Regular Paper

# Dalvik Bytecode Acceleration Using Fetch/Decode Hardware Extension

Surachai Thongkaew[1,a)] Tsuyoshi Isshiki[1,b)] Dongju Li[1,c)] Hiroaki Kunieda[1,d)]

Received: May 6, 2014, Accepted: November 10, 2014

**Abstract:** The Dalvik virtual machine (Dalvik VM) is an essential piece of software that runs applications on the Android operating system. Android application programs are commonly written in the Java language and compiled to Java bytecode. The Java bytecode is converted to Dalvik bytecode (Dalvik Executable file) which is interpreted by the Dalvik VM on typical Android devices. The significant disadvantage of interpretation is a much slower speed of program execution compared to direct machine code execution on the host CPU. However, there are many techniques to improve the performance of Dalvik VM. A typical methodology is just-in-time compilation which converts frequently executed sequences of interpreted instruction to host machine code. Other methodologies include dedicated bytecode processors and architectural extension on existing processors. In this paper, we propose an alternative methodology, "Fetch & Decode Hardware Extension," to improve the performance of Dalvik VM. The Fetch & Decode Hardware Extension is a specially designed hardware component to fetch and decode Dalvik bytecode directly, while the core computations within the virtual registers are done by the optimized Dalvik bytecode software handler. The experimental results show the speed improvements on Arithmetic instructions, loop & conditional instructions and method invocation & return instructions, can be achieved up to 2.4x, 2.7x and 1.8x, respectively. The approximate size of the proposed hardware extension is 0.03 $mm^2$ (equivalent to 10.56 Kgate) and consumes additional power of only 0.23 mW. The stated results are obtained from logic synthesis using the TSMC 90 nm technology @ 200 MHz clock frequency.

**Keywords:** Dalvik processor, Dalvik hardware extension, Android, Virtual Machine acceleration

## 1. Introduction

Android, a Linux based operating system, is currently one of the most popular and highly rated open source mobile OS platforms. It provides common standards for the communication and connectivity to most mobile devices. It allows developers to develop additional software and change and/or replace functionalities without limitations. A typical Android Architecture is shown in **Fig. 1**, which consists of a number of layers: Applications, Application framework, Libraries, Android runtime and Linux kernel. The most important part is the Android Runtime which includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language.

Every Android application runs in its own process, with its own instance of the Dalvik virtual machine (VM) [4]. Dalvik VM is a major component of the Android platform which is optimized for low memory requirements and is designed to allow multiple VM instances to run at the same time. The Java language is used to program Dalvik VM, however unlike the stack-based Java VM, Dalvik VM is a register-based architecture where the Java class files generated by the Java compiler are further transformed into .dex format [5]. This .dex file is optimized for minimal memory footprint. The Dalvik interpreter source code [6] of ARM v5TE is shown in **Figs. 2** and **3**. The core of the original version was implemented as a single C function (Fig. 2), but to improve performance they rewrote it in assembly language (Fig. 3).

The original all-in-one-function of the C version still exists as the "portable" interpreter [7], and is generated using the same sources and tools that generate the platform-specific versions. The architecture-specific configuration files determine what goes into two generated output files (InterpC-<arch>.c and InterpAsm-<arch>.S). Depending on the architecture, instruction-to-instruction transitions may be done as either "computed goto" or "jump table." In the computed goto variant, each instruction handler is allocated with a fixed-size area (e.g., 64 bytes) and the "Overflow" code will be tacked on to the end.

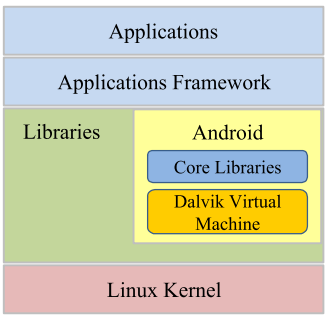


**Fig. 1** Android architecture.

[1] Department of Communications and Computer Engineering, Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[a)] surachai.th@vlsi.ce.titech.ac.jp
[b)] isshiki@vlsi.ce.titech.ac.jp
[c)] dongu@vlsi.ce.titech.ac.jp
[d)] kunieda@vlsi.ce.titech.ac.jp

```
/* File: c/OP_MOVE_FROM16.cpp*/
HANDLE_OPCODE(OP_MOVE_FROM16 /*vAA, vBBBB*/)
  vdst = INST_AA(inst);
  vsrc1 = FETCH(1);
  ILOGV("|move%s/from16 v%d,v%d %s(v%d=0x%08x)",
      (INST_INST(inst) == OP_MOVE_FROM16) ? "" : "-object"
      , vdst, vsrc1, kSpacing, vdst, GET_REGISTER(vsrc1));
  SET_REGISTER(vdst, GET_REGISTER(vsrc1));
  FINISH(2);
OP_END
```

**Fig. 2** Typical Dalvik interpreter C source code.

```
/* File: armv5te/OP_MOVE_FROM16.S */
 .balign 64
.L_OP_MOVE_FROM16: /* 0x02 */
/* for: move/from16, move-object/from16 */
   /* op vAA, vBBBB */
   FETCH(r1, 1)                  @ r1<- BBBB
   mov     r0, rINST, lsr #8     @ r0<- AA
   FETCH_ADVANCE_INST(2) @ advance rPC, load rINST
   GET_VREG(r2, r1)              @ r2<- fp[BBBB]
   GET_INST_OPCODE(ip)           @ extract opcode from rINST
   SET_VREG(r2, r0)              @ fp[AA]<- r2
   GOTO_OPCODE(ip)               @ jump to next instruction
```

**Fig. 3** Typical Dalvik interpreter Assembly source code.

In the jump table variant, all of the instruction handlers are contiguous and may vary in sizes. Therefore, as shown in Fig. 3, the interpretation of one bytecode consists of many host machine codes.

The interpretation starts fetching the *first byte* of the bytecode to compute the corresponding handler address and then jumping to that opcode handler. Inside the opcode handler, the remaining bytecode is fetched, then each bytecode field is translated into operand fetch, computation and writeback within the virtual registers. Here, the bytecode size varies in lengths of 2, 4, 6 or 10 bytes, and thus affect the remaining bytecode fetching process. The last part of the opcode handler deals with fetching the first byte of the next bytecode, computing the corresponding handler address and finally jumping to the next opcode handler. As we can see here, each opcode handler that corresponds to executing a single Dalvik bytecode requires many host machine codes, and therefore is much slower than executing native-compiled codes. In order to accelerate the Dalvik VM execution, several approaches are proposed that can be classified into three categories as follows: software acceleration, dedicated bytecode processor and co-processor, and architectural extension.

(1) Software acceleration: this approach utilizes software techniques, such as Just-In-Time Compiler, to accelerate bytecode interpretation on top of the existing VM.
(2) Dedicated bytecode processors and co-processors: these are designed to execute a large portion of bytecodes directly as part of machines' native instruction set.
(3) Architectural extension: this approach utilizes dedicated hardware logic added on top of an existing processor for executing the bytecode.

Our proposed technique in this paper is categorized as the architecture extension, where we have added a dedicated hardware logic for fetching and decoding Dalvik bytecode that is designed to drastically reduce the operation steps in the opcode handler software.

This paper is organized as follows. Section 2 discusses the related works on VM accleration techniques in detail. Our proposed hardware extensions and software optimization technique is described in Section 3. Experimental results of our proposed technique applied to our processor (TCT processor) and also on an ARM processor is given in Section 4. Future works and conclusion are summarized in Sections 5 and 6 respectively.

## 2. Related Works

Related works cover the works associated with improving performance of both the Dalvik VM and Java VM. There are similarities between both the Dalvik VM and Java VM; both techniques accelerate the speed of bytecode interpretation during the runtime phase. The related works can be classified into 3 groups as mentioned in the introduction section.

### 2.1 Software Acceleration

For the software acceleration approach, a widely used bytecode interpretation technique is the JIT (Just-in-time) compiler. JIT compilation [8], [9] attempts to bridge the gap between the two approaches to program translation: compilation and interpretation. In the JIT compilation process, starting with the interpreter, some features of a static compiler are built into the system. Typically, a JIT compiler will isolate some sections of the code at run-time which are accessed more often and then compile them to native code, aggressively optimizing the above in the process. The sections of code that are to be statically compiled can be identified in many ways and these sections of code are commonly called hot-paths. Hot-path detection is done at either the method level or at the trace (or a string of instructions which start with a loop head) level. In method based JIT, as the name suggests, the potential hot-paths are marked at the beginning of each method implementation. However, the trace-based JIT compiler for a Dalvik VM [1] has been implemented on Android 2.2. The trace-based JIT [10] compiler is the most prevalent and effective method which compiles the sections of the code that are most likely to frequently be called. These could include certain obvious choices like targets of backward branches [11]. The trace is ended when it forms a cycle in the buffer, executes another backward branch, calls a native method or throws an exception [12]. These potential traces are profiled with some additional meta-data to keep track of their execution counts.

Although JIT compilers are effective for desktop computers, JIT compilers are not appropriate to provide acceleration mechanisms for mobile applications because they require extra resources [13]. Typical compilers are more than 100 Kbytes in size, and compiled code typically expands by a factor of six or eight times, requiring a large RAM cache. A JIT compiler is typically slow to initiate, resulting in pauses and user input disruptions. JIT compilers also make heavy demands on the CPU during the compilation phase which means greater memory requirements, more processing power and ultimately more expenses.

### 2.2 Dedicated Bytecode Processor and Co-processor

The related works of dedicated bytecode processor and co-processor techniques have not officially been found for Dalvik VM but often found for Java VM; for instance, Sun's picoJava and JSTAR, etc.

Sun's picoJava [14] is the Java processor most often cited in research papers. The first version of picoJava was introduced in 1997. The processor was targeted at the embedded systems market as a pure Java processor with restricted support of C. Simple Java bytecode is directly implemented in hardware, and most of them execute in one to three cycles. Other performance critical instructions, for instance invoking a method, are implemented in microcode. The picoJava stuck on the remaining complex instructions, such as creation of an object, and emulates this instruction. To access memory, internal registers and for cache management, picoJava implements 115 extended instructions are 2-byte opcodes. These instructions are necessary to write system-level code to support the JVM. The architecture of picoJava is a stack-based CISC processor, which can be implemented within 440 Kgates (128K for the logic and 314K for the memory component: 284x80 bits microcode ROM, 2x192x64 bits FPU ROM and 2x16 KB caches).

Although Dedicated Java processors appear to offer acceptable performance but they represent a significant overhead and additional integration and development complexity [13]. They do not support existing applications or established operating systems, they must always operate alongside another processor.

For Nazomi JA108 [15], previously known as JSTAR, a Java co-processor sits between the native processor and the memory subsystem. JA108 fetches Java bytecodes from memory and translates them into native microprocessor instructions. JA108 acts as a pass-through when the core processor's native instructions are being executed. The JA108 is targeted for mobile phone usages to increase performance of Java multimedia applications. The co-processor is available as a standalone package or with included memory and can be operated up to 104 MHz. The resource usage for JSTAR is known to be approximately 30 Kgate plus 45 Kbit for the microcode.

Java co-processors translate Java bytecode into the existing core's instructions. This acceleration process often requires a significant hardware and software integration effort and is difficult to incorporate into the existing OSs. Co-processors also require extra space for the gates, and extra power to operate, and are expensive to manufacture. In addition, they tend to run relatively slowly because they are loosely coupled with the core processor.

### 2.3 Architectural Extension

Unlike the dedicated bytecode processors explained in the previous subsection, its architectural extension is an approach in which a dedicated bytecode hardware is extended on top of the existing processors. Jazelle [16] is an hardware extension (of the ARM 32-bit RISC processor) to execute Java bytecode, similar to the Thumb state (a 16-bit mode to reduce memory consumption) [17]. The Jazelle DBX (Direct Bytecode eXecution) is integrated into the ARM processor. The hardware bytecode decoder logic is implemented in less than 12 Kgate and 140 Java bytecodes are executed directly in hardware, while the remaining are emulated by sequences of ARM instructions. This solution also uses code modification with quick instruction to substitute certain object related instructions after link resolution. All Java bytecode, including the emulated sequences, are re-startable to enable a fast interrupt response time. A new ARM instruction (BXJ), puts the processor into Java State. Bytecodes are fetched and decoded in two stages, compared to a single stage in the ARM state. Four registers of the ARM core are used to cache the top stack elements. Stack spill and fill is handled automatically by hardware. Additional registers are reused for the Java stack pointer, the variable pointer, the constant pool pointer and locate variable 0 (the pointer in method). Keeping the complete state of the Java mode in ARM registers simplifies its integration into the existing operating system.

A Dalvik Accelerator [2] is a hardware extension to accelerate execution in a Dalvik VM, an additional pipeline execution path for direct execution of Dalvik bytecodes as native codes is added. Thus, instruction Fetch and Decode stages for Dalvik bytecode have been added in addition to the existing instruction Fetch and Decode stages in a pipeline of the processor. A Dalvik Accelerator and existing native code decoder are selected by a selector according to value of status register to change executing modes of the processor. A dedicated instruction which changes execution modes is added to a processor for the mode to be migrated by controlling the status register. Each Dalvik bytecode which has variable length is fetched in the Fetch stage. Then, the corresponding native instruction codes, which consist of 1 to 14 instructions, are generated in a Decode stage from the Dalvik bytecode. Since such switching overhead is accumulated in execution time, the execution speed decreased during frequent switching. Therefore, two methods were proposed to eliminate the above switching overhead. First is the bytecode prediction, whether a Dalvik bytecode can be executed by hardware or not. If the instruction is not continued, the mode migration is avoided. The second method, a register set which is used by a Dalvik VM and a Dalvik Accelerator at the same time is duplicated in order for the registers to be saved and restored independently. The above implementation is called the "Register Window" to switch the required register sets.

ARM-Jazelle and Dalvik Accelerator are an enhanced single processor solution that directly executes Java/Dalvik bytecode alongside existing OSs, middle ware and application code. By placing an additional instruction set (to support the architectural extension) inside the processor, an architectural extension can reuse all existing processor resources without the need to re-engineer existing the architecture or add cost, power or budget resources. An extended core can efficiently run both bytecode and native code, giving developers the ability to leverage the existing base of applications and operating system expertise while achieving a successful balance of Java portability and native performance for their application.

## 3. Proposed Solution and Architecture

According to the analysis of strengths and weaknesses of various techniques to optimize the efficiency of VM execution, we have chosen to explore the architectural extension approach and introduce new architecture and software optimization techniques for a practical solution. The strength of ARM-Jazelle and Dalvik Accelerator are hardware extensions in an existing processor which are used for fetching, decoding and executing bytecode di-

**Table 1** Comparison of hardware extension techniques.

| Operation | ARM-Jazelle | Dalvik Accelerator | Proposed technique |
|---|---|---|---|
| **VM supporting** | Java VM | Dalvik VM | Dalvik VM |
| **Fetching** | HW | HW<br>Predictor | HW<br>*Dalvik bytecode fetch logic (DFE)* |
| **Decoding** | HW | HW | HW<br>*Dalvik Decode logic (DDC)* |
| **Executing**<br>**-Simple bytecode**<br>**-Complex bytecode** | <br>HW<br>Handler SW | <br>HW<br>Handler SW | <br>*Optimized handler SW*<br>*Optimized handler SW* |

rectly. However, both ARM-Jazelle and Dalvik Accelerator can execute only simple Java/Dalvik bytecodes directly, while the remaining complex instructions, such as creation of instances and invoking method need a large amount of complex accessing in a virtual machine. Thus, the complex instructions are emulated by handler software which is the sequence of native instructions. Comparison of hardware extension techniques of ARM-Jazelle, Dalvik Accelerator and our proposed techniques are summarized in the **Table 1**.

- Fetch and decode of bytecodes are done on dedicated hardware logic for all three cases.
- While ARM-Jazelle and Dalvik Accelerator contains dedicated hardware logic for executing simple bytecodes, our technique utilizes optimized handler software to reduce the software execution overhead while maintaining the hardware overhead small. The drawback of our approach is the limited acceleration of simple bytecodes, whereas the strength is that our technique can be utilized to accelerate complex bytecodes as well.

In the next subsection, we summarize our overall proposed techniques, and then explain each details in the following subsections.

### 3.1 Summary of Architectural Features and Software Optimization Techniques

**Table 2** shows a typical Dalvik bytecode handler, which corresponds to "move vA, vB" bytecode. The left column shows the original SW handler codes and the right column shows the optimized SW handler codes utilizing the proposed HW extensions. As shown in the Table 2, each opcode handler consists of the following steps.

( 0 ) At the entry of the opcode handler, the current byte code is already fetched and decoded.
( 1 ) Load source/destination operand addresses (vreg indices) to physical registers.
( 2 ) Fetch next byte code.
( 3 ) Load source operands from memory and compute result.
( 4 ) Write back result to destination operand address.
( 5 ) Decode next bytecode and jump to the corresponding SW handler address.

Our proposed HW extensions are designed to eliminate steps 1, 2 and 5 of the above SW opcode handler codes, which are summarized as follows:

- Dalvik bytecode fetch logic (DFE) for one cycle bytecode fetch operation on frequently used 2-byte and 4-byte bytecodes (eliminate step 2). For 6-byte and 10-byte bytecodes, it requires additional cycles to fetch from 4-byte wide program memory.

**Table 2** Typical Dalvik bytecode execution with proposed technique.

| **move vA, vB instruction** | |
|---|---|
| **Original Dalvik handler codes** | **Optimized Dalvik handler with proposed HW extension** |
| **Read source operand index (step 1)** | |
| SHR R1,R7,12,0 *@ r1<- B from 15:12* | **Pre-loader (DDC) code removed** |
| **Read destination operand index (step 1)** | |
| SHR R0,R7,8,0 *@ r0<- A from 11:8* | **Pre-loader (DDC) code removed** |
| **Fetch next bytecode (step 2)** | |
| ADD R4,R4,2 *@ advance rPC, load rINST*<br>LDMUS R7,R4 | **Dalvik Fetch (DFE) code removed** |
| **(Compute Result)** | |
| **Load source operand value (step 3)** | |
| SHL BRP2,R1,2 *@ r2<- fp[B]*<br>ADD BRP2,SP,R1<br>LDM R2,BRP2<br>AND R0,R0,15 | SHL BRP2,R1,2 *@ r2<- fp[B]*<br>ADD BRP2,SP,R1<br>LDM R2,BRP2<br>AND R0,R0,15 |
| **(Write back destination)** | |
| **Store source value to destination operand (step 4)** | |
| SHL BRP2,R0,2 *@ fp[A]<- r2*<br>ADD BRP2,SP,R0<br>ADD BRP2,SP,BRP2<br><br>STM R2,BRP2 | SHL BRP2,R0,2 *@ fp[A]<- r2*<br>ADD BRP2,SP,R0<br>ADD BRP2,SP,BRP2<br>**BXD 0** @For switch to Dalvik mode<br>STM R2,BRP2 @ Delay-slot scheduling |
| **Extract next opcode (step 5)** | |
| AND R12,R7,255 *@ ip<- opcode from rINST* | **Bytecode classifier (DDC) code removed** |
| **Compute next handler address (step 5)** | |
| SHL R12,R12,6<br>ADD R0,R8,R12<br>JMP_REG R0 | **Handler address computing unit (DDC) code removed** |

**Table 3** The goto instruction execution with proposed technique.

| **goto AA instruction** | |
|---|---|
| **Original Dalvik handler codes** | **Optimized Dalvik handler with proposed HW extension** |
| **Read target address (step 1)** | |
| SHL R0,R7,16 *@ r0<- AAxx0000*<br>SHR R1,R0,24,1 *@ r1<- ssssssAA (sign-extended)*<br>ADD R2,R1,R1 *@ r2<- byte offset, set flags*<br>ADD R4,R2,R4 *@ update rPC, load rINST* | **goto-instructions controller (DDC) code removed** |
| **Fetch next bytecode (step 2)** | |
| LDMUS R7,R4 *@ advance rPC, load rINST* | **Dalvik Fetch (DFE) code removed** |
| **Extract next opcode (step 3)** | |
| AND R12,R7,255 @ ip<- opcode from rINST | **Bytecode classifier (DDC) code removed** |
| **Compute next handler address (step 3)** | |
| SHL R12,R12,6<br>ADD R0,R8,R12<br>JMP_REG R0 | **Handler address computing unit(DDC) code removed** |

- Dalvik bytecode decode logic (DDC) for calculating the corresponding SW handler address (eliminate step 5) and preloading source/destination operand address (Vreg indices) to physical registers (eliminate step 1).

Steps 3 and 4 are executed in native mode, where eliminated steps 1, 2 and 5 are executed in the explained HW extensions in Dalvik mode. Switching from Dalvik mode to native mode is done automatically by the Dalvik HW logic, whereas switching from native mode to Dalvik mode is done by BXD instruction that is added to the native mode instruction set.

**Table 3** shows a special Dalvik bytecode handler for goto instruction whose task is simply to jump to the bytecode location specified by the constant offset value. This goto bytecode handler consists of the following steps.

( 1 ) Compute the location of the goto target bytecode (add offset value to the current Dalvik program counter).
( 2 ) Fetch next byte code.
( 3 ) Decode next bytecode and jump to the corresponding SW handler address.

While steps 2 and 3 are eliminated by the proposed Dalvik fetch logic (DFE) and Dalvik decode logic (DDC), another HW extension called "Goto instructions controller" is added to elimi-

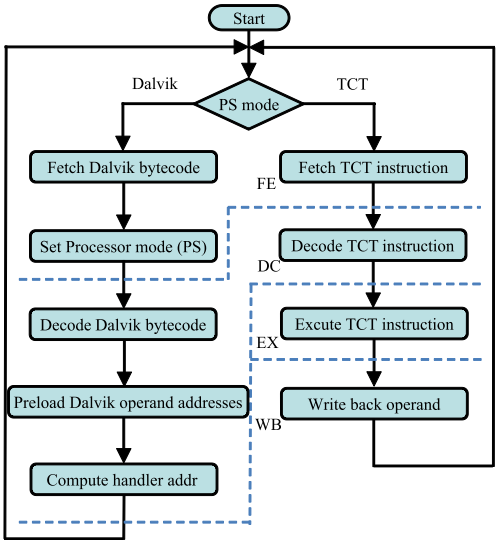


**Fig. 4** Behavior of the proposed processor architectural framework.

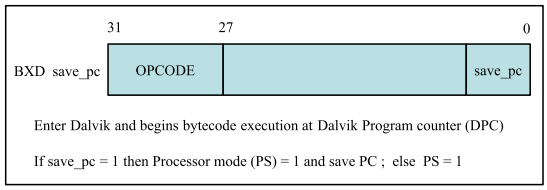


**Fig. 5** Change to execute Dalvik instruction (BXD).

nate step 1 as well, thereby totally elimination the need to switch to native mode in this case.

We extended hardware to our existing in-house processor which is a Tightly-Coupled-Thread (TCT) processor [18], [19]. The TCT Processor is a Harvard architecture, 4-stage pipelines, 32-bit RISC with load-store architecture. The four pipeline stages are instruction fetch (FE), instruction decode (DC), execute and memory-read access (EX), and (register or data memory) write back (WB). Multiplication and division operations are implemented in the ALU using multi-cycle modules. Behavior of the proposed processor architectural framework is shown in **Fig. 4**.

The data processing begins with the examination of processor modes (either Dalvik or TCT mode). If it is TCT mode, the processor will process as a normal procedure in 4 pipeline stages, which are fetch, decode, execute and write back. On the other hand, if it is Dalvik mode, the processor will fetch Dalvik bytecode, set processor mode, decode Dalvik bytecode, preload Dalvik operand and compute handler address respectively. The processor can switch from TCT mode to Dalvik mode via particular instruction of the TCT processor which is a change to execute Dalvik bytecode (BXD) instruction as shown in **Fig. 5**. The principle function of this instruction is to configure the processor status (PS) signal and then Dalvik bytecode is fetched at Dalvik the Program Counter (DPC) in the next cycle. The BXD instruction also performs a test on the save_pc bit. If the save_pc bit is set, the processor then will store the current PC for returning to main TCT mode. If it is not, the current PC will not be stored for returning to the Dalvik handler in TCT mode.

The mode switching from Dalvik to TCT is done automatically by Dalvik bytecode fetch logic (DFE: explained in section

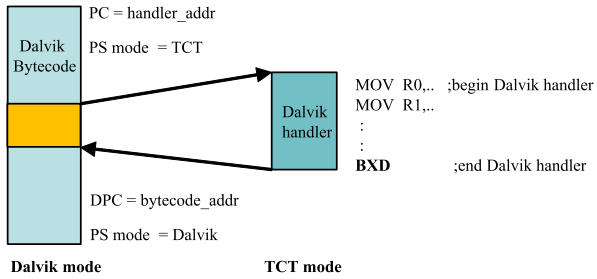


**Fig. 6** Proposed mode switching.

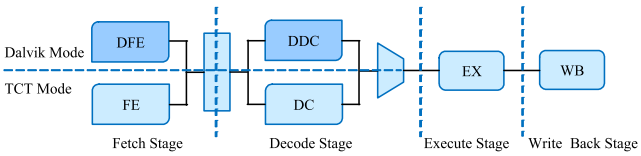


**Fig. 7** Processor pipeline structure with Dalvik hardware extension.

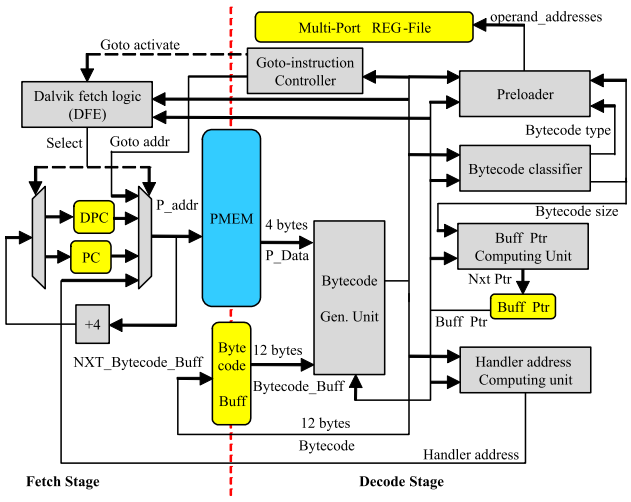


**Fig. 8** Block diagram of Dalvik hardware extension.

3.3), and therefore, the overall process of the processor can now operate in dual modes. The first mode is TCT instruction execution with 4-stage pipelined (TCT mode) and the second mode is Dalvik bytecode execution (Dalvik mode). This processor could be called "a 2 in 1 processor" as shown in **Fig. 6**. The PS mode is set to TCT automatically by DFE for executing the Dalvik handler and then PS is set to Dalvik by the BXD instruction for switching back to fetch the next Dalvik bytecode. The implementation of proposed hardware is shown in **Figs. 7** and **8**. The proposed extension is an acceleration in executing Dalvik bytecode to the existing processor without degrading the original architecture.

### 3.2 Internal Registers

#### 3.2.1 Dalvik Program Counter (DPC)

Dalvik Program Counter (DPC) acts as pointer to fetch the next bytecode. By adding the above program counter, switching between Dalvik and TCT can be rapidly done without any typical saving and restoring program counter before switching mode, since each mode has its own program counter.

#### 3.2.2 Bytecode Buffer (Bytecode_Buff) and Buffer Pointer (Buff_Ptr)

Bytecode buffer (Bytecode_Buff), which is located on the

pipeline boundary between fetch (FE) and decode (DC) stages, has a width of 12 bytes and acts as an instruction register in native processor mode (32-bit instruction) and as temporary bytecode storage for Dalvik mode. In Dalvik mode, bytecode fetch occurs on the 4-byte word boundary from PMEM (P_data in Fig. 8) which is the same for native mode, where the fetched P_data and Bytecode_Buff is appropriately shifted and combined to generate the current bytecode and the next Bytecode_Buff data (NXT_Bytecode_Buff). Shift operation on P_data and Bytecode_Buff is controlled by a buffer pointer (Buff_Ptr in Fig. 8) which keeps track of how many bytes reside in Bytecode_Buff.

As we will explain in the following subsections, Dalvik bytecode fetch logic (DFE) automatically performs the 4-byte fetch from PMEM as long as Bytecode_Buff contains 8 or less bytes after decoding, and Dalvik bytecode decode logic (DDC) automatically performs the decoding as long as fetched P_data and Bytecode_Buff contains the entire data of the current bytecode. With this scheme, 2-byte and 4-byte bytecodes can be fetched in one cycle, whereas 6-byte and 10-byte bytecodes require an additional one or two cycles, depending on how many bytes reside in Bytecode_Buff.

### 3.3 Dalvik Bytecode Fetch Logic (DFE)

Two principle operations of Dalvik bytecode fetch logic (DFE) are fetching Dalvik bytecode of variable length formats and switching to TCT mode. The Dalvik bytecode have unit size of 1, 2, 3, 5 times larger than 2 bytes (2, 4, 6, and 10 bytes). The majority of the instruction sizes are 2 and 4 bytes. A 4 bytes fetching is implemented in DFE, this allows a faster execution than those of 2 bytes fetching. The Dalvik DFE will activate the PS mode signal instantly for fetching TCT instruction in the next cycle. If Dalvik bytecode fetching meets the requirements to execute one Dalvik instruction then the processor mode is changed to TCT immediately by default.

### 3.4 Dalvik Decode Logic (DDC)

The DDC block consists of 4 components, which are Dalvik bytecode classifier, operand address preloader, Dalvik handler address computing unit and goto-instructions controller, respectively. They are described as follows:

#### 3.4.1 Dalvik Bytecode Classifier

Dalvik bytecode classifier categorizes the instruction into 2 main groups based on the length of the instruction and the type of operand loading in Dalvik handler. The length of the instruction is divided into 4 types, they are 2, 4, 6 and 10 bytes and after the classifier indicates the length and type of the instruction, the operand address preloader and the bytecode buffer pointer will be loaded with an appropriate value accordingly.

#### 3.4.2 Operand Address Preloader

Since Dalvik virtual registers (`vreg`) are mapped to memory space in the Dalvik VM, it requires two steps to access one Dalvik VM operand on `vreg`. The first step is a taking Dalvik register addresses (`vreg` indices) from Dalvik instruction, which is reading Dalvik register addresses from program memory to physical registers. The second step is loading the register values from data memory (Dalvik virtual register) to physical registers. For this reason, various Dalvik operand addresses (`vreg` indices) are written into physical registers before accessing the Dalvik virtual registers during Dalvik handler is being executed. The operand address preloader hardware processes as first step, which is taking Dalvik register addresses (`vreg` indices) from Dalvik instruction to physical registers. The second step (loading operand values) will be done in optimized handler. Therefore, the addition of the operand address preloader into the processor will increase the speed of interpretation. The operand address preloader hardware processes the first step, that is, writing the Dalvik register addresses (`vreg` indices) to physical registers. The multi-port register file is used in the proposed technique in order to write various Dalvik operand addresses into various physical registers within one cycle. The writing of Dalvik operand addresses to physical registers in different handler were categorized by the type of handler source code [6] and format of Dalvik instruction [20] in order to make a minimal size of preloader.

#### 3.4.3 Dalvik Handler Address Computing Unit

The handler address computing unit will compute the handler address using technique called “compute goto.” This technique will define the size of handler code in the fixed size of 64 bytes and overflow code will be attached at the end of handler table. Therefore, the computation of the base address of each handler can be done with obtained fetched opcode and then multiply by 64, which can be implemented by the simple shift-left hardware.

The computed handler address in the decode stage is delivered via a direct feedback path to TCT fetch logic (FE) as shown in Fig. 8. This allows the first TCT instruction of the Dalvik bytecode handler to be fetched without causing any pipeline bubbles when switching from Dalvik to TCT mode.

#### 3.4.4 Goto Instructions Controller

The goto-instructions controller (as shown in **Fig. 9**) is the alternative hardware on the TCT processor to obtain faster bytecode interpretation for the goto-instruction group. When processor is decoding goto-instructions in the decode stage, the goto-instructions controller sends value of the destination address which obtained from goto-instructions to Dalvik bytecode fetch logic (DFE). The DFE will fetch the Dalvik instruction from destination address which obtained from goto-instructions controller promptly send them to the Dalvik decode unit (DDC). When the

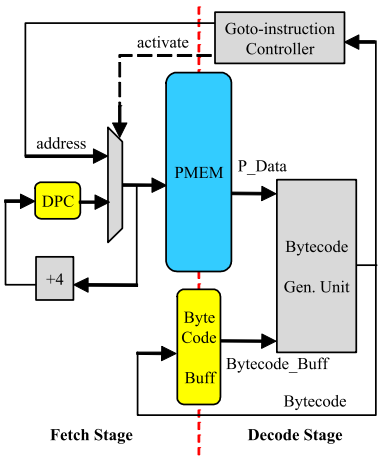


**Fig. 9** Proposed goto instructions controller.

**Table 4** Native Instruction comparison between TCT & TCT+DXT.

| Dalvik Instructions | | Native Instruction count | |
|---|---|---|---|
| **Opcode** | **Mnemonic** | **TCT** | **TCT+DXT** |
| 0x01 | move vA,vB | 16 | 5 |
| 0x07 | move-object vA, vB | 16 | 5 |
| 0x0A | move-result vAA | 11 | 4 |
| 0x0E | return-void | 25 | 23 |
| 0x0F | return vAA | 20 | 18 |
| 0x12 | const/4 vA,#B | 14 | 3 |
| 0x13 | const/16 vAA,#B | 18 | 3 |
| 0x15 | const/high16 vAA,#BBBB0000 | 14 | 7 |
| **0x28** | **goto +AA** | **10** | **1** |
| 0x32 | if-eq vA, vB, +CCCC | 22 | 7 |
| 0x35 | if-ge vA, vB, +CCCC | 22 | 7 |
| 0x71 | invoke-static | 75 | 69 |
| 0x84 | lont-to-int vA,vB | 15 | 5 |
| 0x90 | add-int vA,vB | 22 | 8 |
| 0x9B | add-long vA,vB | 31 | 18 |
| 0xB7 | xor-int vAA,vBB,vCC | 20 | 9 |
| 0xC2 | xor-long vAA,vBB,vCC | 25 | 15 |
| 0xD8 | add-int/lit8 vAA, vBB, #+CC | 19 | 6 |

* DXT = Dalvik hardware extension

```
op0x13: ;/* const/16 vAA, #+BBBB */
.balign 64

;FETCH_S(r0, 1) @ r0<- ssssBBBB (sign-extended)
ADD             BRP2,R4,2
LDMSS           R0,BRP2
;@ r3<- AA
SHR             R7,R7,8,0
MOV             R3,R7
;FETCH_ADVANCE_INST(2) @ advance rPC, load rINST
ADD             R4,R4,4
LDMUS           R7,R4
;SET_VREG(r0, r3)        @ vAA<- r0
SHL             R3,2
ADD             BRP2,SP,R3
STM             R0,BRP2
;GET_INST_OPCODE(ip)     @ extract opcode from
rINST
AND             R12,R7,255
;GOTO_OPCODE(ip)         @ jump to next instruction
SHL             R12,R12,6
ADD             R0,R8,R12
JMP_REG         R0
```

**Fig. 10** Original Dalvik handler.

```
op0x13: ;/* const/16 vAA, #+BBBB */
.balign 64

;SET_VREG(r0, r3)        @ vAA<- r0
ADD             BRP2,SP,R3
BXD             0
STM             R0,BRP2
```

**Fig. 11** Software handler after 2nd step optimization.

goto-instruction is being decoded, goto instruction controller will activate PS mode signal to Dalvik mode. This implies that it is unnecessary to use handler in execution for the goto-instruction group that means the goto-instruction is able to work directly with the extended hardware. This finding is confirmed by the number of goto-handler is 1 in **Table 4** after the optimization. Moreover, the designed goto-instructions controller works accordingly with bytecode buffer and buffer pointer. When the goto-instructions are encountered, the bytecode buffer will be flushed and next instruction is loaded from jump address to the bytecode buffer automatically. However, the goto-instruction controller can accelerate only the goto-instructions group interpretation, the conditional branch instructions (if-eg, if-ne… etc.) will be executed with optimized handler.

The goto-instructions controller hardware was implemented in the original TCT processor [18], [19] which can run properly at 200 MHz. It could be the alternative to apply with the modern mobile-class processor, the Dalvik goto-instruction group can be executed by optimized handler and along with proposed Fetch & Decode hardware extension without the goto-instructions controller hardware in case of the modern mobile-class processor does not match up with the designed goto-instructions controller.

### 3.5 Dalvik Handler Optimization

Together with the previously explained hardware extensions, software handler codes need to be optimized to remove redundant operations already implemented in extended hardware logic. This software handler optimization process is already outlined in Table 2 previously. Here, we will use **Figs. 10** and **11** to explain the optimized handler codes.

The Figs. 10 and 11 show the typical example of handler software optimization. Figure 10 is a original Dalvik handler on the TCT processor that handles const/16 instruction which consists of 13 instructions before the optimization takes place. The 1st step optimization is a reduction of 8 instructions in the extended Dalvik fetcher and Decoder by removing the instruction that fetches remained bytecode, fetches next bytecode, computes next handler address and jump to next opcode handler. Moreover, Fig. 11 shows the preloader can reduce 3 instructions (SHR, MOV and SHL) on the 2nd step optimization. A comparison between the original Dalvik handler and the optimized Dalvik handler is shown in Table 4. The result confirms that the Dalvik instruction interpretation on TCT processor with Dalvik hardware extension (DXT) uses less number of instructions than those of the original TCT processor due to the proposed Dalvik hardware extension. For example, the move vA, vB instruction is interpreted with 16 instructions on the original TCT processor but it needs only 5 instructions interpretation on the TCT processor with proposed Dalvik hardware extension.

### 3.6 Mode Switching Overhead Elimination Technique

According to the move vA, vB (Dalvik instruction) is used as an example in Table 2, this section will also use move instruction to explain the overhead elimination. **Table 5** shows the sequence of Dalvik and TCT instructions,which are processed in the TCT processor with proposed hardware extension. The symbol column shows the instruction type (D = Dalvik, T = TCT) and sequence number of instructions, the instructions column shows the sequence of Dalvik and TCT instructions. The [T_1 to T_9] are TCT instructions in the optimized handler of move Dalvik instruction [D_0], [D_10] (const/16 vAA, #+BBBB) is the next Dalvik instruction and [T_11 to T_n] are TCT instructions in the optimized handler of const/16 Dalvik instruction [D_10]. The operation rough steps of Dalvik bytecode execution in the processor with proposed hardware extension can be described as follows:

(1) Enter to Dalvik mode for fetch and decode bytecode.
(2) Switch to TCT mode which can be done by Dalvik fetch logic (DFE).
(3) Enter to TCT mode to execute Dalvik handler.
(4) Switch to Dalvik mode which can be done by BXD instruc-

**Table 5** Sequence of instruction execution.

| Symbol | Instructions |
|---|---|
| [D_0] | move vA, vB |
| [T_1] | SHL BRP2, R1, 2 |
| [T_2] | ADD BRP2, SP, R1 |
| [T_3] | LDM R2, BRP2 |
| [T_4] | AND R0, R0, 15 |
| [T_5] | SHL BRP2, R0, 2 |
| [T_6] | ADD BRP2, SP, R0 |
| [T_7] | ADD BRP2, SP, BRP2 |
| [T_8] | BXD 0 for swich to Dalvik |
| [T_9] | STM R2, BRP2 (delay-slot of BXD) |
| [D_10] | const/16 vAA, #+BBBB |
| [T_11] | ADD BRP2, SP, R3 |
| ... | ... |

* D_x = Dalvik instruction number x
* T_y = TCT instruction number y

**Table 6** Switching overhead elimination.

| Time | Fetch | Decode | Execute | Write back |
|---|---|---|---|---|
| **0** | [D_0] move | ... | ... | ... |
| **1** | [T_1] SHL | [D_0] move | ... | ... |
| **2** | [T_2] ADD | [T_1] SHL | <NOP> | ... |
| **3** | [T_3] LDM | [T_2] ADD | [T_1] SHL | <NOP> |
| ... | ... | ... | ... | ... |
| **7** | [T_7] ADD | [T_6] ADD | [T_5] SHL | [T_4] AND |
| **8** | [T_8] BXD | [T_7] ADD | [T_6] ADD | [T_5] SHL |
| **9** | [T_9] STM | [T_8] BXD | [T_7] ADD | [T_6] ADD |
| **10** | [D_10] const/16 | [T_9] STM | <NOP> | [T_7] ADD |
| **11** | [T_11] ADD | [D_10] const/16 | [T_9] STM | <NOP> |
| ... | ... | [T_11] ADD | <NOP> | [T_9] STM |

tion in TCT Decode logic (DC).

From those steps, one concern with our proposed HW extension scheme of having Dalvik mode for fetch and decode together with native (TCT) mode for SW handling is the overhead for mode switching, since each Dalvik bytecode execution requires switching from Dalvik mode to native mode and then back to Dalvik mode again. Here, we propose two techniques for eliminating the mode switching overheads as follows:

- Dalvik to Native mode switching controlled by Dalvik bytecode fetch logic (DFE).
- Native to Dalvik mode switching implemented by BXD instruction in SW handler, where delay-slot instruction scheduling technique is applied to hide mode switching latency.

#### 3.6.1 Dalvik to TCT Mode Switching Overhead Elimination

The Dalvik to TCT mode switching overhead elimination can be summarized as follows:

- Mode switch from Dalvik to TCT activates after the whole bytecode has been fetched within one to three cycles (it depends on size of Dalvik instruction and the remaining bytecodes in Bytecode-buffer).
- Mode switch from Dalvik to TCT is activated at fetch stage in Dalvik bytecode fetch logic (DFE).
- The first TCT instruction of handler is fetched immediately with TCT fetch logic (FE) at the next cycle.
- Dalvik Handler address is computed in Dalvik Decode logic (DDC) and sent to TCT fetch logic (FE) simultaneously.

As shown in **Table 6**, the DFE activates TCT mode while [D_0] is being fetched at cycle 0, then the first instruction of move instruction handler [T_1] is fetched immediately by TCT fetch logic and [D_0] decoded by Dalvik Decode logic (DDC) at cycle 1. Here, the DFE can eliminate the next Dalvik instruction fetching at cycle 1, then the remaining TCT instructions [T_2 to T_9] are executed continuously until to the end of handler in the TCT mode. The Dalvik instruction is processed only in DFE and DDC stages.

#### 3.6.2 TCT to Dalvik Mode Switching Overhead Elimination

The TCT to Dalvik mode switching overhead elimination can be summarized as follows:

- BXD instruction is recognized at decode stage in order to implement mode switch from native to Dalvik.
- Mode switch needs to take effect at fetch stage (since fetch logic is different for the two modes).
- Our TCT processor has one cycle latency between fetch and decode stages, thus 1 delay-slot scheduling is applied to hide the pipeline structural hazard.
- In general, if there is N-cycle latency between the first pipeline stage and the decode stage, we can still utilize the N delay-slots on this mode switch instruction to hide the N-cycle pipeline hazard.

As shown in Table 6 the BXD instruction [T_8] is placed at one instruction before end of handler in order to hide one cycle latency between fetch and decode stages. Here, the 1 delay-slot scheduling is applied to our proposed technique. The BXD instruction actives mode switch from TCT to Dalvik by TCT decode logic (DC) in decode stage and the last instruction of handler [T_9] is also fetched by TCT fetch logic (FE) in fetch stage at cycle 9th, then next Dalvik instruction [D_10] will be fetched immediately by Dalvik fetch logic (DFE) in Davik mode at cycle 10.

This delay-slot scheduling technique is applicable to all handler codes except in the following cases. That is, for conditional branch handlers (if-eq, if-ne, etc.), conditional branch instruction exists at the end of the handler code which forces the BXD instruction to be placed at the end of the handler, and thus the delay slot cannot be utilized. In the future, we plan to extend our BXD instruction such that the condition check instruction can be placed in the delay slot.

## 4. Experimental Results

In the experiment, we have evaluated the performance of Dalvik bytecode interpretation by applying the proposed techniques to TCT processor and ARM (V5TE) processor.

### 4.1 Evaluation of TCT Processor Dalvik Extension

The existing TCT processor was compared to the TCT processor plus extended Dalvik fetch and decode hardware. The methodology and procedure of the experiment is shown in **Fig. 12**, they are explained as follows.

The Java applications are converted to Dalvik bytecodes and stored to program memory of both processors then these bytecodes will be interpreted by TCT platform interpreter. The Dalvik interpreter for TCT platform will be generated from Dalvik source code [6]. The process in generating platform-specific code can be processed by using configuration file and generation tool [21] that is provided by Google. After generating TCT

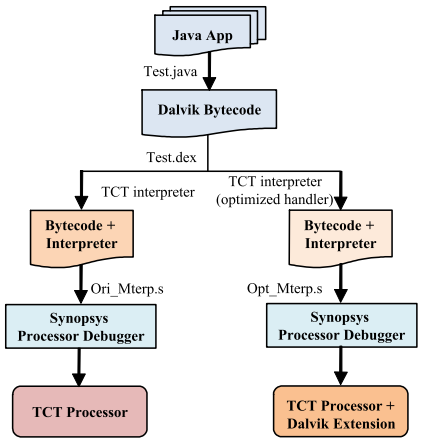


**Fig. 12** Proposed evaluation methodology.

platform interpreter, particular role assignment of registers on the interpreter must be done to allow the interpreter to work correctly according to Dalvik Virtual Machine specification. The role assignment of registers are as follows, R5 (interpreted frame pointer) is used for accessing locals and argument, R6 is thread pointer and R8 is the interpreted instruction base pointer. We can use generated interpreter in TCT processor for interpreting the Dalvik bytecode afterwards. However, the TCT processor with Dalvik hardware extension must optimize the assembly code in each handler before we can apply as we mentioned in proposed solution.

Since TCT processor and Dalvik hardware extension have been developed by Language for Instruction Set Architectures (LISA) on Synopsys Processor Designer [22], the Synopsys Processor Debugger software [23] will be used to measure the processing efficiency of both processors. The Synopsys Processor Debugger allows us to observe, debug and profile the executed application source code including the state of the processor by visualizing all processor resources and the output which provided by the executed application.

The efficiency comparison of both processors, are focused in 3 areas, they are as follow: arithmetic and logic, loop and condition, and method invocation and return. Java application for evaluating the efficiency of the arithmetic and logic is from Google source code (012-math) [6] which verifies the accuracy of Dalvik Virtual Machine. The evaluations of arithmetic and logic are divided into 2 types, they are: 32 bits and 64 bits data processing. Java application that processes 32 bits is transformed to Dalvik bytecode, consists of 24 Dalvik instructions size 74 bytes and those instructions aim to process with integer operands; for example, add-int, mul-int, shl-int and xor-int etc. Another Java application processes 64 bits is transformed to Dalvik bytecode which consists 49 Dalvik instructions size 152 bytes and those instructions aim to process with long operands; for example, add-long, mul-long, shl-long and xor-long etc. Likewise, Java application for evaluating the efficiency of loop and condition is obtained from Google source code (090-loop) [6] as well. Loop and condition program will be transformed to be 40 Dalvik instructions size 112 bytes. This program includes 2 for-loops, conditions and instructions that are for increasing the value of counters in loop; for example, goto, if-ge, if-ltz and add-int etc. The Java application for

```
;[0033ac] java.lang.Character.codePointCountImpl:([CII)I

.byte 0x90 ,0x02 ,0x05 ,0x06                ;0000: add-int v2, v5, v6
.byte 0x12 ,0x00                            ;0002: const/4 v0, #int 0 // #0
.byte 0x35 ,0x25 ,0x1c ,0x00                ;0003: if-ge v5, v2, 001f // +001c
.byte 0xd8 ,0x01 ,0x00 ,0x01                ;0005: add-int/lit8 v1, v0, #int 1 // #01
.byte 0xd8 ,0x00 ,0x05 ,0x01                ;0007: add-int/lit8 v0, v5, #int 1 // #01
.byte 0x49 ,0x03 ,0x04 ,0x05                ;0009: aget-char v3, v4, v5
.byte 0x71 ,0x10 ,0x30 ,0x00 ,0x03 ,0x00    ;000b: invoke-static {v3},
;Ljava/lang/Character;.isHighSurrogate:(C)Z // method@0030
.byte 0x0a ,0x03                            ;000e: move-result v3
.byte 0x38 ,0x03 ,0x11 ,0x00                ;000f: if-eqz v3, 0020 // +0011
.byte 0x35 ,0x20 ,0x0f ,0x00                ;0011: if-ge v0, v2, 0020 // +000f
.byte 0x49 ,0x03 ,0x04 ,0x00                ;0013: aget-char v3, v4, v0
.byte 0x71 ,0x10 ,0x3f ,0x00 ,0x03,0x00     ;0015: invoke-static {v3},
;Ljava/lang/Character;.isLowSurrogate:(C)Z // method@003f
.byte 0x0a ,0x03                            ;0018: move-result v3
.byte 0x38 ,0x03 ,0x07 ,0x00                ;0019: if-eqz v3, 0020 // +0007
.byte 0xd8 ,0x05 ,0x00 ,0x01                ;001b: add-int/lit8 v5, v0, #int 1 // #01
.byte 0x01 ,0x10                            ;001d: move v0, v1
.byte 0x28 ,0xe5                            ;001e: goto 0003 // -001b
.byte 0x0f ,0x00                            ;001f: return v0
.byte 0x01 ,0x05                            ;0020: move v5, v0
.byte 0x01 ,0x10                            ;0021: move v0, v1
.byte 0x28 ,0xe1                            ;0022: goto 0003 // -001f
```

**Fig. 13** codePointCountImpl bytecode.

**Table 7** TCT processor evaluation result.

| Test Application | Dalvik | TCT | | TCT+DXT | |
|---|---|---|---|---|---|
| | Insn | Insn No. | cycle | Insn No. | cycle |
| ALU 32 bits | 24 | 486 | 490 | 180 | 209 |
| ALU 64 bits | 49 | 1,247 | 1251 | 519 | 651 |
| Loop & Condition | 655,362 | 9,994,238 | 9,994,238 | 3,014,666 | 3,670,028 |
| java.lang.Character. isHighSurrogate | 8 | 135 | 135 | 58 | 67 |
| java.lang.Character. isLowSurrogate | 8 | 135 | 135 | 58 | 67 |
| java.lang.Character. codePointCountImpl | 156 | 3,243 | 3,243 | 1,609 | 1,765 |
| Java.lang.String. codePointCount | 167 | 3,489 | 3,489 | 1,787 | 1,954 |

* Insn = Instruction, No. = Number, DXT = Dalvik hardware extension

evaluating the efficiency of method invocation and return, are obtained from java library because java.lang.String.java [24] is considered as fundamental class that is frequently used. The chosen codePointCount in the referred class to evaluate Dalvik bycode consists of 4 methods as follows; the String.codePointCount, Character.codePointCountImpl, Character.isHighSurrogate and character.isLowSurrogate. The invoke instruction in the codePointCount method will invoke the codePointCountImpl method which is main principal process and Dalvik instruction of this method is shown in **Fig. 13**.

As noted, this method has 2 invoke and 2 return instructions in for-loop (goto, if-ge), thus, this method will invoke the isHighSurrogate and isLowSurrogate method indefinitely to the end of loop and then return to process on codePointCount method. For this purpose, the number of instructions and the code size of Dalvik instructions are obtained from the String.codePointCount, Character.codePointCountImpl, Character.isHighSurrogate and character.isLowSurrogate are 14 instructions 52 bytes, 21 instructions 70 bytes, 8 instructions 28 bytes and 8 instructions 28 bytes respectively.

The results of interpretation of all applications/methods are shown in the **Table 7** which will be explained as follows. This Table 7 shows the number of Dalvik instructions that is actually interpreted while processing but may not be the whole instruction number of each method. Therefore, in this table shows how many instructions and how many cycles that Dalvik instructions is interpreted with native instructions; for example, the arithmetic and logic (32-bit) application are processed with 490 cycles in TCT processor by using 486 TCT instructions interpreted 24 Dalvik

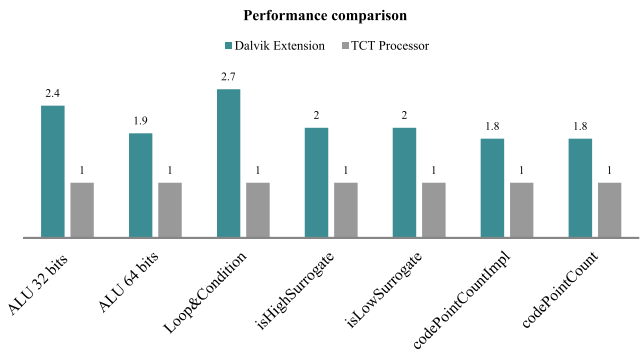


**Fig. 14** Performance comparison.

instructions. But the processing in TCT processor with Dalvik hardware extension is used only 180 TCT instructions and 290 cycles.

The evaluation graph in **Fig. 14** shows the efficiency comparison between 2 types of processor. TCT processor with proposed Dalvik hardware extension can process loop & condition 2.7 times faster than TCT processor due to Dalvik instruction interpretation in goto-instruction group is able to execute directly in the designed goto instructions controller. The evaluation of arithmetic and logic of 32 and 64 bits processing, TCT processor with proposed Dalvik hardware extension can process 2.4 and 1.9 times faster than TCT processor respectively. The reason of 64 bits processes slower is the accessing of 2 or 3 pairs of 32-bit register, which are used for computing 64-bit result, additional instructions are required to improve the 64-bit scheme. The measuring efficiency of process of invoke and return instruction can be checked the results from the parent method (CodepointCount). It proved that TCT processor with proposed Dalvik hardware extension can process 1.8 times faster than TCT processor because of invoke and return instructions is complex instruction so a numerous numbers of TCT instruction are used up to 69 and 23 instructions in interpretation as you can see in Table 4.

### 4.2 Evaluation of ARM (V5TE) Processor Dalvik Extension

The proposed techniques are simple to apply to other processors which can be done with just in 2 steps. The 1st step is the hardware extension and the 2nd step is the Dalvik handler optimization. We assumed that Dalvik Fetch and Decode hardware have already extended into ARM (V5TE) processor. In the 2nd step, we optimized Dalvik handler of ARM processor in Android open source code [6]. As indicated in the Dalvik handler optimization section, the instructions which fetch remained bytecode, fetch next bytecode, compute handler address and jump to opcode handler can be replaced by the instruction (BXD) for entering to Dalvik mode and utilizing Dalvik fetch and Decode hardware. Utilizing Dalvik operand address preloader in this 2nd step, more instructions related to loading Dalvik operands can be removed. A comparison between the original ARM (V5TE) handler and the optimization handler is shown in **Table 8**.

For the instance, Dalvik instruction (move vA, vB) is interpreted with 8 ARM instructions on the original ARM processor but it needs only 3 ARM instructions interpretation on the ARM processor with proposed Dalvik hardware extension. An

**Table 8** Native Instruction Comparison between ARM (V5TE) & ARM+DXT.

| Dalvik Instruction | | Native Instruction count | |
|---|---|---|---|
| **Opcode** | **Mnemonic** | **ARM** | **ARM+DXT** |
| 0x01 | move vA,vB | 8 | 3 |
| 0x07 | move-object vA, vB | 8 | 3 |
| 0x0A | move-result vAA | 6 | 3 |
| 0x0E | return-void | 19 | 17 |
| 0x0F | return vAA | 23 | 20 |
| 0x12 | const/4 vA,#B | 8 | 2 |
| 0x13 | const/16 vAA,#B | 6 | 2 |
| 0x15 | const/high16 vAA,#BBBB0000 | 7 | 2 |
| 0x28 | goto +AA | 7 | 0 |
| 0x32 | if-eq vA, vB, +CCCC | 13 | 4 |
| 0x35 | if-ge vA, vB, +CCCC | 13 | 4 |
| 0x71 | invoke-static | 68 | 63 |
| 0x84 | lont-to-int vA,vB | 8 | 3 |
| 0x90 | add-int vA,vB | 11 | 5 |
| 0x9B | add-long vA,vB | 14 | 9 |
| 0xB7 | xor-int vAA,vBB,vCC | 10 | 5 |
| 0xC2 | xor-long vAA,vBB,vCC | 13 | 8 |
| 0xD8 | add-int/lit8 vAA, vBB, #+CC | 10 | 4 |

* DXT = Dalvik hardware extension

```
;[000108] count10.main:([Ljava/lang/String;)V

.byte 0x12, 0x00          ;0000: const/4 v0, #int 0 // #0
.byte 0x13, 0x01, 0x0a, 0x00   ;0001: const/16 v1, #int 10 // #a
.byte 0x35, 0x10, 0x05, 0x00   ;0003: if-ge v0, v1, 0008 // +0005
.byte 0xd8, 0x00, 0x00, 0x01   ;0005: add-int/lit8 v0, v0, #int 1 // #01
.byte 0x28, 0xfa          ;0007: goto 0001 // -0006
.byte 0x0e, 0x00          ;0008: return-void
```

**Fig. 15** Dalvik bytecode test program.

**Table 9** ARM (v5TE) processor evaluation result.

| OP | Count to 10 | ARM | | | ARM+DXT | | | ARM+DXT | |
|---|---|---|---|---|---|---|---|---|---|
| | | cycle | | | cycle | | | Switching Overhead | |
| | Java application program | Insn No. | (insn) | (program) | Insn No. | (insn) | (program) | Bytecode fetching | BXD latency |
| 0x12 | const/4 vA,#B | 8 | 12 | 12 | 2 | 4 | 4 | 1 | 1 |
| 0x13 | const/16 vAA,#B | 6 | 10 | 100 | 2 | 4 | 40 | 10 | 10 |
| 0x35 | if-ge vA, vB, +CCCC | 13 | 19 | 190 | 4 | 8 | 80 | 10 | 20 |
| 0xd8 | add-int/lit8 vAA, vBB, #+CC | 10 | 17 | 170 | 4 | 8 | 80 | 10 | 10 |
| 0x28 | goto +AA | 7 | 11 | 110 | 0 | 1 | 10 | 10 | 10 |
| 0x0E | return-void | 19 | 25 | 25 | 17 | 21 | 21 | 1 | 1 |
| **summation(10 loops)** | | **387** | | **607** | **119** | | **235** | **42** | **52** |

* OP = Opcode, Insn = Instruction, No. = Number

efficiency comparison between the original ARM processor and the ARM processor with proposed Dalvik hardware extension was conducted based on the summation of native instructions and cycles which interprets all bytecodes on typical Java application program. The number of cycles can be obtained from the simulation of each handler on ARM RealView Debugger (RVDS) [25]. On this evaluation, a simple Java program which count to 10 is used as a test program. This program consists of 6 Dalvik instructions (size 18 bytes) and the instructions are const/4, const/16, if-ge, add-int/lit8 and goto as shown in **Fig. 15**.

The function of each instruction is explained as follows. The const/4 instruction assigns the default value (0) to counter (v0), the const/16 instruction assigns the end value (10) of counter to v1, the if-ge instruction verify the condition of counter, the add-int/lit8 instruction increases counter value by one, the goto instruction repeatedly jumps to the const/16 instruction and the program is terminated by return-void instruction. **Table 9** is the evaluation result based on the cycle count of instructions in handler with the latency of BXD and the cycle count of bytecode fetching

for both ARM configurations.

As we explained in the proposed architecture Section, the processor with proposed HW extension runs in two modes (Native and Dalvik), One Dalvik instruction is fetched by DFE and the optimized handler (including the BXD instruction) is executed by Native mode. The ARM Realview Debugger could not give us the cycle count of Dalvik bytecode fetching and the latency of BXD instruction. The cycle count of Dalvik bytecode fetching in the Dalvik bytecode fetch logic (DFE) is 1 to 3 cycles. On this test program the cycle count of every instructions are 1 cycle because the instruction size of them are 2 to 4 bytes. Thus, on this test program 1 cycle is applied to the bytecode fetching for ARM processor with proposed hardware extension and the latency of BXD instruction is 1 cycle (2 cycles for conditional branch handlers) in the same way as proposed hardware.

All instructions in the test program are run in 10 loops except const/4 and return-void instructions are run in 1 pass. A total of 42 Dalvik instructions will be interpreted by ARM instructions while running this program. The results of the tests are shown on the last line of Table 5. To interpret 42 Dalvik instructions, the original ARM processor needs 387 ARM instructions and 607 cycles, the ARM processor with proposed Dalvik hardware extension (DXT) needs only 119 ARM instructions within cycles of optimized instructions in handler, 52 cycles of BXD latency and 42 cycle of bytecode fetching. On the test program, the proposed Dalvik hardware extension is able to speedup the interpretation by 2.2 times (approximately). However, the efficiency of Dalvik bytecode interpretation obtained from the proposed hardware extension will depend on the Java application program. If the Java application program contains many simple instructions (ALU, Loop & Condition), the result is the Dalvik bytecode interpretation will have a higher efficiency. On the other hand, the loaded interpretation of complex instruction (invoke, return and instance-obj) shows only a small increases in efficiency. For example, the interpretation speedup of const/4, if-ge and goto (simple) instructions are increased to 3x, 2.7x and 11x respectively but the interpretation speedup of return-void (complex) instruction is only 1.2x.

### 4.3 The Proposed Hardware Synthesis

The Synopsys Design Compiler [26] is used for synthesis TCT processor with Dalvik hardware extension by using TSMC 90 nm technology with tcbn90lphplvttc library at 200 MHz. We show the result of synthesis TCT processor before and after extended Dalvik hardware in **Table 10**.

From Table 10, the total area of Dalvik hardware extension is slightly increased by 0.3 $mm^2$ or equivalent to 10.56 Kgate (NAND) [3]. Moreover, it consumes the total power which also has a minor increase at only 0.23 mW. The Dalvik hardware extension slightly uses more resources because less complicated hardware are being utilized as much as possible.

**Table 10** Synthesis comparison result.

| | Clk freq (MHz) | Total Area (μm2) | Total Gate count | Total Power (mW) | Library (tcbn90lphplvttc) |
|---|---|---|---|---|---|
| TCT | 200 | 70,072.43 | 28,174 | 5.93 | TSMC |
| TCT+DXT | | 100,524.72 | 38,733 | 6.16 | |
| DXT | | 30,452.29 | 10,559 | 0.23 | |

## 5. Future Work

The knowledge extension and development of our proposed techniques have 2 directions as follows;

( 1 ) The focus on the efficiency development of the better Dalvik bytecode interpretation which we have the concept in hardware development to support Dalvik bytecode interpretation directly in case of the instruction is not so complicated. Afterwards, it will be integrated to the proposed technique so it can extremely accelerate the speed of the Dalvik bytecode interpretation.

( 2 ) The proposed techniques can apply to other popular Virtual Machines; such as, Java Virtual Machine and Parrot Virtual Machine. The proposed techniques utilize the hybrid solution which the adopted hardware and software can be processed together and it is easy to do so. We could state that using the existing software of Virtual Machine may invoke minor adjustments and the proposed hardware can be adapted to another processor easily without degrading the existing characteristics.

## 6. Conclusion

We proposed a new method to enhance the efficiency of Dalvik bytecode interpretation in Android Operating System by using hybrid technique which is the hardware extension into the existing processor and operate with Dalvik handler software perfectly. The achievement that we acquired was the Dalvik hardware extension enabled existing processor interpreted Dalvik bytecode to the maximum speed up to 2.7x @200 MHz by consuming the additional area of 0.03 $mm^2$ and power increment of 0.23 mW. The new solution that we proposed also has many other distinctive features; such as, the Dalvik hardware that we designed to have less complications. It is simple to expand in any processor without degrading the original characteristics in them. Another thing is the Dalvik hardware can easily process together with existing Dalvik handler software with a few adjustments. The experiment result provides us the ideas to significantly improve the efficiency of Dalvik bytecode interpretation in the future. The key is the designing extension of the existing Dalvik hardware to support the basic instructions and integrate the proposed techniques that we presented to support the complicated instruction of Dalvik. For this reason, processor will process completely and extremely faster. Moreover, it not only presenting the efficiency development of Dalvik bytecode interpretation but we also proposed future techniques to apply to other Virtual Machines to have a better performance. In addition, this mentioned application can be used with both stack based Virtual Machine and register based Virtual Machine; for example, Java VM and Parrot VM by slightly modifying the existing Virtual Machine. Furthermore, it could be said that our Dalvik hardware has particular feature is Virtual Machine Independent.

**Acknowledgments** This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo and Synopsys, Inc.

## References


[1] Cheng, B. and Buzbee, B.: A JIT Compiler for Android's Dalvik VM, Google, available from ⟨http://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf⟩ (accessed 2014-03-09).

[2] Ohta, A.,Yoshizane, D. and Nakajo, H.: Cost Reduction in Migrating Execution Modes in a Dalvik Accelerator, *Proc. 1st IEEE Global Conf. Consumer Electronics*, pp.502–506 (2012).

[3] Taiwan Semiconductor Manufacturing Company Limited: *TSMC 90nm Core Library Application Note*, Release 1.2 (2006).

[4] Bornstein, D.: Dalvik VM Internals, Google I/O Conf. 2008 Presentation Slides, available from ⟨http://sites.google.com/site/io/dalvik-vm-internals⟩ (accessed 2014-03-09).

[5] Google Android: Dalvik Executable Format, available from ⟨https://source.android.com/devices/tech/dalvik/dex-format.html⟩ (accessed 2014-03-09).

[6] The Android Open Source code, available from ⟨https://source.android.com/source/index.html⟩ (accessed 2014-03-09).

[7] Porting Android to Devices, available from ⟨https://source.android.com/devices/index.html⟩ (accessed 2014-03-09).

[8] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T.: A dynamic optimization framework for a Java just-in-time compiler, *Proc. 16th ACM SIGPLAN Conf. Object-oriented Programming, Systems, Languages, and Applications* (*OOPSLA '01*), pp.180–195 (2001).

[9] Aho, A., Lam, M., Sethi, R. and Ullman, J.: *Compilers: Principles techniques and tools*, Vol.1009, Pearson/Addison Wesley (2007).

[10] Gal, A. et al.: Trace-based Just-In-Time Type Specialization for Dynamic Languages, *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp.465–478 (2009).

[11] Gal, A., Probst, W. and Franz, M.: An effective jit compiler for resource-constrained devices, *Proc. 2nd International Conf. Virtual Execution Environments*, *VEE '06*, pp.144–153, ACM (2006).

[12] Inoue, H., Hayashizaki, H., Wu, P. and Nakatani, T.: A trace-based java jit compiler retrofitted from a method-based compile, *Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, *CGO '11*, pp.246–256, IEEE Computer Society (2011).

[13] Steve, S.: Accelerating to meet the challenge of embedded Java, *White paper of ARM Limited*, Cambridge, UK (2001).

[14] Puffitsch, W. and Schoeberl, M.: picoJava-II in an FPGA, *Proc. 5th International Workshop on Java Technologies For Real-Time and Embedded Systems*, *JTRES '07*, Vol.231, pp.213–221, ACM (2007).

[15] M. Schoeberl: Hardware Support for Embedded Java, available from ⟨http://www.jopdesign.com/doc/chap_javahw.pdf⟩ (accessed 2014-03-09).

[16] Jazelle-ARM Architecture Extensions for Java Applications, available from ⟨http://cadal.cse.nsysu.edu.tw/seminar/seminar_file/sywang_0917.pdf⟩ (accessed 2014-03-09).

[17] ARM: *The Architecture for the Digital World*, Technical Reference.

[18] Urfianto, Z., Isshiki, T., Khan, A. and Li, D.: A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development, *IEICE Trans. Fundamentals*, Vol.E91-A, No.4, pp.1185–1196 (2008).

[19] Isshiki, T., Urfianto, Z., Khan, U., Li, D. and Kunieda, H.: Tightly coupled thread: A new design framework for multiprocessor system-on-chips, *Proc. DA Symposium 2006 IPSJ Symposium Series*, Vol.2006, No.7, pp.115–120 (2006).

[20] Android: Bytecode for the Dalvik Virtual Machine, available from ⟨https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html⟩ (accessed 2014-03-09).

[21] Android-platform, Dalvik Porting Guide, available from ⟨https://groups.google.com/forum/#!topic/android-platform/-4epsQnp1CM⟩ (accessed 2014-03-09).

[22] Synopsys, Inc.: *Processor Designer Reference Manual*, F-2011.06 (June 2011).

[23] Synopsys, Inc.: *Processor Designer*: *Debugger Reference Manual*, F-2011.06 (June 2011).

[24] Source code for the Java library classes, available from ⟨http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Class.java⟩ (accessed 2014-03-09).

[25] *ARM RealView Debugger User Guide*, Version 4.0 (2008).

[26] Synopsys, Inc.: *Design Compiler User Guide*, Version D-2010.03-SP2 (2010).


**Surachai Thongkew** received his B.E. degree from Sripatum University and M.E. degree from Asia Institute of Technology, Thailand, in 1999 and 2002 respectively. He was a lecturer at Department of Computer Engineering in Sripatum University. Since 2010 he has been studying for his Ph.D. degree in Department of Communications and Computer Engineering, Tokyo Institute of Technology. His interests include MPSoC hardware/software co-design and VLSI architecture design for Virtual Machine.

**Tsuyoshi Isshiki** has received his B.E. and M.E. degrees in electrical and electronics engineering from Tokyo Institute of Technology in 1990 and 1992, respectively. He received his Ph.D. degree in computer engineering from University of California at Santa Cruz in 1996. He is currently an Associate Professor at Department of Communications and Computer Engineering in Tokyo Institute of Technology. His research interests include MPSoC programming framework, high-level design methodology for configurable system, bit-serial synthesis, FPGA architecture, image processing, fingerprint authentication algorithms, computer graphics, and speech synthesis. He is a member of IEEE CAS, IPSJ and IEICE.

**Dongju Li** received her Ph.D. degree in Electrical and Electronics from Tokyo Institute of Technology in 1998. She is currently an Associate Professor at Department of Communications and Computer Engineering, Graduate School of Science and Engineering, Tokyo Institute of Technology. Her current research interests include embedded algorithm for fingerprint authentication, fingerprint authentication solution for smart phone, VLSI architecture design and methodology and SOC design for multimedia applications such as fingerprint and video CODEC. She is a member of IEEE CAS and IEICE since 1998.

**Hiroaki Kunieda** was born in Yokohama in 1951. He received his B.E., M.E. and Dr. Eng. degrees from Tokyo Institute of Technology in 1973, 1975 and 1978, respectively. He was a Research Associate in 1978 and an Associate Professor in 1985, at Tokyo Institute of Technology. He is currently a Professor at Department of Communications and Computer Engineering in Tokyo Institute of Technology. He has been engaged in researches on Distributed Circuits, Switched Capacitor Circuits, IC Circuit Simulation, VLSI CAD, VLSI Signal Processing and VLSI Design. His current research focuses on fingerprint authentication algorithms, VLSI Multimedia Processing including Video CODEC, Design for System On Chip, VLSI Signal Processing including Reconfigurable Architecture, and VLSI CAD. He is a member of IEEE CAS, SP society, IPSJ and IEICE.