行政院國家科學委員會專題研究計畫 成果報告

低耗能 Java 虛擬機器之設計
研究成果報告(精簡版)

計 畫 類 別 ：個別型
計 畫 編 號 ：NSC 95-2221-E-002-137-
執 行 期 間 ：95 年 08 月 01 日至 96 年 07 月 31 日
執 行 單 位 ：國立臺灣大學資訊工程學系暨研究所

計 畫 主 持 人：陳俊良

計畫參與人員：博士班研究生-兼任助理：林俊杰

處 理 方 式 ：本計畫可公開查詢

中 華 民 國 96 年 12 月 12 日

行政院國家科學委員會補助專題研究計畫 ■ 成 果 報 告
□期中進度報告

低耗能 Java 虛擬機器之設計
Design of a Low Power Java Virtual Machine

計畫類別：■個別型計畫　□整合型計畫
計畫編號：NSC95-2221-E-002-137
執行期間：95 年 08 月 01 日至 96 年 07 月 31 日

計畫主持人：陳俊良

成果報告類型(依經費核定清單規定繳交)：■精簡報告　□完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
□出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：■除產學合作研究計畫、提升產業技術及人才培育研究計畫、列
管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立台灣大學資訊工程學系

中　華　民　國　96　年　11　月　　日

行政院國家科學委員會專題研究計畫成果報告

低耗能 Java 虛擬機器之設計

Design of a Low Power Java Virtual Machine

計畫編號：NSC 95-2221-E-002-137
執行期限：95 年 8 月 1 日至 96 年 7 月 31 日
主持人：陳俊良　國立台灣大學資訊工程學系

中文摘要

在行動式裝置中，嵌入式 Java 系統以及 NAND 快閃記憶體扮演了重要的角色。本研究提出了一個系統化的方法來最佳化 Java 虛擬機器；考慮的平台是該 Java 虛擬機器執行於 NAND 快閃記憶體而且只搭配少量的 RAM。我們的方法可以有效降低快閃記憶體尋頁錯失。模擬實驗顯示，針對 8KB RAM，經過我們的方法改善後，Java 虛擬機器可以減少 51%的執行時間以及 53%的耗電量。

因為 Java 虛擬機器直譯器的控制流圖有特殊的形狀，一般利用控制流圖來進行最佳化的方法無法直接引用。藉著抓取執行時的資訊，我們建立了一個特殊的控制流圖。再藉著此一特殊的控制流圖，我們重新排列 Java 虛擬機器直譯器的程式碼。排列動作的考量是想辦法讓最相關的程式碼集中在 NAND 快閃記憶體的同一頁裡，進而降低尋頁錯失的個數，以及耗電量。

關鍵詞： 低耗能、Java 虛擬機器、快閃記憶體、尋頁錯失

Abstract

Embedded Java and NAND flash memories play important roles in mobile devices. In this paper, we proposed a systematic method to optimize embedded Java virtual machines executed in NAND flash memories (XIP) with limited RAM buffer. Our approach is efficient in reducing cache misses generated by the virtual machine. The experimental simulation showed the refinement approach reduces 51% of execution time and 53% of power consumptions in the case with 8KB RAM buffer.

Because of the nature of interpreters, the shape of the control flow graph (CFG) of Java virtual machine is unique. Our approach constructed an indirect CFG by capturing dynamic bytecode traces. The Indirect CFG helped the refinement process to rearrange bytecode handlers within the source codes of Java virtual machine. The arrangement algorithm picks the most relevant program fragments and put them into NAND flash pages. This approach helps to reduce read access to NAND flash memories.

Keywords: low power, Java virtual machine, flash memory, page fault

報告內容

已發表於 2007 Inyternational Symposium on Communication and Information Technologies (ISCIT 2007), Sydney, Australia, November 2007, pp. 152-157。

計畫成果自評

已達成既定目標。

Source Code Arrangement of Embedded Java Virtual Machine for NAND Flash Memory

Chun-Chieh Lin, Chuen-Liang Chen, and Ching-Hsu Tseng
Department of Computer Science and Information Engineering
National Taiwan University
Taipei 10764, TAIWAN
{d93020,clchen,r92104}@csie.ntu.edu.tw

Abstract— **Embedded Java and NAND flash memories play important roles in mobile devices. In this paper, we proposed a systematic method to optimize embedded Java virtual machines executed in NAND flash memories (XIP) with limited RAM buffer. Our approach is efficient in reducing cache misses generated by the virtual machine. The experimental simulation showed the refining approach reduces 51% of execution time and 53% of power consumptions in the case with 8KB RAM buffer. Because of the nature of interpreters, the shape of the control flow graph (CFG) of Java virtual machine is unique. Our approach constructed an indirect CFG by capturing dynamic bytecode traces. The Indirect CFG helped the refining process to rearrange bytecode handlers within the source codes of Java virtual machine. The arrangement algorithm picks the most relevant program fragments and put them into NAND flash pages. This approach helps to reduce read access to NAND flash memories.**

I. INTRODUCTION

Java technology has become an important player in embedded systems. There are numerous Java applications designed for mobile phones. The performance of the embedded Java virtual machine (KVM in J2ME CLDC) [1] is a significant issue. Especially, the large interpreter within the JVM hungers for computation time and energy. Performance issue becomes crucial if we want to have JVMs "execute-in-place" (XIP) in NAND flash memories on mobile phones. The reason is that the cache miss penalty is extremely high in this configuration.

During the past decades, NOR flash memories have become the main stream of the "code flash" market, because a NOR flash chip can be connected to processor's host bus and it is good for programs to execute-in-place (XIP) without extra hardware. Its programming interface (erasing and writing) is quite straightforward, and designers do not have to worry about bad block management.

NAND flash memories which originally aimed for "data flash" market have gained tremendous success in storage medias like Compact Flash, SD, and MMC. The NAND flash technology has advantages in semiconductor process so that it is easier to offer higher capacity within the same chip area. Nevertheless, there are drawbacks in accessing NAND flash memories. Both read and write operations are page-oriented, and they are far slower than performing same operations on NOR flash memories. A NAND flash memory has serial bus so that it cannot be connected to CPU host bus directly. Nevertheless, the cost-effectiveness and high capacity features have made NAND flash technology an attractive replacement of NOR flash memories. NOR flash memories are cost-effective in low capacity market segments (1MB-4MB) but they lose the competence from 16 MB and beyond [2].

Although virtual machine is a special class of software, it is an important branch of system programs. Our goal was to refine interpreters and simulators, such as Java virtual machine, so that they will generate less cache misses when running on embedded devices with a limited amount of cache memories and NAND XIP. For example, system-on-chips (SOC) usually offer very limit on-chip SRAM. Our approach has several contributions in solving this kind of problem: First, it revised the code placement of bytecode handlers within the interpreter by rewriting its source code. On the contrary, similar approaches typically manipulate machine codes. Second, we introduced an aspect called "indirect CFG" to express the temporal block execution orders. Third, by observing the "indirect CFG", we concluded that the optimization of code placement is a derivation of the famous graph partition problem. The experimental result demonstrated that this approach efficiently reduced the numbers of cache misses, and it led to shorter execution time and less power consumption.

II. RELATED WORK

Park *et al*., in [3], proposed a hardware module connecting with NAND flash to allow direct program execution over NAND flash memory, as so-called execute-in-place. In this approach, program codes stored in NAND flash pages are loaded into SRAM cache on-demand instead of moving entire contents into DRAM. Their work is a universal hardware-based solution without considering application characteristics.

Samsung Electronics offers a commercial product called "OneNAND" [2] based on the same concept of above approach. It is a single-die chip with standard NOR flash interface, but inside the chip, there is a NAND flash memory array for permanent storage. The chip vendor tries to provide a cost-effective alternative to NOR flash memory used in existing designs. The internal structure of OneNAND comprises a block

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

1-4244-0977-2/07/$25.00 © 2007 IEEE

of NAND flash memory, control logics, hardware ECC, and 5KB buffer RAM (or say cache memory). The design divides the 5KB buffer RAM into three parts: 1KB for boot RAM, and a pair of 2KB buffers used for bi-directional data buffers.

Park *et al*., in [4], proposed another pure software approach to archive 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 sum up sizes of basic blocks along the calling tree. Although special hardware is no longer required, a tailor-made compiler is necessary as compared to their previous work in [3].

Conventional studies of refine code placement to minimize cache miss could extent to NAND flash cache. Parameswaran *et al*., in [5], uses the bin-packing approach. Their method reorders program codes by examining the execution frequency of basic blocks, and placing code segments with high execution frequency 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 at machine code level. The approach analyzes program flow graph, identifies and packs basic blocks within the same loop. Their approach can identify loops within the program. It is not possible to break its interpreter into individual loops for a Java virtual machine because all the loops share the same starting point.

III. BACKGROUND

A. Using NAND as Code Memory

Block-oriented access is the most significant nature of NAND flash memory. Unlike ROM and NOR flash memory, a processor cannot randomly access arbitrary byte or word in a NAND flash memory. Besides, the memory controller must transmit a series of commands to the NAND flash memory through its serial interface in order to load blocks from it. The whole procedure is complex and slow as compared with accessing to NOR.

In practice, there are obstacles in using NAND flash memories as code memory. Most implementations treat NAND flash memories as secondary storage devices like hard drives, they just load entire contents, including program code and data, from NAND flash memories to RAM before running the program. Although this is a straightforward technique, there are several drawbacks. First, it requires RAM large enough to hold everything in NAND flash memories, sometimes up to 1 GB. After system boot-up, NAND flash memories become useless. This strategy is not economic for SOC or small-scale embedded systems. Second, the system would suffer from long boot latency because it wastes a lot of time to move everything from the NAND flash to RAM. Downloading entire content from a 512MB NAND flash memory may take about 15 seconds. This is long enough to boot embedded Linux. Third, if the program code grows beyond original design, both NAND flash memories and RAM modules must upgrade together.

Yet another approach realizes XIP using NAND flash memories. Figure 1 illustrates a modest hardware configuration. Fundamental programs such as boot loaders or operating systems could permanently resident in ROM. The approach put applications and libraries, such as embedded Java environment and class files, in NAND flash memories. Based on the principle of memory hierarchy, it needs a memory management unit (MMU) and a few RAM as cache memory. The size of RAM would be enough if it were sufficient for holding a small amount of NAND flash pages. Program codes always resident in NAND flash memory. A processor will fetch instructions from the cache. When the processor is about to run a code fragment absented in the cache, MMU will load fragments of those machine codes from pages in NAND flash memory to the cache. A system may implement the MMU by either hardware (like [3]) or the virtual memory manager of an operating system. The XIP approach efficiently utilizes NAND flash memory, and retains precious RAM resource to system programs.

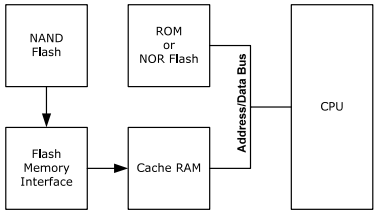

Fig. 1. Organization of a NAND/NOR mix structure

B. Basic Block Control Flow of a KVM Interpreter

The life cycle of a KVM can be divided into several parts: KVM initialize, class loading and resolving, fetching and interpreting bytecode instructions, heap memory garbage collection, cleanup and termination.

In our experiments running Embedded Caffeine Benchmark [7], the interpreter contributed 96% of total memory accesses. These evidences concluded 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 2 shows the outline of the KVM interpreter. Its skeleton is a large switch-case dispatcher enclosed by a loop which iteratively fetches bytecode instructions. Each ‘case’ sub-clause, say bytecode handler, deals with a bytecode instruction. We can split the flow of the KVM interpreter into several rough steps:

(i) Rescheduling

The virtual machine deals with thread context-switch, and stack frame adjustment.

(ii) Fetching

It fetches a bytecode instruction from memory and dispatches the job to corresponding handlers. In terms of source code, this part is a large “switch…case” compound statement (as demonstrated in Figure 2) which comprises about 200 “case” sub-clauses. After translating the interpreter into assembly code, the compiler converts the “switch” statement to a large

jumping table which dispatches program flow to distinct bytecode handlers.

(iii) Execution

It jumps to corresponding program blocks, say bytecode handlers. Each of them deals with a bytecode instruction. For complex instructions like "new" and "invokevirtual", the virtual machine jumps to supporting sub-functions. However, most instructions are simple enough so that a single code fragment can carry out a bytecode instruction. Upon completion, the virtual machine jumps back to rescheduling point and goes on for the next instruction.

(iv) Branch

This is the entry point that takes care of all branch instructions and exception handling. It alters the program counter properly according to different type of branch instructions.

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

Fig. 2. Pseudo code of KVM interpreter

IV. Problem Modeling

A. Indirect Control Flow Graph

Typical approaches derive the code placement of a program from its control flow graph (CFG). However, the CFG of an interpreter is a special case, its CFG is flat spanning tree enclosed by a loop. All bytecode handlers are sibling code blocks, and it is hard to tell the temporal order between each bytecode handler from the CFG.

There are repetitious patterns in the bytecode instruction stream executed by a virtual machine. Our research found that instruction patterns are one of the key factors caused cache misses.

Java compiler is one of the sources that generate those repetitious patterns, since it generates fixed patterns for specific Java statements. For example, Java compiler always translates an empty class constructor to a sequence of these three bytecode instructions: <aload_0, invokespecial, return>. Consider a simple experiment that uses two versions of JVMs to run the same Java application. Both JVMs are built for different code placement. JVM-A gathers the handlers of those three bytecode instructions into the same flash page, but JVM-B distributes those handlers to three individual pages. Assume 30% of the bytecode instruction trace of a Java program consists of calling to dummy class constructors. The number of cache misses generated by JVM-A is 20% less than that of JVM-B. The phenomenon indicates that the optimization strategy should consider the dynamic bytecode instruction sequence rather than analyzing the static CFG of an interpreter. Therefore, we proposed a concept called "Indirect Control Flow Graph" (ICFG); it uses the dynamic instruction sequence to construct the dual CFG of the interpreter.

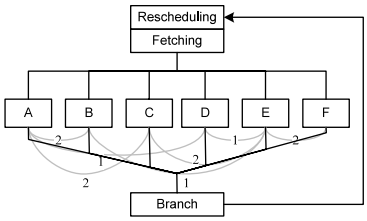

Fig. 3. Bytecode handlers calling flow

Consider a virtual machine with six bytecode instructions: A, B, C, D, E, F; the bytecode sequence of a Java application is <A, B, A, C, A, D, E, F, E, C, E, B>. In Figure 3, the graph connected by solid edges is the static CFG of the interpreter, and dashed arcs connect those six bytecode handlers upon the order of the bytecode sequence. Actually, the CFG connected by dashed edges is its ICFG as showed in Figure 4. The number of each vertex denotes the size of the bytecode hander, and the number of each edge is the time that one bytecode instruction comes after another in the instruction sequence. The ICFG is a manageable form and presents the flow between all bytecode handlers. Our approach will find the ideal code placement by partitioning the ICFG.

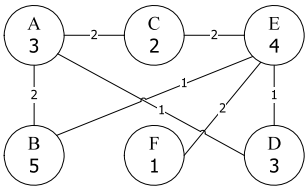

Fig. 4. The ICFG of the simplified VM

B. The Intra-Block Order That Affects Cache Misses

The code block layout of a program concerned with the sequence of loading NAND flash pages. Figure 5 illustrates the program layout in the flash memory of the simplified virtual machine. Apparently, the flash pages containing the loop header that is the rescheduling and fetching program fragments are the most frequent referenced pages. This characteristic is consistent no matter what bytecode instructions the interpreter has executed. If the cache block replacement algorithm is optimal, the flash pages contained the loop header will stay in the cache forever. Therefore, NAND flash pages that contain bytecode handlers must compete with each other to get the chance to stay in the cache.

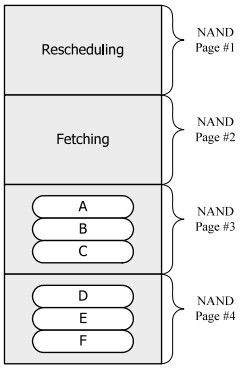

Fig. 5. Layout of code blocks of the simplified virtual machine in flash pages

Consider using two JVMs with different program layout to run the bytecode sequence as described in last section, and assume that there is only one free cache page for swapping bytecode handlers in. In version #1, bytecode handlers of A and B were placed in the same page and the others are put in distinct pages. The cache miss count will be 10. In version #2, bytecode handlers of C, E and F are placed in the same page, and the others are put in distinct pages. The cache miss count will be 8. This short example explained that properly relocating bytecode handlers to flash pages could reduce cache misses. In other words, assumed the interpreter decoded instruction A in the current iteration; there would be a cache hit if the bytecode handler of the next instruction X lived in the same page which contained the bytecode handler of A.

Thus, our research models the problem of bytecode handler placement within the JVM interpreter on NAND flash memories as the following definition. The object is to divide the ICFG of JVM into several disjoint partitions $\{G_1, G_2 \ldots G_k\}$. We defined the following notations:

- $S(i)$ – the code size of bytecode handler B_i.
- $P(I,j)$ –number of times that two bytecode instructions I and j executed after each other.
- Q – the size of one NAND flash page.
- $F(G_k)$ – the size of partition G_k. It is $\Sigma S(m)$ for all $B_m \in G_k$.
- $V(G_k)$ – the value of partition G_k. It is $\Sigma P(I,j)$ for all $Bi, Bj \in G_k$.

Our goal is to find a partition that satisfies the following constrains.

[DEFINITION] The problem is to divide G into K disjoint partitions $\{G_1, G_2 \ldots G_k\}$. For each G_k that $F(G_k) \leq Q$. Such that maximize $\Sigma V(G_i)$ for all $G_i \in \{ G_1, G_2 \ldots G_k \}$.

Actually, this problem model is equivalent to the graph partition problem. That is the size of the partition must satisfy the constraint (NAND flash page size) and the total cost of all intra-partitions edges must be minimal. The graph partition problem is NP-complete [8]. Therefore, we can find the optimal combination of bytecode handlers by using a graph partition algorithm.

How many bytecode handlers does one NAND flash page can keep? According to our experimental statistics, in a KVM built for ARM7 32-bit instruction mode, the average size of each handler is 154 bytes, and there are 60 handlers smaller than 60 bytes. That is, a 512-bytes page could roughly carry 4 to 8 bytecode handlers at a time, and there were 49 pages used to hold all bytecode handlers. A 2048-bytes page could contain more than 14 bytecode handlers, and there were 14 pages used to hold all bytecode handlers in the experiment.

C. The Process of Rewriting the Virtual Machine

As discussed in previous sections, the arrangement of bytecode handlers in the NAND flash memory affects cache miss rate. A bytecode handler is a sub-clause of "case" statement in the interpreter of KVM. 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 codes form in premeditated order. The following steps are the outline of the refining procedures.

(i) Profiling

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

(ii) 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 the intermediate files generated by the compiler.

(iii) 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,D,F) to a bundle and (B,C,E) to another.

(iv) 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. In the previous example, the order of all "case" statements is shown as Figure 6.

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

Fig. 6. The output of rearranged case statements

V. EXPERIMENT

A. Simulation Environment

The purpose of the simulation environment was to record the flash memory accesses from a J2ME KVM running Java benchmark programs on an embedded processor. In this experiment, we have ported KVM to uClinux/ARM7 [9], which is a popular Linux-variation for embedded devices, so that the experiment setup is close to real world applications. We modified the memory access module in GDB so that it will keep track of memory accesses while running uClinux. Table 3 lists all the tools used in our experiment.

TABLE 1. SOFTWARE TOOLS AND VERSIONS USED IN THE EXPERIMENT

Title	Version
uClinux distribution	20041215
Linux kernel	2.4.22-uc0
ARM binutil	2.10
ARM gcc	2.95.3
ARM gdb	5.0
uClibc	0.9.18
J2ME (KVM)	CLDC 1.1
elf2flt	20040326
Genromfs	0.5.1

We assumed there was a custom hardware unit acting as the NAND flash memory controller (as suggest in [3]) and MMU which takes response for loading NAND flash pages to a small piece of RAM buffer. This experiment assumed all the NAND flash memory operations are invisible to the operating system. In other words, our approach is platform-independent, thus it is not necessary to modify the operating system for the experiment. In the meanwhile, we chose "Embedded Caffeine Mark 3.0" [7] to be the experimental benchmark, since its bytecode instruction mix is similar to real embedded Java applications. Figure 7 illustrates the hierarchy diagram of the experimental setup. The operating system was contained in ROM. The KVM was stored in the NAND flash memory, and it loaded Java class files into RAM upon execution. The customized GDB only traced the memory accesses of KVM since our setup put the KVM in the NAND flash memory.

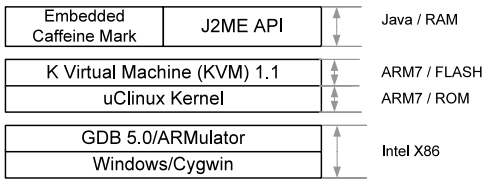

Fig. 7. Hierarchy of simulation environment

Although there are only 512-bytes and 2048-bytes NAND flash memory commodities available in the market, we assumed there are four kinds of page size: 512, 1024, 2048, and 4096 in the experiment. Therefore, the refining program generated four versions of KVM for each page size option, and the experiment collected the performance statistics from these four KVMs.

B. Experimental Result

Table 2 lists the cache miss counts generated from all configurations. The page replacement policy was FIFO. Numbers under the column "greedy" are the statistics from the KVM with refined bytecode handlers' arrangement. The "original" column refers to statistics from the original KVM, in which bytecode handlers were arranged in numeric order. In the third column, we generated a KVM whose bytecode handlers were arranged in alphabetic order of bytecode names. The column "random" are the mean values from 20 distinct random placements. The improvement ratio is a comparison between the original and refined KVM. The experiment assumed the maximal size of the cache is 64K bytes. For each different NAND flash page size, the number of cache blocks starts from four to (*64K / NAND flash page size)*.

TABLE 2. TABLE OF CACHE MISS COUNTS

		greedy	original	alphabetic	random	improv%
512	4	22,871,780	25,242,319	28,711,248	29,255,675	9.39%
	8	6,508,217	11,269,029	12,489,099	19,803,321	42.25%
	16	2,028,834	2,472,373	2,679,979	6,769,623	17.94%
	32	75,632	145,005	145,762	321,854	47.84%
	64	11,485	11,933	18,009	25,735	3.75%
	128	2,459	2,507	2,768	2,490	1.91%
1024	4	9,180,472	15,988,106	15,631,951	22,688,645	42.58%
	8	2,717,027	5,086,130	3,765,627	11,012,242	46.58%
	16	130,921	486,765	132,380	2,831,233	73.10%
	32	25,413	23,395	26,248	37,291	-8.63%
	64	2,928	3,230	4,123	4,151	9.35%
2048	4	2,373,841	10,813,688	8,611,925	17,355,743	78.05%
	8	105,157	2,341,042	1,128,333	5,367,094	95.51%
	16	25,388	68,756	70,640	137,590	63.08%
	32	4,080	4,294	4,697	5,712	4.98%
4096	4	1,992,734	4,899,778	5,050,800	12,213,574	59.33%
	8	72,580	422,512	395,942	1,063,494	82.82%
	16	6,983	8,995	12,607	12,845	22.37%

In Figure 8, each line corresponds to the improvement ratio of "greedy" to "original". The result showed that the KVM refined by our approach outperforms than all others. The improvement ratios tend to be concave curves. It meant the improvement for the case of eight pages was greater than the one of four pages. It benefit from the smaller working set size of the optimized KVM. Therefore, the cache could hold more working sets, and this is helpful in reducing cache misses.

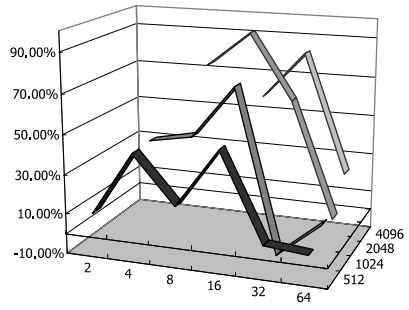

Fig. 8. Improvement ratio of cache misses. The x-axis represents the size of the cache in KB.

At the case of 1024 bytes * 32 blocks, the refined version lost the game, because there are errors between expected and eventual values of start address and code size of a bytecode handler in the compiled executable files. Since our approach rearranged the order of bytecode handlers at source level, it cannot precisely control the quality of the compilation output and generates errors. These errors are "noise" interfering with cache-hit rates.

VI. CONCLUSIONS

The experimental result showed the refined virtual machine outperformed in both execution time and power consumptions. Especially in the configurations with limited cache pages, the improved version was obviously better. This property made our approach ideal for resource-limited systems, such as system-on-chips.

The approach refines the code placement by manipulating source codes of KVM. Typically, it is a compiler's duty to do the similar operation. A compiler may deal with code placement at code generation stage. However, customizing a compiler to suit for a specific purpose is a complex work as compared with our approach. Our convincible experimental results proved that programmers would get satisfying performance simply by rewriting the source codes.

Although we assumed there was a piece of hardware MMU dedicated for buffering NAND flash memory in the experimental setup, the MMU might be a software implementation in practice. For example, an operating system can implement NAND flash execution-in-place mechanism with its virtual memory manager. Our approach is a platform-independent method so that it suits all kinds of implementations.

Our systematic method could 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 execution flow of these sub-blocks is determined by incoming data, so that we can plot an ICFG for analyzing the dynamic trace. In practice, our approach could apply to other virtual machines, like Microsoft .NET Common Language Runtime, or an XML-driven processing program.

REFERENCES

[1] Sun Microsystems. *J2ME Building Blocks for Mobile Devices*. Sun Microsystems, Inc. May 19, 2000

[2] Samsung Electronics. *OneNAND Features & Performance. Samsung Electronics*, November 4, 2005

[3] 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*, October 2003

[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, 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 Conference on ICCAD, 2001*, 635–641

[6] 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] CaffeineMark 3.0, Pendragon Software Corp, http://www.benchmarkhq.ru/cm30

[8] Garey M R, and Johnson D S. *Computer and Intractability - A Guide To The Theory of NP-Completeness*. Bell Telephone Laboratories, 1979

[9] uClinux, Embedded Linux / Microcontroller Project, http://www.uclinux.org .