On Demand Loading of Code in MMUless Embedded System

Sunil R Gandhi *, Chetan D Pachange, Jr.**
Mandar R Vaidya***, Swapnilkumar S Khorate****

*Pune Institute of Computer Technology, Pune INDIA (Mob- 8600867094; e-mail sunil.gandhi007@gmail.com).
** Pune Institute of Computer Technology, Pune INDIA ( e-mail:pachange.chetan@gmail.com)
*** Pune Institute of Computer Technology, Pune INDIA ( e-mail:mandar2174@gmail.com)
**** Pune Institute of Computer Technology, Pune INDIA ( e-mail:swapnilkhorate@gmail.com)

Abstract: The paper proposes a technique for on-demand loading of code in MMUless embedded system using binary re-writing. This technique will be used for re-writing of code so that it runs on low-end embedded systems that lack MMU-support. Such re-writing of code can be helpful for processors having constrained on-chip memory and lack support for virtualization. This will reduce cost and add functionality to hardware using system on chip. We first analyze the problem using Mathematical modeling and based on analysis we propose the system for modifying the code. We put forth problems that need to be handled during loading of code and possible solutions to them.

1. INTRODUCTION

Many embedded applications use low end processors due to its low cost and low power consumption. These processors usually have constrained resources. They lack MMU support, are small in size and have less on-chip memory. Processor speed is increasing and so is possibility of running complex programs on these embedded processors. But memory size in low-end processor remains the same. Even though processing power is present, this power cannot be utilized due to lack of memory and MMU support. Usually embedded system programmers reinvent on-demand loading techniques every time they face problem of memory constraint. We try to solve this problem by our code generation mechanism which will support on demand loading. Thus code modified by our system can run even if it is bigger than on-chip RAM available. This will reduce the need of external RAM and thus will also reduce its cost.

2. APPROACHES FOR ON-DEMAND LOADING OF CODE

On demand loading in MMUless processors can be achieved using two approaches: 1) Static translation and 2) Dynamic Translation. Dynamic translation is done using technique called Dynamic Binary Translation (DBT). This technique translates the code at run-time just before it gets executed. This type of translation incurs run-time overhead of translating code every time it jumps to new location which is not already translated. This overhead can be reduced by use of code cache. But since low-end processors like ARM cortex-M3 don’t have cache, huge run-time overhead will be incurred in DBT. Also the memory required for DBT is large compared to static translation and we are designing system for low-end microcontrollers which have less on-chip RAM, thus use of dynamic translation is not recommend. But, since DBT and static translation have some problems in common, we have reviewed DBT and tried to understand its working.

Static translation is done during compilation of code. In this the modified binary is generated which can run even if its size is bigger than size of RAM. Although static translation does not incur runtime overhead, it can be used only for code regions that can be identified statically. It cannot be used for self modifying code and indirect control transfers.

Since, Both approaches have their limitations, we are using both approaches in our system, so that minimum overhead is incurred and all cases are handled. We are modifying code during its compilation, But for indirect control transfers, a fault will be generated which will handle on demand loading.

3. RELATED TERMS

3.1 System on chip

The System-on-a-chip or system on chip (SoC or SOC) is integrated circuit (IC) which will have all the functionality like flash, RAM, processor on single integrated chip. This is done to reduce the size, cost and power consumption of embedded devices. It also increases speed of execution. Use of SOC in embedded systems is increasing day by day.
.3.2 Binary re-writing

A binary re-writing means modifying a program to functionally equivalent program which has some extra features like it is optimized or has smaller code size. We are using Binary re-writing to get new program which can be executed on RAM smaller than program size.

.3.3 On-demand loading of code

A code is loaded into memory only when that code is referenced. Such type of loading of code is called on-demand loading.

4. MATHEMATICAL MODELLING

Mathematical modelling is way of expressing a problem in form of set theory, relations and functions. This way of expressing problem helps us to understand system better and determine its input, output, success, failure and other related parameters. It helps in finding out efficient ways of implementing the system and various algorithms using which it can be implemented.

4.1 Mathematical Model

Following is mathematical model of loading data in size constraint embedded system,

Let S be a System,

\[ S = \{ \text{MC}_1, \text{MC}_2, R_{\text{size}}, P_{\text{size}}, P, P', P'', P_c, A_p, N, M \} \]

\text{MC}_1: \text{Input Binary Program}

\text{MC}_2: \text{Output Binary Program}

R_{\text{size}}: \text{Ram size}

P_{\text{size}}: \text{Page size}

P: \text{Set of partitions in secondary memory}

P': \text{Set of partitions in Ram}

P'': \text{Set of modified partitions}

P_c : \text{Program Counter}

A_p : \text{Active partition}

N : \text{Number of partitions that Ram can hold}

M : \text{Number of partitions present in program}

Here, each partition is of same and fixed size.

\[ P = \{ P_1, P_2, \ldots, P_n \} \]

where \( 0 < n < \text{constant} \)

\[ P_i = \{ I_1, I_2, \ldots, I_n \} \]

\[ I_i = \{ \text{Ad}, \text{Mne}, [\text{Reg}], [X] \} \]

Where,

\( \text{Ad} = \text{Address} \)

\( \text{Mne} = \text{Mnemonic} \)

\( \text{Reg} = \text{Register} \)

\( X = \text{Register memory address} \)

4.1.1 Mapping Functions

A Initially, \( P' = \{ \emptyset \} \)

where \( P' = \text{set of partitions modified RAM} \)

\[ F(\text{PC}, A_p) \odot I_i \tag{1} \]

Program counter PC fetches the instruction at address from active partition \( A_p \).

\[ F_1(I_i) \odot P_j \tag{2} \]

Instruction \( I_i \) is executed and for branch gives the next required partition.

\[ F_2(\text{PC}, P_j) \odot A_p \tag{3} \]

Routine to denote partition is in RAM that makes partition active

\[ F_3(P) \odot P' \tag{4} \]

Loads partition from secondary memory into primary memory.

\[ F_4(P) \odot P'' \tag{5} \]

Denotes partitions loaded into the RAM.

Denotes the partition is modified.

4.1.2 Polymorphism

\[ F_2(\text{PC}, P_j) \odot A_p \]

\[ F_2(\text{PC}, P_p) \odot A_p \]
If the required partitions are in RAM then processor will fetch next instruction otherwise it will load partition from secondary memory jump to loaded functions. In this system the corresponding address instruction will be replaced by the code to handle the memory access.

4.1.4 Search

Once the address have been modified, the list has to be searched sequentially and compared with the initial address of current instruction and if an operand has address greater than the one present in initial address register modify the address.

4.1.5 Success State

P'' = {P_1, P_2, P_3, ...., P_n}

All the partitions have been successfully modified.

4.1.6 Failure State

F_1 (I_i) ⊙ P_j  where j>M

Or

If the address in the PC is out of program i.e. invalid address then system goes in invalid state and in order to come back to valid state it aborts the execution.

4.1.7 System constraint

Output file will be produced for single target machine architecture.

5. ON DEMAND LOADING IN SIZE CONSTRAINED MEMORY

We can achieve on-demand loading of code using both static and dynamic approaches. But they both have their own limitations. So, we will use both approaches to get optimum performance and ensure that all cases like indirect control transfer are handled. Fig1 shows all phase’s code has to go through before on-demand loading. Thus our system works at two phases namely

1) Code Generation Phase and

2) Execution Phase.

Working of These phases is as follows:

5.1 Code Generation Phase:

Our System works at code generation phase as shown in fig.1 In this phase, we link an (interrupt service routine) ISR to input code of an exception which will be generated whenever there is jump no execute (NX) region. Then we will modify all addresses in binary file to point to NX region.

Following are the steps followed in Code Generation Phase:

1. Write ISR routine for NX fault and link it to input code.

2. Change address of each jump to point to NX region. We can trick compiler in doing this by placing our code in NX region by modifying linker script.

5.2 Execution Phase:
In this phase, we have to handle all control transfer instructions (CTI) and ensure that code is loaded in RAM before executing it. Also the context in which the CTI instruction is executed must not change. Otherwise it may result in erroneous results especially in case of conditional jumps. This can be achieved through exceptions or faults. Before every control transfer instruction NX fault is generated which will check if code to be executed is loaded in RAM or not. We ensure that fault is occurred by changing branch addresses to point to NX region in code generation phase. Since exception handlers will be frequently used, we will always keep all the data and exception handlers in RAM. The Table is created and initialized with name of every function and appropriate values of logical address, physical address, used bit and size of function. Execution of code in MMUless processor takes place as follows:

1. Every instruction is executed step by step until control transfer instruction.
2. Since we have modified addresses in CTI instructions, NX fault will be generated.
3. Inside Fault we can easily re-generate address of branch instruction.
4. Check whether corresponding function is present in RAM or not.
5. If it is present in RAM then find actual address of loaded function.
6. Copy the jump instruction to other location with actual address.
7. Change Link register to point to this instruction.
8. And execute the branch instruction to transfer control to loaded code.
9. Else
10. Use Buddy system or any other replacement algorithm to load the function.
11. Copy the jump instruction to other location with actual address.
12. Change Link register to point to this instruction.
13. And execute the branch instruction to transfer control to loaded code.

This flow of execution is diagrammatically represented in Fig.2. It shows processing done whenever there is a function call.

6. REPLACEMENT ALGORITHMS

We can use table based approach for replacement of algorithm. In this approach we will store logical address, physical address, used bit and size of function in table at bottom of RAM. Logical address represents the address which is present in actual code. This address will point to NX region. Physical address represents address where function is loaded in RAM. Whenever there is nesting of functions calling function used bit is set to one. This function is not replaced from memory. In this approach we can also consider the size of free memory above and below a function to be replaced. This helps in reducing external fragmentation. But it cannot be completely avoided. We can also use buddy system for loading of functions. Fragmentation problem will be solved using buddy system.

6. LIMITATIONS

This algorithm will not work in 2 conditions. These conditions are as follows:

1. Whenever size of one function is bigger than RAM our algorithm fails.
2. Whenever all the functions are nested. And total size of active functions is greater than RAM.

But both these conditions are very rare. So, according to us, it is not necessary to handle these conditions as it would increase execution overhead.

7. RELATED WORK

Hae-woo Park, Kyoungjoo Oh, Soyoung Park, Myoung-min Sim, Soonhoi Ha. in “Dynamic code overlay of SDF-modeled programs on low-end embedded systems” have proposed 2 techniques for Dynamic loading in MMU-less embedded systems namely full shadowing and DBT.

Full shadowing: - This is a most commonly used technique for running code in MMU-less embedded systems. Full shadowing copies the entire contents of a program’s binary from flash to main memory. This approach can be adopted when the size of main memory is greater than size of program’s binary. Also with this approach the loading time of program increases as it has to load complete program beforehand.

DBT: - Dynamic binary translation looks at a short sequence of code—typically on the order of a single basic block—then translates it and caches the resulting sequence. Code is only translated as it is discovered and when possible, and branch instructions are made to point to already translated and saved code (memorization).
1. This technique is difficult to implement in size constraints embedded system because of following reasons:

2. It requires setting up table for mapping between addresses in cache and original program.

3. There must be a DBT present in cache always to translate the instruction in cache.

4. MMU support is not present.

Due to constrained memory it is difficult to use DBT for dynamic loading.

Siddharth Choudhuri and Tony Givargis in “Software virtual memory management for MMU-less embedded systems” shows various approaches in which data can be virtualized, their advantages and disadvantages. But it only explains virtualization of data. The system proposed in this paper is applicable to both data and code.

8. CONCLUSIONS

This paper presents a concept for dynamic loading of code and data in MMU-less embedded system using binary re-writing.

9. REFERENCES

Moore, Hae-woo Park, Kyoungjoo Oh, Soyoung Park, Myoung-min Sim, Soonhoi Ha., 2006. Dynamic code overlay of SDF-modeled programs on low-end embedded systems. In 1st conference on Design, automation and test, in Europe.

José A. Baiocchi, Bruce R. Childers., 2011. Demand code paging for NAND flash in MMU-less embedded systems In Department of Computer Science University of Pittsburgh Pittsburgh, Pennsylvania 15260 USA.
