Scalable Modeling and Optimization of Mode Transitions Based on Decoupled Power Management Architecture

Dexin Li, Qiang Xie and Pai H. Chou
Center for Embedded Computer Systems
University of California, Irvine, CA 92697-2625 USA
{dexinl,qxie,phchou}@uci.edu

ABSTRACT

To save energy, many power management policies rely on issuing mode-change commands to the components of the system. Efforts to date have focused on how these policies interact with the external workload. However, the energy savings are ultimately limited by the set of power-saving modes available to the power manager. This paper exposes new power-saving opportunities to existing system-level power managers by handling each desired mode change in terms of an optimal *sequence* of mode transitions involving multiple components. We employ algorithms to optimize these transition sequences in polynomial time, making them applicable to static and dynamic policies. The decoupling between policies and mechanisms also makes this approach modular and scalable to devices with complex modes and intricate dependencies on other devices in the system. Experimental results show significant energy savings due to these sequentialized mode-change opportunities that would otherwise be difficult to discover manually even by experienced designers.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based System**]: Real-time and embedded systems; D.2.11 [**Software Architectures**]: Domain-specific architectures

General Terms

Algorithms, Design

Keywords

power management, power model, component mode, mode transition, mode change, compex system

1. INTRODUCTION

As embedded systems grow in complexity, designers must rethink power management in the context of complex components. With the advent of systems-on-chip (SoC), today's system will become tomorrow's component, and individual components will contain their own local power managers. It will no longer be viable for a system-level power manager to continue controlling power the way it is done today. Today's power management approaches respond to a variable stream of events by selecting a new power mode for each component. If the overhead for this mode change cannot be amortized over the break-even time, then the power manager will not make the mode change. Such an approach is adequate for relatively simple systems such as laptop computers with hard disks and displays. The power manageable components operate mostly independently of each other. As a result, power managers today need only to model the available modes and some workload profile without concerns for the inter-component interactions.

For the next generation embedded systems, where the components appear more like nodes in a distributed multiprocessor system, power management must be done very differently. Not all mode transitions are available at all times due to the inter-component dependency. Even if all mode transitions are available at all times, deciding when or whether they should be taken is fast becoming a non-trivial task. This is because a mode change request for one component will trigger a chain of mode transitions for its subcomponents, and even across multiple components. If these dependencies are not considered, attempts to save power by locally greedy mode changes will often result in globally higher energy consumption or even missing deadline [5].

We believe that a new power management architecture will be required in order to manage power correctly and effectively for a new generation of complex embedded systems. Current power management in embedded systems and computer systems is performed by a power manager, most likely residing in the operating system. The power manager monitors the device status, makes decisions on the changes of power modes, and sends commands to device drivers to actuate the changes. However as the systems become complicated and the applications have to satisfy multi-dimensional constraints, the power manager may be overwhelmed by details of the large number of power-manageable devices. For example, a software-defined radio system might consist of more than ten power manageable components and mode changes of two devices may be dependent upon each other because of the application constraints. The traditional power manager can handle this system in theory, but in practice it may have difficulty in making correct decisions in a time-efficient and energy-efficient manner. In the operating system, power management itself could become a heavy duty task and a large source of bus traffic simply because it has to monitor working states of and send mode transition commands to a large number of devices, and satisfy all system constraints as well.

In this paper, we purpose a new power management architecture for large and complicated embedded systems. We model power modes with inter-component dependencies. We modularize the system power manager by separating the power manager core and

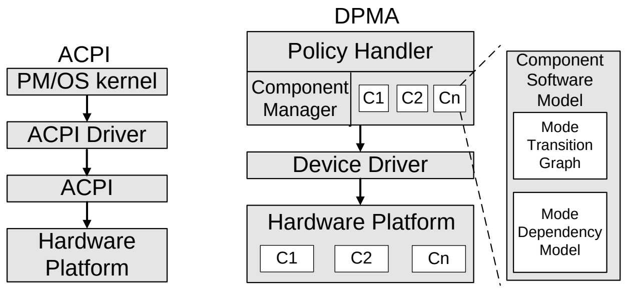

Figure 1: System architectural model: ACPI vs. DPMA (our model).

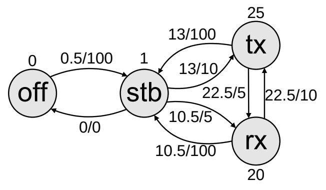

Figure 2: Mode transition graph for a power amplifier in a software-defined radio system. Vertices: component modes with power number(w); edges: mode transitions with power(W) and timing(ms).

a special unit (called Component Manager) handling component-level details. The advantage of modularization is that the power manager does not have to consider component details when making high-level policies. A light-weight simulation engine provides detailed mode change information to the power manager so that it does not have to incorporate all component-level details. The purpose of this work is to demonstrate power mode modeling and simulation aspects of power management architecture. We use a fairly simple DPM technique in comparing power management with our model and without our model. Experimental results show we can obtain feasible sequences of mode transitions and optimized system total energy.

This paper is organized as follows. Section 2 briefly introduces our power model and reviews related work. Section 3 presents detailed power modeling. We give our algorithm in Section 4 and discuss the experimental results in Section 5.

2. RELATED WORK

Many existing techniques for system-level power management assume single device schemes. In task scheduling, researchers have paid most attention to a single processor with voltage/clock scaling capability while power modes and power consumption on peripheral devices are not considered. The cost of mode changes on the processor is often reasonably neglected [6, 9, 10]. In dynamic power management techniques, researchers concentrated on systems of a single device without strong timing guarantees [2, 7, 11]. Tradeoffs are made between the power consumption and system performance. In [8] the authors did model multiple servers and relationships in the system. However, the modeled servers have identical behaviors (handling incoming requests) with the only difference in server parameters (handling capacity, etc.). The modeled relationships of synchronization and concurrency are actually the relationships among *services*, rather than among the *components* themselves. Complex systems with complicated inter-component relationships were not the main focus of the research. On the other hand, our power model encompasses multiple *heterogeneous* components, their power modes, the costs of mode changes, and inter-component dependencies.

ACPI[3] is a system interface in an operating system between the power manager and the hardware devices. It enables universal power management of plug-and-play devices from different manufacturers. Similar to the concept of ACPI, our power model is a middle layer between the system power manager and the device driver, supporting flexible and complicated system-level mode changes. Our model is different from ACPI in several ways: first, ACPI is at the level of the device driver layer whereas our model is above the device driver layer (see Fig. 1); second, the purpose of ACPI is to *bridge* the operating system and the hardware while our model emphasizes the *support* for the power manager by providing component-level knowledge; third, we incorporate application-level knowledge by modeling the dependencies among components, whereas the ACPI mainly models device details without application knowledge. Furthermore, our component manager takes partial responsibility from the power manager by generating the correct sequence of mode transitions for a global mode change, whereas ACPI itself is merely a common driver interface that does not have any power management responsibility.

3. POWER MODELING

In this section we present our component power model and the Decoupled Power Management Architecture (DPMA). We partition the functionality of a traditional power manager into two parts: the policy handler and the component manager (see Fig. 1). The policy handler deals with high-level policy-making while the component manager handles component-level mode changes using the Component Software Model. A light-weight simulation engine, which can be either online or offline, keeps track of the details of mode changes and informs the policy handler of the cost of the mode changes. Our architecture is highly modular. The policy handler can change power management policies without knowledge about the components. The component manager can change its core algorithm without modifying the Component Software Model. The detailed model of a component can be easily updated without changing the policy handler or the component manager.

3.1 Component Software Model

In the context of this paper we model components in terms of their power modes. The component can be an individual device or may consist of subcomponents. All power-manageable components in a system form a set U. Each component has a set of modes $M = \{m_i | i = 1, \ldots, k\}$. We use M_u to denote the set of modes for component u. A mode for the component u is represented as $u.m$ where $u \in U$ and $m \in M_u$.

By “mode transition”, we mean a *direct* switch from one component mode to another. By “mode change” we mean alternating a component from its current mode to a target mode, which may be either a direct mode transition or a sequence of mode transitions. Mode transitions of a component are modeled as a Mode Transition Graph (MTG) (see Fig. 2), similar to the model in [1]. An MTG $G = (M, E, P, H)$ is a directed graph, where M is a set of vertices representing a set of component modes, and E is a set of directed edges representing all possible mode transitions. The power function $P : M \rightarrow R$ maps a component mode to a power number. The overhead function $H : M \times M \rightarrow R \times R$ maps a mode transition

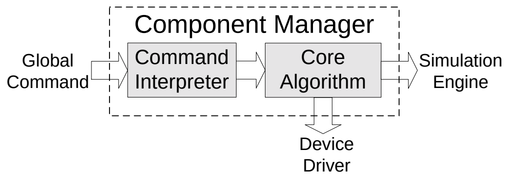

Figure 3: The component manager diagram.

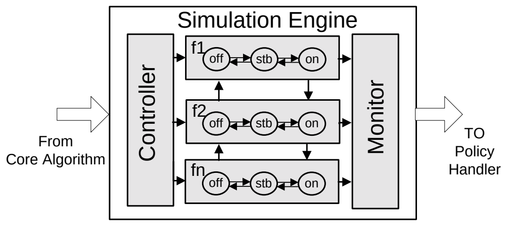

Figure 4: Simulation engine diagram.

to a power number and a timing number. If two vertices v and v' are not directly connected, they have to go through a sequence of mode transitions $\pi = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ such that $v_0 = v$, $v' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$ to accomplish a mode change, if possible.

A mode dependency between two components is represented as "$u.m \mapsto v.n$" where $u, v \in U$, $m \in M_u$ and $n \in M_v$ (M_u and M_v are mode sets of components u and v, respectively). The dependency follows an "only-if" interpretation: the component u is in mode m only if the component v is in mode n. In other words, if the component v is not in mode n, the component u cannot be in mode m. All the dependencies for a set of components U form a set $D \subseteq \bigcup_{i \in U} M_i \times \bigcup_{j \in U} M_j$. We call "$u.m$" (the left side of the arrow) the *key* and "$v.n$" the *value* of the dependency. We use the symbol $\overset{\prec}{\mapsto}$ and $\overset{\succ}{\mapsto}$ to denote the temporal relationships between the key and the value. The former represents the key starting its mode change before the value finishes its mode change, while the latter represents the key starting its mode change after the value finishes its mode change.

The "only-if" interpretation suggests a necessary dependent relationship between two components. We use it to identify and eliminate illegal modes that violate the constraints. Meanwhile, it does not make unnecessary restrictions, allowing full opportunities for design space exploration and system optimization. The definition of the temporal relationships enables us to model the partial ordering in a sequence of mode transitions.

3.2 System Model

Our system model consists of a policy handler, a component manager, component software models, the device driver, and the hardware platform. A separate simulation engine can be integrated with our system model to obtain the cost information of a mode change. The policy handler makes high-level power management decisions and issues global commands for system-level mode changes. A global command can be either an *action* that causes a mode change or a *query* to the simulation engine for information about the cost of a mode change, when necessary. The component manager handles component-level details of the mode changes. The Component Software Models, which are described in the previous subsection, capture the details of component modes. The device driver is an interface between the component manager and the Hardware platform.

The component manager consists of the command interpreter and the core algorithm (see Fig. 3). A global command from the policy handler may contain system-level semantics that the components may not understand. The command interpreter translates a global command $\gamma \in \Gamma$ into a set of local commands Λ_γ. A local command $\lambda_i = u.m \in \Lambda$ is to set the component u to the mode m where Λ is the set of all local commands. The role of the command interpreter can be represented by the function $F : \Gamma \to \Lambda^*$ where Λ^* is the transitive closure of Λ. For example, the command interpreter may translate the global command "set to system ready" into "component A to standby" and "component B to on" ($\Lambda' = \{A.\mathsf{standby}, B.\mathsf{on}\}$). The core algorithm takes the set of local commands $\Lambda' \in \Lambda^*$ as an input and determines a feasible sequence of mode transitions $\sigma = \langle v_0, v_1, v_2, \ldots, v_s \rangle$ where $v_i = u.m, u \in U$ and $m \in U_m$ for $i = 0, 1, \ldots, s$ is a component mode. Note that even if a global command may ask for a mode change on one specific component, modes on multiple components might be changed due to the mode dependencies. The algorithm also tries to optimize the sequence to reduce the energy consumption during the mode change. The output sequence is either sent to the device driver or fed into the simulation engine.

3.3 Mode Simulation

The simulation engine takes σ as an input, simulates the entire of the mode change and obtains the timing cost and the energy cost of a mode change which will be used by the policy handler. When the simulation is performed online, it is able to handle variable costs of mode changes at runtime. For example, when plugging in a new component, the costs of mode changes on other components might be changed due to the mode dependencies; the power-up cost of a component in a cold environment may be a function of the ambient temperature.

The simulation encompasses every detail of a mode change. A mode transition in one component may cause mode transitions in other components. A transition in one component may be required to happen before or after a mode transition in another component. As the simulation of a mode change is completed, the information about both timing and energy becomes available. In the context of this paper we assume that the component manager can continuously send mode transition commands without delay if there is no dependency among the involved components.

The simulation engine contains a controller, the component state machines, and a monitor, as shown in Fig. 4. A component is implemented as a state machine, and a state in the state machine represents one of the component modes. A controller reads a sequence of mode transitions from the Component Manager and dispatches it to the correspondent components. Commands are dispatched in parallel unless dependencies among components prevent doing this. Because of the dependencies, some component may enable a mode change on another component by sending out a control signal. A global clock is used to synchronize all the components. The monitor is used to check the status of the components and track the power and timing information of each component. As the simulation is completed, it generates the power and timing information of the mode change.

4. ALGORITHMS

To generate feasible sequences of mode transitions that satisfy all mode dependencies, we have developed a polynomial-time core algorithm for the component manager. It optimizes the energy cost

```
LocalSeqGen($G, u, i, j$)
# Input: an MTG $G$, current mode $u.i$, target mode $u.j$
# Output: a sequence of mode transitions $\sigma$
$\sigma \leftarrow \emptyset$
$\pi \leftarrow$ ShortestPath($G, i, j$)
for each $u.m$ in $\pi$
    if $u.m$ not in $D.keys()$
        $\sigma$.append($u.m$)
    else
        $v.n \leftarrow D[u.m]$
        $i \leftarrow$getCurrentMode($v$)
        if $u.m$.type() == '$\prec$'
            $\sigma$.append($u.m$)
            $\sigma$.append(LocalSeqGen($G, v, i, n$))
        else
            $\sigma$.append(LocalSeqGen($G, v, i, n$))
            $\sigma$.append($u.m$)
return $\sigma$
```

Figure 5: Generating mode transitions for a local command.

```
CompleteSeqGen($\Lambda, D, G$)
# Input: $\Lambda$: a set of local commands
#        $D$: a set of mode dependencies
#        $G$: an MTG
# Output: a sequence of mode transitions $\sigma$
topo_sort($\Lambda$, $D$) # by power of target modes
$\sigma \leftarrow \emptyset$
for each $\lambda$ in $\Lambda$
    $u \leftarrow \lambda$.getComponent()
    $i \leftarrow$getCurrentMode($u$)
    $j \leftarrow \lambda$.getTargetMode()
    temp $\leftarrow$ LocalSeqGen($G, u, i, j$)
    $\sigma$.append(temp)
    UpdateComponentStatus() # obtain component modes.
return $\sigma$
```

Figure 6: Generating mode transitions for a set of local commands.

for a mode change by reordering the mode transitions under dependency constraints such that the larger power consumers are powered up as late as possible. In this section we first give the problem statement, followed by the algorithm details.

4.1 Problem Statement

As we mentioned earlier, for a component u in a mode m, a local command $\lambda = u.n$ is to set u to a target mode n. Because of the mode dependencies, modes on multiple components might be changed. Therefore, the entire sequence of mode transitions for λ is $\sigma = \langle v_0, v_1, v_2, \ldots, v_s \rangle$ where v_i is a component mode. A feasible sequence of σ is the one that all mode transitions satisfy the dependency set D.

The time cost and the energy cost of mode transitions for π_λ are:

$$
\begin{aligned}
t_\lambda &= \sum_{u \in U'} \sum_{(m_i, m_{i+1}) \in \pi_u} H_t(m_i, m_{i+1}) \quad (1)\\
e_\lambda &= \sum_{u \in U'} \sum_{(m_i, m_{i+1}) \in \pi_u} H_t(m_i, m_{i+1}) H_p(m_i, m_{i+1})\\
&\quad + \sum_{v \in U' - \{u\}} \left(\sum_{(m_i, m_{i+1}) \in \pi_v} H_t(m_i, m_{i+1}) \right) \times P_v(m)\\
&\quad + \sum_{w \in U - U'} t_\lambda P_w(m) \quad (2)
\end{aligned}
$$

respectively, where H_t and H_p are the projections of the cost function H on time and power, respectively, and $P_v(m)$ is the power number of the component v in mode m (current mode). The first term in (2) represents the energy cost for mode transitions on the relevant components. The second term represents the energy cost for relevant components during non-transition time periods. The third term represents the energy cost for components without mode transitions.

Given a set of local commands $\Lambda_\gamma \subseteq \Lambda$ that implement a global command γ, the total time and energy costs are:

$$
\begin{aligned}
t_\gamma &= \sum_{\lambda \in \Lambda} t_\lambda \quad (3)\\
e_\gamma &= \sum_{\lambda \in \Lambda} e_\lambda \quad (4)
\end{aligned}
$$

respectively. The two equations above assume all mode transitions are serialized. The simulation engine uses the same cost functions except that it also explores opportunities where mode transitions may be parallelized. Our problem is to generate σ, a sequence of mode transitions, that is: 1) feasible and 2) with minimum cost e_γ.

4.2 Core Algorithm

Our core algorithms consists of two parts: LocalSeqGen generates a sequence of mode transitions for a local command while CompleteSeqGen generates a complete sequence of mode transitions that implement a global mode change.

For a local command $\lambda = u.n$, if the current mode of u is $m \neq n$, then we apply a single-source-shortest-path algorithm (the Bellman-Ford algorithm [4]) in LocalSeqGen to obtain a sequence of mode transitions from m to n with the minimum delay. To avoid repetitive calls to Bellman-Ford when the costs of mode changes are all static, an alternative all-source-shortest-path algorithm (the Floyd-Warshall algorithm [4]) can be used to generate a table that contains the shortest paths for all the mode changes in an MTG. In case other components have mode dependencies with u, LocalSeqGen changes modes for the dependent components, and generates a combined sequence of mode transitions as the output.

CompleteSeqGen (in Fig. 6) generates a sequence of mode transitions for a set of local commands. We apply a version of topological sort over the target modes of local commands based on the partial ordering defined by the mode dependencies. The sorting reorders the local commands by power consumption of the target modes under all dependency constraints. When a component is to be powered up, or to be set to a higher power mode, we try to postpone its mode change (under dependency constraints) because once its mode is changed, it will keep consuming high power to the end of the global mode change.

5. EXPERIMENTAL RESULTS

We applied our modeling to two examples. One is a power amplifier in a software-defined radio system, and the other is a complete software-defined radio channel. We are able to characterize component modes at both levels of system hierarchy using our system model. We model the inter-component mode dependencies derived from application requirements. Our algorithm optimizes the

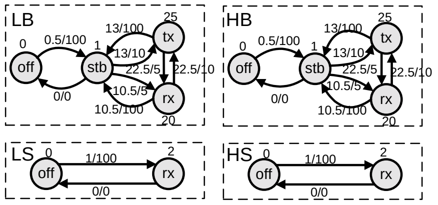

Figure 7: Mode transition graphs for the power amplifier.

Global Mode	Local Mode
1-2GHz send	HB.tx
1-2GHz receive	HB.rv
0.2-1GHz send	LB.tx
0.2-1GHz receive	LB.rv
bypass	HB.off, LB.off
off	HB.off,LB.off HS.off,LS.off

(a)

HB.stb$\overset{\succ}{\mapsto}$LB.off
HB.tx$\overset{\succ}{\mapsto}$LB.off
HB.rx$\overset{\succ}{\mapsto}$LB.off
LB.stb$\overset{\succ}{\mapsto}$HB.off
LB.tx$\overset{\succ}{\mapsto}$HB.off
LB.rx$\overset{\succ}{\mapsto}$HB.off
HS.off$\mapsto$ HB.off
LS.off$\mapsto$ LB.off

(b)

Table 1: (a) Mode mapping for the power amplifier. (b) Mode dependencies for the power amplifier.

sequences of mode transitions and achieves significant energy savings even with the simplest power management policy. Since our model is orthogonal to the power management policy, we expect more energy savings when using advanced policies with our model.

5.1 Power Amplifier Example

The power amplifier (PA) consists of four components: a low-band amplifier (LB), a low-band temperature sensor (LS), a high-band amplifier (HB) and a high-band temperature sensor (HS). The amplifiers have four modes each(off, standby(stb), transmit(tx) and receive(rx)) and the sensors have two modes each (off and on) (see Fig. 7). The PA has six global modes, mapped to a combination of component modes, as shown in Table 1(a). The application requires that at most one amplifier can be in a non-off (stb, tx or rx) mode at any time. The sensor must be on if the amplifier of the same band is on. The mode dependencies are listed in Table 1(b).

Table 2 shows the sequences of mode transitions (the second column) generated for three consecutive local commands (LB.tx, HB.rx, and HB.off) and the sequences optimized (the third column) for energy savings. The command LB.tx is to set the LB from off to tx. Our algorithm determines a shortest transition path (off$\rightarrow$stb$\rightarrow$tx) for it. Because of the mode dependency, the mode of the LS must be changed as well. Since the LB has a higher

Local Cmd.	Generated Sequence	Tims (ms)	Energy (J)	Optimized Sequence	Time (ms)	Energy (J)
LB.tx	LB: off,stb,tx LS: off,on	210	2.975	LS: off,on LB:off,stb,tx	210	0.475
HB.rx	LB:tx,stb,off LS: on,off HB:off,stb,rx HS:off,on	305	3.878	LB:tx,stb,off LS:on,off HS:off,on HB:off,stb,rx	305	1.878
HB.off	HB:rx,stb,off	100	1.1	HB:rx,stb,off	100	1.1

Table 2: Generating sequences of mode transitions: a power amplifier example. Initial component modes are all off.

working power consumption than the LS, our algorithms choose to turn on the LS first and the LB second to reduce the on time for the LB during the mode change. The command HB.rx is to set the HB from off to rx. It actually changes modes of all four components. The LB is changed from tx to off before the HB is changed to stb because the application requires at most one component can be on (stb/tx/rx), which is correctly characterized by our dependency model. The LS is changed from off to on as the HB is set to stb. Note that when the LB is turned off, the LS is not turned off because there is no such dependency specified. This is compatible with the application scenario that the sensor is kept on even if the power amplifier is turned off just in case the amplifier is overheated. For the optimized sequences for command LB.tx and HB.rx, while the time costs remain the same, the energy costs are reduced down to less than 50% because we choose to power up a high power device (an amplifier) after powering up a lower power one (a sensor), which reduces the total amount of energy consumed during the mode change.

5.2 Software-Defined Radio Channel

A software-defined radio (SDR) system supports multiple communication protocols on multiple transmission bandwidths for multiple domains of applications which could possibly update and reconfigure itself through software at runtime. Therefore it has a highly flexible and dynamic behavior that incurs a large number of mode changes. Moreover, the cost of a global mode change is non-negligible and must be considered seriously. For example, waveform reconfiguration may take minutes or more in time. Optimizing the cost of mode changes in such systems will contribute significantly to the system energy reduction.

The SDR channel consists of an antenna, a power amplifier, a transceiver, a modem, an unprotected processor, an unprotected I/O interface, an unprotected power supply, a protected processor, a protected I/O interface, a protected power supply, an encryption unit, a domain controller, and a system power supply, for a total of 13 power-manageable components. The input to the protected processor is from the unprotected processor encrypted through the encryption unit. Each component has a number of power modes. A mode change is associated with costs in terms of both time and power. The whole system has five global modes: off, standby, ready, protected_on and unprotected_on. Each global mode is mapped to a combination of component modes.

This system is rich in mode dependencies that are derived from application requirements. The protected processor is on only if the unprotected processor is on. The protected/unprotected power supply is off only if the protected/unprotected processor is off. The encryption unit is on only if the protected processor is on. The unprotected/protected I/O is on only if the unprotected/protected processor is on.

The system handles incoming requests both from the antenna and from I/O interfaces. The Policy Handler makes decisions to change (or not to change) global power modes before and after each requested service. We assume that the arrival rate and the service time of an incoming request follow Poisson and exponential distributions, respectively. We fix the arrival rate to 5000/sec and vary the average service time. The time costs of mode transitions in the system range from 1ms to 10^3ms. An incoming request is queued if the previous requests are not completed, and we assume the queue is long enough to avoid overflow.

We illustrate the benefits of our power modeling by comparing four cases, all using a fixed time-out policy except Case 1. Case 1 assumes all the components are on at all times, and this is the actual case in the real system due to the lack of an appropriate

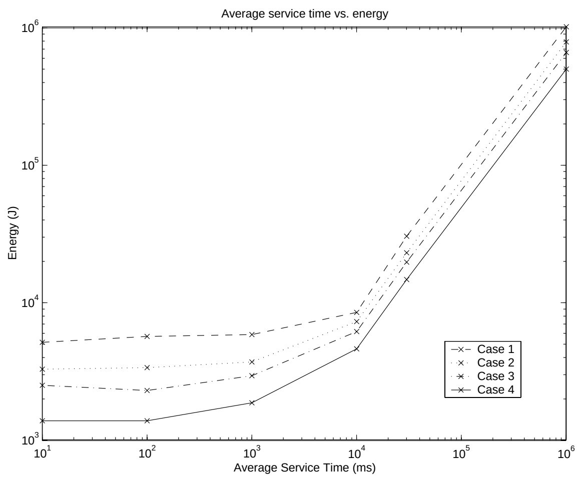

Figure 8: Average service time vs. energy consumption.

AST(ms)	10	100	1000	10000	30000
Case 1	49754.0	53579.1	61023.5	88382.3	315779.1
Case 2	71822.8	76237.2	85656.5	104185.3	331273.7
Case 3	49754.3	54136.1	61080.4	97779.7	329243.0
Case 4	49754.2	54092.0	61036.5	99747.7	327463.2

Table 3: Total execution time for the simulated requests (in ms): AST= average service time.

power model that handles system dependencies. Case 2 assumes two global modes: on and off. Case 3 has five global modes: off, standby, ready, protected_on and unprotected_on. The costs of mode changes in Case 2 and 3 are conservatively estimated by serializing every mode transition due to the lack of detailed component information. Case 4 assumes the same global modes as in Case 3, but uses our power model and the simulation engine to determine the precise costs on mode changes.

We vary the average service time from 1ms to 10^6ms and simulate for 100 requests. The results are shown in Fig. 8 and Table 3. Case 1 has the worst power consumption while the total mission time is the shortest. Case 2 saves some energy by applying the time-out policy, but its execution time is the longest because it assumes two global modes with the largest costs associated with the mode changes. Case 3 avoids the large costs of Case 2 by utilizing multiple power modes whose mode changes bring relatively smaller costs. Our modeling and simulation approach in Case 4 achieves the best result. We avoid overestimating the cost of mode changes by determining the shortest path for mode transitions, by optimizing the sequence of mode transitions, and by parallelizing some mode transitions while still satisfying all dependencies. The curves in Fig. 8 show that our approach in Case 4 always saves energy. When the average service time is comparable to the time cost of a mode transition, our approach achieves more than 80% energy savings over Case 1 and about 50% energy savings over Case 3. When the average service time becomes larger, the energy saving becomes smaller. However, even when the average service time is 1000 times larger than the maximum cost of a mode transition, we still have more than 20% energy savings over Case 3. This means that our savings do not rely on large costs of mode changes but is truly achieved by exploring multiple components and complex system dependencies.

6. CONCLUSION

This paper presents a power modeling technique based on a decouple power management architecture. The cost of mode changes in a large system may be both complex and expensive. A global mode change may involve multiple components and over a sequence of mode transitions. A traditional power manager that closely couples the policy-making and component-level details will be overwhelmed by the large scale of the system and the complex behavior. Our model handles complex systems consisting of multiple component with inter-component dependencies by decoupling the component-level details from system-level power management policies. This way the system power manager is able to concentrate on making high-level policies without considering component details. Our Component Manager generates sequences of mode transitions in polynomial time, and our light-weight simulation engine provides the system-level power manager with time and power information of mode changes. Future work includes enhancements to the core algorithm and the simulation engine to handle more complex application scenarios (e.g., mode change preemptions). In addition, we will develop appropriate power management policies that work best with our model for system-wide energy optimization.

Acknowledgement

This research was sponsored in part by DARPA under contract F33615-00-1-1719 and in part by Natinal Science Foundation under grant CCR-0205712. The authors gratefully acknowledge Professor Nader Bagherzadeh, Dr. David Jensen, and his technical staff at Rockwell Collins for their technical discussion and assistance. We would also like to thank Professor Daniel Gajski for providing the SpecC license used in the study.

7. REFERENCES

[1] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. on VLSI Systems*, 8(3):299–316, June 2000.

[2] L. Benini, G. Paleologo, A. Bogliolo, and G. De Micheli. Policy optimization for dynamic power management. *IEEE Trans. Computer-Aided Design*, 18:813–833, June 1999.

[3] Compaq, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface. In *http://www.acpi.info/index.html*.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[5] D. Li, P. Chou, and N. Bagherzadeh. Mode selection and mode-dependency modeling for power-aware embedded systems. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 697–704, 2002.

[6] J. Luo and N. K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Proc. of DAC*, pages 444–449, 2001.

[7] Q. Qiu, Q. Wu, and M. Pedram. Stochastic modeling of a power-managed system construction and optimization. In *Proc. 1999 International Symposium on Low Power Electronics and Design*, pages 194–199, August 1999.

[8] Q. Qiu, Q. Wu, and M. Pedram. Dynamic power management of complex systems using Generalized Stochastic Petri Nets. In *Proc. of DAC*, pages 352–356, 2000.

[9] G. Quan and X. S. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proc. of DAC*, pages 828–833, 2001.

[10] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proc. of ICCAD*, pages 365–368, 2000.

[11] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proc. of DAC*, pages 524–529, June 2001.