

Power Aware Operating Systems: Task-Specific CPU Throttling

Intitial Draft
Sunny Gleason
COM S 790
Fall 2001

1 Introduction

With the explosive growth in notebook, handheld and embedded computers, mechanisms to facilitate strategic power management are becoming more and more important. One important relationship in synchronous processors is the superlinear increase in power consumption with respect to increasing CPU clock speed. Modern processors (such as the Transmeta Crusoe TM5600) incorporate hardware mechanisms for adjusting CPU speed “on the fly;” however, the processors themselves only have access to a stream of instructions and memory references. We believe that the incorporation of OS-, application-, and user-level feedback is vital to the success of any power management system.

This paper describes a first step in building more power-aware systems: the incorporation of user feedback in setting CPU speed for tasks in the Linux operating system. One long-term goal is to enable input of power-management feedback in multiple stages of program development and execution: from strategic placement of “hooks” during programming and compilation, to user- and OS-level feedback at run-time.

This document consists of three main sections. Section 2 describes the details of our mobile platform. Section 3 contains descriptions of the various interfaces that we have implemented, while Section 4 describes the actual implementation.

2 Platform Description

The computer itself is a Sony Vaio “Picturebook” notebook computer (model #C1VN) running the Red Hat Linux 7.1 Operating System. We worked with several kernels in the 2.4.x series; our latest build was based on kernel version 2.4.12. The CPU is a Transmeta Crusoe TM5600 with Longrun technology. The CPU supports 4 different CPU settings, presented in the table below:

Boost %	MHz	Volts	
0	300	1.300	
33	400	1.350	(1)
66	500	1.400	
100	600	1.600	

Without the power management features we have implemented, a typical Linux user interaction is as follows. Using the Longrun command-line utility provided by Transmeta, the user (as *root*) specifies global *low* and *high* values for the CPU boost %. The CPU itself decides which level to assume between the user-specified values; if *low* = *high*, the CPU assumes the specified value.

The Longrun utility works by writing the two integer values to a model-specific register (MSR) using the *wrmsr* instruction. However, since *wrmsr* is

a privileged instruction, the user's command must go through several hoops before the actual *wrmsr* takes place. The Longrun MSR is accessible through the MSR dev entry located at “/dev/cpu/0/msr”. In order to change the value, the user program must open the device, read the current value, and write the updated value to the MSR.

This scenario has 2 other issues of note. First, only the *root* user has permission to change the CPU speed; even if other users were given the ability to change the speed, simultaneous conflicting changes would yield undesirable performance a multi-user system. More importantly, the CPU speed is tied to 2 system-wide constants (*low* and *high*); between those values, the CPU boost percentage is decided solely at the hardware level.

3 Interfaces

In a system where CPU boost changes occur frequently (at the limit of our example, during every context switch), it is important to bypass the 3-system call overhead associated with setting the MSR. To do this, we add an integer *boost* value to every process control block in the system, and modify the kernel context switch routine to change the values of *low* and *high* when switching between 2 processes of different boost values. We implement two system calls to interact with boost values in the PCB: *getboost* and *setboost*. These two system calls are the heart of the system; using these two system calls, we build a command-line utility (similar to *nice*), a Perl native library, as well as a graphical user interface using Perl/Tk.

3.1 System Call API

In our preliminary implementation, user programs interact with process boost values using two system calls: *getboost* and *setboost*. The system-call approach seems to yield the simplest implementation and the fastest performance; it is conceivable that the interface may switch to a design based on a kernel module interacting through the */proc* filesystem.

The *getboost* system call takes an integer argument (the PID of the process to be boosted) and returns an integer (the boost level of the specified process, or a negative error code value).

The *setboost* system call takes two integer arguments (the desired boost value, as well as the PID of the process to be boosted) and returns an integer result code (positive for success, or a negative error code value).

3.2 Command-Line Interface

The command line interface, called *boost*, is very similar to the *nice* command, used to set the priority of processes in UNIX systems. The *boost* command may

be called with one or two arguments. When called with one integer argument, it interprets the argument as a PID which is used in a *getboost* system call; the result of the call is printed to standard output. When called with two arguments, the values are interpreted as a PID and a boost value which are used in a *setboost* system call; the result of the call is printed to standard output.

3.3 Perl Native Library

The Perl native library consists of a single Perl module, named *Boost*, created using the *h2xs* utility that is part of the standard Perl 5.6 distribution. The module exports 2 functions, *Boost::getboost* and *Boost::setboost*, which provide the same functionality as the C system call API (along with some additional Perl overhead for marshalling/demarshalling arguments).

3.4 Graphical User Interface

The graphical user interface, called *Boostgui*, is implemented using Perl/Tk. The interface consists of a process list which is loaded when the GUI is started, as well as a button to refresh the process list. The process list contains a list of process command lines, along with the associated users and PIDs. To change the boost value of a process, the user double-clicks its entry in the process list. A modal dialog containing 5 buttons is presented to the user (4 boost values, plus Cancel), with the old boost value highlighted by default. The user chooses the new boost value from the dialog by pressing the appropriate button, or presses Cancel to abort the change. If the GUI is unable to process the boost change (for example, if the user does not own the process), the GUI displays an appropriate error message to the user.

4 Implementation

The system implementation is surprisingly simple: the main source of complexity is are the two system calls *getboost* and *setboost*. Details of our system implementation are provided in the following section: from the system call API to the implementation of the boost GUI.

4.1 System Call API

To implement the system calls, we made several small changes to the Linux 2.4.12 kernel. These changes involved modifying the process control block, (or *task_struct*), implementing *getboost* and *setboost* in kernel space, adding entries to the kernel system call table, and modifying the kernel context switch routine.

In each of the following subsections, we give all paths relative to a */linux* root folder, which is expected to contain the source files and directories of the linux kernel distribution.

4.1.1 Modifying the PCB

The *task_struct* struct is contained within the file */linux/include/linux/sched.h*. To accommodate a separate boost value for each process, we added a single integer *boost* field to the *task_struct* struct. In addition, we modified the *INIT_TASK(tsk)* macro to set the default boost value to zero.

4.1.2 System-call Implementation: Kernel Space

We based the implementation of *sys_getboost* and *sys_setboost* on the existing *sys_getpriority* and *sys_setpriority* functions, respectively. The *priority* functions are located in the file */linux/kernel/sys.c*, so we placed our *boost* functions there, too. The functions make sure their arguments are within range, lock the task list, perform the necessary operations on the task list, unlock the task list, and return the appropriate values.

Of course, since the functions operate in kernel mode on kernel data structures, the *sys_* calls cannot be called directly from user space. Instead, the user must follow the Linux system call protocol. Roughly, the protocol consists of the following:

1. Place the integer identifier of the desired system call in *eax*
2. Place the system call arguments in other registers, and onto the user stack as necessary
3. Execute an *int 0x80* software interrupt instruction
4. *Execution jumps to the kernel system call interrupt handler, which jumps to the instruction pointed to by the function pointer in the kernel system call table indexed by the specified identifier*
5. *the kernel executes the implementation of the system call in privileged mode, places the return value in the *eax* register, and executes a return from interrupt instruction*

In order to be able to call the system calls from user space, we must place the appropriate function entries within the kernel system call table. This table is located in */linux/arch/i386/kernel/entry.S*. We add two new entries at the bottom of the *sys_call_table* entry, containing pointers to the *sys_getboost* and *sys_setboost* functions. To provide meaningfully-named constants to user applications, we also add entries to the file */linux/include/asm-i386/unistd.h*. The entries in *unistd.h* are simple C preprocessor definitions of the symbols *__NR_getboost* and *__NR_setboost* to the integers 225 and 226, representing their offsets in the kernel symbol table.

4.2 Task-Specific CPU Throttling

To implement CPU boost value changes at context-switch time, we modified the context-switch preparation routine, called `__switch_to`, within the file `/linux/arch/i386/kernel/process.c`. Using the `read_msr` and `write_msr` routines contained within the `msr.c` file in the same directory, and the hardware MSR values specified in the source code of the Transmeta Longrun utility, we created two inline functions called `rd_msr` and `wr_msr`. Inside the `__switch_to` function, we:

1. disable interrupts (to avoid pre-emption at this critical moment!)
2. read the present value of the Longrun MSR
3. check whether it is different from the desired boost value
4. if it is different, write the desired value to the MSR
5. restore interrupts

Although we omit the additional bit-flipping magic that is necessary to get the boost value in the correct format for the Longrun MSR, the above procedure fully describes our kernel-space implementation of process-specific CPU throttling.

4.3 The Boost System Library

To facilitate invocation of boost system calls from user space, we create a library called `boostlib` containing the stub functions `getboost` and `setboost`. The files are contained within the `/boostlib/boostlib.c` file. These stub functions are similar to the stub functions that are found in standard `libc` implementations (such as `glibc`): they place the user arguments and system call identifier into the necessary registers (and stack locations, if necessary), invoke the `int 0x80` instruction, and return the return value to the user upon completion of the system call.

4.4 Command-Line Interface

The command-line interface (contained within `/boostcmd/boost.c`) is simply a wrapper around the `boostlib` library. If any error codes are returned, it uses the `strerror()` function to print a human-understandable error message.

4.5 Perl Native Library

The Perl native library was created using the `h2xs` program included with the standard Perl 5.6 distribution. The file `/boostperl/Boost/Boost.xs` contains two function declarations at the bottom, describing the types of arguments and return values that it should use for the `getboost` and `setboost` functions described

in */boostlib/boostlib.h*. The file */boostperl/Boost/Boost.pm* contains the Dynaloader (dynamic library loader) directive, as well as *perldoc* documentation for the *Boost* package.

4.6 Graphical User Interface

The *boostgui* program is implemented in Perl/Tk; it is a simple GUI wrapper around the *Boost* package, which is a Perl wrapper around the *boostlib* C library.

5 Conclusion

In this paper, we have described our implementation of process-specific CPU throttling for Transmeta Crusoe TM5600 systems running the Linux operating system. Our implementation is surprisingly simple, enabling us to provide support for CPU throttling in multiple programming languages. This experiment is an important proof-of-concept; it paves the way for more sophisticated OS- and user- and compiler-driven control of power management functionality in mobile and embedded systems.