Design of a real-time optimized emulation method

Timo Kerstan  
University of Paderborn  
Fuerstenalle 11  
33102 Paderborn  
Germany  
timo.kerstan@uni-paderborn.de

Markus Oertel  
OFFIS Institute for Information Technology  
Escherweg 2  
26121 Oldenburg  
Germany  
markus.oertel@offis.de

Abstract—Virtualization has become a key technology in the design of embedded systems. Within the scope of virtualization, emulation is a central aspect to overcome the limits induced by the heterogeneity of complex distributed embedded systems. Most of the techniques developed for the desktops and servers are not directly applicable to embedded systems due to their strict timing requirements. We will show the problems of existing emulation methods when applying them to embedded real-time systems and will propose a metric to determine the worst-case overhead caused by emulation. Based on this metric we then propose an emulation method minimizing the worst-case overhead.

I. INTRODUCTION

Virtualization has become a key technology in the desktop and server markets. There exist a huge number of products offering hardware virtualization at the bare metal level or at the host level. These products are available for system administrators to help consolidating whole server farms and they are available for end users to bring different operating systems at one time on their desktop. In the case of server consolidation virtualization helps to better utilize or load balance servers saving costs and energy. Distributed embedded systems used in automotive and aeronautical systems consist of a number of microcontrollers where every microcontroller executes a dedicated task. This is mainly done because of isolation reasons to prevent a fault from spreading over the whole network. Beside this aspect the utilization of a single microcontroller may also be very low. Applying virtualization to distributed embedded systems helps to increase their scalability while keeping the needed isolation, safety and dependability. Unfortunately it is not possible to apply directly the virtualization techniques used in the server and desktop systems because of the inherent timing constraints of such distributed embedded systems. These timing constraints add the requirement of temporal isolation to virtualized embedded systems. For example the consolidation of a distributed embedded system into a virtualized architecture using less processing elements, is restricted due to the heterogeneity of the Instruction Set Architectures of the processing elements. Emulation overcomes this restriction. Research on emulation techniques has resulted in a number of methods optimizing their steady state performance, but they lack the support of real-time aspects like bounded execution times or minimizing the Worst Case Execution Times (WCET).

This paper describes a method for emulating embedded programs under hard real-time constraints. The approach guarantees holding all deadlines while estimating the WCET of the emulation in a non conservative way. Further it is possible to adapt the emulated program freely to the available memory on the target platform, while reaching an optimal WCET for the final program size.

II. RELATED WORK

The basic interpretation technique uses a decode and dispatch loop to fetch and execute the source instructions using emulation methods called within the loop. The performance of interpretation is very poor. Therefore several interpretation techniques like Indirect Threading [1], [2], Direct Threading [3], Subroutine Threading [4], Context Threading [5] and optimization techniques like Inlining [6] and Replication [7] have been developed to improve the performance of interpretation. However the resulting performance is still poor in contrast to the native execution.

To overcome the performance problems of interpretation techniques, binary translation tries to map a sequence of source instructions to a sequence of target instructions. These instructions can be directly executed on the target with a better performance than interpretation. The Code Location Problem and the Code Discovery Problem[8] make it very difficult to translate a complete source program to the targets ISA. Dynamic Binary Translators address these problems by translating so-called basic blocks into a block of natively executable code for the target using runtime information. There are two kinds of basic blocks: the static basic blocks and the dynamic basic blocks. Static basic blocks contain an instruction sequence with a single entry point and a single leave point. In essence, static basic blocks begin and end at all branch or jump instructions and all branch or jump targets. Static basic blocks are the biggest atomic instruction sequences and can be translated in advance. In contrast, dynamic basic blocks are determined by the actual flow of a program as it is executed. A dynamic basic block always begins at the instruction executed immediately after a branch or jump, follows the sequential instruction stream, and ends with the next branch or jump. Jump or branch targets within this flow do not terminate the dynamic basic block. This shows that dynamic basic blocks tend to be larger than static basic blocks.
Another thing to notice is that dynamic basic blocks can overlap each other and will be compiled just-in-time and kept in a so called block cache as they depend on the execution flow while this is not the case with static basic blocks. The block cache is very problematic for real-time applications, as the contents of the block cache is not deterministic. Because of this the worst case, the dynamic block not being present in the block cache, has to be assumed. This leads to a very large WCET in the case of dynamic binary translation.

An interesting approach is to improve the performance of interpretation by selective inlining. Piumarta und Riccardi [6] proposed a method to split the source program into blocks. Every emulation method for a source instruction is available as a precompiled binary code. Within a block every source instruction is replaced by the binary code of its corresponding emulation method. This means a translated block consists of a concatenation of the binary code of its emulation methods. This improves the performance, since the jumps between each instruction implementation in a block can be omitted, but increases the memory needed on the target system as the emulation methods were copied into several different blocks.

All these emulation techniques presented here have not been developed to consider real-time aspects like minimization of the WCET and deterministic behaviour. Their main goal is to maximize the steady-state-performance of an emulated program. In the following parts of this paper we will show how these techniques can be used and improved to minimize the WCET and guarantee deterministic behaviour.

Since the slow-down of emulated programs has already been discussed [9], we concentrate on developing an emulation with a small WCET.

### III. Design Aspects

Emulation of a real-time system consists of two fundamental tasks: First, the result of the emulated program has to be identical to the result of the program being emulated and second, the timing constraints have to remain fulfilled in all cases. The first task has already been solved by a variety of different techniques introduced in section II. A solution of the second task is presented in the following chapters.

#### A. Metrics

Emulating a program usually takes more cycles on the target system, since there is not always an equivalent replacement for instructions of the source architecture. This is in particular even more problematic if the bus-width of the source and the target system differs, since all condition codes have to be calculated manually. For still holding all deadlines, the target system has to be faster by a program-dependant factor. This factor is called $CGR_{\text{max}}$ (Maximum Cycle Gain Ratio) and is based, as the name indicates, on cycles. The CGR defines a performance ratio between the emulated execution on the host and the native execution on the source system. The aim of the approach presented in this paper is to determine $CGR_{\text{max}}$ and minimize it at the cost of additional memory consumption.

#### Definition 1: A program $p$ is a ordered sequence of instructions $i_k \in ISA$ with length $n \geq 1$:

$$p_{1:n} = (i_1, i_2, \ldots, i_n) \in ISA^n$$

The instructions occuring in $p$ form the set of used instructions in $p$:

$$I_p = \{i | i \in p\} \subseteq ISA$$

10pt

#### Definition 2: $SC(x)/TC(x)$ denotes the number of cycles needed on the source ISA to execute/emulate the source instruction $x \in ISA$:

$$SC : ISA \rightarrow N^+$$

$$TC : ISA \rightarrow N^+$$

#### Definition 3: The Cycle Gain Ratio represents the time delay factor introduced by the emulation of an ordered instruction subsequence $p_{m:r}$ with $m \leq r$ from instruction $i_m$ up to instruction $i_r$ of $p_{1:n}$:

$$CGR(p_{m:r}) = \frac{\sum_{i \in p_{m:r}} TC(i)}{\sum_{i \in p_{m:r}} SC(i)}$$

All considerations rest upon a worst-case scenario, so the cycles per instruction are assumed to be constant and equal for each instruction, since a possible pipeline of the target system is neglected. Because of this, the timings of a real system will differ from the calculated behavior hence superscalar architectures usually speed up a system quite a bit. But it is not possible to make assumptions about the state of the pipeline at a particular point of the program flow. This is mainly caused by unpredictable thrown interrupts and input dependant branches.

To calculate $CGR_{\text{max}}$ based on single instruction emulation $CGR_{\text{max}}$ of a program $p$ is determined by the instruction $i \in I_p$ that expands the most:

$$CGR_{\text{max}}(I_p) = \max\{CGR(i) | i \in I_p\}$$

A target system using pure interpretation of single instructions and being $CGR_{\text{max}}$ times faster than the source system would hold all deadlines, since even the most complicated instruction could be executed on the new target in the same time as on the original system. This would be sufficient, but we have already shown that pure interpretation suffers from bad performance and a preciseness of one instruction is not necessary. The biggest independantly ordered instruction sequence which is always contiguously executed is the static basic block. This is the reason why $CGR(s)$ is based on an ordered instruction sequence. The benefit of considering static basic blocks instead of single instructions is quite huge, since within one block the number of cycles from all instructions can be accumulated (see Definition 3). A single instruction having a big $CGR$ value can be compensated by other, smaller instructions, leading to a smaller WCET.
Let the set $SBB$ be the set of all static basic blocks of $p$, then $CGR_{\text{max}}$ based on the set $SBB$ of static basic blocks can be calculated as follows:

$$CGR_{\text{max}} = \max\{CGR(b)|b \in SBB\} \quad (7)$$

**Definition 4:** The set of critical static basic blocks with $CGR_{\text{max}}$ is called $CSBB$:

$$CSBB = \{b|b \in SBB \land CGR(b) = CGR_{\text{max}}\} \quad (8)$$

**B. Design-goal for the implementation**

While designing an emulator, there is no information about the composition of instructions inside the blocks nor their execution flow known, since this information is program-dependent. Some emulators speed up the execution time of some instructions at the cost of other instructions because they make the assumption that the usage of instructions is not equally distributed. This assumption is useful in non real-time systems, but it is not useful for real-time systems, because this may raise the upper bound of the WCET of the emulation.

When taking a closer look at the emulation of a single instruction there are several performance aspects that have to be taken into account:

- The **startup-time** describes the time spent in operations before the emulation starts. These operations include the pre-decoding of the source-binary or different kinds of binary optimizations. The effort spent in this period is irrelevant for the calculation of $CGR_{\text{max}}$ since only the timing behavior at runtime matters. Optimizations that take place at this moment are to be preferred.
- The **first-time-effort** is the effort caused by the first execution of an instruction. Especially dynamic binary translators tend to have a huge first-time-effort compared to the other executions of the instruction. Regarding $CGR_{\text{max}}$ this effort is critical and should be kept at a minimum.
- The effort spend for a single execution which is not the first one is called **steady-state-performance**.

The **overall-performance** of an instruction $i$ is a fraction of the combination of its first-time-effort $FTE_i$ and its steady-state-performance $SSP_i$ for $n \in \mathbb{N}^+$ executions:

$$TC(i)_{\text{overall}} \approx \frac{FTE_i + (n-1) \cdot SSP_i}{n} \quad (9)$$

$n$ is assumed to be large in desktop and server systems and hence the overall-performance is often used as a metric to optimize modern virtualization systems because the FTE does not carry much weight. In real-time emulation systems this assumption cannot be made, because in the case of real-time applications the worst case for emulating an instruction $i$ has to be considered:

$$TC_{WC}(i) = \max(FTE_i, SSP_i) \quad (10)$$

So a solution for $FTE = SSP$ has to be found while obtaining still a good SSP. This approach differs a bit from the common virtualization techniques, which usually try to optimize the overall performance.

**IV. Real-Time Optimized Emulation**

Interpretation and static binary translation are the techniques that could be used for real-time emulation. The pros and cons act quite oppositional. Static translated program could reach a better $CGR_{\text{max}}$ but need more space. Interpreted programs use less space and process slower. Both, memory and processing power can be critical in small embedded systems.

Because of this our approach combines both techniques to enable the user to make the emulator fulfill the requirements in the best possible way.

**A. The general idea**

Piumarta and Riccardi [6] already proposed an approach increasing the speed of interpretation by the help of static translated code (see Section II). The target code for the implementation of the instructions of the source ISA was stored in memory. The source program was also divided into blocks. To speed up a block, the target code of the instruction implementations gets concatenated. The benefit of this approach is, that there are no jumps between the instruction implementations inside a block. But this approach has some major disadvantages. Through the block-building process not every instruction can be assigned to a block and so only a part of the instructions can be translated. Further, the operands of the instruction still have to be fetched from the threading table, which normally could be prevented by the use of static translation. Another disadvantage is the lack of the capability to optimize the code block-wide. Block-wide optimization means, that a block is treated as a unit. So the order of target instructions can vary and is not bound to the source instruction frame. It can even be possible to save some store instructions to write temporary data back to the context block since this data can be kept in the registers of the target system.

In a real-time environment this approach can be enhanced to avoid all the mentioned disadvantages. Our approach uses static basic blocks to divide the program. These blocks are statically translated with embedded operands and block-wide optimizations. The main problem of the static translation is that the target binary gets very big. Even with quite small instruction implementations a multiplier of ten is not uncommon. The problem of partly and offline translated programs can not be solved for desktop applications, since there is no data available how frequently a block gets executed. Considering real-time execution the overall-performance does not matter and so the frequency of block execution is not needed either. Important is the cycle gain ratio of a block. To reduce $CGR_{\text{max}}$ for a program the critical set of blocks has to be statically translated. Using the $CGR$ it is possible to sort the blocks in the order of importance for a static translation. Having a memory limitation it is possible to choose the most important blocks for translation while fulfilling the memory constraint.
For all instructions inside a block the critical set of blocks can be determined by summing up all the cycles necessary for an interpreted execution of an instruction (see Def. 4). The blocks can be translated offline until the memory limit has been reached. To increase platform independence and to reduce the complexity of the translation process the blocks are translated to C-code which later will be compiled for the target system. This empowers our approach to have optimizations across the instruction borders, since the C-code can be translated with all possible compiler optimizations. The advantage of block wide optimizations is huge. Operands can be embedded in the code and do not have to be extracted out of the threading table and a lot of stores to the context block can be saved, since a lot of registers get overwritten inside a single block.

![Diagram]

Fig. 1. Stages of the emulator generation process

Fig. 1 depicts the basic steps of our approach:

- In the first step, the program is disassembled and stored as a predecoded array. In this step optimization is already active. Some instructions have switch bits, which change the operation. These instructions are split into separate ones to avoid a computation of the switch bits at runtime. In many architectures there are special registers that configure the behavior of the CPU or peripheral devices. Usually there is only one instruction to set the value of a register which would result in the necessity to check the target register type every time the instruction would have been called. To avoid this, the setting of configuration registers are also separated in different instructions.
- In the second step the program is divided into static basic blocks. Depending on the use of indirect jumps the C-code of the application is necessary
- In the third step the critical blocks are identified to minimize the WCET.
- As a next step, the C-code for the identified block is generated.
- In a last step, the generated code including the necessary real-time emulation core is compiled for the target system and can then be started.

V. CONCLUSION & FUTURE WORK

This paper presented a real-time emulation method for minimizing the $CGR_{max}$ of arbitrary programs. In the worst case scenario the system performing the target system has to be $CGR_{max}$ times faster than the source system to guarantee that all deadlines will still be met. $CGR_{max}$ can be minimized to a certain degree depending on the memory constraint and the possibility to divide the program in static basic blocks. Our approach can be used in the scope of self-optimizing systems, because we are working on ways to change the set of pre-compiled blocks at runtime. Minimizing $CGR_{max}$ can be used as an optimization goal which is limited by the required memory.

Our approach can be seen in the middle of interpretation and static binary translators which combines the advantages of the good portability and the simple implementation of interpreters with the speed of binary translators without the nearly unpredictable effort for compilation at runtime.

We are currently working to finalize our case study. We have chosen the ATmega8 micro-controller from AVR as an 8-bit system, that should be emulated on a 32-bit PowerPC 405 from IBM. In this case the effort to compute the condition codes of the 8-bit instructions on the 32-bit system is quite big, because of the different number representation. The first results of our case study are promising good results in minimizing the $CGR_{max}$.

REFERENCES