches bytecode instructions from Java applications, and each "case" sub-clause deals with one bytecode instruction. The control flow graph of the interpreter, as illustrated in Figure 4, is a flat and shallow spanning tree. There are three major steps in the interpreter,

(1) Rescheduling and Fetching. In this step, KVM prepares the execution context and the stack frame. Then it fetches a bytecode instruction from Java programs.

(2) Dispatching and Execution. After reading a bytecode instruction from Java programs, the interpreter jumps to corresponding bytecode handlers through the big "switch…case…" statement. Each bytecode handler carries out the function of the corresponding bytecode instruction.

(3) Branching. The branch bytecode instructions may bring the Java program flow away from original track. In this step, the interpreter resolves the target address and modifies the program counter.

```
ReschedulePoint:
RESCHEDULE
opcode = FETCH_BYTECODE ( ProgramCounter );
switch ( opcode )
{
            case ALOAD: /* do something */
              goto ReschedulePoint;
            case IADD: /* do something */
              …
            case IFEQ: /* do something */
              goto BranchPoint;
              …
}
BranchPoint:
            take care of program counter;
            goto ReschedulePoint;
```

Fig. 3. Pseudo code of KVM interpreter

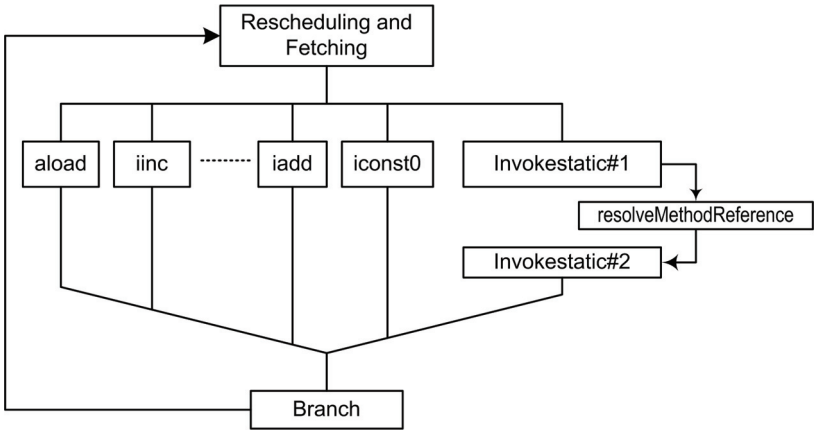


Fig. 4. Control flow graph of the interpreter

Assembly Level. Our analysis of the source files revealed the peculiar program structure of the VM interpreter. Analyzing the code layout in the compiled executables of the interpreter helped this study to create a code placement strategy. The assembly code analysis in this study is restricted to ARM and *gcc* for the sake of demonstration, but applying our theory to other platforms and tools is an easy job. Figure 5 illustrates the layout of the interpreter in assembly form (FastInterpret() in interp.c). The first trunk *BytecodeFetching* is the code block for rescheduling and fetching, it is exactly the first part in the original source code. The second trunk *LookupTable* is a large lookup table used in dispatching bytecode instructions. Each entry links to a bytecode handler. It is actually the translated result of the "switch…case…case" statement.

The third trunk *BytecodeDispatch* is the aggregation of more than a hundred bytecode handlers. Most bytecode handlers are self-contained which means a bytecode handler occupies a contiguous memory space in this trunk and it does not

jump to program codes stored in other trunks. There are only a few exceptions which call functions stored in other trunks, such as "invokevirtual." Besides, there are several constant symbol tables spread over this trunk. These tables are referenced by the program codes within the *BytecodeDispatch* trunk.

The last trunk *ExceptionHandling* contains code fragments related with exception handling. Each trunk occupies a number of NAND flash pages. In fact, the total size of *BytecodeFetching* and *LookupTable* is about 1200 bytes (compiled with *arm-elf-gcc-3.4.3*), which is almost small enough to fit into two or three 512-bytes-page. Figure 6 shows the size distribution of bytecode handlers. The average size of a bytecode handler is 131 bytes, and there are 79 handlers smaller than 56 bytes. In other words, a 512-bytes-page could gather 4 to 8 bytecode handlers. The inter-handler execution flow dominates the number of cache misses generated by the interpreter. This is the reason that our approach tries to rearrange bytecode handlers within the *BytecodeDispatch* trunk.

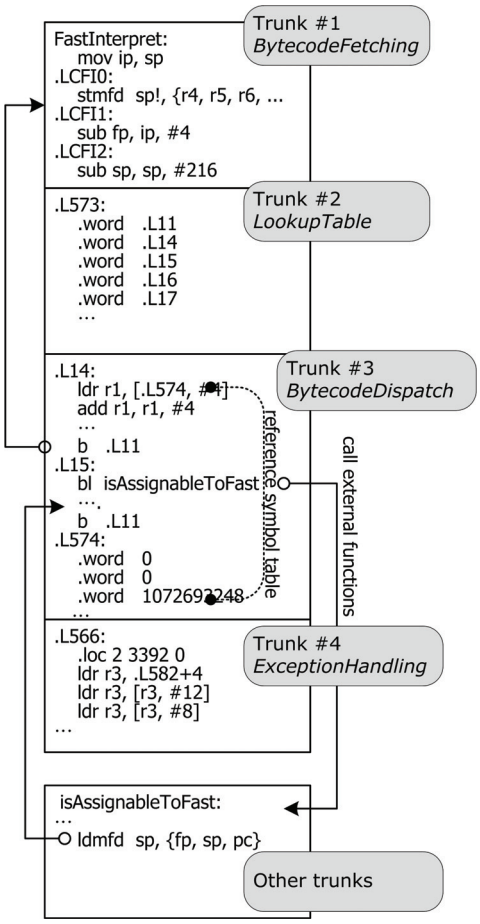


Fig. 5. The organization of the interpreter at assembly level

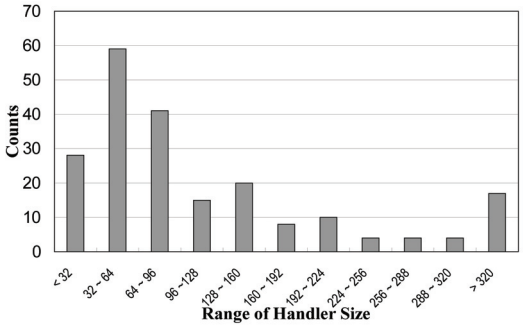


Fig. 6. Distribution of Bytecode Handler Size (compiled with *gcc-3.4.3*)

4 Analyzing Control Flow

4.1 Indirect Control Flow Graph

Static branch-prediction and typical code placement approaches derive the layout of a program from its control flow graph (CFG). However, the CFG of a VM interpreter is a special case, its CFG is a flat spanning tree enclosed by a loop. The CFG does not provide information to distinguish the temporal order between each bytecode handler. If someone wants to improve the program locality by observing the dynamic execution order of program blocks, the CFG is apparently not a good tool to this end. Therefore, we propose a concept called "Indirect Control Flow Graph" (ICFG); it uses the real bytecode instruction sequences to construct the dual CFG of the interpreter.

Consider a simplified virtual machine with 5 bytecode instructions: A, B, C, D, and E, and use the virtual machine to run a very simple user applet. Consider the following short alphabetic sequence as the instruction sequence of the user applet:

A-B-A-B-C-D-E-C

Each alphabet in the sequence represents a bytecode instruction. In Figure 7, the graph connected with the solid lines is the CFG of the simplified interpreter. By observing the flow in the CFG, the program flow becomes:

[Dispatch] – [Handler A] – [Dispatch] – [Handler B]…

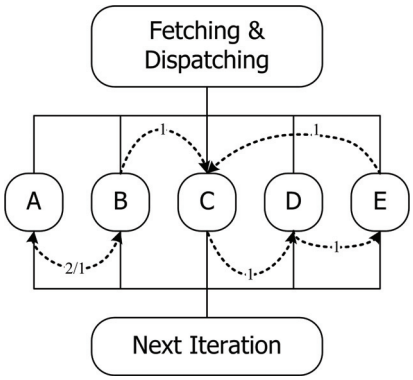


Fig. 7. The CFG of the simplified interpreter

It is hard to tell the relation between handler-A and handler-B because the loop header hides it. In other words, this CFG cannot easily present which handler would be invoked after handler-A is executed. The idea of the ICFG is to observe the patterns of the bytecode sequences executed by the virtual machine, not to analyze the structure of the virtual machine itself. Figure 8 expresses the ICFG in a readable way, it happens to be the sub-graph connected by the dashed directed lines in Figure 7.

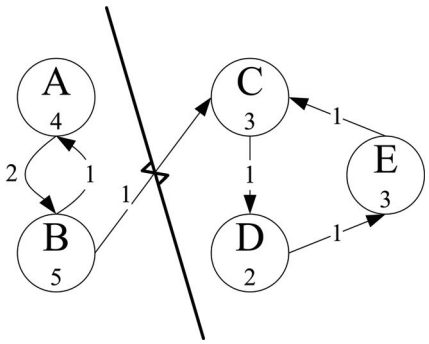


Fig. 8. An ICFG example. The number inside the circle represents the size of the handler.

4.2 Tracing the Locality of the Interpreter

As stated, the Java applications that a KVM runs dominate the code locality of the interpreter. Precisely speaking, the incoming Java instruction sequence dominates code locality. Therefore, the first step to exploit the code locality is to consider the bytecode sequences executed by the virtual machine. Consider the previous example sequence, the order of accessed NAND flash pages is supposed to be:

[*BytecodeFetching*]–[*LookupTable*]–[A]–[*BytecodeFetching*]–[*LookupTable*]–[B]–[*BytecodeFetching*]–[*LookupTable*]–[A]…

Obviously, memory pages containing *BytecodeFetching* and *LookupTable* are much often to appear in the sequence than those containing *BytecodeDispatch*. As a result, pages containing *BytecodeFetching* and *LookupTable* are favorable to last in the cache. Pages holding bytecode handlers have to compete with each other to stay in the cache. Thus, we induced that the order of executed bytecode instructions is the key factor impacts cache misses.

Consider an extreme case: In a system with three cache blocks, two cache blocks always hold memory pages containing *BytecodeFetching* and *LookupTable* due to the stated reason. Therefore, there is only one cache block available for swapping pages containing bytecode handlers. If all the bytecode handlers were located in distinct memory pages, processing a bytecode instruction would cause a cache miss. This is because the next-to-execute bytecode handler is always located in an uncached memory page. In other words, the sample sequence causes at least eight cache misses. Nevertheless, if both the handlers of A and B are grouped to the same page, cache misses will decline to 5 times, and the page access trace becomes:

fault-A-B-A-B-*fault*-C-*fault*-D-*fault*-E-*fault*-C

If we extend the group (A, B) to include the handler of C, the cache miss count would even drop to four times, and the page access trace looks like the following one:

fault-A-B-A-B-C-*fault*-D-*fault*-E-*fault*-C

Therefore, the core issue of this study is to find an efficient code layout method partitioning all bytecode instructions into disjoined sets based on their execution relevance. Each NAND flash page contains one set of bytecode handlers. We propose partitioning the ICFG reaches this goal.

Back to Figure 8, the directed edges represent the temporal order of the instruction sequence. The weight of an edge is the transition count for transitions from one bytecode instruction to the next. If we remove the edge (B, C), the ICFG is divided into two disjoined sets. That is, the bytecode handlers of A and B are placed in one page, and the bytecode handlers of C, D, and E are placed in the other. The page access trace becomes:

fault-A-B-A-B-*fault*-C-D-E-C

This placement causes only two cache misses, which is 75% lower than the worst case! The next step is to transform the ICFG diagram to an undirected graph by merging reversed edges connecting same vertices, and the weight of the undirected edge is the sum of weights of the two directed edges. The consequence is actually a variation of the classical MIN *k*-CUT problem. Formally speaking, we can model a given graph $G(V, E)$ as:

- $\boldsymbol{V_i}$ – represents the *i*-th bytecode instruction.
- $\boldsymbol{E_{i,j}}$ – the edge connecting *i*-th and *j*-th bytecode instruction.
- $\boldsymbol{F_{i,j}}$ – number of times that two bytecode instructions *i* and *j* executed after each other. It is the weight of edge $E_{i,j}$.
- $\boldsymbol{K}$ – number of expected partitions.
- $\boldsymbol{W_{x,y}}$ – the inter-set weight. $\forall\ x \neq y$, $W_{x,y} = \Sigma F_{i,j}$ where $V_i \in P_x$ and $V_j \in P_y$.

The goal is to model the problem as the following definition:

Definition 1. The MIN *k*-CUT problem is to divide *G* into *K* disjoined partitions $\{P_1, P_2, \ldots, P_k\}$ such that $\Sigma W_{i,j}$ is minimized.

4.3 The Mathematical Model

Yet there is an additional constraint in our model. It is impractical to gather bytecode instructions to a partition regardless of the sum of the program size of consisted bytecode handlers. The size of each bytecode handler is distinct, and the code size of a partition cannot exceed the size of a memory page (e.g. NAND flash page). Our aim is to distribute bytecode handlers into several disjoined partitions $\{P_1, P_2,\ldots,P_k\}$. We define the following notations:

- $\boldsymbol{S_i}$ – the code size of bytecode handler V_i.
- $\boldsymbol{N}$ – the size of a memory page.
- $\boldsymbol{M(P_k)}$ – the size of partition P_k . It is ΣS_m for all $V_m \in P_k$.
- $\boldsymbol{H(P_k)}$ – the value of partition P_k . It is $\Sigma F_{i,j}$ for all V_i , $V_j \in P_k$.

Our goal is to construct partitions that satisfy the following constraints.

Definition 2. The problem is to divide G into K disjoined partitions $\{P_1, P_2,\ldots,P_k\}$. For each P_k that $M(P_k) \le N$ such that $W_{i,j}$ is minimized, and maximize $\Sigma H(P_i)$ for all $P_i \in \{P_1, P_2,\ldots,P_k\}$.

This rectified model is exactly an application of the graph partition problem, i.e., the size of each partition must satisfy the constraint (size of a memory page), and the sum of inter-partition path weights is minimal. The graph partition problem is NP-complete [16]. However, the purpose of this paper was neither to create a new graph partition algorithm nor to discuss difference between existing algorithms. The experimental implementation just adopted the following algorithm to demonstrate our approach works. Other implementations based on this approach may choose another graph partition algorithm that satisfies specific requirements.

Partition (G)

1. Find the edge with maximal weight $F_{i,j}$ among graph G, while the $S_i + S_j \le N$. If there is no such an edge, go to step 4.
2. Call ***Merge (V_i , V_j)*** to combine vertices V_i and V_j.
3. Remove both V_i and V_j from G, go to step 1.
4. Find a pair of vertices V_i and V_j in G such that $S_i + S_j \le N$. If there is not a pair satisfied the criteria, go to step 7.
5. Call ***Merge (V_i , V_j)*** to combine vertices V_i and V_j.
6. Remove both V_i and V_j out of G, go to step 4.
7. End.

The procedure of merging both vertices V_i and V_j is:

Merge (V_i , V_j)

1. Add a new vertex V_k to G.
2. Pickup an edge E connects V_t with either V_i or V_j . If there is no such an edge, go to step 6.
3. If there is already an edge F connects V_t to V_k.
4. Then, add the weight of E to F, and discard E.
5. Else, replace one end of E which is either V_i or V_j with V_k.
6. End.

Finally, each vertex in G is a collection of several bytecode handlers. The refinement process is to collect bytecode handlers belonging to the same vertex and place them into one memory page.

5 The Process of Rewriting the Virtual Machine

Our approach emphasizes that the arrangements of bytecode handlers affects cache miss rate. In other words, it implies that programmers should be able to speed up their programs by properly changing the order of the "case" sub-clauses in the source files. Therefore, this study tries to optimize the virtual machine in two distinct ways. The first approach revises the order of the "case" sub-clauses in the sources of the virtual machine. If our theory were correct, this tentative approach should show that the modified virtual machine performs better in most test cases. The second version precisely reorganizes the layout of assembly code blocks of bytecode handlers, and this approach should be able to generate larger improvements than the first version.

5.1 Source-Level Rearrangement

The concept of the refining process is to arrange the order of these "case" statements in the source file (execute.c). The consequence is that after translating the rearranged source files, the compiler will place bytecode handlers in machine code form in meditated order. The following steps are the outline of the refining procedures.

A. Profiling. Run the Java benchmark program on the unmodified KVM. A custom profiler traces the bytecode instruction sequence, and it generates the statistics of inter-bytecode instruction counts. Although we can collect some patterns of instruction combinations by investigating the Java compiler, using a dynamic approach can capture further application-specific patterns.

B. Measuring the size of each bytecode handler. The refining program compiles the KVM source files and measures the code size of each bytecode handler (i.e., the size of each 'case' sub-clause) by parsing intermediate files generated by the compiler.

C. Partitioning the ICFG. The previous steps collect all necessary information for constructing the ICFG. Then, the refining program partitions the ICFG by using a graph partition algorithm. From that result, the refining program knows the way to group bytecode handlers together. For example, a partition result groups (A, B) to a bundle and (C, D, E) to another as shown in Figure 8.

D. Rewriting the source file. According to the computed results, the refining program rewrites the source file by arranging the order of all "case" sub-clauses within the interpreter loop. Figure 9 shows the order of all "case" sub-clauses in the previous example.

```
switch ( opcode ) {
    case B:        …;
    case A:        …;
    case E:        …;
    case D:        …;
    case C:        …;
}
```

Fig. 9. The output of rearranged case statements

5.2 Assembly-Level Rearrangement

The robust implementation of the refinement process consists of two steps. The refinement process acts as a post processor of the compiler. It parses intermediate files generated by the compiler, rearranges program blocks, and generates optimized assembly codes. Our implementation is inevitably compiler-dependent and CPU-dependent. Current implementation tightly is integrated with *gcc* for ARM, but the approach is easy to apply to other platforms. Figure 10 illustrates the outline of the processing flow, entities, and relations between each entity. The following paragraphs explain the functions of each step.

A. Collecting dynamic bytecode instruction trace. The first step is to collect statistics from real Java applications or benchmarks, because the following steps will need these data for partitioning bytecode handlers. The modified KVM dumps the bytecode instruction trace while running Java applications. A special program called TRACER analyzes the trace dump to find the transition counts for all instruction pairs.

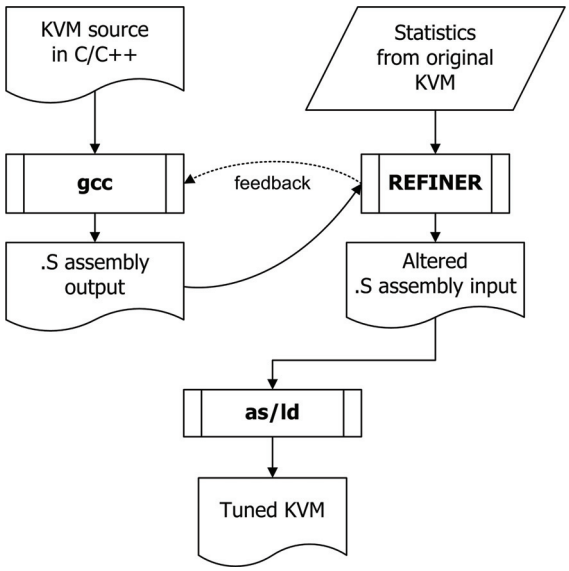


Fig. 10. Entities in the refinement process

B. Rearranging the KVM interpreter. This is the core step and is realized by a program called REFINER. It acts as a post processor of *gcc*. Its duty is to parse bytecode handlers expressed in the assembly code and organize them into partitions. Each partition fits into one NAND flash page. The program consists of several sub tasks described as follows.

(i) Parsing layout information of the original KVM. The very first thing is to compile the original KVM. REFINER parses the intermediate files generated by *gcc*. According to structure of the interpreter expressed in assembly code introduced in

§3.2, REFINER analyzes the jump table in the *LookupTable* trunk to find out the address and size of each bytecode handler.

(ii) Using the graph partition algorithm to group bytecode handlers into disjoined partitions. At this stage, REFINER constructs the ICFG with two key parameters: (1) the transition counts of bytecode instructions collected by TRACER; (2) the machine code layout information collected in the step A. It uses the approximate algorithm described in §4.3 to divide the undirected ICFG into disjoined partitions.

(iii) Rewriting the assembly code. REFINER parses and extracts assembly codes of all bytecode handlers. Then, it creates a new assembly file and dumps all bytecode handlers partition by partition according to the result of (ii).

(iv) Propagating symbol tables to each partition. As described in §3.2, there are several symbol tables distributed in the *BytecodeDispatch* trunk. For most RISC processors like ARM and MIPS, an instruction is unable to carry arbitrary constants as operands because of limited instruction word length. The solution is to gather used constants into a symbol table and place this table near the instructions that will access these constants. Hence, the compiler generates instructions with relative addressing operands to load constants from the nearby symbol tables. Take ARM for example, its application binary interface (ABI) defines two instructions called LDR and ADR for loading a constant from a symbol table to a register [17]. The ABI restricts the maximal distance between a LDR/ADR instruction and the referred symbol table to 4K bytes.

Besides, it would cause a cache miss if a machine instruction in page X loads a constant s_i from symbol table S_Y located in page Y. Our solution is to create a local symbol table S_X in page X and copy the value s_i to the new table. Therefore, the relative distance between s_i and the instruction never exceeds 4KB neither causes cache misses when the CPU tries to load s_i.

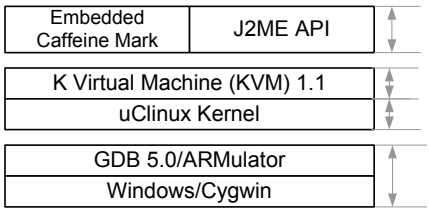
(v) Dumping contents in partitions to NAND flash pages. The aim is to map bytecode handlers to NAND flash pages. Its reassembled bytecode handlers belong to the same partition in one NAND flash page. After that, REFINER refreshes the address and size information of all bytecode handlers. The updated information helps REFINER to add padding to each partition and enforce the starting address of each partition to align to the boundary of a NAND flash page.

6 Evaluation

In this section, we start from a brief introduction of the environment and conditions used in the experiments. The first part of the experimental results is the outcome of source-level rearranged virtual machine. Those positive results prove our theory works. The next part is the experiment of assembly-level rearranged virtual machine. It further proves our refinement approach is able to produce better results than the original version.

6.1 Evaluation Environment

Figure 11 shows the block diagram of our experimental setup. In order to mimic real embedded applications, we have implanted Java ME KVM into uClinux for ARM7 in the experiment. One of the reasons to use this platform is that uClinux supports FLAT executable file format which is perfect for realizing XIP. We ran KVM/uClinux on a customized *gdb*. This customized *gdb* dumped memory access traces and performance statistics to files. The experimental setup assumed there was a specialized hardware unit acting as the NAND flash memory controller, which loads program codes from NAND flash pages to the cache. It also assumed all flash access operations worked transparently without the help from the operating system. In other words, modifying the OS kernel for the experiment is unnecessary. This experiment used "Embedded Caffeine Mark 3.0" [15] as the benchmark.



Title	Version
arm-elf-binutil	2.15
arm-elf-gcc	3.4.3
uClibc	0.9.18
J2ME (KVM)	CLDC 1.1
elf2flt	20040326

Fig. 11. Hierarchy of simulation environment

There are several kinds of NAND flash commodities in the market: 512-bytes, 2048-bytes, and 4096-bytes per page. In this experiment, we model the cache simulator after the following conditions:

1. There were four NAND flash page size options: 512, 1024, 2048 and 4096.
2. The page replacement policy was full associative, and it is a FIFO cache.
3. The number of cache memory blocks varied from 2, 4 … to 32.

6.2 Results of Source-Level Rearrangement

First, we rearranged the "case" sub-clauses in the source codes using the introduced method. Table 1 lists the raw statistics of cache miss rates, and Figure 12 plots the charts of normalized cache miss rates from the optimized KVM.

Each of the four columns is the result from a special kind of KVM. The "original" column refers to statistics from the original KVM, in which bytecode handlers were arranged in machine code order. The second column "optimized" is the result from the KVM refined with our approach. The improvement ratio is a comparison between the original and the optimized KVM. The experiment assumed the maximal cache size is 64K bytes. For each NAND flash page size, the number of cache blocks starts from 4 to (*64K / NAND flash page size)*.

The result showed that the optimized KVM outperforms than the other in most cases. There is 95% improvement in the best case. Looking at the charts, the curves of normalized cache miss rates (i.e., *optimized_miss_rate* / *original_miss_rate*) tend to

be concave. It means the improvement for the case of eight pages is greater than the one of four pages. It benefits from the smaller "locality" of the optimized KVM. Therefore, the cache could hold more localities, and this is helpful in reducing cache misses. After touching the bottom, the cache is large enough to hold most of the KVM program code. As the cache size grows, the numbers of cache misses of configurations converge.

However, the miss rate at 1024 bytes * 32 blocks is an exceptional case. This is because our approach rearranges the order of bytecode handlers at source level, and it cannot enforce the predicted and the real start address and code size of a bytecode handler to be the same. Therefore, it causes the cache miss rates to increase.

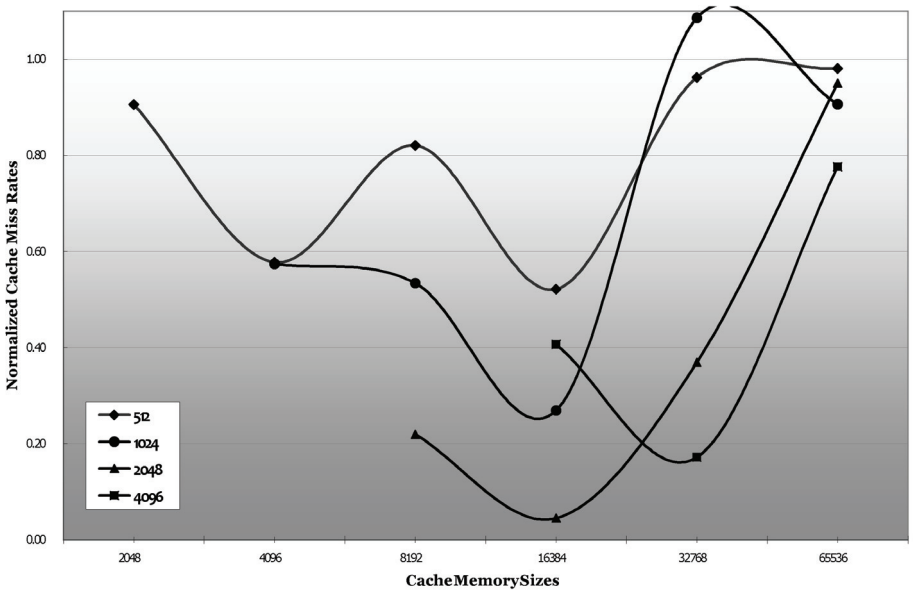


Fig. 12. The charts of normalized cache-miss rates from the source-level refined virtual machine. The x-axis is the size of the cache memory (*number_of_blocks* * *block_size*).

Table 1. Normalized cache miss rates generated from source-level modified virtual machines.

512 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
4	9.39%	25,242,319	22,871,780
8	42.25%	11,269,029	6,508,217
16	17.94%	2,472,373	2,028,834
32	47.84%	145,005	75,632
64	3.75%	11,933	11,485
128	1.91%	2,507	2,459
Total Access		*567,393,732*	*567,393,732*

1024 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
4	42.58%	15,988,106	9,180,472
8	46.58%	5,086,130	2,717,027
16	73.10%	486,765	130,921
32	-8.63%	23,395	25,413
64	9.35%	3,230	2,928
Total Access		567,393,732	567,393,732

2048 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
4	78.05%	10,813,688	2,373,841
8	95.51%	2,341,042	105,157
16	63.08%	68,756	25,388
32	4.98%	4,294	4,080
Total Access		567,393,732	567,393,732

4096 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
4	59.33%	4,899,778	1,992,734
8	82.82%	422,512	72,580
16	22.37%	8,995	6,983
Total Access		567,393,732	567,393,732

6.3 Results of Assembly-Level Rearrangement

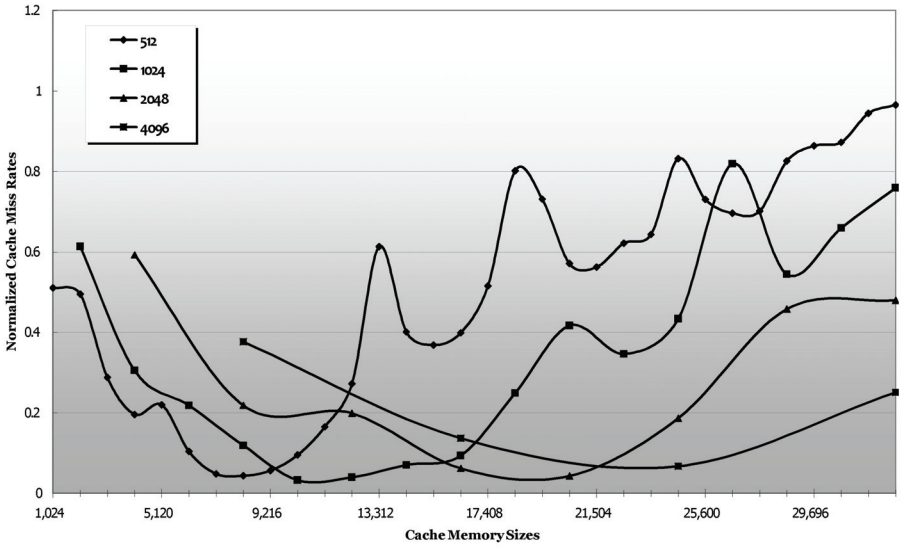


Fig. 13. The chart of normalized cache-miss rates from assembly-level rearranged virtual machines. The x-axis is the size of the cache memory (*number_of_blocks * block_size*).

The last experiment proved the theory should work but with a few exceptions. The assembly-level rearrangement method is a remedy. We tuned four versions of KVM; each of them suited to one kind of page size. All the experimental measurements are

compared to those from the original KVM. Table 2 is the highlight of experimental results and shows the extent of improvement of the optimized versions as well.

In the test case with 4KB/512-bytes per page, the cache miss rate of the tuned KVM is less than 1%, in contrast to the cache miss rate of the original KVM that is greater than 3%. In the best case, the cache miss rate of the tuned KVM is 96% lower than the value from the original one. Besides, in the case with only two cache blocks (1KB/512-bytes per page), the improvement is about 50%. It means tuned KVMs outperform on devices with limited cache blocks.

Figure 13 is the chart of the normalized miss rates. The envelope lines of these charts are tending to be concave. In the conditions that the amounts of cache blocks is small, the cache miss rates of the tuned KVM decline faster than the rates of the original version, and the curve goes downward. Once there is enough cache blocks to hold the entire locality of the original KVM, the tuned version gradually loses its advantages, and the curve turns upward.

In both experiments, the normalized miss rate curves are tending to be concave. We conclude this is a characteristic of our approach.

Table 2. Experimental cache miss counts. Data of 21 to 32 blocks are omitted due to being less relevant.

512 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
2	48.94%	52106472	25275914
4	50.49%	34747976	16345163
6	71.19%	26488191	7249424
8	80.42%	17709770	3294736
10	78.02%	12263183	2560674
12	89.61%	9993229	986256
14	95.19%	6151760	280894
16	95.63%	4934205	204975
18	94.37%	3300462	176634
20	90.48%	1734177	156914
Total Access		*548980637*	*521571173*

1024 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
2	38.64%	29760972	17350643
4	69.46%	21197760	6150007
6	78.15%	13547700	2812730
8	88.11%	8969062	1013010
10	96.72%	6354864	197996
12	96.02%	3924402	148376
14	92.97%	1735690	115991
16	90.64%	1169657	104048
18	75.11%	380285	89934
20	58.30%	122884	48679
Total Access		*548980637*	*521571046*

2048 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
2	40.74%	25616314	14421794
4	78.17%	14733164	3055373
6	80.10%	8284595	1566059
8	93.80%	4771986	281109
10	95.66%	2297323	94619
12	81.33%	458815	81395
14	54.22%	96955	42166
16	52.03%	62322	28403
18	24.00%	26778	19336
20	10.08%	18390	15710
Total Access		*548980637*	*521570848*

4096 Bytes/Blk		Miss Count	
# Blks	Improve.	Original	Optimized
2	62.32%	14480682	5183539
4	86.32%	7529472	978537
6	93.27%	2893864	185037
8	74.91%	359828	85762
10	33.39%	88641	56096
12	-89.68%	25067	45173
14	0.08%	16547	15708
16	-33.81%	7979	10144
18	-17.08%	5484	6100
20	-24.69%	3536	4189
Total Access		*548980637*	*521570757*

7 Conclusion

In this study, we present a refinement process to distribute bytecode handlers into logical partitions that can map to pages of NAND flash memory. The technique we used to profile the virtual machine analyzes not only the CFG of the interpreter but also the patterns of bytecode instruction streams, since we observe the input sequence drives the program flow. From this point of view, we conclude it is a kind of graph partition problem.

We use two different approaches in the experiments. By modifying either source codes or assembly codes, the refined KVMs effectively cause lower cache misses than the unmodified version. The success in source code modification even implies that our technique can help programmers to write efficient programs without the knowledge of modifying compiler-backend. Certainly, the assembly-level (or machine-code-level) rewriting tool is definitely the best solution, which provides the ultimate performance.

The most important of all, the refined virtual machine performs well on the device with limited cache memory blocks. Consider the case of 8KB/512-bytes per page, the cache miss rate of the tuned KVM is 0.6%. Compare to the 3.2% of the original KVM, this is a significant improvement. Undoubtedly, if the cache size is large, the miss rate will not be an issue. However, our approach can ensure that the KVM generates lower cache misses at smaller cache sizes. This technique also enables SOC to integrate a small block of embedded cache RAM and still execute the KVM efficiently.

Comparing our improvement on the KVM interpreter with JIT (dynamic compilation) is an interesting issue. The outcome of JIT is usually good so that it seems the effort on improving interpreter is in vain. However, a JIT VM usually consumes huge amount of memory that a small-scaled embedded device cannot afford, it is still worthwhile to refine the interpreter VM. The experimental results in [18] by Anderson Faustino da Silva *et al*. suggest that an interpreter VM is between 3 to 11 times slower than a JIT VM. However, by taking timing parameters of real NAND flash memory and DRAM into our formula, the performance boost by our improvement helps an interpreter VM runs as faster as a JIT VM.

Actually, our approach is not exclusively for interpreters. Our investigation shows our approach is applicable to the part of translating bytecodes to native codes in a JIT VM. We left this issue for future development.

Furthermore, our systematic method can apply to any program with the following two properties. First, its program flow branches to a large number of sibling sub-blocks, i.e., a big “switch… case… case…” compound statement in the interpreter. Second, the input data patterns drive the execution flows of those sibling sub-blocks, so that we can plot an ICFG to capture the dynamic trace. In practice, our approach can apply to other virtual machines, like Microsoft .NET Common Language Runtime, or an XML-driven processing program besides KVM.

8 Reference

1. Sun Microsystems. J2ME Building Blocks for Mobile Devices. Sun Microsystems, Inc. May 19, 2000.
2. C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim. A Low-Cost Memory Architecture with NAND XIP for Mobile Embedded Systems. In ISSS+CODES 2003: First IEEE/ACM/IFIP International conference on Hardware/Software Codesign and System Synthesis, ACM Press, New York, NY, October 2003.
3. Samsung Electronics. OneNAND Features & Performance. Samsung Electronics, November 4, 2005.
4. Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min. Compiler Assisted Demand Paging for Embedded Systems with Flash Memory, In Proceedings of the 4th ACM international conference on Embedded software (EMSOFT'04) (September 27–29, 2004, Pisa, Italy.). ACM Press, New York, NY, 2000, pp. 114-124.
5. Parameswaran, S., and Henkel, J. I-CoPES: Fast Instruction Code Placement for Embedded Systems to Improve Performance and Energy Efficiency. In Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design, IEEE Press, Piscataway, NJ, USA, 2001, pp. 635–641.
6. A. Janapsatya, S. Parameswaran and J. Henkel. REMcode: relocating embedded code for improving system efficiency. In IEE Proc.-Comput. Digit. Tech., Vol. 151, No. 6, November 2004
7. Karl Pettis, Robert Hansen. Profile-guided code positioning. In the Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation PLDI '90, Volume 25 Issue 6. ACM Press, New York, NY, 1990, pp. 16-27.
8. C. Young and M. D. Smith. Improving the Accuracy of Static Branch Prediction Using Branch Correlation, In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), Oct. 1994.
9. Thomas Ball and James R. Larus. Optimally profiling and tracing programs, ACM Transactions on Programming Languages and Systems, 16(4), pp. 1319-1360, July 1994.
10. Van Put, L., Chanet, D., De Bus, B., De Sutler, B., and De Bosschere, K. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In the Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005, IEEE Press, Piscataway, NJ, USA, 2005, pp. 7-12.
11. Michael Santarini. NAND versus NOR-Which flash is best for bootin' your next system? EDN October 2005. Reed Business Information, a division of Reed Elsevier Inc. October 13, 2005, pp. 41-48.
12. Micron Technology, Inc. Boot-from-NAND Using Micron® MT29F1G08ABA NAND Flash with the Texas Instruments™ (TI) OMAP2420 Processor, Micron Technology, Inc., 2006.
13. Sony Ericsson. Java™ Support in Sony Ericsson Mobile Phones. Sony Ericsson Mobile Communications AB, 2003.
14. Sébastien Lafond, Johan Lilius. An Energy Consumption Model for Java Virtual Machine, In Turku Centre for Computer Science TUCS Technical Report No 597, TUCS, March 2004.
15. CaffeineMark 3.0, Pendragon Software Corp, http://www.benchmarkhq.ru/cm30.
16. Garey M R, and Johnson D S. Computer and Intractability - A Guide to the Theory of NP-Completeness. Bell Telephone Laboratories, 1979.
17. Steven Fuber. ARM System-on-Chip Architecture (2nd Edition). Addison-Wesley Professional, August 25, 2000, pp. 49-72.
18. Anderson Faustino da Silva and Vitor Santos Costa. An Experimental Evaluation of JAVA JIT Technology. Journal of Universal Computer Science, vol. 11, no. 7 (2005), Graz University of Technology, pp. 1291-1309.