Dynamic Power Management for Embedded Systems

IBM and MontaVista Software

Version 1.1, November 19, 2002

Introduction

Power management for computer systems has traditionally focused on regulating the power consumption in static modes such as *sleep* and *suspend*. These are de-activating states, often requiring a user action to re-activate the system. There are usually significant latencies and overheads for entering and exiting these states, and in desktop and server systems a firmware layer typically supports these modes.

Dynamic power management refers to power management schemes implemented while programs are running [**1**]. Many architectures provide the equivalent of a *halt* instruction that reduces CPU power during idle periods. The operating system and device drivers may also manage power of peripheral devices, for example spinning down disks during periods of inactivity. Highly integrated processors with on-board peripherals often include software-controlled clock management capabilities to reduce power consumed by inactive peripherals and peripheral controllers. The memory subsystem also provides a profitable area for dynamic power management, either through the memory controller implementation or through software-based schemes.

Recent advances in processor design techniques have led to the development of systems that support very dynamic power management strategies based on dynamic voltage and frequency scaling. Since CPU power consumption typically decreases with the cube of voltage while frequencies scale linearly with voltage, significant opportunities exist for tuning the power-performance tradeoff to the needs of the application. Processors such as the Transmeta™ Crusoe™, Intel® StrongARM™ and XScale™ processors, and the recently announced IBM® PowerPC™ 405LP allow dynamic voltage and frequency scaling of the processor core in support of these dynamic power management strategies. Aside from the Transmeta system, all of the processors named above are highly integrated system-on-a-chip (SOC) processors designed for embedded applications. The applications of these processors typically do not include a traditional BIOS, therefore control of the dynamic power state of the system must be implemented in the operating system.

The IBM Low-Power Computing Research Center, IBM Linux® Technology Center and MontaVista™ Software are currently developing a general and flexible dynamic power management architecture for embedded systems. The proposal covered in this paper is primarily concerned with the power management implications of dynamic scaling. Several research and production implementations of processor voltage and frequency scaling exist; however, our proposal augments the capabilities of these systems in several important ways. Dynamic power management is still a very active area of research, and research efforts have typically been targeted to investigate a particular strategy or optimization [**2**, **3**, **4**]. Production implementations dictate a more or less fixed power management policy [**5**]. This proposal attempts to standardize a dynamic power management and policy *framework* that will support different power management strategies, either under control of operating system components or user-level policy managers. The flexible framework proposed here will help enable the excellent research being done in this area to find its way into a wider range of commercial products.

The concepts developed here should be applicable to a broad class of operating systems. MontaVista's primary interest is enabling dynamic power management capabilities for the Linux operating system. Although the IBM PowerPC 405LP is used extensively as an example in this paper, both IBM and MontaVista are committed to developing a dynamic power management architecture whose high-level specification is portable across a number of hardware platforms.

Requirements

We recognize that the overriding power management goal in portable systems is to reduce *system-wide* energy consumption. The current generation of embedded processors are so power-efficient that the processor may no longer be the major energy-consumer in systems that include high-performance memories and large color displays. Therefore, a dynamic power management system that is only concerned with voltage and frequency scaling the processor core may be of limited use. Instead, we are committed to enabling aggressive power management strategies that encompass the entire system. For example, scaling bus frequencies can drive significant reductions in system-wide energy consumption. Our dynamic power management architecture supports the ability of processors like the IBM PowerPC 405LP to rapidly scale internal and external bus frequencies, in concert with or even independent of the CPU frequency. A large part of this proposal also deals with the requirement to aggressively manage power consumption based on the states of peripheral devices.

Another key observation is that the breakdown of system-wide energy consumption as well as the most effective way to manage energy consumption are highly application[1]-dependent. Therefore, a dynamic power management architecture needs to be flexible enough to support multiple platforms with differing requirements. Part of this flexibility is the requirement to support "pluggable" power management policies that allow device manufacturers to specialize policies for their applications and differentiate their products based on their own unique approaches to power management. We believe that the requirements for simplicity and flexibility are best served by leaving the workings of the dynamic power management system completely transparent to most tasks, and even to

[1] Throughout this paper we use the terms *system* and *application* in the sense of a complete embedded system, e.g., a cellular phone or PDA, and the terms *program* and *task* to refer to software.

the core of the operating system itself. An implementation based on this proposal need not require any changes to programs. Further, it requires only trivial changes to the well-understood process management implementation of the operating system to achieve significant results.

In highly energy-constrained systems such as cellular phones, however, we believe that task-specific dynamic power management will become a hard requirement. Similar to the way that real-time scheduling policies are used to guarantee predictability, our architecture supports the ability of tasks to set their own power-performance characteristics for those cases where this is required.

Finally, we are aware of the trends in SOC processor design that promise higher levels of integration, symmetric and asymmetric multiprocessing on a single chip and more flexible dynamic power management schemes. IBM Microelectronics, IBM Research and MontaVista Software will take leadership roles in the definition of the hardware and software architectures of these systems. Our power management architecture is based on the capabilities of state-of-the art systems and techniques and also looks forward to next-generation technologies.

Architectural Overview

A high-level overview of our proposed architecture is given in Figure 1. The low-level implementation of the dynamic power management architecture (DPM) is resident in the kernel of the operating system, and power management strategies come from outside of the system. Note that DPM is *not* a self-contained device driver. The low-level implementation of DPM requires enhancements at a few key places in the operating system.

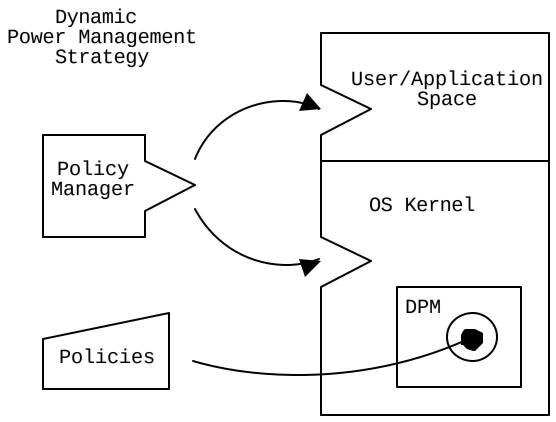


Figure 1:A high-level view of our dynamic power management proposal.

As shown above, we expect a complete dynamic power management strategy to be defined in advance for each application, by a system designer familiar with the characteristics of the embedded system and its special features and requirements. The strategy is communicated to DPM in two ways: as a predefined set of policies and as an application/policy-set specific policy manager that manages them.

DPM *policies* are named data structures. As we explain later, policies may be defined that exert very fine-grained control over the dynamic state of the system. Therefore, policies must be installed into the operating system kernel for efficiency. Policies specify the component and device-state transitions that ensure reliable operation in line with the power management strategy. The structure of DPM policies and the effects of policies on the system are covered in *Policy Architecture* on Page 4. A major component of the policy mechanism deals with the interaction of device states with policies. This feature of the architecture is covered in *Device Constraint Management* on Page 9.

DPM *policy managers* are executable programs that activate policies by name. Policy managers implement user-defined and/or application-specific power management strategies. They can execute either as part of the kernel or in user space (or both) as required by the strategy. Policy managers may be very active, responding in real time to changes in application power/performance requirements, or may be more passive, for example by changing policies on a longer timescale in response to changes in available battery power. In fact, DPM supports strategies that do not require any policy manager at all. Effective strategies for some applications may consist of a single policy installed at system initialization, perhaps modified by critical applications that interact directly with DPM. Some example policy managers and their associated policies are described at a high level in *Example Strategies* on Page 17.

Policy Architecture

A DPM policy is a named data structure, installed into the DPM implementation in the operating system, and managed by a policy manager that may be outside of the operating system. Once a DPM system is initialized and activated, the system will always be executing under a particular DPM policy. The structure of a DPM policy is a hierarchy of abstract objects. In this Section we describe both the policy objects and what they represent. The discussion begins with the concept of a system operating point and system operating states, and concludes with a description of how DPM policies are constructed.

Operating Points

At any given point in time, a system is said to be executing at a particular *operating point*. The operating point may be described by such parameters as the core voltage, CPU and bus frequencies and the states of peripheral devices. A dynamic power management system could properly be defined as the set of rules and procedures that move the system from one operating point to another as events occur. The concept of an operating point also extends to non-operational sleep and hibernate states. The fact that modern voltage and frequency scalable systems support multiple operating points, and the fact that the proper selection of an operating point has a critical impact on system energy consumption are central to this proposal.

The *operating point* is the lowest-level object in the DPM system hierarchy. An operating point object encapsulates the minimal set of inter-dependent, physical and discrete parameters that define a specific system performance level along with an associated energy cost. A good example of inter-dependency is the relation between voltage and frequency of a CPU core. The core voltage limits the maximum operating frequency of a voltage-scalable CPU, and the frequency of the processor cannot be considered without also considering the voltage. As discrete parameters we consider things like SDRAM timing parameters, e.g., “CAS 2”, that are critical for correctness and constrained by other parameters in the operating point, e.g., the SDRAM interface frequency.

By their nature, operating points for advanced processors will be processor- and system-dependent. Under our proposal the system designer is responsible for defining as many operating points as are necessary for the application’s power management needs. As an example, operating points for the IBM PowerPC 405LP currently specify a core voltage level, CPU and bus frequencies, memory timing parameters and other clocking related data. A detailed description of the operating points for the 405LP appear in *Appendix: Operating Points for PowerPC 405LP* on Page 23. Abbreviated details of three 405LP operating points for a particular evaluation platform appear as Table 1. Further below, we explain how a DPM system selects an operating point and transitions from one operating point to another.

Table 1: Three abbreviated operating point descriptions for an IBM PowerPC 405LP reference design.

OPERATING POINT	“33/33”	“200/100”	“266/133”
Core Voltage	1.0 V	1.5 V	1.8 V
PLL VCO Frequency	800 MHz	800 MHz	533 MHz
CPU Frequency (VCO:CPU)	33 MHz (24:1)	200 MHz (4:1)	266 MHz (2:1)
PLB Frequency (CPU:PLB)	33 MHz (1:1)	100 MHz (2:1)	133 MHz (2:1)
EBC Frequency (PLB:EBC)	33 MHz (1:1)	33 MHz (3:1)	33 MHz (4:1)
SDRAM Timing	CAS 2	CAS 2	CAS 3

Operating States

Given that a system supports multiple operating points, some rules and mechanisms are required to move the system from one operating point to another. Current dynamic control mechanisms may set operating points in response to changes in activity or in response to the requests of key programs. The fact that advanced processors like the IBM PowerPC 405LP can scale frequencies with a latency measured in a few microseconds, voltages with a latency measured in tens of microseconds, and all without interrupting system operations in the meantime, means that much more aggressive and finer-grained policies can now be contemplated.

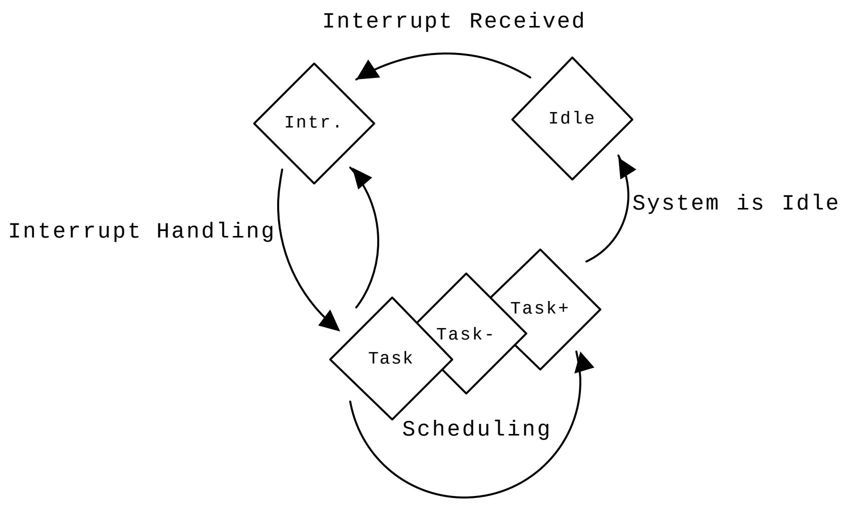


Figure 2: Operating states and state transitions that might be recognized by a DPM implementation.

More generally, Figure 2 illustrates how the operating system can be thought of as a state machine moving through different states in response to events: tasks are scheduled, the system goes idle, interrupts are received and handled, etc. We refer to these system states as *operating states*. In an aggressive dynamic power management policy, each operating state may be associated with an operating point specific to the requirements of that state.

The introduction of the concept of the operating state was first motivated by the observation that significant system-wide energy savings can be achieved by reducing CPU and bus frequencies, and core voltage while the system is idle. Therefore a mechanism is required to specify a different operating point during the time that programs are executing, and the times that the system is idle. This naturally leads to a distinction between an *active* state and an *idle* state, each with a potentially different operating point. The transition from the *active* state operating point to an *idle* state operating point and back is smoothly and efficiently managed by the DPM implementation in the operating system. Others have also explored the possibilities of this type of fine-grained control of the operating point [**6**].

The concept of an operating state also provides for task-specific operating points for power-aware tasks. This requires multiple task-specific *active* states or *task* states. The DPM architecture allows for any number of *task* states. The default *task* state is expected to be used by the large majority of tasks, analogous to the way that most tasks now use the default scheduling policy of the operating system. Tasks with special requirements may specify, or be specified to run in different *task* states, each of which may be associated with a different operating point. Note that tasks never explicitly specify an operating point. Instead, the operating point is implied by the *task* state and the current policy. Task states are discussed in more detail in *Implementation and Effects of Task-Specific Operating States* on page 15, and illustrated in the examples that appear under *Example Strategies* starting on page 17.

Operating states also appear in the DPM policy architecture. Conceptually, a DPM policy simply associates an operating point with each of the system operating states, and changing to a new DPM policy simply changes the association. The actual structure of a DPM policy is much richer in capabilities, however, as explained in the next Section.

Device Management and Operating Point Congruence Classes

The states of on-board and external peripheral devices have a tremendous influence on system-wide energy consumption, and on the choice of operating point. For example, the IBM PowerPC 405LP has an on-board LCD controller which uses a framebuffer stored in external SDRAM. If the LCD controller is enabled, then any valid operating point for the system must specify a memory bus frequency high enough to satisfy the refresh rate of the display, which in turn is determined by the variable pixel clock frequency also specified in the operating point. When the LCD is disabled (for example, when a PDA is used simply as an MP3 player), significant system-wide energy reductions may be achieved by reducing these frequencies.

Our power management architecture relieves the policy manager from the responsibility of managing device states and from having to respond to changes in device states. We rely on low-level device drivers or other system tasks to aggressively manage the power consumption of the devices they control. For example, if a PowerPC 405LP system is not currently producing or consuming audio data, the device driver for the audio CODEC interface may power-down the external CODEC chip as well as command the on-board clock and power manager to remove the clock from the CODEC interface peripheral. From the perspective of DPM, since the CODEC is a DMA peripheral these changes alter the bandwidth (frequency) requirements for the on-board peripheral bus, and it might be profitable to also trigger a change in the operating point since the system is no longer constrained by the DMA requirements of the audio subsystem.

No individual device driver has the global view of the system or the power management strategy required to completely specify the operating point, however. Instead this information is centralized in the DPM policy structure as a *congruence class* of operating points. This object groups together operating points that the system designer considers equivalent for specific operating states modulo a power management strategy. This means that any of the operating points in the class would be acceptable as an operating point for the system in the given operating state, although device constraints might render some members of the class invalid, and power considerations might cause one operating point to be preferred over other valid operating points in the class. At any given point in time a valid DPM policy will designate one member of each congruence class as the selected operating point for that class. The examples under *Device Constraint Management* on Page 9 should help clarify this concept.

Devices specify their requirements as sets of constraints associated with particular device states. For example, an LCD controller might specify a pixel clock in the range of 16 to 25 MHz while active, and no constraint while inactive. When devices change state, and hence their requirements for system resources change, these requirements are communicated to DPM. Simple rules are defined to invalidate inappropriate operating points and to automatically select one of possibly several valid operating points from the congruence class under the new constraints.

This mechanism frees the policy manager to focus on high-level management, while ensuring that the system always operates at the best operating point (as defined by the system designer) consistent with the current policy and device states. This mechanism also supports systems that include simple policy managers, or that do not implement any run-time DPM policy management at all. The system designer may be able to describe a suitable dynamic power management strategy using only a single DPM policy, based on operating point congruence classes that anticipate the significant states of peripheral devices with regard to power management.

Policies and Policy Managers

The highest-level abstraction of the DPM architecture is the *policy*, which maps each operating state to a congruence class of operating points. A power management strategy will specify at least one policy, and may specify as many different policies as necessary for different situations. The policy in effect at any given point in time controls the operating point of the system in every operating state. The complete DPM system hierarchy is illustrated in Figure 3.

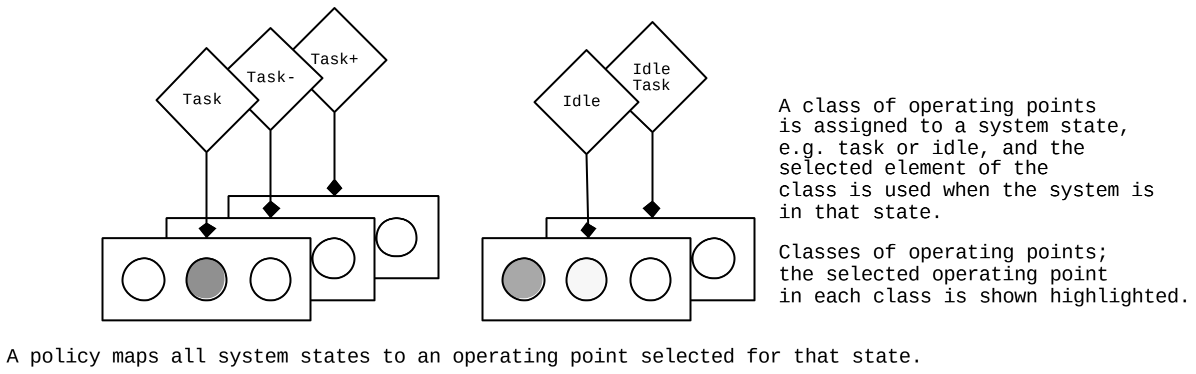

Task
Task-
Task+
Idle
Idle Task
A class of operating points is assigned to a system state, e.g. task or idle, and the selected element of the class is used when the system is in that state.
Classes of operating points; the selected operating point in each class is shown highlighted.
A policy maps all system states to an operating point selected for that state.

Figure 3: A fully enumerated DPM policy assigning each operating state to a congruence class of operating points. Only one operating point from a class is selected (shown highlighted) at any given time.

Note that the DPM architecture does not require the presence of any operating states other than a single *task* state common to all platforms. The number of *task* states may vary from platform to platform; however on all platforms the task states will only be given a meaning by the policies and the policy manager. The examples used in this paper (which show three *task* states, an *idle* state and an *idle-task* state) are representative examples only; DPM does not *require* this system structure.

If multiple policies are needed, then a *policy manager* must exist in the system to coordinate the activation of different policies. The policy manager may collect information from the operating system, user preferences, running programs, configuration files and/or physical devices to make its policy decisions. The "location" of the policy manager (kernel space or user space), the types of information required, and all of the actions taken in response to that information are not specified. The intention of this architecture is simply to define a consistent way for policy developers to express policies that are controlled by the policy manager and implemented by DPM. See *Policy Examples* on page 17 for several examples of how policies and policy managers might work under DPM.

The DPM hierarchy extends to multiprocessing systems as well. In a symmetric multiprocessing system where each processor is identical but independently controlled, it might be advantageous to define an operating point for each processor subsystem. In this case the active policy on each of the processors could actually be different, although derived from a common set of policies. In the case of an asymmetric multiprocessor, if the processors were truly independent then each independent processor subsystem would by necessity have a different type of operating point with a different set of parameters, and each subsystem might also recognize different operating states. This type of asymmetric multiprocessor would require different types of policies for each processor subsystem

Device Constraint Management

The automatic selection of operating points as devices change state is a central feature of DPM. Embedded systems may not have a BIOS or machine abstraction layer to insulate the operating system from low-level device and power management. Therefore this task will fall to the operating system and its device drivers. As the complexity of embedded systems increases, and the interrelationships between clock sources and power management modes become more complex, this becomes an increasingly difficult task. Under the DPM abstraction the system designer becomes an “oracle” for the power management system by pre-selecting sets of meaningful operating points for the application, and organizing these operating points within power management policies that are suitable for the application. Note that there is nothing in the architecture that would restrict a very self-aware system from performing this role as well.

The most aggressive power management strategies will also require the system designer to carefully consider the influence of attached devices on the strategy. This Section uses an example to illustrate the operation of the DPM architecture with respect to dynamically varying requirements from peripheral devices. The process being described in this Section is the automatic mechanism by which the DPM system selects a preferred operating point from a class of equivalent operating points as system devices change state.

The following example is based on reference designs for the IBM PowerPC 405LP. As background, the example design includes a VGA (640 x 480) LCD panel and an external security chip for secure key management functions. The LCD controller is on board the 405LP, and receives a variable-speed pixel clock generated by on-chip clock dividers. The pixel clock frequency determines the LCD refresh rate as well as the Processor Local Bus[2] (PLB) bandwidth to the SDRAM framebuffer required to service the LCD. The security chip requires the 405LP to source a precise 33 MHz clock via an external clock port. Variations in this frequency while the security chip is active may be interpreted as attempts to compromise the system, and cause the security chip to shut down. However, the security chip does allow the clock to be removed when the device is inactive. System energy is conserved by only sourcing the 33 MHz clock while the security chip is performing key management, which is an infrequent occurrence, while correctness requires that exactly 33 MHz be sourced on this port while the chip is active.

[2] The PLB is the on-board system bus in the 405LP, and connects the PowerPC 405 core with the memory controller and all other on-board peripherals.

Figure 4 illustrates a simplified example policy for the PowerPC 405LP for this design. The operating point classes for two operating states, *task* and *idle*, are illustrated. In this simplification[3], an operating point is described by a core voltage (in volts), and a CPU, PLB, pixel clock and external clock frequency specified in MHz. This is a high performance policy that specifies a 1.8 V, 266 MHz operating point for the *task* state, and lower voltage and frequency operating points for the *idle* state.

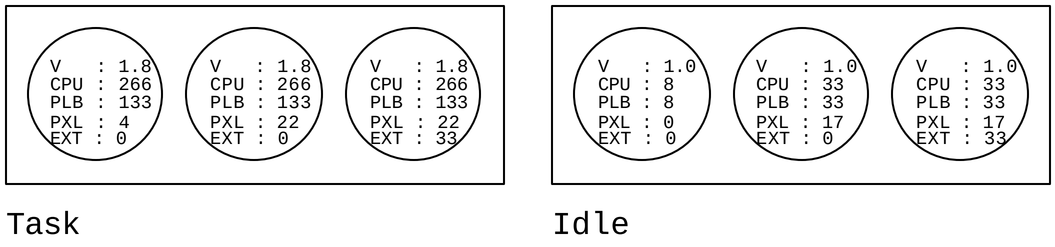

V : 1.8 CPU : 266 PLB : 133 PXL : 4 EXT : 0
V : 1.8 CPU : 266 PLB : 133 PXL : 22 EXT : 0
V : 1.8 CPU : 266 PLB : 133 PXL : 22 EXT : 33
V : 1.0 CPU : 8 PLB : 8 PXL : 0 EXT : 0
V : 1.0 CPU : 33 PLB : 33 PXL : 17 EXT : 0
V : 1.0 CPU : 33 PLB : 33 PXL : 17 EXT : 33
Task
Idle

Figure 4: A simplified policy for a PowerPC 405LP reference design. Each annotated circle represents an operating point within the boundaries of congruence classes of operating points for the task *and* idle *states. Simplified operating points specify a core voltage in Volts, and CPU, PLB, pixel clock and external clock source frequencies in MHz.*

Three operating points are specified for each of the two states. In both cases, the leftmost operating point of each set is the lowest-energy state in which the pixel clock is effectively disabled (at 16 bpp the VGA display requires a pixel clock of at least 17 MHz for acceptable visual performance), and the external clock is completely disabled. The other two operating points differ simply in whether the external clock is sourced.
.

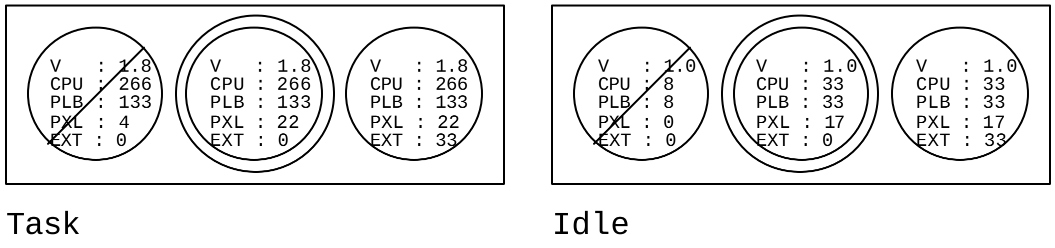

V : 1.8 CPU : 266 PLB : 133 PXL : 4 EXT : 0
V : 1.8 CPU : 266 PLB : 133 PXL : 22 EXT : 0
V : 1.8 CPU : 266 PLB : 133 PXL : 22 EXT : 33
V : 1.0 CPU : 8 PLB : 8 PXL : 0 EXT : 0
V : 1.0 CPU : 33 PLB : 33 PXL : 17 EXT : 0
V : 1.0 CPU : 33 PLB : 33 PXL : 17 EXT : 33
Task
Idle

Figure 5: The policy during "normal" operation where the LCD controller is enabled but the security chip remains disabled. The selected operating points are highlighted.

Figure 5 illustrates the situation during the "normal" operation of the application, where the LCD controller is active but the security chip is offline. In this state the first operating point of each state is invalid due to an insufficient pixel clock frequency. The LCD controller's pixel clock requirement is communicated to the system whenever the LCD controller changes state, therefore when the LCD controller is enabled the DPM system will invalidate the indicated operating points. The next operating points in each class are valid, so this is a valid policy. Here we assume that the DPM implementation

[3] The actual operating points for the 405LP system include several other parameters as detailed in *Appendix: Operating Points for PowerPC 405LP* on Page 23.

uses a pre-defined sorting of the operating points in the class (priority descending left to right in the figure) to make a determination of which of multiple valid operating points to select, and the operating system will smoothly transit between these two operating points (with the associated dynamic voltage and frequency scaling) as the operating system moves from the *task* state to the *idle* state and back

Note that the pixel clock frequency at idle is slightly lower than during the *task* state. We observed that this particular display provides adequate performance for static images at lower frequencies than are required for displaying dynamic images. Since by definition the image is static at idle, the *idle* operating point specifies a slightly lower pixel clock frequency than the *task* operating points. Lowering the pixel clock frequency lowers the SOC power consumption, the SDRAM power consumption (due to decreased memory bandwidth), and the power consumption of the LCD panel electronics, and this adds up to a tangible system-wide energy savings with no perceived loss of visual performance. This type of energy optimization is made possible by the fine-grained structure of DPM policies coupled with a processor architecture specifically designed for aggressive power management.

A hidden detail is the fact that the pixel clock is generated from the PLB clock, therefore the clock divider ratio changes between the *task* and *idle* states. In the *task* states the PLB/pixel ratio is 6:1, moving down to 2:1 at *idle*. This divider change is also encoded in the operating point in order that the operating state changes can occur without involving the LCD controller device driver. The DPM architecture allows isolation of this and all other low-level details of the frequency changes from device drivers, and does this efficiently by using pre-computed operating points.

We realize that in all cases in some systems, and in certain cases in any system, changing a policy or operating point may require notification of device drivers for some action, e.g., reprogramming bus controllers for a new frequency to insure correct and efficient operation. It is up to the underlying DPM implementation to handle this requirement during the change in the operating point. In general, the most efficient policies for flexible systems like the 405LP will minimize operating point changes that require device driver notification.

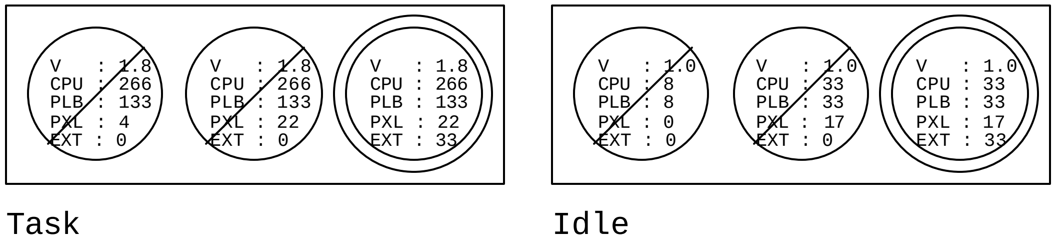


Figure 6*: The state of the policy while the LCD controller is active and the security chip is performing key management.*

Figure 6 shows the state of the policy when the security chip is online. In this situation only one operating point for each operating state remains valid, namely the operating point that enables the external clock port at 33 MHz. As soon as the security chip device driver determines that the security chip can be taken offline, this will be communicated to the power management system and the policy will revert to that shown in Figure 5.

Figure 7 illustrates the policy when both the LCD controller and the security chip are disabled. This state might arise if a user were using the device simply as an audio player, and had disabled the display to conserve energy. In this situation the lowest-energy operating point from each class is selected. Since the LCD controller is disabled, there are very limited demands for PLB and memory bandwidth while idle (8 MHz is *much* more than adequate to support DMA of CD-quality audio data to an external CODEC). Therefore, the *idle* state operating point specifies an extremely low energy state, with just enough processing power to register the event that takes the system out of idle and return to the *task* state.

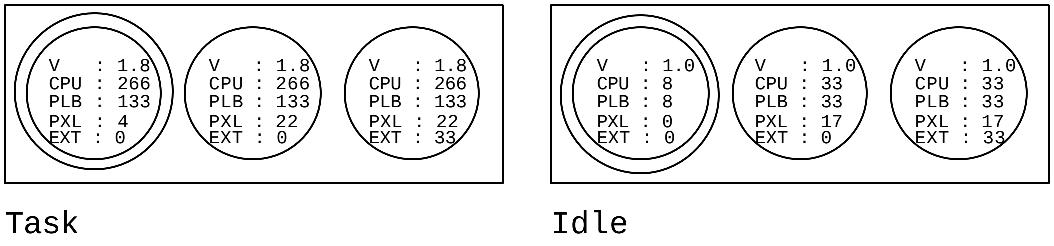


Figure 7: The state of the policy when both the LCD controller and the security chip are inactive.

However, note what would happen if any key management activities were initiated while in this system state. Prior to commencing activity, the security chip driver would communicate the requirement for the 33 MHz external clocks back up to the power management system. This would cause the policy to revert back to the situation shown in Figure 6, which is the only state supporting a 33 MHz external clock in all operating states. Again, the 33 MHz external clock is generated by dividing down the PLB clock, and the low-level code that actuates an operating point handles the divider change during the operating state changes without requiring any other action of the security chip driver.

Up to this point we have illustrated the classes of operating points assigned to operating states as an explicit enumeration. This is a compromise between simplicity and flexibility that we have adopted in our prototype implementation. However, the DPM architecture also allows operating points to be specified as a set of possible values for each parameter along with the mechanism for applying device constraints and strategy rules against the possible range of values to generate the explicit operating points at run time. Regardless of the method used to arrive at the operating point, the intention of the architecture is that operating points (either explicit or with well-defined mechanisms to derive them at run-time), congruence classes and policy mappings are pre-specified, and changes in device constraints modify the set of operating points available in a policy. These changes can be made transparent to the core of the underlying operating system, which is free to move the application from state to state without regard to any particular operating point or device state.

Abstract Implementation

The previous Sections presented a high-level design of the DPM architecture. To review, the architecture is a hierarchy of objects: operating points and congruence classes of operating points, operating states, and policies composed of mappings from operating states to congruence classes of operating points. This is a straightforward architecture that could be implemented in an operating system in several ways. This Section gives our preferred implementation and the rationale behind the choices made in the implementation.

Although the framework of the power management system described here is simple, accounting for all of the possible interactions of user-level polices and the influence of device constraints is a challenging task. Ultimately, regardless of the implementation, the system designer who creates the power management policies for the system is responsible for understanding all of the constraints imposed by the application with respect to the power management system.

Two of the challenges with respect to implementing this system include:

- Changes in device constraints may invalidate operating points. Automating these transitions is the primary mechanism by which the architecture relieves the high-level power management task from having to deal with device states. This leads to several obvious conflicts, however.

- Operations on the DPM implementation may block. Blocking could arise at the very lowest level of the implementation, where power management device drivers use system I/O ports to control voltages and frequencies. In some cases, changes in the operating point will require notification of device drivers that frequencies have changed or will change, and in some systems preparing the device for these changes may require temporary blocking. At a higher level, we recognize that certain critical tasks may need to lock the power management system against any change in operating point for periods of time, for example during a user-initiated change in the DPM policy.

The following system design recognizes and accounts for these challenges in a consistent way.

Abstract API

At an abstract level, the power management implementation supports 5 high-level entry points that that may trigger a change in the operating point, or otherwise change the state of DPM: *assert_constraint(), remove_constraint(), set_operating_state(), set_policy()* and *set_task_state()*. The first three entry points, *assert_constraint(), remove_constraint()* and *set_operating_state(),* are only required to be visible from a kernel context. Device constraints are asserted and removed by device drivers as devices change state. The operating system's process management and event handling code controls changes in the operating state and notifies DPM as these state changes occur for potential changes in the operating point. A policy manager outside of DPM sets policies and the policy manager or the tasks themselves may set task states. Therefore the entry points *set_policy()* and *set_task_state()* are only required to be visible from a user context, for example via a system call, although they may also be available in a kernel context for the use of in-kernel policy managers.

The system is assumed to be fully operational at the point where the operating system activates the DPM system. No state transitions caused by the DPM functions are allowed that would render the system inoperable or incapable of moving to a valid operating point. A straightforward definition of *validity* is used: an operating point is valid if it satisfies all device constraints, a congruence class of operating points is valid if at least one of the class is valid, and a policy is valid if every operating state in the policy maps a valid class of operating points. Assuming that the system is initially set to a valid policy, the DPM implementation ensures that the current policy will never become invalid, and the system is never allowed to move to an invalid policy.

The remainder of this Section goes into more detail on the implications of the abstract API with respect to an implementation. Special emphasis is given to the concept of task-specific operating states.

set_operating_state()

An important principle of the DPM design is the concept that an operating state is independent of any particular policy, since every valid policy must define an operating point for every state. Since the system will always be executing in a valid policy, and every operating state in any valid policy will have at least one valid operating point, the call to *set_operating_state()* can be decoupled from its eventual completion, at least to the extent that the completion of an operating state change requires a change in the operating point. This is critical because *set_operating_state()* will be called from process management code that will not be able to block and may not be able to effectively deal with errors. Even if the power management system is temporarily locked by some other operation, or the implied operating point change requires blocking due to dynamic scaling constraints, device driver notification or otherwise completes asynchronously, the *set_operating_state()* call will complete without error to the caller. The system will continue to execute at the current (valid) operating point until the DPM implementation is able to process the request to change the operating point to be consistent with the new the operating state.

It is possible that in some situations the operating state transitions will occur faster than the system can set the operating points. Note that in this situation there is no need to queue more than one request (the latest request) to *set_operating_state()*. For situations where the system cannot proceed until the new operating state is completely activated, blocking and failing variants of *set_operating_state()* may also be provided.

assert_constraint()

The device driver calls to *assert_constraint()* may always block and/or fail. If the assertion of a device constraint would invalidate the current policy it cannot be allowed to complete. Resolving this conflict may require notifying the policy manager that a change in policy is needed. Changes in device constraints will typically occur in response to system calls (*open(), close(),* etc.) that execute in a process context where a blocking call is acceptable. Device drivers that change device states in response to interrupts will need to be carefully coded to avoid problems with activating their constraints. Perhaps the best approach will be for system designers to insure that every policy will be valid regardless of the device constraints, and only use device constraints to fine-tune the selection of the operating point for optimum power management.

remove_constraint()

The call to *remove_constraint()* to remove a device constraint is also decoupled from its eventual completion. Since removing a constraint will never invalidate a policy, this call need never block the caller or fail. Again, however, this entry point is expected to be activated from a system call context (most likely *close()*) where blocking semantics would be acceptable (and more straightforward to implement).

set_policy()

Calls to *set_policy()* will fail if the target policy is not valid. A call to *set_policy()* may also block temporarily if the power management system is locked by another task. Since policy managers are expected to be implemented as user- or kernel-level daemons in a process context, they can easily be coded to handle blocking and failure of the *set_policy()* call.

Implementation and Effects of Task-Specific Operating States

One of the key features of DPM is the concept of task-specific operating points, implemented by assigning different *task* operating states to different tasks. The implementation of this feature in the core operating system is straightforward, as it simply requires the task structure to carry a descriptor of the *task* operating state to use when the task is scheduled. In our current Linux prototype the task state of each task is inherited across *fork()*, beginning from the initial task started at system boot. The task state of a task is changed by the *set_task_state()* entry point, which may be exported to the user level as a system call. In our current Linux prototype tasks with sufficient privileges may change their own task states, or the task states of other tasks. Thus a system could be constructed where a single intelligent policy manager controlled the task states of critical programs for improved power/performance efficiency, without requiring any changes to the programs.

The DPM architecture does not require the process scheduler to interpret the task state, but simply to call *set_operating_state()* with the new task state descriptor prior to context switching to a new task. The operating point associated with the task is then implied by the current policy, which is controlled by the policy manager. It is possible that some efficiency improvements could be gained by a scheduler that considered operating state affinity in its scheduling algorithm, since every operating point change involves some system overhead. It is also possible that a “hook” in the scheduler would be useful to the policy manager; however neither of these two enhancements is necessary for a DPM implementation.

It is generally believed that critical tasks may need to participate in the selection of an optimum operating point for their execution [2]. However, we do not believe that tasks running in a general-purpose system should explicitly set the system operating point. This would tie programs to a particular application, and usurp the authority of a policy manager to implement policies that might, for example, put a cap on power consumption by limiting the range of allowed operating points in a policy, perhaps to conserve battery life, or perhaps even in critical response to thermal overload conditions. Allowing a program to set the system operating point might also leave the system in an inappropriate (high-energy) state between active episodes of a periodic task. Similarly, we do not believe that a program should set the system DPM policy unless the program is prepared to act as a complete DPM policy manager for the system. We arrived at a compromise between tasks having direct control on operating points (to perform task-dependent power management) and a power management system completely outside the control of individual tasks (for global power management and system reliability) with the *set_task_state()* entry point being made available to authorized tasks.

We currently propose a set of ordered task states, with a default state and other states implying more or less power/performance. Our current Linux prototype includes 9 task states: *task-4,…,task,…,task+4*. We view the assignment of tasks to task states as somewhat analogous to a process priority scheme. A process priority has no meaning until it is considered in the context of the entire system, and the process priorities assigned to competing processes. Similarly, task states are only given meaning by the DPM policies. Numerous mechanisms are available for tasks and the policy manager to determine the appropriate task state for a task in those cases where the default state is not sufficient.

Our Linux prototype also introduces the concept of a *no state* task state. Tasks marked as *no state* are simply run at the system's current operating point. The *no state* task state is assigned to system threads that perform small amounts of work on behalf of other tasks and device drivers (*keventd* and *softirqd_** in Linux), to avoid short-duration changes to the operating point. It also turns out that changes to the current operating point may complete asynchronously in the context of one of these kernel daemons, and without the *no state* concept the activation of the daemon itself might override the change in operating point it was being called on to complete! In general the system design may also assign the *no state* task state to tasks that run periodically for very brief periods, e.g., a DPM policy manager daemon, as a way to minimize the impact of operating point changes that might occur when ephemeral tasks are scheduled and run.

In our current prototype, interrupts are always handled at the then-current operating point. Thus interrupt handlers are similar to *no state* tasks. We have considered systems that would include operating states specifically for interrupt handling, but have not yet found a need for this added complexity in the general case. However the DPM architecture would easily accommodate a generic *interrupt* operating state, an *interrupt* operating states specific to each task state, or even an *interrupt* state specific to each particular interrupt.

Finally, although DPM task states are similar to process priorities they are not necessarily correlated to static or real-time process priority mechanisms, and they must be considered separately. It is tempting to assume that high scheduling priority and high-performance operating points would go hand-in-hand, but this is not necessarily the case. Consider for example an MP3 player task. MP3 playback only requires a fraction of the processing power available at the highest-efficiency, low-voltage operating point of the PowerPC 405LP. Although the MP3 player might be scheduled under a real-time policy to guarantee predictability in scheduling, the MP3 player by itself would never require a high-performance operating point to meet its real-time constraints.

Example Strategies

The previous Sections have introduced an architecture for dynamic power management of embedded systems, given examples of how the architecture handles constraints from devices, and explored some of the low-level implementation issues. This Section concludes the technical presentation by exploring three example strategies. In this paper our intention is only to explore a few of the possibilities of DPM by using simple examples. More complex interactions between tasks, policy managers and the operating system are certainly possible and are under consideration.

These examples are based on the current Linux prototype implementation of DPM for the PowerPC 405LP. In the examples, operating point classes are named by a CPU frequency, PLB frequency and a core voltage. The prototype implementation currently includes 9 *task* states, although for simplicity only 3 *task* states are used in the examples. The *idle* state is used during system idle periods, more specifically during those periods where the CPU is halted pending an interrupt. The *idle-task* state is introduced for the idle thread itself, and especially to handle interrupts that occur during idle, allowing the system to enter a higher-performance state for interrupt handling without necessarily moving to a *task* state. When the prototype platform is truly idle, the *idle/idle-task/idle* transitions occur approximately 200 times per second in response to the timer interrupt. Interrupts that occur during *task* states are handled at the then-current operating point as previously mentioned.

Static Strategies

The simplest use of this architecture is to base a strategy on a single, “static” policy. DPM does not *require* a run-time policy manager. A single policy may be installed at system initialization and allowed to remain active indefinitely. Figure 8 below is an example of such a strategy. All *task* states are assigned to a common class of operating points and there are also different classes of operating points associated with the *idle* and *idle-task* states.

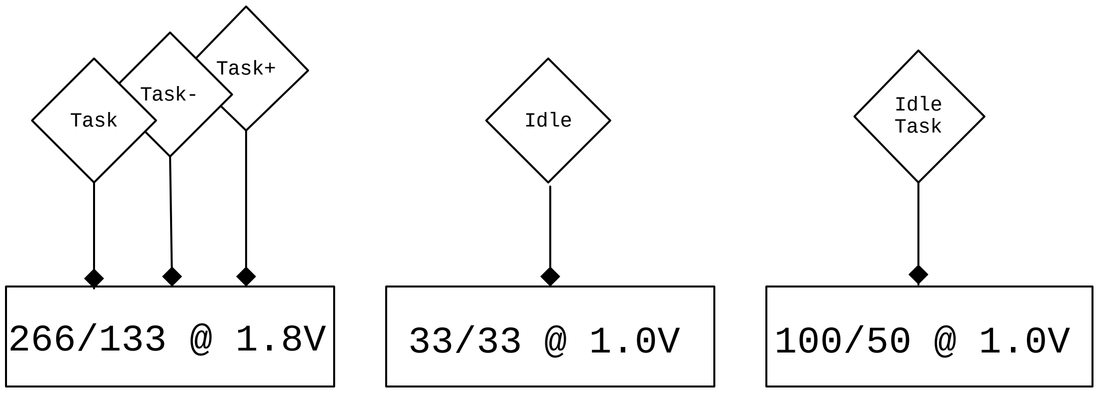


Figure 8: A simple, one-policy instantiation of the DPM framework.

For certain systems this may be an effective power management approach, since it insures that applications have top performance when they are active while the system still has a low-power idle state to save energy. Researchers are sometimes disappointed that dynamic scaling strategies do not have as large an effect as had been hoped for, either because the system has a large background power consumption that does not scale (e.g., a big LCD panel), because of the difficulty of devising general-purpose scaling heuristics, or because a lack of real-time facilities in the OS means that soft real-time tasks are forced into inefficient polling loops for short delays rather than allowing the system to return to a more power-efficient idle state [**7**]. In these systems a simple strategy like that in Figure 8 might provide close to the best possible energy savings, and efforts to conserve energy might be better focused on low-power system design, general application performance tuning, and implementing real-time extensions to the OS rather than on complex policy managers.

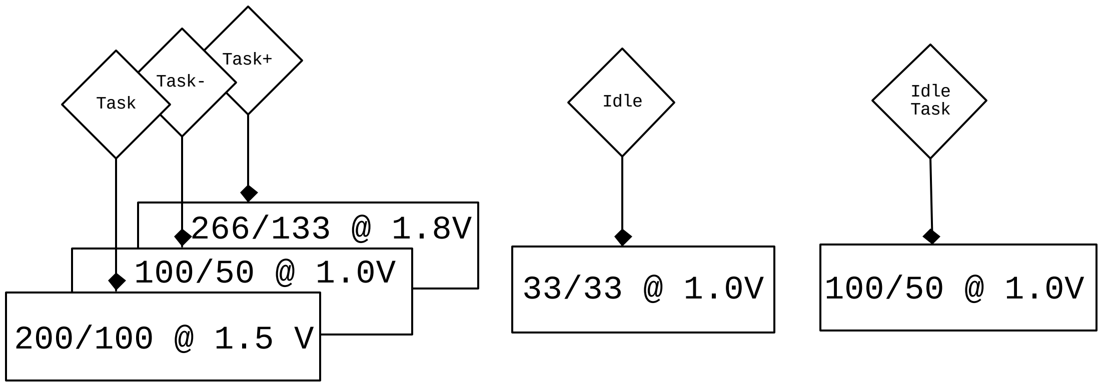


Figure 9: *Another complete yet simple instantiation of the DPM framework.*

Figure 9 above is another example of a simple, one-policy strategy. Here, different operating points are assigned to the different task states. In this simple system, most tasks will run at 200 MHz, while those tasks requiring more or less system performance can also be accommodated by requesting a different task state. It would also be within the bounds of this proposal to have a single policy like the one shown, and then have a policy manager change the task state assignments for tasks dynamically based on whatever information was required to make that decision.

Simple Dynamic Scaling

As the next example, we consider the implementation of a simple activity-based power manager for a dynamic voltage and frequency scalable system. Systems like this use CPU utilization to drive the dynamic power management policy. As system activity increases, the policy manager increases the system frequency (and the core voltage) in an attempt to provide adequate performance for the workload while minimizing power consumption. These simple types of policy managers have been well studied and are in commercial use [5].

Implemented under DPM, the example policy manager uses the mechanism of setting a policy to move the system from one voltage and frequency level to another. Note that the DPM abstraction relieves the policy manager from all of the low-level details: The policies describe consistent operating points for the idle states as well as the task states, regardless of the state of peripheral devices, and if special operating points are required for non-default task states, these are transparently encoded by the congruence class mappings for those states. In fact, the policy manager need not even be aware of the particular voltages and frequencies associated with the policies. The policy manager could simply interpret a set of abstract rules, specified by the system designer, that describe the events that should move the system from one power policy to another.

An example of the policies that might be appropriate for this type of strategy is shown in Table 2. In this example, policies are associated with CPU core voltages, and no distinction is made between the task states. The policies use increasingly higher performance and higher energy operating points as we go from Low to High. The policy manager operates by periodically querying the system as to the amount of time the application has been executing in the various operating states. When system activity increases past a certain threshold, indicated by the ratio of time spent in the *task* states vs. the *idle* state, a rule set would cause the policy manager to move to a higher power-performance (higher voltage and frequency) policy. Decreases in system activity would trigger rules that move to a lower power-performance policy. This is such a generally useful type of policy trigger that it is reasonable to require a DPM implementation to maintain these statistics.

Table 2: Example policies for a simple dynamic voltage and frequency scaling policy manager.

POLICY	TASK[+/-]	IDLE	IDLE-TASK
Low (1.0 V)	100/50 @ 1.0 V	33/33 @ 1.0 V	100/50 @ 1.0 V
Medium (1.5 V)	200/100 @ 1.5 V	33/33 @ 1.5 V	200/100 @ 1.5V
High (1.8 V)	266/133 @ 1.8 V	33/33 @ 1.8 V	266/133 @ 1.8V

Note that the policy manager might also receive meta-information from the system that would affect its management algorithm. For example, as available battery energy decreases, the policy manager might have rules to cap the energy consumption by not allowing certain policies to be activated.

The above strategy attempts to reduce latency going in and out of idle by not voltage scaling at idle, which might have a longer latency than simple frequency scaling. In practice, a policy manager using the above policy might change from the *Low* to the *Medium* policy when system activity increased above 50% over some time interval, and from *Medium* to *High* when system activity increased above 75%.

One of the drawbacks of an activity-based strategy like the one suggested here is that these strategies impose an energy penalty on the system when the system runs tasks that are highly active but have no real-time performance constraints, since the policy manager scales voltage and frequency without any direct information on the performance requirements. A simple alternative that would obtain higher energy savings is to assign the *task-* state to these busy tasks that have low performance requirements and have the policy manager ignore activity in the *task-* state when making scaling decisions. In other words, the policy manager would never scale frequency and voltage if the majority of process activity took place in the *task-* state. Similarly, tasks with soft real-time requirements could be placed in the *task+* state, and the policy manager could interpret activity in the *task+* state as a hint to scale more aggressively. This would be one way of using DPM to implement a job classification scheme like that suggested in [**8**].

Task-State Specific Dynamic Scaling

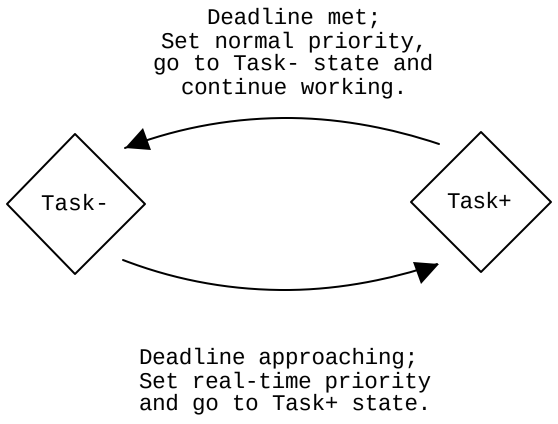
Classic scaling theory suggests that if there is work to be performed on a CPU, it will be performed more energy-efficiently at a lower voltage and frequency. Consider for example a simple video decoder that has met a deadline and is prepared to sleep until the beginning of the next frame. If the system is otherwise unloaded, and if the program could continue to make progress (e.g., begin decoding the next video frame), then another alternative would be for the decoder to continue to work at a more power-efficient operating point until the next deadline approaches.

Table 3: Example policies for a task-state specific strategy.

POLICY	TASK+	TASK	TASK-	IDLE-TASK	IDLE
Battery Critical	100/50 @1.0 V	100/50 @ 1.0 V	100/50 @ 1.0V	100/50 @ 1.0 V	33/33 @1.0V
Battery Low	266/133 @ 1.8V	100/50 @ 1.0 V	100/50 @ 1.0V	100/50 @ 1.0 V	33/33 @1.0V
Battery Good	266/133 @ 1.8V	200/100 @ 1.5 V	100/50 @ 1.0V	100/50 @ 1.0 V	33/33 @1.0V

The final example briefly explores this scenario, using the policies detailed above in Table 3. In this scenario the policy manager does not play much of a role, merely changing policies in response to changes in the state of the battery as suggested by the policy names. As long as battery power is good, most tasks are allowed to run at 200 MHz, scaling back to 100 MHz as available energy decreases, although specific tasks can run in higher energy states while the situation is still not critical.

Under this type of strategy a power-aware video decoder might modulate its scheduling priority and task state in response to impending deadlines. In this way a task could make use of the pre-existing capabilities of the operating system for guaranteed execution, coupled with the facilities of DPM that allow a task to indirectly set its own operating point. A simple two-state machine the task might implement is illustrated below.



Note that the above example assumed only minimal changes to the video decoder. This would be expected to provide some energy savings over simply running the decoder at full speed all of the time. If more extensive changes to the decoder are possible, and if the decoder is to be delivered in a system with a dedicated policy manager and special-purpose policies, then very significant reductions in energy may be possible [**9**].

Discussion

Preliminary versions of this paper have been reviewed, and these reviews brought out some issues worthy of further discussion. Some of these issues are addressed below.

Portability

One current criticism of this proposal has to do with the fact that DPM operating points are platform-specific, and the belief therefore that DPM policies will not be portable across multiple platforms. The question arises as to the possibility of simplifying the proposal, for example by using a single CPU frequency as a portable operating point, and defining a pair of min/max frequencies and an activity-based scaling policy as a portable policy manager.

The above suggestion might make sense for less energy constrained, and less flexible desktop and server systems. Note that the DPM framework easily accommodates simple operating points and policy managers for these types of systems. However, we do not see how we will be able to aggressively manage energy consumption to the extent needed by coming generations of portable devices unless we are able to fully exploit all of the energy management capabilities of the underlying hardware. We currently believe that platform-specific operating points and application-specific policies and policy managers are the best way to capture this requirement. Note that since DPM policies are based on a hierarchy of *named* objects, all levels of the hierarchy above the operating point definitions are potentially portable.

It is also difficult to deny that SOC architectures will continue to have more and more complex power management architectures, whose use will vary from application to application. Another benefit of the DPM model is that many of the platform-/product-specific details of how the various power and clocking modes of the system are used in the application has been removed from the code space of the operating system into a configuration data space, where this information is easier to manage and maintain.

Scope of the Proposal

Another current criticism of this proposal is that we have not developed a single, unified architecture for dynamic power management of every system component and attached device. Such a proposal can be found in [**10**]. Early on in the development of this proposal we made a conscious decision not to include device states in the operating point definitions, but instead to restrict operating points to the minimal set of parameters that needed to be considered together to define a core operating point. Power management mechanisms for many types of devices already exist in operating systems, and we do not see the advantage of taking on the large amount of work required to fit everything into a unified mechanism, and the representation, maintenance and state explosion problems that would result.

For example, the 405LP has an integrated PCMCIA socket controller that includes a way to control the voltage applied to the card. The state of the socket and the card voltage are not part of the 405LP operating point. The PCMCIA socket driver system manages power to socket, card drivers are expected to manage power states of the cards, and the only interaction with DPM is the constraint that if a card is inserted, then a particular bus frequency must be sufficient to run transactions to the card.

As another extreme example, the LCD backlight intensity controller widget on a PDA functions very well as a stand-alone program. Although the use of this program may have a significant effect on system energy consumption, it is not clear that there is any benefit to tightly coupling this user convenience function with a DPM policy manager that may need to operate on a millisecond timescale, and may vary from product to product. Instead we tend to favor system-wide power management approaches based on sets of cooperating power managers. In these types of schemes a DPM policy manager would be one part of the overall power management solution for the application.

A very simple example of this idea is present in the Linux PDA reference platform being developed for the 405LP. The "light and power manager" widget controls the backlight intensity, display off times, etc. This widget has also been enhanced to allow the user (or the widget itself) to specify "meta-policies" to the DPM policy manager for the PDA, e.g., "full performance", "low battery". These meta-policy commands are sent as short messages whenever conditions warrant, and the DPM policy manager implements the meta-policies using sets of pre-defined policy management rules.

Conclusion

This paper has proposed an architecture supporting aggressive dynamic power management for embedded systems. This architecture is based on the capabilities of current and next-generation processors and their application requirements. We first introduced an abstraction based on *policies*, defined as mappings of *operating points* to be used during the *operating states* of the system. We explained the interaction of devices and device constraints with the model, briefly explored some implementation issues and finished with a few examples. IBM and MontaVista are currently working to implement the system described here under Linux for the IBM PowerPC 405LP and other embedded processors. The architecture and the implementation will be evaluated by developing and characterizing dynamic power management strategies and policy managers for real workloads. This work will undoubtedly lead to further refinements of this proposal. We welcome comments from interested readers.

Contacts

This ideas contained in this document were developed by Hollis Blanchard[4], Bishop Brock[5], Matthew Locke[6], Mark Orvek[6], Robert Paulsen[4] and Karthick Rajamani[5].

Comments on this document may be posted to **linux-pm-devel**@lists.sourceforge.net.

For further technical information regarding this document contact Bishop Brock (bcbrock@us.ibm.com) or Matthew Locke (mlocke@mvista.com).

For more information on MontaVista Software's Embedded Linux Solutions contact sales@mvista.com or visit http://www.mvista.com.

For product information on the IBM PowerPC 405LP contact Thomas Marts (tmarts@us.ibm.com) or visit http://www.chips.ibm.com/products/powerpc.

Appendix: Operating Points for PowerPC 405LP

The challenge of building power-optimized systems based on processors like the IBM PowerPC 405LP is one motivation behind the DPM specification. Table 4 below details the information contained in the operating points for the 405LP in the current Linux prototype of the DPM system. The operating points are defined by physical parameters that may derive other parameters. Each parameter or group of parameters is annotated with an explanation of how it bears on a power management strategy.

[4] IBM Linux Technology Center
[5] IBM Research, Austin Center for Low-Power Computing
[6] MontaVista Software

Table 4: Components of the PowerPC 405LP Operating Point under Linux

PHYSICAL PARAM(S).	DERIVED PARAM(S).	NOTES
Core Voltage		The operating point specifies the core voltage, rather than deriving it from frequency constraints, in order to allow policies that use only frequency scaling in places where voltage scaling would have a negative impact on latency. Although the 405LP supports voltage scaling across its full operating range in as little as 80 µS, the latency of voltage scaling is ultimately based on the power supply design and the mechanism used to control it, and only the system designer can make the necessary tradeoffs for the application's power management policy.
System Clock Source		The clock tree is driven by the PLL VCO (the normal case), the system reference clock (PLL bypass during relocking), or the RTC timebase (for very low-power active standby).
PLL Multiplier and Divider	CPU Frequency	The 405LP accepts a wide range of system clock frequencies. Therefore the CPU frequency and most other frequencies are only uniquely specified based on a PLL multiplier and divider. Although the PLL divider can be changed at will, changing the multiplier requires a short PLL relock interval. Policies requiring minimum latencies between critical operating states will require a common PLL multiplier in the operating points in the policy.
CPU/PLB Divider	PLB Frequency	The Processor Local Bus (PLB) frequency is used as the basis for most other clocks in the system, and is also the SDRAM frequency.
PLB/[Bus] Dividers	Bus Frequencies	Several bus frequencies are derived from the PLB, as well as the LCD pixel clock frequency. Bus protocol timings are often optimized for the particular bus frequency, and bus controller drivers may request to be notified when bus frequencies change to insure optimal performance. For minimal latency between transitions, policies can be written such that bus *frequencies* never change between operating states, although the bus *dividers* will change as the PLB frequency changes.
External Clock Control	External Clock Sources	The 405LP sources two external clocks with programmable dividers from system bus frequencies. Clock dividers may change across operating points even though the policies may ensure that the derived frequencies do not change.
	Bus/SDRAM Target Frequencies	In certain cases the system designer may be able to trade critical latencies against power/performance implications by running the system in non-optimal states. For example, the lowest-latency transition from an 8 MHz *idle* state to a 266 MHz *task* state will occur if the *idle* state is specified to run with 266 MHz memory timings, which are not optimal for 8 MHz operation. These fields support these kinds of tradeoffs.
	DCR Values	At the SOC level, system timing is controlled by three Device Control Registers (DCRs) associated with clock distribution and two DCRs associated with SDRAM timing. The operating point includes pre-derived values for the DCRs to reduce latencies during transitions between operating points.

Disclaimers

References

The following is only a representative sample of the large body of work on the problems of dynamic power management and dynamic voltage and frequency scaling. See [**11**] for a more comprehensive online bibliography.

1 Benini, et al., "A survey of Design Techniques for System-Level Dynamic Power Management", *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, Vol. 8, No. 3, June 2000.
2 Pouwelse, et al., "Application Directed Voltage Scaling", *IEEE Transactions on Very Large Scale Integration (TVLSI)*, Sep 2002. *(To Appear)*
3 Flautner, et al., "Automatic Performance Setting for Dynamic Voltage Scaling", *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM*), July 2001.
4 Simunic, et al., "Dynamic Voltage Scaling and Power Management for Portable Systems", *Proceedings of the 38th Design Automation Conference*, DAC 2001, June 2001.
5 Marc Fleischmann, "LongRun™ Power Management", Transmeta Corporation Whitepaper, January 2001. Available via links from http://www.transmeta.com.
6 Shaffer et al., *Apparatus and Method for Automatic CPU Speed Control Based on Application-Specific Criteria*, United States Patent 6,298,448, October 2001.
7 Grunwald, et al., "Policies for Dynamic Clock Scheduling", *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, Usenix Association, October 2000.
8 Weiser, et al., "Scheduling for Reduced CPU Energy ", *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Usenix Association, November 1994.
9 Pouwelse, et al., "Power-Aware Video Decoding", *2001 Picture Coding Symposium*, April 2001.
10 Rawson, III et al., *Method and System of Power and Thermal Management for a Data Processing System Using Object-Oriented Program Design*, United States Patent 5,535,401, July 1996.
11 Online bibliography maintained by Professor Frank Bellosa at the University of Erlangen-Nürnberg: http://www4.informatik.uni-erlangen.de/Projects/PowerManagement/Bibliography/