Operating System Support for Procedural Abstraction in Embedded Systems

Keun Soo Yim[a], Jeong-Joon Yoo[a], Jae Don Lee[a], and Jihong Kim[b,*]
[a]*Computing Lab., Samsung Advanced Institute of Technology, Yongin 449-712, Korea*
[b]*School of Computer Sci. and Engr., Seoul National University, Seoul 151-742, Korea*
keunsoo.yim@samsung.com

Abstract

Procedural abstraction reduces code size by replacing repeated code fragments with call instructions to a subroutine that executes the repeated fragment. However, in order to build a subroutine, extra instructions are necessary to support the procedural call mechanism. In this paper, we present an operating system level technique which improves the space efficiency of a procedural abstraction-based code compaction technique. The call-related extra instructions are not used in the proposed technique because operating system routines implicitly supports the procedure call and return. The proposed technique consists of three execution modes including one applicable to ROM-based systems. The experimental results show the proposed technique reduces the code size significantly while increasing the execution time slightly.

1. Introduction

The cost and power consumption of embedded real-time systems both depend heavily on the size of memory [1]. A real-time system consists of four main hardware components: processor, memory, and I/O devices. As processor and I/O devices have irreplaceable features, memory (e.g., DRAM) is often pointed out as an optimization point. Memory forms and consumes a large amount of silicon area and electric energy, respectively, thus reducing the memory size surfaces as a critical design objective, particularly in mass-produced real-time systems.

Code compaction reduces program size significantly and at the same time stores the code in an executable form [2]. Specifically, procedural abstraction retrieves repeated instruction sequences and replaces them with call instructions to a newly created procedure that runs the original sequence [3, 4]. For future space efficiency improvement, it translates similar instruction sequences to identical sequences by reordering instructions, renaming registers, and using relative addresses in branch. Figure 1(a) exemplifies the original code where identical letters denote the same instruction sequences. Fragment *B* is repeated twice, and can be abstracted as a procedure shown in Figure 1(b). Although this requires additional two call instructions and one return instruction, this reduces two instructions overall and still maintains the code executable.

* Jihong Kim was supported by the Ministry of Information and Communication, Korea, under the Information Tech. Research Center support program supervised by the Institute of Information Tech. Assessment, IITA-2005-C1090-0502-0031.

However, this occasionally requires larger numbers of extra instructions, particularly when the system stack and return address are required to be managed. Figure 1(c) is an example of abstracted code when fragments *ABC* and *ABE* are a leaf procedure that does not have a call instruction. Thus, original fragments of *ABC* and *ABE* do not store the return address to the system stack using push and pop instructions. Since procedural abstraction has changed *ABC* and *DBE* to a non-leaf procedure, extra push and pop instructions should be used to manage the return address. Due to these stack operations, the code size becomes three instructions larger than the original size, and thus degrades the value of procedural abstraction in terms of code size reduction.

In this paper, an operating system technique is presented for improving the space efficiency of the procedural abstraction. The extra instructions are not required in the proposed technique because operating system routines implicitly process operations executed by the additional instructions. The proposed technique consists of three execution modes, each impacting performance differently. One of them is applicable to ROM-based systems. In utilizing the execution frequency profiling data, the proposed techniques can be effectively used for compacting codes in cooperation with existing code size reduction techniques. The experimental results show that over half of program codes are seldom executed. By utilizing this phenomenon the proposed technique reduces the code size significantly while delaying the execution time slightly.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes the proposed technique, and Section 4 and 5 present the implementation and evaluations. Section 6 concludes this paper.

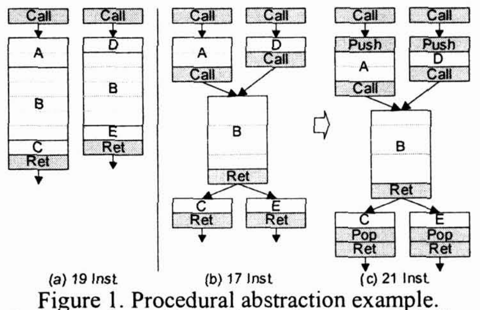

Figure 1. Procedural abstraction example.
* *Identical letters denote identical instruction sequences. Call means a branch and link instruction, Ret means a return instruction, Push stores a link register to stack, and Pop restores the link register from the stack.*

2. Related work

There are two principle technologies that reduce the program code size: compression and compaction [2, 3, 5]. Compression is superior over compaction in terms of reducing code size [6]. However, it suffers from long decompression time and thus hardware decompressors are used to reduce the decompression time. This hardware increases the cost of embedded systems.

Compaction does not suffer from decompression as it reduces code size in direct executable form. Compaction includes the basic compiler optimization techniques [7], e.g., elimination of redundant, unreachable, and dead codes. Its advanced technique is code factoring, which is classified into two types: local factoring and procedural abstraction [3, 12]. Local factoring moves identical instructions from basic blocks to their common predecessor or successor.

The procedural abstraction retrieves identical instruction sequences and replaces them with call instructions to the newly created procedure that executes the identical sequence. Its efficiency in terms of code size reduction can be improved by translating similar instruction sequences to identical sequences. This method can use instruction reordering and register renaming techniques, and can use relative-addressing mode in branch instructions. In addition, predicated instructions can be used to conditionally skip parts of instruction sequences [8].

However, this method requires a considerable number of extra instructions as shown in Figure 1. Figure 2 is an example of when the repeated code is a non-leaf procedure. Here, fragments *BCD* are repeated twice, and thus both can be abstracted as a procedure. Since fragments *BD* include a call instruction to fragment *C*, as shown in Figure 2(c), *BC* must manage the return address using push and pop instructions. Otherwise, the return address is modified by a call instruction in fragment *B*, and fragment *D* returns to an incorrect location. As a result, in order to build a procedure five instructions are used, two call, one push, one pop, and one return instructions.

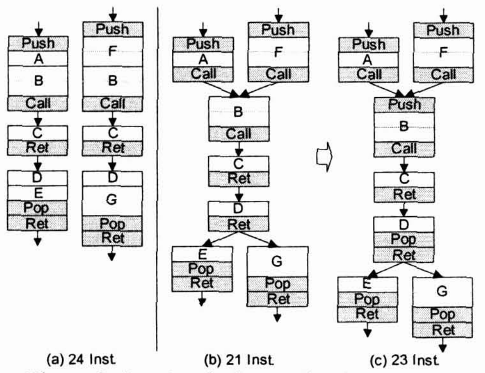

Figure 2. Overhead of procedural abstraction.

In order to address this procedural abstraction overhead, two novel instructions are proposed. These are sequential and bit-mask echo instructions: '*echo.s offset, length*' and '*echo.b offset, mask*' [9, 10]. The sequential echo branches to a target specified by *offset*, executes a certain number of instructions specified by *length*, and returns to the next instruction of this *echo.s* instruction. The bit-mask echo (*echo.b*) conditionally executes the target instructions. It executes *N*-th instruction of the target only if *mask & (1<<(N-1))* is true. These instructions can be fabricated as machine instructions since each of these echo instructions is a combination of micro-instructions. This hides the execution time delay. In theory, echo instructions mimic dictionary-based compression algorithms [5, 6]. That is the reason the technique is superior over other existing compaction techniques.

At the time of writing, no commercial microprocessors support the echo instructions. Thus, microprocessor re-designing and fabrication are required for this to occur, increasing development cost. The increased cost can be larger than the reduced cost of memory achieved using echo. Currently, only a Java virtual machine uses echo [10]. Although, the concept of reducing the code size of the proposed technique is similar to echo, the proposed technique does not require microprocessor modification, this makes the technique a practical and attractive solution in low-cost embedded systems.

Furthermore, code can be compacted if the microprocessor has two instruction sets with different width. For example, ARM processor has two modes: Thumb and ARM. The width of Thumb and ARM is 16 and 32bits, respectively, programs coded by Thumb are typically smaller than programs coded by ARM [11].

3. Proposed code compaction technique

The framework of the proposed code compaction technique is provided in Figure 3. Similar fragments are translated to identical code fragments, and identical fragments are selected for compaction. In order to determine the method of compaction, the execution frequency of the selected code obtained by profiling is used. If the frequency is high, the procedural abstraction is applied as it builds faster code than the proposed technique. However, if the frequency is low or zero, the proposed technique is applied as it achieves greater code size reduction. The proposed technique has three execution modes.

3.1. Exception mode

We use two repeat commands: basic and conditional. Figure 4 shows the basic repeat (*Repeat.B*) applied to the code used in Figure 1(a). In this mode, *Repeat.B* is an undefined instruction that raises the undefined instruction exception on execution. Then, the processor jumps to the exception handler defined by the operating system.

The handler saves the registers used. It detects offset and length of the target fragment by reading operands of the *Repeat.B* instruction. In Figure 4, it stores the next instruction of target fragment *B*, which is the first instruction of fragment *C*. The next instruction is then changed to another undefined instruction of *Return.B*. Finally, this handler restores the saved registers and jumps to target *B*.

The processor starts execution from the target and finally meets the *Return.B* instruction. In checking the opcode of instruction which raised the exception, the handler of undefined instruction exception identifies *Return.B* from *Repeat.B*. In this case, the handler restores the *Return.B* instruction by the original instruction, and then it returns to the next instruction of *Repeat.B*. This handler shall properly manage the program counter, stack pointer, and return address.

The conditional repeat command (*Repeat.C*) operates in the similar way as *Repeat.B*, except for the followings. Figure 5 demonstrates that *Repeat.C* skips some instructions in the target fragment. In the figure, the condition mask is 11011_b, which means a skip of the third instruction of the target. In the mask, the least significant bit is the first instruction of the target. The handler stores the third instruction and replaces it by a no operation instruction. The replaced instruction is restored in the handler of *Return.C*, called after executing the target fragment.

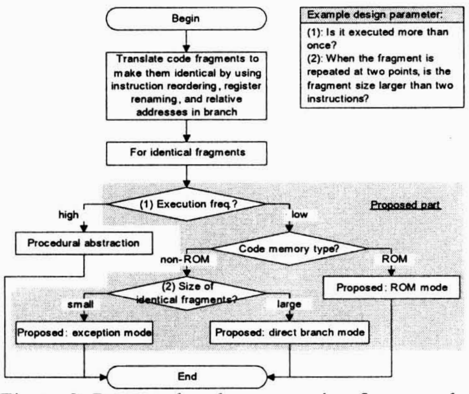
Begin
Translate code fragments to make them identical by using instruction reordering, register renaming, and relative addresses in branch
For identical fragments
(1) Execution freq.?
high
low
Procedural abstraction
Code memory type?
non-ROM
ROM
Proposed: ROM mode
(2) Size of identical fragments?
small
large
Proposed: exception mode
Proposed: direct branch mode
End
Proposed part
Example design parameter:
(1): Is it executed more than once?
(2): When the fragment is repeated at two points, is the fragment size larger than two instructions?

Figure 3. Proposed code compaction framework.

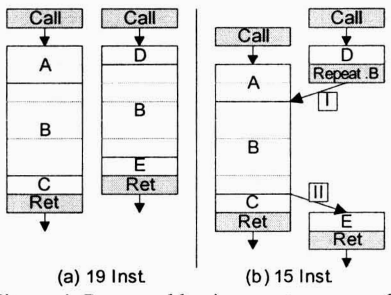

Figure 4. Proposed basic repeat command.
** I: change the next instruction of target fragment to Return.B. II: restore the changed instruction.*

3.2. Branch mode

The exception delays the time of execution due to its indirect branching and pipeline flushing. In order to overcome this overhead, a direct branch mode is presented. This mode uses stack push/pop instructions with a jump (or a system call) instruction. The push instruction stores offset and length of the target fragment, and the jump instruction directly jumps to the *Repeat.B* or *Repeat.C* handler. In this mode, the handler reads the target offset and length from the stack. Except for this, the operation is identical to that of the handler in the exception mode.

While this mode addresses the delay caused by the exception mode, it uses one additional instruction for the stack push operation. The exception mode is only used if the size of the target fragment is small and the execution frequency is low.

3.3. ROM mode

The above two modes modify the code. However, this is impossible in ROM-based systems. Thus, an additional ROM mode is presented. Instead of modifying the instructions in the target fragment, this ROM mode copies them to its buffer placed in RAM. It rebinds the addresses used in the copied instructions and then executes the copied code directly in the buffer.

Figure 6 is an example that uses *Repeat.C*. Since the mask is 10001_b, the first and the fifth instructions of the target fragment (*C1* and *C4*) are requested for execution. Thus, *C1* and *C4* are copied to the RAM buffer (denoted by dotted boxes). It then sets the next instruction of the copied instructions to *Return.C*. This transfers control to the handler when the processor completes execution of all requested target instructions.

For conditional repeat command, this mode can be faster than the branch mode even in non-ROM-based systems. This is because of the fact that the branch mode modifies instructions requested for skipping and copies instructions requested for execution. If the number of skipped instructions is larger than that of executed instructions, this mode can reduce overall memory operation count, which eventually improves execution speed.

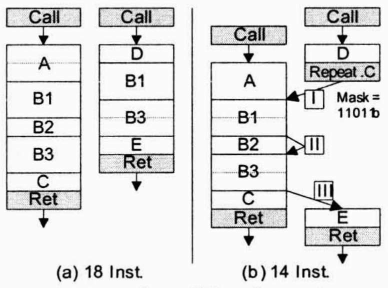

Figure 5. Proposed conditional repeat command.
** I: change B2 to no operation instruction and the next instruction of the target fragment to Return.C. II: skip because it is a no-operation. III: restore the changed instructions.*

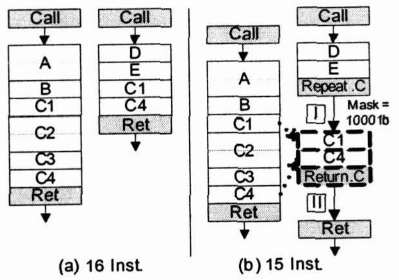

Figure 6. ROM mode with conditional repeat command.
** I: copy the requested instructions, C1 and C4, to the RAM-buffer and set next instruction of the copied instructions to Return.C. II: transfer the control of processor properly.*

4. Implementation

The proposed technique is implemented on a real-time operating system, the Samsung Multiplatform Kernel (SMK). We assume that the hardware platform supports instruction and data cache coherency protocol. The repeat and return instructions are defined using the undefined op-codes. For example, ARM instructions where 27-25th bits are $011_{(2)}$ and 4th bit is $1_{(2)}$ are undefined instructions. Based on this, we define *Repeat.B*, *Return.B*, *Repeat.C*, and *Return.C* as shown in Table 1. The offset field denotes 2^{14}x4B = 64KB address space in ARM mode. Similarly the length and mask each uses only 8 bits, which can be enlarged by using bits allocated for the offset field.

The exception handlers for the software defined instructions must be programmed. Table 2 is a pseudo code where *handler* is an entry point of the handler. It quickly identifies instruction type (line 1-7). If it is *Repeat.B*, the saved registers are restored (line 8). It interprets offset from the operand field of the instruction where an arithmetic shift operation is used to cope with the negative value (line 9-11). Similarly length is calculated (line 12-13). Based on these offset and length, it modifies the next instruction of the target fragment (line 14, 19). Prior to modifying the instruction, both the original instruction and the return address are stored in a custom stack pointed by *buffer* (line 15-18). Due to this stack, the pseudo code is reenterable, allowing nested repeats. The stack is dedicated to each task, and thus it is changed at context switch operation. It finally restores the saved registers and jumps to the target fragment (line 20-22).

After executing the target fragment, the processor will generate another exception, which is handled by *return_b*.

Table 1. Encoding of the proposed commands.

[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

** 0: Undefined instruction space, 1: Repeat.B inst., 2: Return.B inst., 3: Repeat.C inst., 4: Return.C inst.*

Table 2. Pseudo code of handlers.

```
01             .extern buffer                 ; Index of a custom stack
02 handler:    Push R0                        ; Push to stack
03             R0 ← [LR – 4]
04             R0 ← (R0 & 0x01800000) >> 23
05             if (R0 == 0) Jump repeat_b;
06             else if (R0 == 1) Jump return_b;
07             ...
08 repeat_b:   Push R1, R2                    ; Push to stack
09             R0 ← [LR – 4]                  ; Repeat.B inst.
10             R1 ← (R0 << 5) >>> 9           ; Offset operand
11             R1 ← R1 x 4                    ; In ARM mode
12             R2 ← (R0 & 0x1e0) >> 1
13             R2 ← R2 + (R0 & 0xf)           ; Length operand
14             R0 ← (LR – 4) + R1 + R2 + 4    ; Next inst.
15             buffer ← buffer + 4            ; Increase the index
16             [buffer] ← LR                  ; Store LR
17             buffer ← buffer + 4            ; Increase the index
18             [buffer] ← [R0]                ; Store the original
19             [R0] ← #Return.B               ; Set a Return inst.
20             R1 ← LR – 4 + R1
21             Push R1
22             Pop R0, R1, R2, PC             ; Pop from stack
23 return_b:   R0 ← [buffer]
24             buffer ← buffer – 4
25             [LR – 4] ← R0                  ; Restore the inst.
26             R0 ← [buffer]
27             buffer ← buffer – 4
28             Push R0
29             Pop R0, PC                     ; Return
```

* PC: Program counter, LR: Link register, [R]: Memory value where address = R, #: Constant, >>>: Arithmetic shift right.

It restores the next instruction of the target fragment to the original instruction stored in the custom stack (line 23-25). It then returns to the next instruction of the *Repeat.B* instruction (line 26-29). The exception handlers of *Repeat.C* and *Return.C* are similar to these two handlers. The major difference is that the *Repeat.B* handler changes specified instructions in the target fragment to no operation instructions, and the *Return.B* handler restores the modified instructions to the original.

5. Performance evaluations

The performance metrics of the proposed technique are the normalized code size and execution time. The normalized code size of the proposed technique is almost identical to that of the existing echo instructions. Thus, the previous experimental data [3, 9, 10, 12] is used in order to evaluate the performance of the proposed technique as shown in Figure 7. It demonstrates that the proposed technique reduces code size by 30-40% with SPEC2000 benchmark suite and 10-20% with Media Bench suite. This reduction ratio is ~10% higher than that achievable with procedural abstraction.

The proposed technique requires extra memory for handler code memory and custom stack. This memory is relatively small when compared with the whole program code and data size. The size of extra code and data memory are ~512 bytes and ~32 bytes, respectively.

The normalized execution time of the proposed technique is now analyzed, and evidence data where the proposed technique can avoid delay in the execution time is presented. Generally, code using the proposed technique runs slower than code using procedural abstraction. Procedural abstraction delays the execution time by ~15%, and the delay can be reduced to less than 5% when it utilizes profiling data [13]. However, the proposed technique executes more instructions than procedural abstraction. Each handler in Table 2 consists of about 10~50 instructions, while procedural abstraction uses a branch and two push and pop instructions. The handler also can increase the cache miss ratio. If handler is frequently called, the increased miss ratio would be low.

It is observed that in practice the proposed technique can avoid the delay in execution time by taking advantage from the execution frequency profiling data. Specifically, it was found that a large portion of code fragments in embedded software is never or seldom executed with conventional usage patterns. We have implemented a code coverage analysis tool to measure the execution frequency of all codes in the operating system. For example, more than 59% of the operating system code segments were not executed although we run 424 different test applications on top of it. These include unused destructor functions of inter-task communication primitives and destination of untaken branches in boot and initialization codes. The ratio of unexecuted fragments would be much greater in application codes as they are not shared by other software. Thus, if the proposed technique is applied to these unused or seldom used fragments, the delay in the execution time will be negligible.

Finally, we compare the proposed technique with echo instructions. The echo reduces the code size significantly, and does not slow down the execution speed if supported

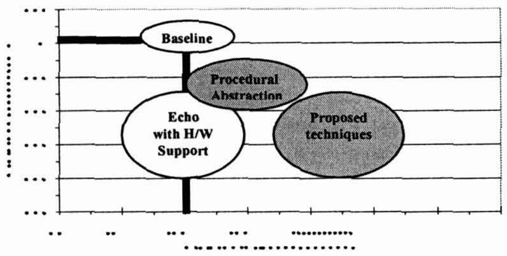

Figure 7. Code size vs. execution time.

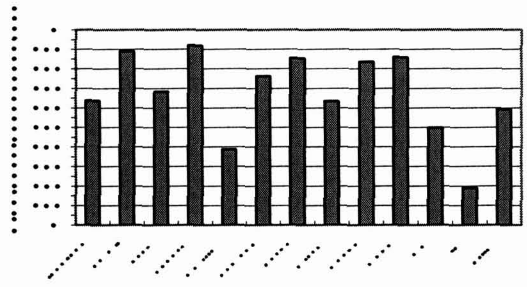

Figure 8. Ratio of unexecuted fragments in operating system components when over 400 test cases are used.

by the hardware. However, so far as we know, no commercial processor supports these instructions. The greatest advantage of the proposed technique is that it does not rely on hardware extension. That is the technical challenge solved in this paper [14].

6. Conclusion

This paper presented an operating system technique for supporting echo instructions without modifying microprocessor architecture. First, the exception mode can reduce program code size if at least two instructions are repeated in at least two points. This mode is most appropriate for tiny embedded systems where code size is critical. Second, the branch mode does not raise an exception, and thus it provides a greater execution speed. This is valuable for frequently-executed codes. Third, the last mode allows these two modes to be applicable to ROM-based systems. The experimental results showed that a large fraction of codes is never executed. If the proposed technique is applied for this fraction, it can reduce the code size largely without delaying the execution time.

References

[1] D. Chanet, B.D, Sutter, B.D. Bus, L.V. Put, and K.D, "System-wide Compaction and Specialization of the Linux Kernel," *Proc. ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 95-104, 2005.
[2] A. Beszedes, R. Ferenc, and T. Gyimothy, "Survey of Code-Size Reduction Method," *ACM Computing Surveys*, Vol. 5, No. 3, pp. 223-267, 2003.
[3] S.K. Derray, W. Evans, R. Muth, and B.D. Sutter, "Compiler Techniques for Code Compaction," *ACM Trans. on Programming Languages and Systems*, Vol. 22, No. 2, pp. 378-415, 2000.
[4] R. Muth, *Alto: A Platform for Object Code Modification*, University of Arizona, Ph.D. Dissertation, 1999.
[5] K.S. Yim, J.-S. Lee, J. Kim, S.-D. Kim, and K. Koh, "A Space-Efficient On-Chip Compressed Cache Organization for High Performance Computing," *Lecture Notes in Computer Science (LNCS)*, Vol. 3358, pp. 952-964, 2004.
[6] D.A. Lelewer and D.S. Hirschberg, "Data Compression," *ACM Computing Surveys*, Vol. 19, No. 3, pp. 261-296, 1987.
[7] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, pp. 660-720, 1986.
[8] W. Cheung, W. Evans, and J. Moses, "Predicated Instructions for Code Compaction," *LNCS*, Vol. 2826, pp. 17-32, 2003.
[9] C.W. Fraser, "Method and System for Compressing Program Code and Interpreting Compressed Program Code," *United States Patent*, No. 2003/0229709 A1, 2002.
[10] J. Lau, S. Schoenmackers, T. Sherwood, B. Calder, "Reducing Code Size with Echo Instructions," *Proc. ACM Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 84-94, 2003.
[11] A. Krishnaswamy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," *Proc. ACM LCTES*, pp. 56-64, 2002.
[12] K.D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," *ACM SIGPLAN Notices*, Vol. 5, pp. 139-149, 1999.
[13] B.D. Sutter, H. Vandierendonck, B.D. Bus, and K.D. Bosschere, "On the Side-Effects of Code Abstraction," *Proc. ACM LCTES*, pp. 244-253, 2002.
[14] K.S. Yim, J. Park, J.-J. Yoo, *et al.*, "Method for Reducing Program Code Size on Code Memory," *Republic of Korea Patent*, Filed No. P2005-0088927, 2005.