OS Virtualization: A TinyOS Example

Abstract
An idea as old as time-sharing, virtualization is proving to be an ever-more useful tool to solving problems ranging from trusted computing to VLIW migration to distributed computing. Used to migrate resource management into user space or encapsulate runtime environments, new virtualization developments promise to dramatically affect our use of computers in the near future. An interesting application of virtualization is TinyOS. TinyOS is a micro-threaded, event-driven operating system designed for use in sensor networks on hardware constrained by very-small memory footprints (<8 KiB). It achieves task concurrency without locking (and while using a single, unprotected memory address space) by exploiting compile-time data race checking provided by the nesC programming language. TinyOS uses virtualization in several ways, including the use of an event-driven simulator that allows creation of a virtual sensor network of thousands of identical nodes. Other research projects have successfully modified the TOS simulator to create different nodes each running in its own virtual environment. With the advent of increased computing power, TinyOS also might incorporate more powerful uses of virtualization in the future. Currently, TinyOS allows the compiling and linking of only one application into the TinyOS runtime. To run multiple applications, the source code to each
application must be linked together manually. A consequence of this is incorrect program
execution as a result of the lack of high-level synchronization and separation in key
components of the OS. The project examined the potential difficulties associated with
adding a user-level multiplexer that would allow multiple TOS applications to be linked
together into one TinyOS executable easily, ensuring correct program execution.

Report

Virtualization Background

Interest in creating virtual environments that mimicked the hardware capabilities of
physical machines originated from research and development in time-sharing systems.
(Indeed, a heavy-weight approach to time-sharing is to create a virtual machine and load
the OS and applications into its environment.) By the mid-1970s, the canonical
requirements and methods of creating virtual machine (VM) environments had been
identified and well-documented. Since interest was generally in running applications
compiled and/or linked for the instruction set architecture of the hardware, most
implementations involved using a virtual machine monitor (VMM), running as the sole
system application, to take control of the machine and provide appropriate memory,
storage, I/O, and processor allocations for each VM it created. The VMM would also
start execution of a single-user OS in VM and connect the VM’s I/O layer with
appropriate terminal hardware. To provide protection for the VMM, to prevent a VM
from affecting other VMs, and to provide for the greater potential memory requirements,
hardware memory protection, in the form of a memory management unit (MMU), and memory paging (supported by the newly added MMU) was required [Meyer70].

An additional consequence of the need to protect and partition areas of memory was the requirement for different processor modes that would permit or deny machine instructions from directly accessing physical memory, changing registers, etc, and have the ability to notify the system program of attempts to run privileged instructions. Since software being run inside a VM would be executed directly on the processor without any translation, being able to trap “sensitive” machine instructions was essential; after a trap to the main system code (in this case, the VMM), register states could be examined, the reason for the fault determined, appropriate action taken (using software routines to emulate the desired hardware instruction, for instance), and then control returned to the application. (It is this need for context switching and software processing of normally machine-handled instructions that forms the largest portion of virtualization overhead to this day.) Sensitive instructions, as noted in [Galley73], include examples such as jump to subroutine, jump and save program counter, and push down and jump were sensitive instructions for the PDP-10 since they checked or altered the processor’s mode bit. By 1974, as part of his thesis work and through other papers, Robert Goldberg had classified instruction behavior and provided a formal, theory-based framework to determine if an architecture was capable of supporting a virtual environment, based, in part, upon its ability to handle these special-case, sensitive instructions [Goldberg74].

Several universities successfully implemented virtual machines on top of PDP-10 [Galley73] and PDP-11/45 architectures [Popek75, Goldberg74]. Much early research on VM was centered at IBM, though: their Control Program-67/Cambridge Monitor System
(CP-67/CMS) machines [Meyer70] was the first machine to use virtualization to provide complete virtual machines to users on a time-shared mainframe. Successor machines in IBM’s VM/370 line would continue to advance the technology and improve virtualization performance, especially as virtualization played increasingly important roles in IBM’s product line. One method developed was the use of shadow translation tables [Gum83]. Typically, the VMM or host operating system would contain a page table for each VM, resulting in reduced performance, as any TLB miss would require a lookup in both the VM page table and the VMM page table before the page could actually be loaded. With a shadow page table, the VM is given read-only access to a page table containing the VM virtual address and the actual physical memory addresses, thereby reducing lookup time and context switching.

An important IBM development was the conception and use of hardware *assists* to reduce the time cost of sensitive instruction handling noted above [Gum83]. An assist is an extension of a given machine’s ISA, and is used automatically by the hardware when certain VM-related system calls are made. Some typical assists are used to access, manipulate, and validate a VM’s shadow page table, or to switch from user to supervisor mode [Gum83]. Reflecting IBM’s pricing structure, many assists “have the characteristic that correct execution of the...program does not depend on their use, their value lying instead in their ability to improve performance” [Gum83, page 533]: thus different models in IBM’s VM/370 line had “different assist capabilities and some [had] different levels of the same assist” [Gum83, page 533]. Gum notes the effectiveness, though, of the assists; typically, benchmarks compared virtualization speed of an application or process to the speed of the application or process on native hardware; with assists, virtualization
performance from 35% of the native-machine execution time to 70% and greater [Sum83, page 533]. IBM has also exploited manipulation of the hardware-software interface level by building large multi-processor machines (first starting with the 3090 PR/SM) that can be partitioned via hardware into separate logical machines [Borden89].

Virtualization has become a core feature of IBM’s high-end server line, becoming a very convenient solution to a problems ranging from support of legacy systems, testing and development, to high availability and easy backup and recovery [Borden89]. Since each VM is a complete machine implementation, different operating system and/or application software versions can be run in each VM, supporting legacy systems while migrating to newer implementations. Of course, duplicate copies of system can be running for instant fail-over, or newer software can be tested on the actual hardware it is destined for while older versions are kept in production. With advancements in the VMM, VMs can be migrated to different processors or memory banks, permitting on-the-fly hot swapping of failed hardware without any overall system downtime.

Evolution of Virtualization

Since the 1980s, virtualization has been leveraged to solve (or attempt to solve) a wide range of problems. Perhaps one of the most visible directions virtualization has taken is the migration of VMM down to personal computer hardware. Commercial applications such as VMware’s Workstation have sold hundreds of thousands of copies [Smith03] and run one or more guest operating systems inside hosted virtual machines, with all software compiled for the same architecture [Sugarman01]. The VMware’s VMM runs as an application inside the host OS, intercepting VM memory and I/O calls before passing the requests to the host system, but otherwise letting as much guest code run on the native
hardware as possible. Given the wide variety of hardware available for personal computers, this hosted method permits the VMM to focus on only providing a generic set of hardware interfaces for the guest operating systems to use. System call overhead due to multiple traps to the VMM and hosted OS, as well as several extra data integrity checks and copies, however, means that the virtualized environment pays an even higher performance penalty relative to, for instance, the VM of the 1970s detailed above. Although [Sugarman01] details methods of amortizing the cost of many small guest OS system calls or interrupt requests by essentially batching the calls, fundamental increases in performance are basically only possible through modification of guest and host operating systems to be more aware of the virtualization process. For instance, even while running as a guest OS, Linux will context switch to the idle task—causing several expensive page tables switches; adding the idle thread page table to every running thread halved MMU associated virtualization overhead [Sugarman01]. Similarly, while running Linux as the host OS, for instance, a packet passed from the guest OS will force a copy of the data from the guest OS, through the VMM, to the system OS—a very significant source of overhead for network transactions. By giving the host kernel and guest OS some pre-allocated shared memory space, network transfer overhead could be greatly reduced [Sugarman01]. Of course, given native drivers, the VMM could bypass the host OS and interact directly with the card; as noted before, though, the maintenance requirement for this would be extremely large due to the abundance of commodity hardware available for the PC.

Given the capability of the VMM to intercept instructions and alter the (virtual) registers of the VM, the ability to run binary software compiled for an architecture
different from that of the physical machine was a natural extension of the power of virtualization. Purely software approaches to dynamically translating instructions include applications (Virtual PC) do exist, but the performance penalty until, perhaps, recently inhibited wide-spread use of the method. (Performance penalties are especially severe if the architecture being hosted contains more registers than the physical machine [Smith03].) Hardware methods, though, of dynamic translation have been developed, and even commercially available. Perhaps the most important motivation for this development has been the desire to utilize VLIW architectures with commodity software. IBM’s DAISY architecture, for instance, emulates a full PowerPC system, including BIOS initialization routines, dynamically recompiling PowerPC instructions into VLIW atoms and performing trace scheduling on VLIW words to improve performance of commonly-repeated instructions [Ebcioglu01]. Similar techniques are utilized in Transmeta’s Crusoe microprocessor, although much of the translation is performed in software on the chip, allowing larger groups of instructions to be recognized, translated, and the translation cached [Klaiber00].

Academic research has always been long been interested in utilizing VM for secure (now “trusted”) computing. Indeed, just as the IBM CP-67 was going into production, [Madnick73] demonstrated how using a VMM with a single-user OS in each VM “provides substantially better software security than a conventional multiprogramming operating system approach.” More recently, both [Garfinkel03] and [Lie03] have devised systems exploiting trusted, well-secured hardware systems to provide VMs running programs with varying degrees of security. Work on virtualized systems has also included developments that completely change the model of
computation, such as packaging individual VMs to form self-contained units that can be migrated across networks or NUMA machines [Bugnion97], or allowing the VMs to play dictate resource management [Engler95].

Ultimately, though, the desired end results are the all same: permit multiple applications to cleanly run side-by-side with the best possible performance—similar goals for investigating how to modify TinyOS from its single, dedicated application model to a model permitting multiple applications to be compiled and run simultaneously.

TinyOS

TinyOS Background [adopted from project description]

TinyOS is a micro-threaded, event-driven operating system designed for use in sensor networks constrained by very-small memory footprints (<8 KiB) but requiring near real-time interrupt handling to multiple I/O sources. It achieves task concurrency without locking (and while using a single, unprotected memory address space) by exploiting compile-time data race checking provided by the nesC programming language. An extension of C, the nesC language’s use in TinyOS motivated the development of TOS into a modular design with clearly defined component interfaces and inter-task communication mechanisms. The combination of modular design with low resource utilization has resulted in the increasingly popular use of TinyOS in embedded systems. This popularity destines TinyOS to be utilized in an increasing large variety of systems and projects. With the onward advance of Moore’s Law, affecting microprocessor and microcontroller capacity, memory, and size, the near-future will promote TinyOS to
newer platforms that lessen many of the constraints posed by the original sensor systems. This advancement will also be motivated by the processing overhead and bandwidth required by future generations of microcontroller and mote peripherals (audio microphones, etc.). Indeed, when given the option to process data locally or offload raw data for processing elsewhere, the current trend in embedded system design dictates that the data be processed locally due to the high power cost of transmitting the data wirelessly [Savvides04]. Many mote devices may continue to shrink, reducing power consumption while leaving unaffected computer capacity or memory size; other devices, however, will not follow this trend. As the authors of nesC have noted: “Motes have very limited physical resources, due to the goals of small size, low cost, and low power consumption. We do not expect new technology to remove these limitations: the benefits of Moore’s Law will be applied to reduce size and cost, rather than increase capacity.” [Gay03] While this comment will be applicable to many devices TinyOS runs on, the author predicts this comment will not be applicable to all devices.

The evolution of TinyOS as an embedded platform will have many consequences to the design and feature set of the OS. Some of the limitations of the system—such as the lack of a heap, no memory protection, no thread-based concurrency, a rather simplistic task scheduler, shared resources, etc.—will need to be reevaluated and perhaps redesigned. Many of the advantages of the operating system—low latency, easy access to low-level hardware, small memory footprint, fast context switching, clean and modular code base—will continue to be leveraged in higher performance platforms, however. An example of this would be a port of TinyOS to the ARM family of microprocessors. The latest ARM chips offer up to 64x the memory capacity and performance of the AVR
processors TinyOS currently runs on; these processors would permit TinyOS to operate on embedded systems more advanced than 4 MHz mote devices—performing computational-intensive tasks such as packet routing, data encoding, etc.—but not as advanced as embedded PCs.

TinyOS Technical Overview (derived from [Lymberopoulos])

Oversimplifying, TinyOS is essentially a scheduler that runs tasks posted to it by applications. A task is essentially a thread. Each application is a set of components that have been “wired” together. A component is contained completely in a stack frame. Each component has a “public” part called the interface, and a “private” part called the implementation. The interface itself has two parts: the set of functions it offering and will run when ordered to by a higher component, called commands, and the set of signals it will send to higher components when it needs to communicate to them, called events.

When a component sends a command (which can include parameters) to a component below it, control returns immediately to the caller; the callee’s task will eventually be run once the caller’s task completes. When the callee’s task completes, it will signal up to the caller, invoking the caller’s even handler and passing up whatever information is needed.

The basic unit of execution in TinyOS is the task. As noted before, tasks are basically threads. Tasks are posted to the OS to be run at its convenience, and run until completion unless interrupted by an event (hardware interrupts can cause events). The task run queue is, essentially, an array cycled through repeatedly.

TinyOS Coding Details and Examples

A typical TinyOS code directory will usually have files with three different extensions: .nc, .c, and .h. The .c and .h files are pure ANSI C files as expected. The .nc
files are nesC files that can be generally arranged into two types: configuration files and
module files. A configuration file, as mentioned above, is used to dictate what interfaces
the files provides, what interfaces the file needs, and the implementation portion
mandates what other modules are needed, and how all of the interface connections
between components are made. A module file has nearly the same form, but with one
important difference: it cannot wire any interface connections together. Rather, the
module file is where the actual implementation of the interface functions resides: the
implementation portion contains actual C code along with special nesC and TinyOS
directives. For an example of a configuration .nc file, one could look at the original
TimerC.nc file:

```nesC
configuration TimerC {
    provides interface Timer[uint8_t id];
    provides interface StdControl;
}
implementation {
    components TimerM, NoLeds, HPLPowerManagementM;
    TimerM.Leds -> NoLeds;
    TimerM.Clock -> ClockC;
    TimerM.PowerManagement -> HPLPowerManagementM;
    StdControl = TimerM;
    Timer = TimerM;
}
```

In essence, this file contains two “input ports” that must be resolved at compile time.
Each interface is basically typed, as the interfaces are defined in the tos/interfaces
directory (http://cs490.mine.nu/src/tinyos/tos/interfaces/) and are not permitted to be
overloaded; for instance, the StdControl interface (used to initialize, start, and stop parts
of the task) is defined as:

```nesC
interface StdControl
{
    /**<
     * Initialize the component and its subcomponents.
     */
```
Thus, the commands StdControl.init(), StdControl.start(), and StdControl.stop() must be
defined by a .nc module file and connected to (in this example) TimerC’s StdControl
input. An example module file could be NoStdControl.nc, which just has each function
return success when called:

```c
module NoStdControl {
    provides interface StdControl;
}
implementation
{
    command result_t StdControl.init() {
        return SUCCESS;
    }
    command result_t StdControl.start() {
        return SUCCESS;
    }
    command result_t StdControl.stop() {
        return SUCCESS;
    }
}
```

Returning to the original TimerC.nc, we can see what connections are being made. One
should also note a few important points:

- the Timer interface here is parameterized
when connecting component interfaces, an = signifies complete
bidirectional communication, thus allowing events and signals to pass as
expected; an -> arrow points towards the actual implementation, thus
allowing only commands to be executed

configuration TimerC {
    provides interface Timer[uint8_t id];  //parameterized interface
    provides interface StdControl;
}

implementation {
    components TimerM, NoLeds, HPLPowerManagementM;
    // needed components whose interfaces are not being from the outside

    TimerM.Leds -> NoLeds;     // dummy LED function
    TimerM.Clock  -> ClockC;
    TimerM.PowerManagement -> HPLPowerManagementM;

    StdControl = TimerM;     // implies StdControl = TimerM.StdControl
    Timer = TimerM;           // implies Timer = TimerM.StdControl
}

http://cs490.mine.nu/src/tinyos/tos/platform/pc/TimerC.nc

The canonical TinyOS application example is BlinkM
(http://cs490.mine.nu/src/tinyos/apps/appBlink/). This app uses a timer the blink a led
every second. It is composed of four files: BlinkM.nc (the module defining the
StdControl functions and how often the LEDs blink), BlinkApp.nc (the configuration file
that connects the Main StdControl interfaces with the BlinkM interfaces, connects the
timer interfaces together, and links the Led interfaces together), SingleTimer.nc (another
configuration file which basically acts as a wrapper around the TimerC interfaces), and a
Makefile (which declares which is the main connecting file, and what file contains the
rest of the make rules). The nesC component graph (produced by running “make docs
cpapp” in the app/appBlink folder) listing the is as follows

Compilation

Compiling a TinyOS application can be divided roughly into two distinct phases:

- the nesC preprocessing, encompassing the lexing, parsing, syntactic and semantic checking off all the .nc files; the “wiring” of components together, the generation of an intermediate app.c\textsuperscript{2} file, and passing a finished preprocessed file to
- gcc for standard compilation

The nesC processing system is somewhat baroque: part of the compilation process is a call to ncc, a perl script that performs several safety checks, extracts some variables from the shell environment, extracts a command-line switch containing the available compile targets from the .platform file contained in each platform directory\textsuperscript{3}. Part of the nesC process depends upon David Gay’s RC compiler\textsuperscript{4}; a “region-based” extension to gcc that allocates and frees memory in chunks (provided all external references to the region of memory have been removed). The generated intermediate file is important to examine to understand some of the behaviors of TinyOS, and some of the hurdles that must be overcome to allow multiple programs to be compiled into one instance of TinyOS.

Attempts at virtualization

Fundamental design considerations disallow some of the original goals of the project. Perhaps most importantly, the TinyOS Simulator TOSSIM [Levis03] effectively virtualizes TinyOS, permitting a large number of identical nodes to run simultaneously and interact with each other in a virtual environment. TOSSIM works at the event level, queuing events posted by tasks, and treating interrupts as, essentially, NOPs, thus

\begin{itemize}
\item \url{http://cs490.mine.nu/src/tinyos/apps/bigBlink/build/pcapp/app.c}, for example
\item \url{http://cs490.mine.nu/src/tinyos/tos/platform/pcapp/pc_.platform}
\item \url{http://www.cs.berkeley.edu/~dgay/rc/}
\end{itemize}
preventing true simulation of each TinyOS node. Additional constraints prevent realistic
construction of virtual environments, since the actual hardware TinyOS runs on permit
virtualization of the environment, and the simulator, as noted above, works at the high-
level of events, not at the machine code level. Thus, direction shifted slightly to
investigating the ability to link separate applications into the TinyOS model.

The first job was creating a duplicate of the TOSSIM pc platform called pcapp, so
that modifications could be made without affecting the original implementation. Adding a
platform ultimately required slight modification of a few nesC files\(^5\), as well as some
fixes to ifdefs in the system, makefile, and debug structures\(^6\). The test app toyed with is
bigBlink\(^7\), an app with two different timers, a prototypical example of compiling multiple
programs. Early toying with the application demonstrated some important development
considerations. For example, multiple connections can be made between components as
is demonstrated from a compilation run (pc_Nido contains the Main.StdControl, the
StdControl that initializes, starts, and stops all other StdControls):

```c
# 70 "/home/mjp/Tinyos/tinyos-1.x/tos/interfaces/StdControl.nc"
inline static  result_t pc_Nido$StdControl$start(void){
 #line 70
   unsigned char result;
 #line 70
 #line 70
   result = TimerMmulti$StdControl$start();
 #line 70
   result = rcombine(result, BlinkM1$StdControl$start());
 #line 70
   result = rcombine(result, TimerMmulti$StdControl$start());
 #line 70
   result = rcombine(result, BlinkM2$StdControl$start());
 #line 70
```

\(^5\) [http://cs490.mine.nu/src/nesc/src/nesc-main.c](http://cs490.mine.nu/src/nesc/src/nesc-main.c),
\(^6\) [http://cs490.mine.nu/src/nesc/src/machine/selfapp.c](http://cs490.mine.nu/src/nesc/src/machine/selfapp.c)
\(^7\) [http://cs490.mine.nu/src/tinyos/apps/bigBlink/](http://cs490.mine.nu/src/tinyos/apps/bigBlink/)
Thus, wiring multiple components to one interface does what one might expect: merges them together under (ultimately) one function. Take, for instance, the two different timers for bigBlink are coupled and blink on and off at the same:

Firstly, to enable multiple programs to run requires multiple interfaces in components that must link with more than one application. Adding additional interfaces, though, requires them to be wired to some component; thus, both a null output interface module

```plaintext
module NoLeds {  
  provides interface Leds;  
}  
implementation  
{  
  async command result_t Leds.init() {  
    return SUCCESS;  
  }  
}
```

and a null input interface module

```plaintext
module NoStdControlU1 {  
  uses interface StdControl1;  
}  
implementation  
{  
  int i;  
}
```

http://cs490.mine.nu/src/tinyos/tos/system/NoLeds.nc

http://cs490.mine.nu/src/tinyos/tos/platform/pcapp/NoStdControlU1.nc
must be provided. Additionally, a method of determining which interfaces are actually being used, and which have already been allocated, will need to be determined at compile-time, less either the user be forced to do it manually or checks be placed within the code to test if the interface being used is null and should be ignored. It is also uncertain what effect some of the empty input interfaces have on correct system operation; typically, it seems, the functions seems to be converted to empty, static inlined functions in the app.c file.

Additional problems were realized when several system components lacked high level synchronization (a reasonable consideration in most cases, given only one application is using the structures). One application, though, that appeared to support multiple instances (TimerM.nc) proved to store only enough state information for one instance. Modifications to TimerMmulti.nc provide an example of typical modifications that will have to be made to other parts of the system. After modifications, however, a problem that [Levis03] note began to crop up: “our experience with TinyOS has shown that while failure bugs are usually quickly found, bugs that produce operational but aberrant behavior…are far more difficult to discover” [Levis03]. Ultimately, the project did not successfully implement a working pseudo-multiprogramming version of TinyOS; the lessons learned in the process, however, might provide stepping stones for future revision and expansion of the OS.

**References:** Entries followed by “pdf” are available at [http://cs490.mine.nu/papers/](http://cs490.mine.nu/papers/)


[Engler95] Dawson R. Engler, Frans Kaashoek, and James O. Toole Jr., Exokernel: an operating system architecture for application-level resource management, MIT, 1995


