Multi-processor and Frequency Scaling

Making Your Server Behave Like a Laptop

Paul Devriendt
AMD Software Research and Development
paul.devriendt@amd.com

Abstract

This paper will explore a multi-processor implementation of frequency management, using an AMD Opteron™ processor 4-way server as a test vehicle.

Topics will include:

- the benefits of doing this, and why server customers are asking for this,
- the hardware, for case of the AMD Opteron processor,
- the various software components that make this work,
- the issues that arise, and
- some areas of exploration for follow on work.

1 Introduction

Processor frequency management is common on laptops, primarily as a mechanism for improving battery life. Other benefits include a cooler processor and reduced fan noise. Fans also use a non-trivial amount of power.

This technology is spreading to desktop machines, driven both by a desire to reduce power consumption and to reduce fan noise.

Servers and other multiprocessor machines can equally benefit. The multiprocessor frequency management scenario offers more complexity (no surprise there). This paper discusses these complexities, based upon a test implementation on an AMD Opteron processor 4-way server. Details within this paper are AMD processor specific, but the concepts are applicable to other architectures.

The author of this paper would like to make it clear that he is just the maintainer of the AMD frequency driver, supporting the AMD Athlon™ 64 and AMD Opteron processors. This frequency driver fits into, and is totally dependent, on the CPUFreq support. The author has gratefully received much assistance and support from the CPUFreq maintainer (Dominik Brodowski).

2 Abbreviations

BKDG: The BIOS and Kernel Developer's Guide. Document published by AMD containing information needed by system software developers. See the references section, entry 4.

MSR: Model Specific Register. Processor registers, accessable only from kernel space, used for various control functions. These registers are expected to change across processor families. These registers are described in the

BKDG[4].

VRM: Voltage Regulator Module. Hardware external to the processor that controls the voltage supplied to the processor. The VRM has to be capable of supplying different voltages on command. Note that for multiprocessor systems, it is expected that each processor will have its own independent VRM, allowing each processor to change voltage independently. For systems where more than one processor shares a VRM, the processors have to be managed as a group. The current frequency driver does not have this support.

fid: Frequency Identifier. The values written to the control MSR to select a core frequency. These identifiers are processor family specific. Currently, these are six bit codes, allowing the selection of frequencies from 800 MHz to 5 Ghz. See the BKDG[4] for the mappings from fid to frequency. Note that the frequency driver does need to "understand" the mapping of fid to frequency, as frequencies are exposed to other software components.

vid: Voltage Identifier. The values written to the control MSR to select a voltage. These values are then driven to the VRM by processor logic to achieve control of the voltage. These identifiers are processor model specific. Currently these identifiers are five bit codes, of which there are two sets—a standard set and a low-voltage mobile set. The frequency driver does not need to be able to "understand" the mapping of vid to voltage, other than perhaps for debug prints.

VST: Voltage Stabilization Time. The length of time before the voltage has increased and is stable at a newly increased voltage. The driver has to wait for this time period when stepping the voltage up. The voltage has to be stable at the new level before applying a further step up in voltage, or before transitioning to a new frequency that requires the higher voltage.

MVS: Maximum Voltage Step. The maximum voltage step that can be taken when increasing the voltage. The driver has to step up voltage in multiple steps of this value when increasing the voltage. (When decreasing voltage it is not necessary to step, the driver can merely jump to the correct voltage.) A typical MVS value would be 25mV.

RVO: Ramp Voltage Offset. When transitioning frequencies, it is necessary to temporarily increase the nominal voltage by this amount during the frequency transition. A typical RVO value would be 50mV.

IRT: Isochronous Relief Time. During frequency transitions, busmasters briefly lose access to system memory. When making multiple frequency changes, the processor driver must delay the next transition for this time period to allow busmasters access to system memory. The typical value used is 80us.

PLL: Phase Locked Loop. Electronic circuit that controls an oscillator to maintain a constant phase angle relative to a reference signal. Used to synthesize new frequencies which are a multiple of a reference frequency.

PLL Lock Time: The length of time, in microseconds, for the PLL to lock.

pstate: Performance State. A combination of frequency/voltage that is supported for the operation of the processor. A processor will typically have several pstates available, with higher frequencies needing higher voltages. The processor clock can not be set to any arbitrary frequency; it may only be set to one of a limited set of frequencies. For a given frequency, there is a minimum voltage needed to operate reliably at that frequency, and this is the correct voltage, thus forming the frequency/voltage pair.

ACPI: Advanced Configuration and Power Interface Specification. An industry specification, initially developed by Intel, Microsoft, Phoenix and Toshiba. See the reference section, entry 5.

_PSS: Performance Supported States. ACPI object that defines the performance states valid for a processor.

_PPC: Performance Present Capabilities. ACPI object that defines which of the _PSS states are currently available, due to current platform limitations.

PSB Performance State Block. BIOS provided data structure used to pass information, to the driver, concerning the pstates available on the processor. The PSB does not support multiprocessor systems (which use the ACPI _PSS object instead) and is being deprecated. The format of the PSB is defined in the BKDG.

3 Why Does Frequency Management Affect Power Consumption?

Higher frequency requires higher voltage. As an example, data for part number ADA3200AEP4AX:

2.2 GHz @ 1.50 volts, 58 amps max – 89 watts

2.0 GHz @ 1.40 volts, 48 amps max – 69 watts

1.8 GHz @ 1.30 volts, 37 amps max – 50 watts

1.0 GHz @ 1.10 volts, 18 amps max – 22 watts

These figures are worst case current/power figures, at maximum case temperature, and include I/O power of 2.2W.

Actual power usage is determined by:

- code currently executing (idle blocks in the processor consume less power),
- activity from other processors (cache coherency, memory accesses, pass-through traffic on the HyperTransport™ connections),
- processor temperature (current increases with temperature, at constant workload and voltage),
- processor voltage.

Increasing the voltage allows operation at higher frequencies, at the cost of higher power consumption and higher heat generation. Note that relationship between frequency and power consumption is not a linear relationship—a 10% frequency increase will cost more than 10% in power consumption (30% or more).

Total system power usage depends on other devices in the system, such as whether disk drives are spinning or stopped, and on the efficiency of power supplies.

4 Why Should Your Server Behave Like A Laptop?

- Save power. It is the right thing to do for the environment. Note that power consumed is largely converted into heat, which then becomes a load on the air conditioning in the server room.
- Save money. Power costs money. The power savings for a single server are typically regarded as trivial in terms of a corporate budget. However, many large organizations have racks of many thousands of servers. The power bill is then far from trivial.
- Cooler components last longer, and this translates into improved server reliability.
- Government Regulation.

5 Interesting Scenarios

These are real world scenarios, where the application of the technology is appropriate.

5.1 Save power in an idle cluster

A cluster would typically be kept running at all times, allowing remote access on demand. During the periods when the cluster is idle, reducing the CPU frequency is a good way to reduce power consumption (and therefore also air conditioning load), yet be able to quickly transition back up to full speed (<0.1 second) when a job is submitted.

User space code (custom to the management of that cluster) can be used to offer cluster speeds of “fast” and “idle,” using the `/proc` or `/sys` file systems to trigger frequency transitions.

5.2 The battery powered server

Or, the server running on a UPS.

Many production servers are connected to a battery backup mechanism (UPS—uninterruptible power supply) in case the mains power fails. Action taken on a mains power failure varies:

- Orderly shutdown.
- Stay up and running for as long as there is battery power, but orderly shutdown if mains power is not restored.
- Stay up and running, mains power will be provided by backup generators as soon as the generators can be started.

In these scenarios, transitioning to lower performance states will maximize battery life, or reduce the amount of generator/battery power capacity required.

UPS notification of mains power loss to the server for administrator alerts is well understood technology. It is not difficult to add the support for transitioning to a lower pstate. This can be done by either a cpufreq governor or by adding the simple user space controls to an existing user space daemon that is monitoring the UPS alerts.

5.3 Server At Less Than Maximum Load

As an example, a busy server may be processing 100 transactions per second, but only 5 transactions per second during quiet periods. Reducing the CPU frequency from 2.2 GHz to 1.0 GHz is not going to impact the ability of that server to process 5 transactions per second.

5.4 Processor is not the bottleneck

The bottleneck may not be the processor speed. Other likely bottlenecks are disk access and network access. Having the processor waiting faster may not improve transaction throughput.

5.5 Thermal cutback to avoid over temperature situations

The processors are the main generators of heat in a system. This becomes very apparent when many processors are in close proximity, such as with blade servers. The effectiveness of the processor cooling is impacted when the processor heat sinks are being cooled with hot air. Reducing processor frequency when idle can dramatically reduce the heat production.

5.6 Smaller Enclosures

The drive to build servers in smaller boxes, whether as standalone machines or slim rackmount machines, means that there is less space for air to circulate. Placing many slim rackmounts together in a rack (of which the most demanding case is a blade server) aggravates the cooling problem as the neighboring boxes are also generating heat.

6 System Power Budget

The processors are only part of the system. We therefore need to understand the power consumption of the entire system to see how significant processor frequency management is on the power consumption of the whole system.

A system power budget is obviously platform specific. This sample DC (direct current) power budget is for a 4-processor AMD Opteron processor based system. The system has three 500W power supplies, of which one is redundant. Analysis shows that for many operating scenarios, the system could run on a single power supply.

This analysis is of DC power. For the system in question, the efficiency of the power supplies are approximately linear across varying loads, and thus the DC power figures expressed as percentages are meaningful as predictors of the AC (alternating current) power consumption. For systems with power supplies that are not linearly efficient across varying loads, the calculations obviously have to be factored to take account of power supply efficiency.

System components:

- 4 processors @ 89W = 356W in the maximum pstate, 4 @ 22W = 88W in the minimum pstate. These are worst case figures, at maximium case temperature, with the worst case instruction mix. The figures in Table1 are reduced from these maximums by approximately 10% to account for a reduced case temperature and for a workload that does not keep all of the processors' internal units busy.
- Two disk drives (Western Digital 250 GByte SATA), 16W read/write, 10W idle (spinning), 1.3W sleep (not spinning). Note SCSI drives typically consume more power.
- DVD Drive, 10W read, 1W idle/sleep.
- PCI 2.2 Slots – absolute max of 25W per slot, system will have a total power budget that may not account for maximum power in all slots. Estimate 2 slots occupied at a total of 20W.
- VGA video card in a PCI slot. 5W. (AGP would be more like 15W+).
- DDR DRAM, 10W max per DIMM, 40W for 4 GBytes configured as 4 DIMMs.
- Network (built in) 5W.
- Motherboard and components 30W.
- 10 fans @ 6W each. 60W.
- Keyboard + Mouse 3W

See Table 1 for the sample power budget under busy and light loads.

The light load without any frequency reduction is baselined as 100%.

The power consumption is shown for the same light load with frequency reduction enabled, and again where the idle loop incorporates the hlt instruction.

Using frequency management, the power consumption drops to 43%, and adding the use of the hlt instruction (assuming 50% time halted), the power consumption drops further to 33%.

These are significant power savings, for systems that are under light load conditions at times. The percentage of time that the system is running under reduced load has to be known to predict actual power savings.

system load	4 cpus	2 disks	dvd	pci	vga	dram	net	planar	fans	kbd mou	total
busy	320 90%	32	10	20	5	40	5	30	60	3	525W
light load	310 87%	22	1	15	5	38	5	20	60	3	479W 100%
light load, using frequency reduction	79 90%	22	1	15	5	38	5	20	20	3	208W 43%
light load, using frequency reduction and using hlt 50% of the time	32 40%	22	1	15	5	38	5	20	15	3	156 33%

Table 1: Sample System Power Budget (DC), in watts

7 Hardware—AMD Opteron

7.1 Software Interface To The Hardware

There are two MSRs, the FIDVID_STATUS MSR and the FIDVID_CONTROL MSR, that are used for frequency voltage transitions. These MSRs are the same for the single processor AMD Athlon 64 processors and for the AMD Opteron MP capable processors. These registers are not compatible with the previous generation of AMD Athlon processors, and will not be compatible with the next generation of processors.

The CPU frequency driver for AMD processors therefore has to change across processor revisions, as do the ACPI _PSS objects that describe pstates.

The status register reports the current fid and vid, as well as the maximum fid, the start fid, the maximum vid and the start vid of the particular processor.

These registers are documented in the BKDG[4].

As MSRs can only be accessed by executing code (the read msr or write msr instructions) on the target processor, the frequency driver has to use the processor affinity support to force execution on the correct processor.

7.2 Multiple Memory Controllers

In PC architectures, the memory controller is a component of the northbridge, which is traditionally a separate component from the processor. With AMD Opteron processors, the northbridge is built into the processor. Thus, in a multi-processor system there are multiple memory controllers.

See Figure 1 for a block diagram of a two processor system.

If a processor is accessing DRAM that is physically attached to a different processor, the DRAM access (and any cache coherency traffic) crosses the coherent HyperTransport interprocessor links. There is a small performance penalty in this case. This penalty is of the order of a DRAM page hit versus a DRAM page miss, about 1.7 times slower than a local access.

This penalty is minimized by the processor caches, where data/code residing in remote DRAM is locally cached. It is also minimized by Linux's NUMA support.

Note that a single threaded application that is memory bandwidth constrained may benefit from multiple memory controllers, due to the increase in memory bandwidth.

When the remote processor is transitioned to a lower frequency, this performance penalty is worse. An upper bound to the penalty may be calculated as proportional to the frequency slowdown. I.e., taking the remote processor from 2.2 GHz to 1.0 GHz would take the 1.7 factor from above to a factor of 2.56. Note that this is an absolute worst case, an upper bound to the factor. Actual impact is workload dependent.

A worst case scenario would be a memory bound task, doing memory reads at addresses that are pathologically the worst case for the caches, with all accesses being to remote memory. A more typical scenario would see this penalty alleviated by:

- processor caches, where 64 bytes will be read and cached for a single access, so applications that walk linearly through memory will only see the penalty on 64 byte boundaries,
- memory writes do not take a penalty (as processor execution continues without waiting for a write to complete),
- memory may be interleaved,
- kernel NUMA optimizations for non-interleaved memory (which allocate memory local to the processor when possible to avoid this penalty).

7.3 DRAM Interface Speed

The DRAM interface speed is impacted by the core clock frequency. A full table is published in the processor data sheet; Table 2 shows a sample of actual DRAM frequencies for the common specified DRAM frequencies, across a range of core frequencies.

This table shows that certain DRAM speed / core speed combinations are suboptimal.

Effective memory performance is influenced by many factors:

- cache hit rates,
- effectiveness of NUMA memory allocation routines,
- load on the memory controller,
- size of penalty for remote memory accesses,
- memory speed,
- other hardware related items, such as types of DRAM accesses.

It is therefore necessary to benchmark the actual workload to get meaningful data for that workload.

7.4 UMA

During frequency transitions, and when HyperTransport LDTSTOP is asserted, DRAM is placed into self refresh mode. UMA graphics devices therefore can not access DRAM. UMA systems therefore need to limit the time that DRAM is in self refresh mode. Time constraints are bandwidth dependent, with high resolution displays needing higher memory bandwidth. This is handled by the IRT delay time during frequency transitions. When transitioning multiple steps, the driver waits an appropriate length of time to allow external devices to access memory.

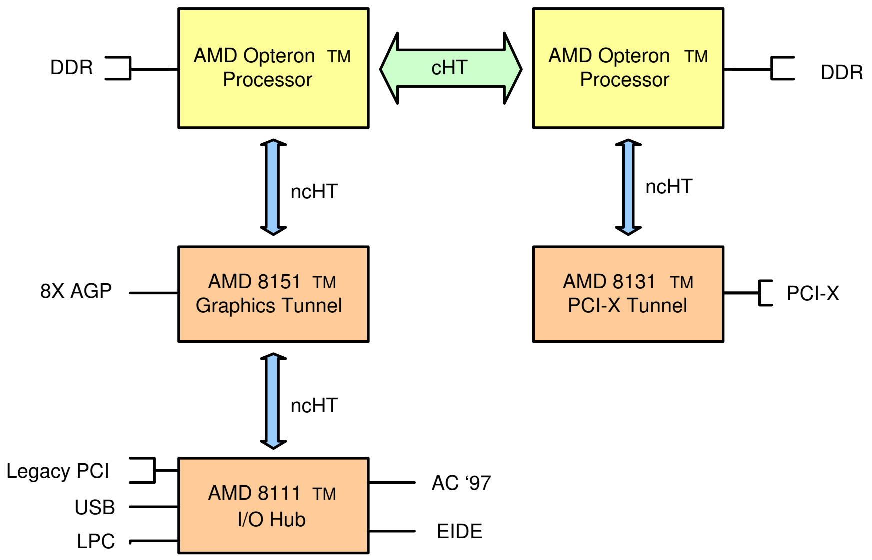

Figure 1: Two Processor System

Processor Core Frequency	100MHz DRAM spec	133MHz DRAM spec	166MHz DRAM spec	200MHz DRAM spec
800MHz	100.00	133.33	160.00	160.00
1000MHz	100.00	125.00	166.66	200.00
2000MHz	100.00	133.33	166.66	200.00
2200MHz	100.00	129.41	157.14	200.00

Table 2: DRAM Frequencies For A Range Of Processor Core Frequencies

7.5 TSC Varying

The Time Stamp Counter (TSC) register is a register that increments with the processor clock. Multiple reads of the register will see increasing values. This register increments on each core clock cycle in the current generation of processors. Thus, the rate of increase of the TSC when compared with "wall clock time" varies as the frequency varies. This causes problems in code that calibrates the TSC increments against an external time source, and then attempts to use the TSC to measure time.

The Linux kernel uses the TSC for such timings, for example when a driver calls udelay(). In this case it is not a disaster if the udelay() call waits for too long as the call is defined to allow this behavior. The case of the udelay() call returning too quickly can be fatal, and this has been demonstrated during experimentation with this code.

This particular problem is resolved by the cpufreq driver correcting the kernel TSC calibration whenever the frequency changes.

This issue may impact other code that uses the TSC register directly. It is interesting to note that it is hard to define a correct behavior. Code that calibrates the TSC against an external clock will be thrown off if the rate of increment of the TSC should change. However, other code may expect a certain code sequence to consistently execute in approximately the same number of cycles, as measured by the TSC, and this code will be thrown off if the behavior of the TSC changes relative to the processor speed.

7.6 Measurement Of Frequency Transition Times

The time required to perform a transition is a combination of the software time to execute the required code, and the hardware time to perform the transition.

Examples of hardware wait time are:

- waiting for the VRM to be stable at a newer voltage,
- waiting for the PLL to lock at the new frequency,
- waiting for DRAM to be placed into and then taken out of self refresh mode around a frequency transition.

The time taken to transition between two states is dependent on both the initial state and the target state. This is due to :

- multiple steps being required in some cases,
- certain operations are lengthier (for example, voltage is stepped up in multiple stages, but stepped down in a single step),
- difference in code execution time dependent on processor speed (although this is minor).

Measurements, taken by calibrating the frequency driver, show that frequency transitions for a processor are taking less than 0.015 seconds.

Further experimentation with multiple processors showed a worst case transition time of less than 0.08 seconds to transition all 4 processors from minimum to maximum frequency, and slightly faster to transition from maximum to minimum frequency.

Note, there is a driver optimization under consideration that would approximately halve these transition times.

7.7 Use of Hardware Enforced Throttling

The southbridge (I/O Hub, example AMD-8111™ HyperTransport I/O Hub) is capable of initiating throttling via the HyperTransport stopclock message, which will ramp down the CPU grid by the programmed amount. This may be initiated by the southbridge for thermal throttling or for other reasons.

This throttling is transparent to software, other than the performance impact.

This throttling is of greatest value in the lowest pstate, due to the reduced voltage.

The hardware enforced throttling is generally not of relevance to the software management of processor frequencies. However, a system designer would need to take care to ensure that the optimal scenarios occur—i.e., transition to a lower frequency/voltage in preference to hardware throttling in high pstates. The BIOS configurations are documented in the BKDG[4].

For maximum power savings, the southbridge would be configured to initiate throttling when the processor executes the `hlt` instruction.

8 Software

The AMD frequency driver is a small part of the software involved. The frequency driver fits into the CPUFreq architecture, which is part of the 2.6 kernel. It is also available as a patch for the 2.4 kernel, and many distributions do include it.

The CPUFreq architecture includes kernel support, the CPUFreq driver itself (`drivers/cpufreq`), an architecture specific driver to control the hardware (powernow-k8.ko is this case), and `/sys` file system code for userland access.

The kernel support code (`linux/kernel/cpufreq.c`) handles timing changes such as updating the kernel constant `loops_per_jiffies`, as well as notifiers (system components that need to be notified of a frequency change).

8.1 History Of The AMD Frequency Driver

The CPU frequency driver for AMD Athlon (the previous generation of processors) was developed by Dave Jones. This driver supports single processor transitions only, as the pstate transition capability was only enabled in mobile processors. This driver used the PSB mechanism to determine valid pstates for the processor. This driver has subsequently been enhanced to add ACPI support.

The initial AMD Athlon 64 and AMD Opteron driver (developed by me, based upon Dave's earlier work, and with much input from Dominik and others), was also PSB based. This was followed by a version of the driver that added ACPI support.

The next release is intended to add a built-in table of pstates that will allow the checking of BIOS supplied data, and also allow an override capability to provide pstate data when not supplied by BIOS.

8.2 User Interface

The deprecated `/proc/cpufreq` (and `/proc/sys`) file system offers control over all processors or individual processors. By echoing values into this file, the root user can change policies and change the limits on available frequencies.

Examples:

Constrain all processors to frequencies between 1.0 GHz and 1.6 GHz, with the performance policy (effectively chooses 1.6 GHz):

```
echo -n "1000000:16000000:
performance" > /proc/cpufreq
```

Constrain processor 2 to run at only 2.0 GHz:

```
echo -n "2:2000000:2000000:
performance" > proc/cpufreq
```

The "performance" refers to a policy, with the other policy available being "powersave." These policies simply forced the frequency to be at the appropriate extreme of the available range. With the 2.6 kernel, the choice is normally for a "userspace" governor, which allows the (root) user or any user space code (running with root privilege) to dynamically control the frequency.

With the 2.6 kernel, a new interface in the `/sys` filesystem is available to the root user, deprecating the `/proc/cpufreq` method.

The control and status files exist under `/sys/devices/system/cpu/cpuN/cpufreq`, where N varies from 0 upwards, dependent on which processors are online. Among the other files in each processor's directory, `scaling_min_freq` and `scaling_max_freq` control the minimum and maximum of the ranges in which the frequency may vary. The `scaling_governor` file is used to control the choice of governor. See `linux/Documentation/cpu-freq/userguide.txt` for more information.

Examples:

Constrain processor 2 to run only in the range 1.6 GHz to 2.0 GHz:

```
cd /sys/devices/system/cpu

cd cpu2/cpufreq

echo 1600000 > scaling_min_freq

echo 2000000 > scaling_max_freq
```

8.3 Control From User Space And User Daemons

The interface to the `/sys` filesystem allows userland control and query functionality. Some form of automation of the policy would normally be part of the desired complete implementation.

This automation is dependent on the reason for using frequency management. As an example, for the case of transitioning to a lower pstate when running on a UPS, a daemon will be notified of the failure of mains power, and that daemon will trigger the frequency change by writing to the control files in the `/sys` filesystem.

The CPUFreq architecture has thus split the implementation into multiple parts:

1. user space policy
2. kernel space driver for common functionality
3. kernel space driver for processor specific implementation.

There are multiple user space automation implementations, not all of which currently support multiprocessor systems. One that does, and that has been used in this project is cpufreqd version 1.1.2 (`http://sourceforge.net/projects/cpufreqd`).

This daemon is controlled by a configuration file. Other than making changes to the configuration file, the author of this paper has not been involved in any of the development work on cpufreqd, and is a mere user of this tool.

The configuration file specifies profiles and rules. A profile is a description of the system settings in that state, and my configuration file is setup to map the profiles to the processor pstates. Rules are used to dynamically choose which profile to use, and my rules are setup to transition profiles based on total processor load.

My simple configuration file to change processor frequency dependent on system load is:

```
[General]
pidfile=/var/run/cpufreqd.pid
poll_interval=2
pm_type=acpi

# 2.2 GHz processor speed
[Profile]
name=hi_boost
minfreq=95%
maxfreq=100%
policy=performance

# 2.0 GHz processor speed
[Profile]
name=medium_boost
minfreq=90%
maxfreq=93%
policy=performance

# 1.0 GHz processor Speed
[Profile]
name=lo_boost
minfreq=40%
maxfreq=50%
policy=powersave

[Profile]
name=lo_power
minfreq=40%
maxfreq=50%
policy=powersave

[Rule]
#not busy 0%-40%
name=conservative
ac=on
battery_interval=0-100
cpu_interval=0-40
profile=lo_boost

#medium busy 30%-80%
[Rule]
name=lo_cpu_boost
ac=on
battery_interval=0-100
cpu_interval=30-80
profile=medium_boost

#really busy 70%-100%
[Rule]
name=hi_cpu_boost
ac=on
battery_interval=50-100
cpu_interval=70-100
profile=hi_boost
```

This approach actually works very well for multiple small tasks, for transitioning the frequencies of all the processors together based on a collective loading statistic.

For a long running, single threaded task, this approach does not work well as the load is only high on a single processor, with the others being idle. The average load is thus low, and all processors are kept at a slow speed. Such a workload scenario would require an implementation that looked at the loading of individual processors, rather than the average. See the section below on future work.

8.4 The Drivers Involved

`powernow-k8.ko` `arch/i386/kernel/cpu/cpufreq/powernow-k8.c` (the same source code is built as a 32-bit driver in the `i386` tree and as a 64-bit driver in the `x86_64` tree)

`drivers/acpi`

`drivers/cpufreq`

The Test Driver

Note that the `powernow-k8.ko` driver does not export any read, write, or ioctl interfaces. For test purposes, a second driver exists with an ioctl interface for test application use. The test driver was a big part of the test effort on `powernow-k8.ko` prior to release.

8.5 Frequency Driver Entry Points

powernowk8_init()

Driver `late_initcall`. Initialization is late as the acpi driver needs to be initialized first. Verifies that all processors in the system are capable of frequency transitions, and that all processors are supported processors. Builds a data structure with the addresses of the four entry points for cpufreq use (listed below), and calls `cpufreq_register_driver()`.

powernowk8_exit()

Called when the driver is to be unloaded. Calls `cpufreq_unregister_driver()`.

8.6 Frequency Driver Entry Points For Use By The CPUFreq driver

powernowk8_cpu_init()

This is a per-processor initialization routine. As we are not guaranteed to be executing on the processor in question, and as the driver needs access to MSRs, the driver needs to force itself to run on the correct processor by using `set_cpus_allowed()`.

This pre-processor initialization allows for processors to be taken offline or brought online dynamically. I.e., this is part of the software support that would be needed for processor hotplug, although this is not supported in the hardware.

This routine finds the ACPI pstate data for this processor, and extracts the (proprietary) data from the ACPI `_PSS` objects. This data is verified as far as is reasonable. Per-processor data tables for use during frequency transitions are constructed from this information.

powernowk8_cpu_exit()

Per-processor cleanup routine.

powernowk8_verify()

When the root user (or an application running on behalf of the root user) requests a change to the minimum/maximum frequencies, or to the policy or governor, the frequency driver's verification routine is called to verify (and correct if necessary) the input values. For example, if the maximum speed of the processor is 2.4 GHz and the user requests that the maximum range be set to 3.0 GHz, the verify routine will correct the maximum value to a value that is actually possible. The user can, however, chose a value that is less than the hardware maximum, for example 2.0 GHz in this case.

As this routine just needs to access the per-processor data, and not any MSRs, it does not matter which processor executes this code.

powernowk8_target()

This is the driver entry point that actually performs a transition to a new frequency/voltage. This entry point is called for each processor that needs to transition to a new frequency.

There is therefore an optimization possible by enhancing the interface between the frequency driver and the CPUFreq driver for the case where all processors are to be transitioned to a new, common frequency. However, it is not clear that such an optimization is worth the complexity, as the functionality to transition a single processor would still be needed.

This routine is invoked with the processor number as a parameter, and there is no guarantee as to which processor we are currently executing on. As the mechanism for changing the frequency involves accessing MSRs, it is necessary to execute on the target processor, and the driver forces its execution onto the target processor by using `set_cpus_allowed()`.

The CPUFreq helpers are then used to determine the correct target frequency. Once a chosen target `fid` and `vid` are identified:

- the cpufreq driver is called to warn that a transition is about to occur,
- the actual transition code within powernow-k8 is called, and then
- the cpufreq driver is called again to confirm that the transition was successful.

The actual transition is protected with a semaphore that is used across all processors. This is to prevent transitions on one processor from interfering with transitions on other processors. This is due to the inter-processor communication that occurs at a hardware level when a frequency transition occurs.

8.7 CPUFreq Interface

The CPUFreq interface provides entry points, that are required to make the system function.

It also provides helper functions, which need not be used, but are there to provide common functionality across the set of all architecture specific drivers. Elimination of duplicate good is a good thing! An architecture specific driver can build a table of available frequencies, and pass this table to the CPUFreq driver. The helper functions then simplify the architecture driver code by manipulating this table.

cpufreq_register_driver()

Registers the frequency driver as being the driver capable of performing frequency transitions on this platform. Only one driver may be registered.

cpufreq_unregister_driver()

Unregisters the driver, when it is being unloaded.

cpufreq_notify_transition()

Used to notify the CPUFreq driver, and thus the kernel, that a frequency transition is occurring, and triggering recalibration of timing specific code.

cpufreq_frequency_table_target()

Helper function to find an appropriate table entry for a given target frequency. Used in the driver's target function.

cpufreq_frequency_table_verify()

Helper function to verify that an input frequency is valid. This helper is effectively a complete implementation of the driver's verify function.

cpufreq_frequency_table_cpuinfo()

Supplies the frequency table data that is used on subsequent helper function calls. Also aids with providing information as to the capabilities of the processors.

8.8 Calls To The ACPI Driver

acpi_processor_register_performance()

acpi_processor_unregister_performance()

Helper functions used at per-processor initialization time to gain access to the data from the _PSS object for that processor. This is a preferable solution to the frequency driver having to walk the ACPI namespace itself.

8.9 The Single Processor Solution

Many of the kernel system calls collapse to constants when the kernel is built without multiprocessor support. For example, `num_online_cpus()` becomes a macro with the value 1. By the careful use of the definitions in smp.h, the same driver code handles both multiprocessor and single processor machines without the use of conditional compilation. The multiprocessor support obviously adds complexity to the code for a single processor code, but this code is negligible in the case of transitioning frequencies. The driver initialization and termination code is made more complex and lengthy, but this is not frequently executed code. There is also a small penalty in terms of code space.

The author does not feel that the penalty of the multiple processor support code is noticeable on a single processor system, but this is obviously debatable. The current choice is to have a single driver that supports both single processor and multiple processor systems.

As the primary performance cost is in terms of additional code space, it is true that a single processor machine with highly constrained memory may benefit from a simplified driver without the additional multi-processor support code. However, such a machine would see greater benefit by eliminating other code that would not be necessary on a chosen platform. For example, the PSB support code could be removed from a memory constrained single processor machine that was using ACPI.

This approach of removing code unnecessary for a particular platform is not a wonderful approach when it leads to multiple variants of the driver, all of which have to be supported and enhanced, and which makes Kconfig even more complex.

8.10 Stages Of Development, Test And Debug Of The Driver

The algorithm for transitioning to a new frequency is complex. See the BKDG[4] for a good description of the steps required, including flowcharts. In order to test and debug the frequency/voltage transition code thoroughly, the author first wrote a simple simulation of the processor. This simulation maintained a state machine, verified that fid/vid MSR control activity was legal, provided fid/vid status MSR results, and wrote a log file of all activity. The core driver code was then written as an application and linked with this simulation code to allow testing of all combinations.

The driver was then developed as a skeleton using printk to develop and test the BIOS/ACPI interfaces without having the frequency/voltage transition code present. This is because attempts to actually transition to an invalid pstate often result in total system lockups that offer no debug output—if the processor voltage is too low for the frequency, successful code execution ceases.

When the skeleton was working correctly, the actual transition code was dropped into place, and tested on real hardware, both single processor and multiple processor. (The single processor driver was released many months before the multi-processor capable driver as the multiprocessor capable hardware was not available in the marketplace.) The functional driver was tested, using printk to trace activity, and using external hardware to track power usage, and using a test driver to independently verify register settings.

The functional driver was then made available to various people in the community for their feedback. The author is grateful for the extensive feedback received, which included the changed code to implement suggestions. The driver as it exists today is considerably improved from the initial release, due to this feedback mechanism.

9 How To Determine Valid PStates For A Given Processor

AMD defines pstates for each processor. A performance state is a frequency/voltage pair that is valid for operation of that processor. These are specified as fid/vid (frequency identifier/voltage identifier values) pairs, and are documented in the Processor Thermal and Data Sheets (see references). The worst case processor power consumption for each pstate is also characterized. The BKDG[4] contains tables for mapping fid to frequency and vid to voltage.

Pstates are processor specific. I.e., 2.0 GHz at 1.45V may be correct for one model/revision of processor, but is not necessarily correct for a different/revision model of processor.

Code can determine whether a processor supports or does not support pstate transitions by executing the cpuid instruction. (For details, see the BKDG[4] or the source code for the Linux frequency driver). This needs to be done for each processor in an MP system.

Each processor in an MP system could theoretically have different pstates.

Ideally, the processor frequency driver would not contain hardcoded pstate tables, as the driver would then need to be revised for new processor revisions. The chosen solution is to have the BIOS provide the tables of pstates, and have the driver retrieve the pstate data from the BIOS. There are two such tables defined for use by BIOSs for AMD systems:

1. PSB, AMD's original proprietary mechanism, which does not support MP. This mechanism is being deprecated.
2. ACPI `_PSS` objects. Whereas the ACPI specification is a standard, the data within the `_PSS` objects is AMD specific (and, in fact, processor family specific), and thus there is still a proprietary nature of this solution.

The current AMD frequency driver obtains data from the ACPI objects. ACPI does introduce some limitations, which are discussed later. Experimentation is ongoing with a built-in database approach to the problem in an attempt to bypass these issues, and also to allow checking of validity of the ACPI provided data.

10 ACPI And Frequency Restrictions

ACPI[5] provides the `_PPC` object, that is used to constrain the pstates available. This object is dynamic, and can therefore be used in platforms for purposes such as:

- forcing frequency restrictions when operating on battery power,
- forcing frequency restrictions due to thermal conditions.

For battery / mains power transitions, an ACPI-compliant GPE (General Purpose Event) input to the chipset (I/O hub) is dedicated to assigning a SCI (System Control Interrupt) when the power source changes. The ACPI driver will then execute the ACPI control method (see the `_PSR` power source ACPI object), which issues a notify to the `_CPUn` object, which triggers the ACPI driver to re-evaluate the `_PPC` object. If the current pstate exceeds that allowed by this new evaluation of the `_PPC` object, the CPU frequency driver will be called to transition to a lower pstate.

11 ACPI Issues

ACPI as a standard is not perfect. There is variation among different implementations, and Linux ACPI support does not work on all machines.

ACPI does introduce some overhead, and some users are not willing to enable ACPI.

ACPI requires that pstates be of equivalent power usage and frequency across all processors. In a system with processors that are capable of different maximum frequencies (for example, one processor capable of 2.0 GHz and a second processor capable of 2.2 GHz), compliance with the ACPI specification means that the faster processor(s) will be restricted to the maximum speed of the slowest processor. Also, if one processor has 5 available pstates, the presence of processor with only 4 available pstates will restrict all processors to 4 pstates.

12 What Is There Today?

AMD is shipping pstate capable AMD Opteron processors (revision CG). Server processors prior to revision CG were not pstate capable. All AMD Athlon 64 processors for mobile and desktop are pstate capable.

BKDG[4] enhancements to describe the capability are in progress.

AMD internal BIOSs have the enhancements. These enhancements are rolling out to the publicly available BIOSs along with the BKDG enhancements.

The multi-processor capable Linux frequency driver has released under GPL.

The cpufreqd user-mode daemon, available for download from `http://sourceforge.net/projects/cpufreqd` supports multiple processors.

13 Other Software-directed Power Saving Mechanisms

13.1 Use Of The `HLT` Instruction

The hlt instruction is normally used when the operating system has no code for the processor to execute. This is the ACPI C1 state. Execution of instructions ceases, until the processor is restarted with an interrupt. The power savings are maximized when the hlt state is entered in the minimum pstate, due to the lower voltage. The alternative to the use of the hlt instruction is a do nothing loop.

13.2 Use of Power Managed Chipset Drivers

Devices on the planar board, such as a PCI-X bridge or an AGP tunnel, may have the capability to operate in lower power modes. Entering and leaving the lower power modes is under the control of the driver for that device.

Note that HyperTransport attached devices can transition themselves to lower power modes when certain messages are seen on the bus. However, this functionality is typically configurable, so a chipset driver (or the system BIOS during bootup) would need to enable this capability.

14 Items For Future Exploration

14.1 A Built-in Database

The theory is that the driver could have a built-in database of processors and the pstates that they support. The driver could then use this database to obtain the pstate data without dependencies on ACPI, or use it for enhanced checking of the ACPI provided data. The disadvantage of this is the need to update the database for new processor revisions. The advantages are the ability to overcome the ACPI imposed restrictions, and also to allow the use of the technology on systems where the ACPI support is not enabled.

14.2 Kernel Scheduler—CPU Power

An enhanced scheduler for the 2.6 kernel (2.6.6-bk1) is aware of groups of processors with different processing power. The power tating of each CPU group should be dynamically adjusted using a cpufreq transition notifier as the processor frequencies are changed.

See `http://lwn.net/Articles/80601/` for a detailed acount of the scheduler changes.

14.3 Thermal Management, ACPI Thermal Zones

Publicly available BIOSs for AMD machines do not implement thermal zones. Obviously this is one way to provide the input control for frequency management based on thermal conditions.

14.4 Thermal Management, Service Processor

Servers typically have a service processor, which may be compliant to the IPMI specification. This service processor is able to accurately monitor temperature at different locations within the chassis. The 2.6 kernel includes an IPMI driver. User space code may use these thermal readings to control fan speeds and generate administrator alerts. It may make sense to also use these accurate thermal readings to trigger frequency transitions.

The interaction between thermal events from the service processor and ACPI thermal zones may be a problem.

Hiding Thermal Conditions

One concern with the use of CPU frequency manipulation to avoid overheating is that hardware problems may not be noticed. Over temperature conditions would normally cause administrator alerts, but if the processor is first taken to a lower frequency to hold temperature down, then the alert may not be generated. A failing fan (not spinning at full speed) could therefore be missed. Some hardware components fail gradually, and early warning of imminent failures is needed to perform planned maintenance. Losing this data would be badness.

15 Legal Information

16 References

1. AMD Opteron™ Processor Data Sheet, publication 23932, available from `www.amd.com`

2. AMD Opteron™ Processor Power And

Thermal Data Sheet, publication 30417, available from www.amd.com

3. AMD Athlon™ 64 Processor Power And Thermal Data Sheet, publication 30430, available from www.amd.com

4. BIOS and Kernel Developer's Guide (the BKDG) for AMD Athlon™ 64 and AMD Opteron™ Processors, publication 26094, available from www.amd.com. Chapter 9 covers frequency management.

5. ACPI 2.0b Specification, from www.acpi.info

6. Text documentation files in the kernel linux/Documentation/cpu-freq/ directory:

   - index.txt
   - user-guide.txt
   - core.txt
   - cpu-drivers.txt
   - governors.txt

Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada