International Journal of Automotive Technology, Vol. ?, No. ?, pp. ?−?(year)

Predicting WCET of Automotive Software Running on Virtual Machine Monitors

Jonghun Yoo[1)], Jaesoo Lee[1)], Yongseok Park[2)] and Seongsoo Hong[1),3)]*

[1)] School of Electrical Engineering and Computer Science, Seoul National University, Republic of Korea
[2)] Digital Media & Communications R&D Center, Samsung Electronics Co., LTD
[3)] Department of Intelligent Convergence Systems, The Graduate School of Convergence Science and Technology, Seoul National University, Republic of Korea

(*Received date ; revised date*)

ABSTRACT−Virtualization is gaining significant interests in the automotive industry since it enables a highly secure and reliable computing environment. More importantly, it maintains the same operating environment for legacy automotive software while exploiting the benefits of widely adopted multicore platforms. To exploit the virtualization technology in an automotive system, it is important to predict the WCET of an automotive application running on a virtual machine monitor (VMM). Unfortunately, it is a challenging task because of difficulties in analyzing tricky interactions between a VMM and a guest OS. There are no known attempts to predict the WCET of an application in such an environment. In this paper, we propose a hierarchical and parametric WCET prediction framework. We divide the problem into two subproblems. First, we model the WCET of an application as a function of WCETs of system calls provided by a guest OS. Second, we model WCETs of a system call as a function of WCETs of VMM services. To do so, we clearly identify the places and times of VMM services invoked during the execution of an application. At deployment time, the WCET of an application is instantiated by composing the WCET models altogether. We have performed experiments with the proposed framework by predicting the WCETs of sample programs on various virtual and real machine platforms. The experimental results effectively prove the viability of the proposed framework.

KEY WORDS: System virtualization; WCET analysis; Hierarchical WCET prediction framework; Multicore ECU

1. INTRODUCTION

Inspired by the huge success of multicore processors in enterprise and desktop computing domains, the automotive industry is rapidly adopting multicore ECUs in vehicular systems (Schneider et al., 2010). Multicore ECUs are particularly attractive since they can enhance system safety and reliability using hardware redundancy, offer improved computing performance with multiple processor cores, and allow for ECU consolidation (Bohm et al., 2010). Unfortunately, migrating a huge real-time automotive software base onto multicore ECUs imposes a tremendous amount of reengineering work to developers. For the lack of automated and systematic tools that can aid in such migration, the process will incur an unacceptable cost and effort and jeopardize the safety and reliability of automotive software. To address this serious challenge in the automotive industry, developers have started to exploit the system virtualization technology.

System virtualization enables multiple legacy operating systems, often referred to as a guest OS, to share a physical and potentially multicore-based machine by providing them with an illusion of exclusive access to imaginary hardware called a virtual machine (VM). Virtualization has been successfully used in enterprise and desktop computing domains for the cost effective sharing and maintenance of computing resources. It is as much useful in an automotive system since it can provide a highly secure and reliable computing environment by isolating secure domains from the other vulnerable parts of a vehicular system (Brakensiek et al., 2008; Heiser, 2008; Kanda et al., 2009). More importantly, it can maintain exactly the

* *Corresponding author*: sshong@redwood.snu.ac.kr

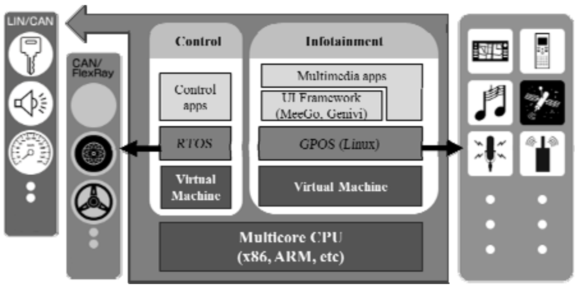


Figure 1. Consolidating control and infotainment functionalities via system virtualization.

same execution environment for legacy automotive software while exploiting the benefits of multicore platforms (Bohm et al., 2010). This helps avoid the multicore migration problem.

Figure 1 shows an anticipated software configuration that utilizes system virtualization in an automotive system. It is an integrated vehicle system that consolidates both control and infotainment functionalities via virtualization. In this example, two VMs are executed on a multicore CPU: one for running control applications on an RTOS and the other for running in-vehicle infotainment applications (Macario et al., 2009; Schroeder, 2010) on an GPOS.

Unfortunately, system virtualization, if used in an automotive system, gives rise to an important technical problem. A large number of automotive systems require real-time guarantees during the execution of their applications. It is thus necessary to predict the worst-case execution times (WCET) of tasks in an automotive real-time application and schedule them in such a way that they can meet their deadlines (Choi et al., 2004; Shin and Sunwoo, 2007). Unfortunately, virtualization brings about uncertainties in WCET prediction. An application running on a VM experiences a varying degree of performance degradation since its access to a physical machine should go through an additional software layer called a virtual machine monitor (VMM). Temporal correctness of an automotive application cannot be guaranteed unless such performance overhead of a VM is accurately assessed.

In the literature, there are no WCET prediction techniques to take into account virtualized software platforms where hardware idiosyncrasies are encapsulated under a VMM. In such an environment, the WCET of an application should be formulated as a function of run-time variables that can be instantiated when a program is deployed on a specific platform. This gives rise to a parametric WCET analysis. In this paper, we present a hierarchical and parametric WCET prediction framework for applications running on a VMM.

Outside the real-time research community, there have been several research attempts to predict the average case performance of an application running on a VMM, by analyzing varying characteristics of diverse VMMs (Benevenuto et al., 2006; Kundu et al., 2010; Menasce, 2005; Tickoo et al., 2010; Wood et al., 2008b; Wood et al., 2008a). Due to difficulties in analyzing VMMs, however, they only stochastically model the performance or resource utilization of an application. To the best of our knowledge, no attempts have been reported to predict the WCET of an application running in a virtualized environment.

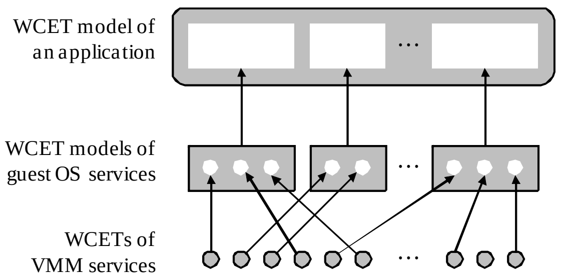

WCET model of an application
WCET models of guest OS services
WCETs of VMM services
…

Figure 2. WCET composition process in the proposed framework when an application is deployed to a VM.

Our framework divides the WCET prediction problem at hand into two subproblems. First, we model the WCET of an application as a function of system calls provided by an underlying OS. Depending on a specific target platform, the WCET of each of the system calls is determined at the deployment time. Second, we model the OS as a function of services provided by a VMM. As a result, the system calls appearing in application code are transparently bound to the corresponding VMM services at the deployment time. The WCETs of VMM services are simply predicted using an existing WECT technique.

The contribution of our work lies in the parametric modeling of a guest OS in terms of services provided by an underlying VMM. This is a challenging task since it is very tricky to find out the places and times of VMM's emulation of instructions in the code of a given guest OS. By composing the WCET models in a transitive manner from the VMM to the application, we can predict the WCET of an application in a virtualized environment. This naturally leads to a hierarchical and parametric WCET framework. Figure 2 illustrates the composition process in the proposed framework when an application is deployed to a VM.

In order to model the WCETs of an application and a guest OS, we make use of the implicit path enumeration technique (IPET) (Li and Malik, 1997). It is one of the most widely used WCET analysis techniques. It formulates the WCET prediction of a given piece of code as an instance of an integer linear programming (ILP) problem. Unlike the original ILP, we allow free variables in linear constraints of the ILP problem so that the WCET is computed after the free variables are instantiated at deployment time.

We have demonstrated the proposed framework with a sample program and a system call designed for our experiment. The experimental results effectively prove that our framework is able to predict the WCET of a program deployed on a virtualized platform. We also show that its overestimation was only 3% of the maximal observed execution time in the experiments.

This paper is organized as follows. Section 2 gives an overview of IPET and existing virtualization techniques to aid in understanding the rest of the paper. Section 3 states the WCET prediction problem of an application running on a VMM and provides an overall solution approach. Section 4 explains the proposed hierarchical WCET prediction framework in detail. Section 5 demonstrates the feasibility of the proposed approach through a series of experiments. Section 6 describes related work and Section 7 provides our conclusion.

2. BACKGROUND

We briefly explain IPET and existing virtualization techniques as a background of our work.

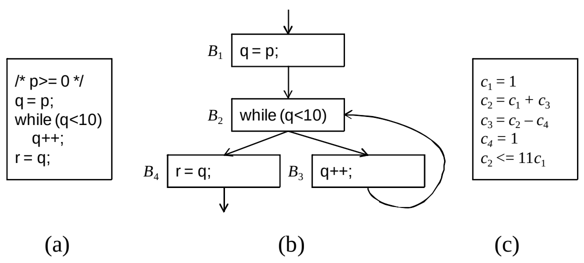
2.1 Overview of Implicit Path Enumeration Technique

We have adopted the implicit path enumeration technique (IPET) to our framework for the static WCET analysis of a given piece of code since it offers a versatile means of formulating a WCET. In IPET, a given program is transformed into a control flow graph (CFG) where each node and directed edge respectively represent a basic block and an execution flow in the program. A basic block is a straight line code with a single entry and a single exit instruction. The execution path of a program varies depending on the run-time states of the system and input variables. Suppose a program consists of *n* basic blocks. For a given execution path of the program, let c_i and t_i be the execution count and WCET of basic block B_i, respectively. Then execution time of the path is calculated by

$$T = \sum_{i=1}^{n} c_i \cdot t_i \qquad (1)$$

The WCET of each basic block is analyzed from micro-architectural hardware modeling (Shaw, 2002) as well as cache modeling (Arnold et al., 2002; Li et al., 1999). The WCET is formulated as an instance of a well-known integer linear programming (ILP) problem where the equation (1) is the objective function to be maximized and a set of constraints are derived from the flow facts of a given program. Flow facts are restrictions on the control flow of a program given by either an automatic analysis of a CFG or manual annotation given by programmers. In general, solving an ILP problem is known to be NP-complete. However, there are many cases in real-world applications where a WCET formulation can be reduced to an LP problem which can be solved with polynomial time complexity (Li and Malik, 1997).

Figure 3 shows an example ILP formulation extracted from (Li and Malik, 1997). In the example, the range of input variable *p* is annotated such that *p* is equal to or greater than 0. Figure 3 (b) and (c) show the CFG of the given code block and constraints derived from it, respectively. The first four constraints are derived from the control structure of the given program while the last one is derived from the range of input variable *p*, given by manual annotation. By solving the ILP problem, we know that the equation (1) is maximized when c_1=1, c_2=11, c_3=10 and c_4=1. It corresponds to a case where input variable *p* gets 0 at the beginning of the program execution. Then the WCET of the given piece of code can be calculated if WCETs of basic blocks are precisely analyzed.



```
/* p>= 0 */
q = p;
while (q<10)
    q++;
r = q;
```

$c_1 = 1$
$c_2 = c_1 + c_3$
$c_3 = c_2 - c_4$
$c_4 = 1$
$c_2 <= 11c_1$

Figure 3. ILP formulation in IPET: (a) example code, (b) its CFG and (c) constraints derived from the example code.

2.2 Overview of Existing Virtualization Techniques

We explain the software architecture of virtualized environments and existing virtualization techniques. We first explain a few terms that are particularly defined in the context of system virtualization (Smith and Nair, 2005). Note that they may have different notions in other domains.

Most modern microprocessors offer at least two modes of operation in order to provide software with different levels of privilege: *user mode* and *kernel mode*. Depending on the execution privilege, an instruction set provided by a CPU is partitioned into two groups. A *privileged* instruction is one that causes a trap whenever executed in the user mode whereas a *non-privileged* instruction can be executed in both modes without a trap. On the other hand, the instruction set can also be partitioned into two groups depending on interaction with hardware resources. A *sensitive* instruction is one that either changes the configuration of hardware or produces different result depending on the operation mode. An *innocuous* instruction is one that is not sensitive. Figure 4 shows the classification of CPU instructions according to these two criteria with examples from x86 instructions set.

Every access from a guest OS to a physical resource should go through a VMM unless the resource is exclusively dedicated to the guest OS. VMM services the requests by providing the guest OS with an illusion of exclusive accesses to the resource. This leads to the run-time performance degradation of an application running on a VMM.

Any instruction issued by a guest OS is either natively executed by the CPU or emulated by the VMM. Innocuous instructions are natively executed since they do not interact with hardware by definition whereas sensitive instructions must be emulated by the VMM. Privileged and sensitive instructions can be easily emulated since they are trapped by the CPU and the VMM gets the control. Non-privileged and sensitive instructions, or simply *critical instructions*, are relatively more difficult to virtualize than privileged instructions since they are not trapped by the CPU. They are replaced by a sequence of non-critical instructions, either dynamically or statically. In either case, guest OS code is modified such that critical instructions are replaced by privileged instructions or *hypercalls*, which are collectively an interface between a guest OS and a VMM.

Figure 5 illustrates an example of execution timeline of an application running on a real machine and a VM. The application invokes a particular system call once during its execution. When it runs on a real machine, its total execution time is $T=T_1+T_2+T_3$ as shown in Figure 5 (a) where T_2 is the execution time of the system call service routine of the underlying OS. On the other hand, the total execution time of the same application running on a virtual machine is $V=V_1+V_2+\ldots+V_8$ as shown in Figure 5 (b). The application running on a VM experiences an extra execution time overhead that amounts to $V - T$.

	Privileged	Non-privileged
Sensitive	E.g., *CLI, STI*	E.g., *SGDT* (Critical instructions)
Innocuous	None	E.g., *ADD, SUB*

Figure 4. Classification of x86 instruction set according to execution privilege and interaction with hardware. (CLI disables external interrupts and STI enables them. SGDT modifies the global descriptor table. ADD and SUB are arithmetic instructions for addition and subtraction, respectively.)

We further explain the components of V in detail. A system call causes a context switch to the VMM, which again delivers the request to the guest OS. This process consumes time V_2. During the execution of the system call service routine, the guest OS issues a privileged instruction to modify an entry in a page table and the VMM emulation of that instruction consumes time V_4. Finally, the guest OS invokes a hypercall to request a write operation to an I/O device. It is serviced by the VMM consuming time V_6. Although this example is simplified to a certain degree, it helps us build a performance model of an application and a guest OS.

3. PROBLEM STATEMENT AND OVERALL SOLUTION APPROACH

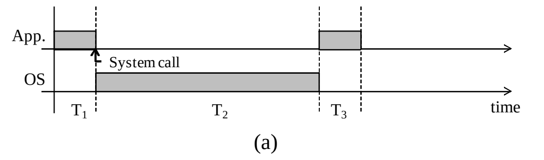
In this section, we formulate our problem and give the overview of our solution approach we name the hierarchical WCET prediction framework.

3.1 System Modeling

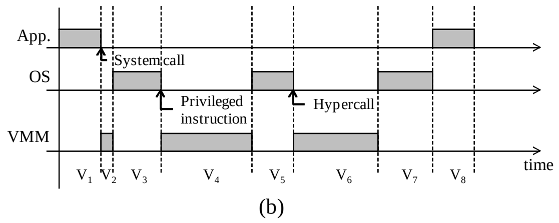
We model the hardware and software of the target system. The hardware consists of a uniprocessor CPU, RAM and I/O devices. We assume a uniprocessor system because we aim to analyze the execution time of a single task excluding any interference from other tasks.

The software consists of three subsystems: an application, an OS and a VMM. They are strictly layered such that each layer is dependent only on the immediately lower layer and there is no upward dependency. When an application makes use of system calls to request some system resources, it is transparently serviced by a VMM.

In this paper, we focus on a native VMM where a VMM runs directly on a bare hardware machine.



(a)



(b)

Figure 5. Execution timeline of an application running on (a) a real machine and (b) a VM.

Nevertheless, our approach can be easily extended to other types of VMMs running on a host OS. For such a software platform, our framework is extended to a four-level hierarchy.

We assume that there is a single guest OS running on a VMM. We further assume that a guest OS is para-virtualized such that it is aware of the underlying VMM. Every critical instruction in the code of the guest OS is replaced with a sequence of non-critical instructions potentially including hypercalls or privileged instructions. Thus, when we model the WCETs of system calls of a guest OS, we take into account VMM services requested explicitly via hypercalls as well as those requested implicitly via protection traps.

3.2 Overview of the Proposed Solution

For a system modeled as above, we aim to estimate a safe WCET of a given application running on a VMM. A WCET analysis is defined as the computation of an upper bound for the execution times of pieces of code for a given application where the *execution time of a piece of code* is defined as the time it takes for the processor to execute that piece of code (Puschner and Burns, 2000). It does not include waiting times due to preemption, blocking or other types of interference.

Our framework divides the WCET prediction problem at hand into two subproblems. First, we model the WCET of an application as a function of the WCETs of system calls provided by the underlying OS. Second, we model the WCET of each system call as a function of the WCETs of services provided by a VMM. As a result, the system calls appearing in application code are transparently bound to the corresponding VMM services at the integration time. The WCETs of VMM services are simply predicted using an existing WECT technique.

To solve the first subproblem, we identify places where system calls are invoked in given application code. This can be done by inspecting the binary code of an application and locating a software interrupt instruction with a specific operand. For example, "*INT* 0x80" invokes a system call in the Linux OS running on the x86 architecture. Using IPET, we formulate the WCET of an application into an ILP problem instance with the execution times of the system call invocation instructions being free variables.

The second problem is more complicated than the first because it is very tricky to find out the times and places of VMM's emulation of instructions in the code of a given guest OS. The simplest case is when a guest OS requests a VMM service via a software interrupt or an instruction dedicated for a virtualized system. Alternatively, a VMM service is executed implicitly by interrupting the sequential execution of a guest OS via a protection trap. The problem becomes more difficult if a VMM dynamically modifies the code in a guest OS at run-time. A VMM does so to reduce the emulation time of the original code. In such a case, we should differentiate the execution time of the original code from that of the modified code in predicting an accurate WCET.

In order to model the WCET of a system call, we carefully examine hypercall invocation instructions, the protection mechanism of the underlying hardware architecture and dynamic replacement of instructions in a guest OS. We then clearly identify a place in the code of a system call in which a VMM service is invoked, and the type of a VMM service invoked at that place. Then we model the WCET of a system call into an ILP problem instance using IPET with execution times of VMM services being free variables.

In modeling the WCETs of an application and a guest OS with IPET, the objective function is to maximize the overall execution time of the given piece of code, and a set of linear constraints are derived from the control flow of the code and manual annotation given by developers. For our purpose, we allow free variables in linear constraints of the ILP problem so that the WCET is computed after the variables are instantiated at integration time.

The WCETs of VMM services are statically analyzed and the WCETs of an application and OS system calls are independently modeled. The WCETs of VMM services are represented as vector **V**. An element in **V** represents the WCET of a VMM service. The WCETs of system calls of an OS are modeled as set **S** of ILP formulations, each of which has free variables to be bound to **V**. Finally, the WCET of an application is

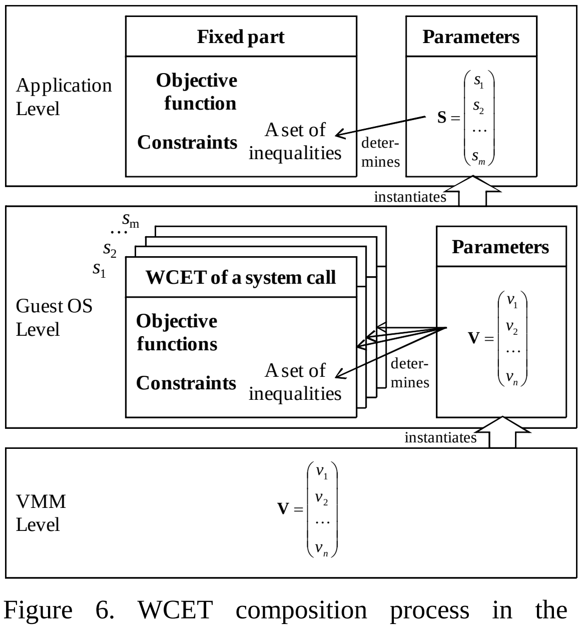


Figure 6. WCET composition process in the proposed framework.

modeled as an ILP formulation with free variables to be bound to **S**.

At integration time, we compose the WCETs of VMM services and the WCET models of a guest OS and an application to instantiate the WCET of the application. In doing so, the WCETs of VMM services are transparently bound to the WCET formulation of an application. The overall composition process is illustrated in Figure 6. In the next section, we describe the proposed framework in detail.

4. HIERARCHICAL FRAMEWORK FOR WCET PREDICTION

In this section, we describe the proposed hierarchical WCET prediction framework in detail. We explain two modeling processes: modeling the WCET of an application and modeling the WCETs of system calls provided by a guest OS. In doing so, we clearly identify which instructions in the code of an application and a guest OS invoke services provided by the underlying software layer. Then we describe model composition process for instantiating the WCET of an application in our framework.

4.1 Modeling an Application

We model the WCET of an application as a function of WCETs of system calls provided by a guest OS. Formally, the WCETs of system calls are represented by a list of free variables **S** shown below.

$$\mathbf{S} = (s_1, s_2, \cdots, s_m) \tag{2}$$

In the equation (2), m is the number of system call types provided by a guest OS and s_i is the WCET of the i-th system call service routine. In our framework, the WCET of the application is dependent only on **S**. The values of the elements of **S** may not be known to developers until the application is actually deployed to a specific platform.

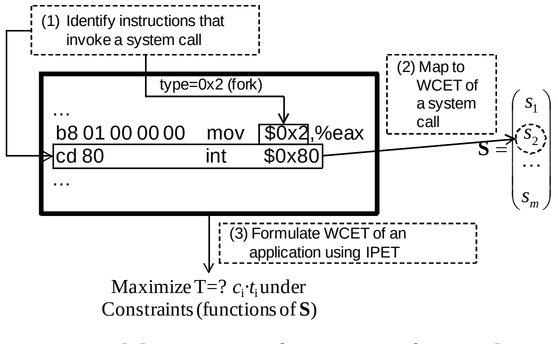

(1) Identify instructions that invoke a system call
type=0x2 (fork)
…
b8 01 00 00 00 mov $0x2,%eax
cd 80 int $0x80
…
(2) Map to WCET of a system call
S =
s_1
s_2
…
s_m
(3) Formulate WCET of an application using IPET
Maximize T=? $c_i \cdot t_i$ under Constraints (functions of S)

Figure 7. Modeling process for WCET of an application.

Figure 7 shows the modeling process and example application code. First, instructions that invoke a system call are identified in the application code. This is can be done by scanning the binary code of the given application and locating a specific opcode that issues a software interrupt. The value of the operand is dependent on the underlying OS. For example, “INT 0x80” invokes a system call in Linux running on the x86 architecture. The system call number is also identified by analyzing the content of argument passing registers. After identifying all instructions that invoke system calls, the WCET of each of the identified instructions is mapped to a corresponding free variable in **S**.

Next, we formulate the WCET of the given application as an ILP problem instance using IPET. In contrast to the conventional IPET, our formulation is allowed to have free variables, which are instantiated at deployment time.

4.2 Modeling a Guest OS

We model the WCET of a system call provided by a guest OS as a function of WCETs of VMM services. To do so, we formulate an ILP problem instance for each system call using IPET. Formally, each ILP problem instance has a list of free variables **V** shown below.

$$\mathbf{V} = (v_1, v_2, \cdots, v_n) \tag{3}$$

In the equation (3), n is the number of VMM service types and v_i is the WCET of the i-th VMM service. In our framework, **S** is dependent only on **V**. We simply use an existing WCET analysis technique to compute the elements of **V**.

Figure 8 shows the modeling process and example system call code. First, instructions that invoke a VMM service are identified from the system call service routine of a guest OS. During the execution of the service routine, a VMM service may be invoked either explicitly or implicitly.

Explicit requests are made in one of two ways. A

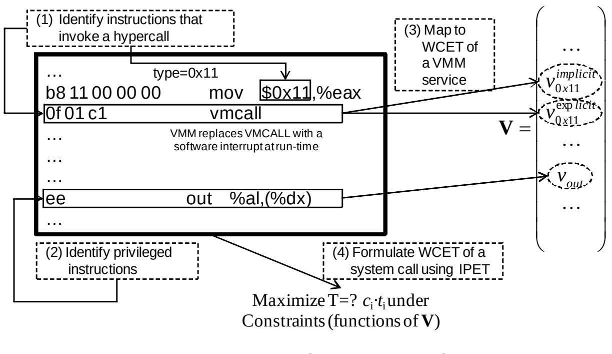

(1) Identify instructions that invoke a hypercall
type=0x11
…
b8 11 00 00 00 mov $0x11,%eax
0f 01 c1 vmcall
VMM replaces VMCALL with a software interrupt at run-time
…
ee out %al,(%dx)
…
(2) Identify privileged instructions
(3) Map to WCET of a VMM service
V =
…
$v_{0x11}^{implicit}$
$v_{0x11}^{explicit}$
v_{out}
(4) Formulate WCET of a system call using IPET
Maximize T=? $c_i \cdot t_i$ under Constraints (functions of V)

Figure 8. Modeling process for WCET of a system call.

guest OS may issue a software interrupt to invoke a hypercall just like an application does to invoke a system call. A specific software interrupt number is dependent on the underlying VMM. For example, "*INT* 0x82" invokes a hypercall to the Xen VMM (Barham et al., 2003) running on the x86 architecture. Alternatively, a guest OS may issue an instruction dedicated to virtualization if the underlying hardware architecture and the VMM support it. For example, Intel x86 processors with virtual machine extensions (VMX) provide the *VMCALL* instruction for invoking a hypercall. We identify these two kinds of instructions and the corresponding hypercall IDs in system call service routines.

Implicit requests are made via protection traps. Because a guest OS runs in the user mode, every privileged instruction issued by the OS is trapped by the CPU and the trap handler of a VMM jumps to a service routine that emulates the instruction. In the x86 architecture, a *general protection fault* occurs in such a case. In practice, there are few privileged instructions in the code of a para-virtualized guest OS since most of them in the original code are replaced with hypercalls and non-privileged instructions. This is because a trap handler exhibits longer execution time than that of a hypercall handler due to the time taken for identifying the cause of the trap and dispatching the request to the corresponding service routine. We identify all the privileged instructions in system call service routines.

After identifying all the instructions that request VMM services, WCET of each of them is mapped to a free variable of the corresponding element in **V**. In doing so, the same instruction may be mapped to different VMM services depending on the accumulated execution count of the instruction. At run-time, some VMMs replace an implicit request with an explicit request in order to reduce emulation time. For example, a Linux OS running on the *lguest* VMM (Russell and OzLabs, 2007) issues *VMCALL* to invoke a hypercall. If the underlying hardware architecture does not support the virtual machine extensions, an *invalid opcode fault* occurs in the x86 architecture. Then the trap handler checks if the fault has occurred by a hypercall and jumps to the corresponding service routine. Before returning to the guest OS, the VMM replaces the *VMCALL* instruction that caused the trap into "*INT* 0x19" to make the OS issue a software interrupt from the next time. Because the WCET for the first execution of *VMCALL* in a guest OS is significantly longer than that of the following executions, they should be assessed differently for accurate WCET prediction.

Mapping the same instruction or basic block to different execution times can be accomplished similarly with a technique that considers the effects of instruction cache (Li et al., 1999). If the *i*-th basic block contains such an instruction, execution count c_i is divided into implicit requests and explicit requests such that $c_i=c_i^{im}+c_i^{ex}$. If the VMM replaces the instruction during the first emulation of the instruction, $c_i^{im}=1$. If the WCETs of two VMM services are t^{im} and t^{ex}, c_it_i term appearing in the equation (1) can be rewritten as $c_it_i = c_i^{im}t^{im} + c_i^{ex}t_i^{ex}$.

Finally, we formulate an ILP problem for each of the system calls. The ILP formulations have list **V** of free variables which are instantiated later with the WCETs of the corresponding VMM services.

4.3 Calculating the WCET of an Application

We compose the WCET models built in the previous steps and the WCETs of VMM services altogether to compute the WCET of a given application. First, free variables in ILP formulations of system calls are bound to the elements of **V**. We solve the ILP problems to compute the elements of **S**. Next, free variables in the WCET model of an application are bound to the values of the elements of **S**. We solve the ILP problem to compute the WCET of the application. In doing so, system calls appearing in the application code are transparently bound to the WCETs of VMM services.

Our framework can predict the WCET of an application regardless of whether it is deployed to a VM or a real machine. WCET of an application is calculated by composing the WCET model of the application and the WCET models of system calls of an OS in the same way. In a real machine, WCET formulations of system calls do not have free variables and are computed directly from the WCET analysis of the system call service routines.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the viability of the proposed framework by predicting the WCET of a test program that intensively accesses hardware resources. We predict the WCET of the program deployed onto varying hardware and software platforms shown in TABLE I. M1 and M2 are virtualized systems with a Xen VMM and a para-virtualized Linux guest OS. M3 and M4 are non-virtualized systems with a native Linux OS. We particularly chose the Linux OS on top of an x86-based processor as our test target since it has been widely considered as a in-vehicle infotainment platform (Macario et al., 2009; Schroeder, 2010).

In order to concentrate on the proposed mechanism by reducing the impacts of other techniques employed in our framework such as IPET and make experimental results easier to analyze, we designed a new system call and implemented it into the Linux OS used in our experiments. This allowed us to avoid pessimism caused

Table I. Target Hardware and Software Platforms

Target Platform	**Hardware**		**Software**	
	CPU	***Memory***	***VMM***	***OS***
M1	Pentium4 at 2.8GHz	1GB	Xen 3.2.1	Para-virtualized Linux 2.6.29
M2	Pentium4 at 2.4GHz	1GB	Xen 3.2.1	Para-virtualized Linux 2.6.29
M3	Pentium4 at 2.8GHz	1GB	None	Native Linux 2.6.29
M4	Pentium4 at 2.4GHz	1GB	None	Native Linux 2.6.29

by the unstructured code and dynamic procedure calls frequently appearing in the existing Linux system calls.

According to the steps defined in the proposed framework, we first modeled the WCET of a given program. We wrote a simple test program that intensively accessed hardware resources via our system call, as shown in Figure 9 (a). We replaced the system call invocation appearing in the test code with an *NOP* instruction and flushed cache at that point to analyze the WCETs of the basic blocks. NOP is an idle instruction which does nothing in a given CPU cycle. We measured the cycle counts of basic blocks using PTLsim (Yourst, 2007) which is a cycle accurate x86 microprocessor simulator that models the out-of-order execution of the full x86 instruction set. As a result, WCET T of the program is formulated as T=148+1000$\times s_{flushcache}$ cycles where $s_{flushcache}$ is the WCET of the system call we have added. The constant 148 was computed by PTLsim and accounts for the execution of instructions for loop initiailization, termination and calling convention for function *test*(). T is maximized when input variable n has its maximum value of 1000.

Next, we modeled the WCETs of the system call invoked by the test program. The source code of the system call in a para-virtualized Linux OS and a native Linux OS is shown in Figure 9 (b) and (c), respectively. It iterates a loop exactly 1,000 times while performing two operations. The first is the cache flush operation implemented via a hypercall in a para-virtualized Linux OS and via privileged instructions *WBINVD* in a native Linux OS. The second is an I/O activity implemented via an *IN* instruction in both OSes. In order to analyze the WCETs of the basic blocks of the system call, we replaced the hypercall and privileged instructions with *NOP* instructions and executed the code using PTLsim.

For a para-virtualized Linux, the WCET of the system call is formulated as $s_{flushcache}$=7702+1000$\times$(v_{mmuext}+v_{emul_in}) cycles. The constatnt 7702 was computed by PTLsim and accounts for the execution of instructions for loop initialization, termination and system call dispatching. It has two free variables: v_{mmuext} is the WCET of the hypercall on line 6 and v_{emul_in} the WCET of the emulation of *IN* instruction on line 8. v_{mmuext} and v_{emul_in} were also analyzed via PTLsim. For a native Linux, the WCET of the system call is simply formulated as $s_{flushcache}$=7847 cycles. This number was also computed by PTLsim.

Figure 10 summarizes the WCET models formulated in the previous steps. We computed the WCET of the test program running on a specific platform by composing the WCET models. As a relative measure of the WCET prediction accuracy, we used the maximal observed execution time. This allowed us to avoid analyzing the exact WCET which is impractical due to too large a state space to explore (Wilhelm et al., 2008).

```
1: // n <= 1,000
2: void test(unsigned int n) {
3:     for(i=0; i<n; i++) {
4:         // Invoke "flushcache? system call
5:         asm ("movl $333, %eax;");
6:         asm ("int $0x80");
7:     }
8: }
```

(a)

```
1: asmlinkage long sys_flushcache(void) {
2:     int i, char k;
3:     struct mmuext_op op;
4:     op.cmd = MMUEXT_FLUSH_CACHE;
5:     for(i=0; i<1000; i++) {
6:         _hypercall4(int, mmuext_op, \
7:             &op, 1, NULL, DOMID_SELF);
8:         k = inb(0x61);
9:     }
10:    return 0;
11:}
```

(b)

```
1: asmlinkage long sys_flushcache (void) {
2:     int i, char k;
3:     for(i=0; i<1000; i++) {
4:         asm ("wbinvd");
5:         k = inb(0x61);
6:     }
7:     return 0;
8: }
```

(c)

Figure 9. Source code for the (a) test program, (b) system call for para-virtualized Linux and (c) system call for native Linux.

	Objective function (WCET in cycles)	Constraints
Application	$T = 148 + n\times s_{flushcache}$	n<=1000
Para-virtualized Linux	$s_{flushcache}$=7702 +1000$\times$(v_{mmuext}+v_{emul_in})	None
Xen VMM	v_{mmuext}=4963	None
	v_{emul_in}=4214	None
Native Linux	$s_{flushcache}$=7847	None

Figure 10. WCET models for (a) the test program, (b) the virtualized platform for M1 and M2 and (c) native platform for M3 and M4.

Table II. WCET Prediction Results for Target Platforms

	M1	M2	M3	M4
Predicted WCET	3.28 sec	3.83 sec	2.80 msec	3.27 msec
Maximal Observed Execution Time	3.21 sec	3.72 sec	2.73 msec	3.18 msec
Overestimation (%)	2.18	2.96	2.56	2.83

Thus, we measured the execution time of the test program by actually running it on a target platform. The predicted WCETs and the maximal observed execution times are shown in Table II. It is clear that our framework is able to predict the WCET of an application running on diverse types of software and hardware platforms, either a VM or a real machine. Our framework is also able to predict an accurate WCET with the overestimation being less than 3% of the maximal observed execution time. The source of the overestimation is in the conservative modeling of modern CPU architecture such as out-of-order execution and branch prediction.

6. RELATED WORK

Virtualization techniques can be classified according to the abstraction level to which a system is virtualized. Two most widely adopted techniques in industry are system virtualization and process virtualization. In this paper, we have addressed the WCET prediction of an application running on a system virtual machine.

For system virtualization, there are a few approaches to predicting the average-case performance of an application running on a VMM. They rely on a stochastic performance model such as a linear model (Wood et al., 2008b), an artificial neural network (Kundu et al., 2010) and a queuing network (Benevenuto et al., 2006; Menasce, 2005). In these approaches, a profiling data set is collected by executing a set of applications on diverse VMMs. Then the model parameters are estimated by training the model using the data set. They aim to predict an application's average-case performance metrics such as resource utilization, memory bandwidth and transactions per second. To the best of our knowledge, no attempts were reported to predict the WCET of an application running on a VMM.

For process virtualization, the problem of predicting the WCET of an application running on a Java Virtual Machine is addressed (Bate et al., 2002; Puschner and Bernat, 2001). These approaches are similar to ours in a sense that they also model the WCET of an application as a function of the WCET required to execute each Java byte code instruction. Our framework has a greater modeling ability since we allow for the hierarchical WCET modeling of a guest OS and VMM services.

There have been significant research efforts for predicting the WCET of a given program since the late 80's. A comprehensive survey on existing WCET analysis techniques are provided in (Puschner and Burns, 2000; Wilhelm et al., 2008). Most existing techniques belong to static analysis and there are some approaches to dynamically estimating WCET depending on the run-time states of a program (Lisper, 2003; Vivancos et al., 2001). Most static analysis techniques involve three phases. First, constraints on feasible execution paths such as maximum loop iterations are found in the *flow analysis* phase. Second, the WCET of each straight-line code segment is derived in the *low level analysis* phase. Third, the WCET of a program is estimated from the flow and low level timing information in the *calculation* phase. IPET is one of the most common techniques proposed for the calculation phase (Lisper, 2006).

7. CONCLUSION

In this paper, we have proposed a framework for predicting the WCET of a given application running on a VMM. We first modeled the WCET of an application as a function of WCETs of system calls provided by the underlying OS. Second, we modeled the WCETs of the system calls as a function of services provided by the underlying VMM. The WCETs of VMM services were simply analyzed by an existing WCET analysis technique. At the integration time of an application, the WCET models and the WCETs of VMM services are composed together to instantiate the WCET of the given application. By doing so, system calls appearing in the application code are transparently bound to the WCETs of VMM services. With experiments, we showed that our framework is a viable means to predict the accurate WCET of an application running on varying software and hardware platforms.

There is a future research direction to further elaborate on our framework. We are looking to incorporate dynamic resource availability into the WCET modeling of system calls. Execution times of I/O system calls are heavily dependent on the run-time states of a VMM or the physical resource availability. For example, the *write*() system call may exhibit a huge variance in its execution time depending on whether a VMM actually issues a disk I/O operation or whether the requested resource is available in the target physical I/O device. Formulating the WCET of the *write*() call as a function of run-time states of a VMM and the resource availability of I/O devices would greatly improve the accuracy in WCET prediction.

ACKNOWLEDGEMENT– The work reported in this paper was supported by Digital Media & Communications R&D Center, Samsung Electronics, Co. Ltd (No. 0421-20110016), by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MEST) (No. 2010-0027809 and No. 2011-0001572), by Automation and Systems Research Institute (ASRI) and by Advanced Institutes of Convergence Technology (AICT), Institute for Green Smart Systems.

REFERENCES

Arnold, R., Mueller, F., Whalley, D. and Harmon, M. (2002). Bounding Worst-Case Instruction Cache Performance, *Proc. the 15th IEEE Real-Time Systems Symposium (RTSS)*, 172-181.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. (2003). Xen and the Art of Virtualization, *Proc. the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 164-177.

Bate, I., Bernat, G., Murphy, G. and Puschner, P. (2002). Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework, *Proc. the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 39-46.

Benevenuto, F., Fernandes, C., Santos, M., Almeida, V., Almeida, J., Janakiraman, G. and Santos, J. (2006). Performance Models for Virtualized Applications, *Proc. the 4th International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, 427-439.

Bohm, N., Lohmann, D., Schroder-Preikschat, W. and Erlangen-Nurnberg, F. (2010). Multi-Core Processors in the Automotive Domain: An AUTOSAR Case Study, *Proc. the Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, 25-28.

Brakensiek, J., Droge, A., Botteck, M., Hartig, H. and Lackorzynski, A. (2008). Virtualization as an Enabler for Security in Mobile Devices, *Proc. the 1st Workshop on Isolation and Integration in Embedded Systems (IIES)*, 17-22.

Choi, J. B., Shin, M. S. and Sunwoo, M. (2004). Development of Timing Analysis Tool for Distributed Real-Time Control System, *Int. J. Automotive Technology,* **5, 4,** 269-276.

Heiser, G. (2008). The Role of Virtualization in Embedded Systems, *Proc. the 1st Workshop on Isolation and Integration in Embedded Systems (IIES)*, 11-16.

Kanda, W., Yumura, Y., Kinebuchi, Y., Makijima, K. and Nakajima, T. (2009). Spumone: Lightweight CPU Virtualization Layer for Embedded Systems, *Proc. the Int. Conference On Embedded and Ubiquitous Computing (EUC)*, 144-151.

Kundu, S., Rangaswami, R., Dutta, K. and Zhao, M. (2010). Application Performance Modeling in a Virtualized Environment, *Proc. the 16th Int. Symposium on High-Performance Computer Architecture (HPCA)*, 1-10.

Li, Y., Malik, S. and Wolfe, A. (1999). Performance Estimation of Embedded Software with Instruction Cache Modeling, *ACM Trans. Design Automation of Electronic Systems (TODAES)*, **4, 3,** 257-279.

Li, Y. T. S. and Malik, S. (1997). Performance Analysis of Embedded Software using Implicit Path Enumeration, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, **16, 12,** 1477-1487.

Lisper, B. (2003). Fully Automatic, Parametric Worst-Case Execution Time Analysis, *Proc. the 3rd Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, 77-80.

Lisper, B. (2006). Trends in Timing Analysis, *From Model-Driven Design to Resource Management for Distributed Embedded Systems,* **225,** 85-94.

Macario, G., Torchiano, M. and Violante, M. (2009). An In-Vehicle Infotainment Software Architecture based on Google Android, *Proc. IEEE Int. Symposium on Industrial Embedded Systems*, 257-260.

Menasce, D. (2005). Virtualization: Concepts, Applications, and Performance Modeling, *Proc. the Computer Measurement Groups Conference* 407.

Puschner, P. and Bernat, G. (2001). WCET Analysis of Reusable Portable Code, *Proc. the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, 45-52.

Puschner, P. and Burns, A. (2000). A Review of Worst-Case Execution-Time Analysis, *Real-Time Systems,* **18, 2-3,** 115-128.

Russell, R. and OzLabs, I. (2007). lguest: Implementing the Little Linux Hypervisor, *Proc. the Ottawa Linux Symposium*, 173-178.

Schneider, J., Bohn, M. and Rößger, R. (2010). Migration of Automotive Real-Time Software to Multicore Systems: First Steps towards an Automated Solution, *Proc. the Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems (ECRTS).*

Schroeder, S. (2010). Introduction to MeeGo, *IEEE Pervasive Computing,* **9, 4,** 4-7.

Shaw, A. C. (2002). Reasoning about Time in Higher-Level Language Software, *IEEE Trans. Software Engineering*, **15, 7,** 875-889.

Shin, M. and Sunwoo, M. (2007). Optimal Period and Priority Assignment for a Networked Control System Scheduled by a Fixed Priority Scheduling System, *Int. J. Automotive Technology,* **8, 1,** 39-48.

Smith, J. E. and Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes,* Elsevier.

Tickoo, O., Iyer, R., Illikkal, R. and Newell, D. (2010). Modeling Virtual Machine Performance: Challenges and Approaches, *ACM SIGMETRICS Performance Evaluation Review,* **37, 3,** 55-60.

Vivancos, E., Healy, C., Mueller, F. and Whalley, D. (2001). Parametric Timing Analysis, *Proc. the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 88-93.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenstrom, P. (2008). The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools, *ACM Trans. Embedded Computing Systems,* **7, 3**.

Wood, T., Cherkasova, L., Ozonat, K. and Shenoy, P. (2008a). Predicting Application Resource Requirements in Virtual Environments, HP Laboratories Report No. HPL-2008-122.

Wood, T., Cherkasova, L., Ozonat, K. and Shenoy, P. (2008b). Profiling and Modeling Resource Usage of Virtualized Applications, *Proc. the 9th ACM/IFIP/USENIX Int. Conference on Middleware,* 366-387.

Yourst, M. T. (2007). PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator, *Proc. the Int. Symposium on Performance Analysis of Systems and Software (ISPASS)*, 23-34.