Multi-Processor Computer System Having Low Power Consumption

C. Michael Olsen and L. Alex Morrow
IBM Research Division, P.O.Box 218, Yorktown Heights, NY 10598, USA
{cmolsen, alex_morrow}@ us.ibm.com

Abstract. We propose to improve battery life in pervasive devices by using multiple processors that trade off computing capacity for improved energy-per-cycle (EPC) efficiency. A separate scheduler circuit intercepts interrupts and schedules execution to minimize overall energy consumption. To facilitate this operation, software tasks are compiled and profiled for execution on multiple processors so that task requirements to computing capacities may be evaluated realistically to satisfy system requirements and task response time. We propose a simple model for estimating the EPC for each processor. To optimize energy consumption, processors are designed to satisfy a particular usage model. Thus, the particular task suite that is anticipated to run on the device, in conjunction with user expectations to software reaction times, governs the design point of each processor. We show that the battery life of a wearable device may be extended by a factor 3-18 depending on users activity.

1 Introduction

A major obstacle for the success of certain types of battery powered Pervasive Devices (PvD) is the battery life. Depending on the device and its usage model, the battery may last anywhere from hours to months. An important mode of device operation is the *user idling mode*. In this mode, the device is always "on" but without being used by the user. "on" refers to that the device is instantly responsive. Wearable devices fall into this category because they form an extension of the user and therefore may be expected to be always instantly available. Secondly, the lower bound of power consumption on the device may be limited by its need to keep time and perform periodic tasks such as polling sensors and evaluate data, regardless of user activity. In other words, the main contributor to the accumulated "on" battery drain is the user idling mode rather than the *user active mode* in which the user actively is using the device. Most PDAs follow this usage model. A user will turn the PDA on and then press one or two buttons and make a selection from the screen. The user then reads the information and then either leave the device idling, or turn it off. In either case, the time the PDA spent idling is generally significantly larger than the time it spent executing instructions associated with the button and screen selections.

PvDs with advanced power management capabilities, such as the Compaq Itsy [1] and the IBM Linux Watch [2], have several stages of power saving modes. In the most efficient "on" low power state, the Itsy may last for 215 hours on its 610mAh battery

while the Linux Watch may last for 64 hours on its 60mAh battery. However, if the Linux Watch, for example, had to perform small periodic tasks more frequently than once per second, it would largely be prevented from taking advantage of its most efficient low power state, and battery life would drop to 8 hours. A battery lifetime of this magnitude, or even a couple of days with a larger battery, is not satisfactory. Users may not be able to recharge or replace batteries at such short intervals. Further, users may be annoyed at the frequent charging requirements, especially if they feel they are not even using the device. Although the battery drains more quickly when the user does use the device, this is more reasonable since the user can develop a sense of how much a given action costs and make usage decisions accordingly.

Another lesson we learned from the Linux Watch was that even if keeping and displaying time was the only task expected of it, the battery life of 64 hours still pales in comparison with commercial wrist watches. Although these devices also use processor chips, they can maintain and display time for several years on a single watch battery. This two-order of magnitude discrepancy in battery life was a primary motivation for this investigation. It led us to think there might be great benefits in off-loading simple repetitive tasks, such as time keeping and sensor polling, from the high-performance processor to, perhaps, a low-speed 8-bit processor with a small cache and a few necessary blocks in the I/O ring. The idea is that the low-speed processor would be specifically designed to execute small simple tasks in such a way as to consume much less active energy as compared to executing equal tasks on the high-performance processor. In other words, there must be a significant differential in energy-per-cycle (EPC) between low end and high end processors. Several means exist to widen this EPC differential, for example, by changing the architecture of the low end processor so that fewer transistors are involved in each cycle. Voltage scaling, transistor device scaling, and the switching scheme known as *adiabatic switching* are circuit techniques that improve EPC [3].

The concept of using more than one processor in a computer for power management is not new. The PC AT used a separate, small, battery-powered microprocessor to maintain time and date when the PC is powered off. The batteries for this function were often soldered in place in early PC's, so it was clearly designed for very low current drain over a long period of time. Further, a number of mobile phone companies have filed patents on computer architectures which utilize multiple computational devices [4,5]. The common thread among these systems is that the systems represent static configurations with prescribed functionality. On the other hand, we are mainly interested in developing a power efficient dynamic, or general purpose, computer system with a functionality like the Palm Pilot, Compaq Itsy and Linux Watch. In other words, a computer platform for which a programmer with relative ease can write new application and driver code and in which said code is executed in the most power efficient manner.

The multi-processor system we are going to propose can not readily be developed since many of the software and hardware components of the system are presently non-existing or require significant modification. In other words, it would take a

considerable effort to properly research and mature such a system. Nevertheless, we still believe that the system has merits from the perspective of Makimoto's *Figure of Merit* formula [6], *Figure of Merit* = (*Intelligence*) / ((*Size*)×(*Cost*)×(*Power*)), which is a qualitative measure of the value of a nomadic device as perceived by the user. Even though the formula is crude, it does suggest that it may be acceptable to trade off Size and Cost for improved Power and Functionality/Intelligence.

The paper is organized as follows. In Chapter 2, the multi-processor system is presented and we walk through a usage example. Next, in Chapter 3 and 4 we present the hypothetical target device to perform energy analysis on and the processor energy model for calculating EPC for each processor. Chapter 5 presents the task suite, user model and discusses the analytical results. Chapter 6 takes a broader look at the whole system. Chapter 7 is a summary.

2 A Low Power Multi-Processor Computer System

Architecture. In this and the next chapter we shall propose a low power multi-processor computer system. It is a first attempt to piece the whole system together in enough detail to facilitate some minimal analysis of the power savings potential. We wish to give readers enough appreciation for how the system may be connected and operated so they can improve or suggest alternatives to the system.

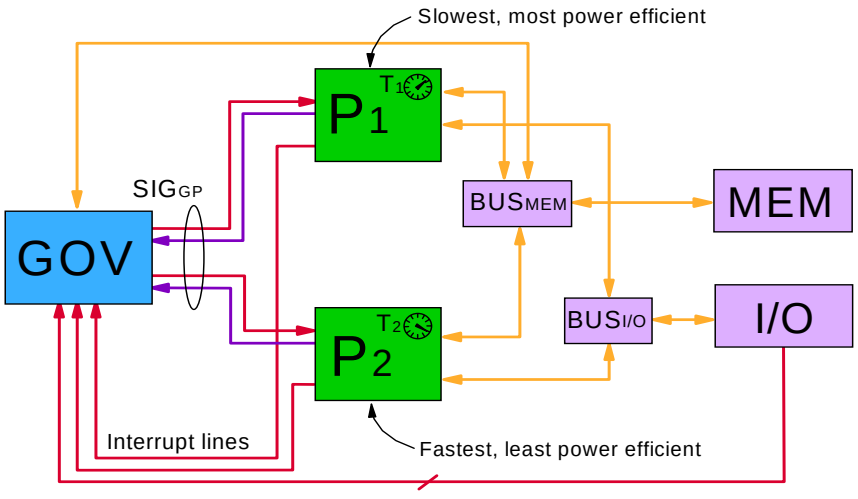


Fig. 1. Multi-processor computer system for power conscious task scheduling.

Figure 1 shows an example of a multi-processor system. It utilizes 2 processors, P_1 and P_2, and a governor circuit, GOV. MEM is the memory space and I/O is the I/O

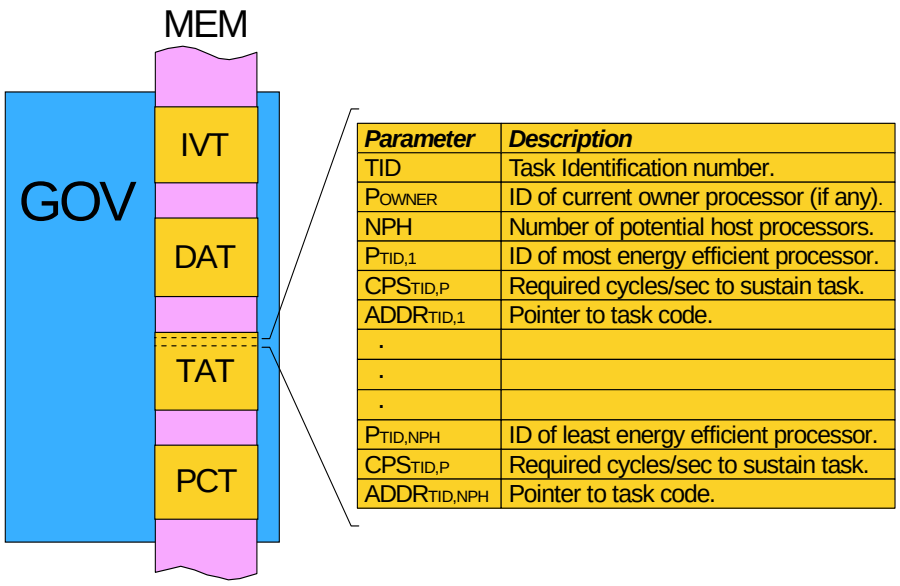
space. SIG_{GP}, BUS_{MEM} and $BUS_{I/O}$ is the governor-processor signal lines, the memory bus and the I/O bus, respectively. P_1 and P_2 execute tasks. P_1 is the most power efficient processor with little computing performance. P_2 is the least power efficient processor but with very high computing performance. All interrupts from I/O space and from the two processors are brought to GOV. GOV intercepts the interrupt signals and determines which of the 2 processors should handle the interrupt. Issues such as interrupt ownership and which processor may execute the task associated with the interrupt in the most power efficient manner, are being considered by GOV and is discussed next.

System Infrastructure. In the following we discuss some of the dynamic aspects of the system operation. The discussion is generic and is not limited to a 2-processor system. All static issues, such as initial setup, software loading, table establishment, and so forth are not discussed. The discussion will shed some light on how the whole computer system may work together. At the end, we give an example of how a calculator application is launched and operated by using a touchscreen. The following assumptions to the system infrastructure are made:

- Processors execute tasks simultaneously in parallel.
- In general the processors do not share code nor data space, and code is never moved from one processors code space to another processors code space. The only memory spaces shared among the processors are device buffer areas, the before mentioned tables in GOV and space for passing parameters.
- Interrupt handlers and tasks have been individually profiled so their computing capacity requirements is known for each of the processors they may execute on.
- Four system tables are used to coordinate energy efficient scheduling of tasks (see Figure 2): Interrupt vector table (IVT), peripheral device attribute table (DAT), process task attribute table (TAT), and the processor capacity table (PCT). As shown in Figure 2, the tables are local to GOV. GOV can access the tables without stealing bus cycles from BUS_{MEM}, and tables may be updated dynamically by the processors through BUS_{MEM}. The IVT contains dynamic pointers to DATs and TATs so GOV can access the proper table upon reception of an interrupt. DAT and TAT structures are identical. The parameters are shown in Figure 2, most of which are self-explanatory. P_{OWNER} is the processor ID of the processor that currently owns the task or handler. NPH is the number of processors which may potentially host (execute) the task, or handler. $\{P, CPS, ADDR\}_{TID,i}$ is the {processor ID, demand to processor bandwidth, code entry address} of the i'th most power efficient processor. Note that processors are listed in order of descending energy efficiency. Processors dynamically update the PCT on each launch or termination of a task or interrupt handler to reflect the processors current instantaneous spare computing capacity. GOV needs this information to properly schedule the execution of tasks and handlers.
- An OS on one processor may utilize the governor to schedule a process task for execution on an OS on another processor.
- A file system may be shared OSs.
- Each OS/processor utilizes a local timer interrupt mechanism. The processors

share a common time base counter for agreeing on instantaneous time.

- The OS utilizes a work dependent timing scheme [2] in which the local hardware timer is dynamically programmed to interrupt the processor only when there is work to be done. Physical timer ticks that do not result in work are skipped, enabling the processor to save power and to shut down more effectively.



Parameter	*Description*
TID	Task Identification number.
P_{OWNER}	ID of current owner processor (if any).
NPH	Number of potential host processors.
$P_{TID,1}$	ID of most energy efficient processor.
$CPS_{TID,P}$	Required cycles/sec to sustain task.
$ADDR_{TID,1}$	Pointer to task code.
.	
.	
.	
$P_{TID,NPH}$	ID of least energy efficient processor.
$CPS_{TID,P}$	Required cycles/sec to sustain task.
$ADDR_{TID,NPH}$	Pointer to task code.

Fig. 2. System tables for energy efficient task/handler scheduling.

Usage Example. The following example demonstrates how the whole system could work together. Assume a process called user interface (UI) is running on the processor P_{BIG}. Assume that UI has opened the touchscreen device, and the system has therefore updated P_{OWNER} in the touchscreen DAT located in GOV so that $P_{OWNER}=P_{BIG}$. Now, the user uses the touchscreen to select the icon representing a calculator application. The touch interrupt is detected by GOV which uses the IVT to find the associated attribute table. GOV then checks in the table if/which processor currently owns the touchscreen, finds P_{OWNER} in the processor list, puts the touchscreen interrupt handler address into a predefined memory slot, and finally signals/interrupts P_{BIG} which in turn jumps to the interrupt handler address. UI may now launch the calculator application, which is yet another process task. But let's assume that the launcher software first peeks into the calculator applications TAT and discovers that it requires very little computing capacity to execute. In this case, the launcher decides not to launch the calculator on P_{BIG} but rather passes the calculator request to GOV for execution on a more power efficient processor. P_{BIG} now updates

its own interrupt entry in the IVT in GOV with the address of the calculator applications TAT and clears the P_{OWNER} field in the attribute table. Then, P_{BIG} interrupts GOV.

Upon reception of the interrupt, GOV, via the IVT, finds the associated attribute table and determines that it is not owned by any processor. GOV will then schedule the calculator application on the most power efficient processor, let's call it P_{LITTLE}, on which another UI process is also running. P_{LITTLE} now changes the owner to $P_{OWNER}=P_{LITTLE}$ in the calculators TAT and then launches the calculator application (say from FLASH). The next time P_{LITTLE} receives a calculator interrupt, it's probably due to the user entering data. So P_{LITTLE} must determine the proper address to jump to in the calculator application upon future calculator inputs. P_{LITTLE} then updates the address in the calculator TAT accordingly.

Since it is now likely that the next screen interrupt will be associated with the calculator application, P_{LITTLE} further opens a touchscreen driver and updates the driver address and P_{OWNER} in the touchscreen DAT accordingly. In this fashion, the next touchscreen interrupted is routed directly to P_{LITTLE} instead of the original owner P_{BIG} which can then be put to sleep for a longer period. Though the shift in ownership of the touchscreen interrupt should only be done if P_{LITTLE} has the spare processor bandwidth as specified in the touchscreen DAT. It is also important that the touch handler and/or the UI manager on P_{LITTLE} can determine if the (x,y)-coordinates belong to the calculator. If the coordinates do not belong to the calculator application, it is equally important that P_{LITTLE} can determine to which application the coordinates do belong, if any, so that it can reflect the interrupt to the proper processor via GOV (assuming the application is not already running on P_{LITTLE}). P_{LITTLE} would do this by putting the (x,y) data in a shared buffer somewhere, update the jump address in P_{LITTLE}'s IVT entry to point to the applications TAT and then finally interrupting GOV. P_{LITTLE} should also be able to launch a new application in the same fashion that P_{BIG} originally launched the calculator application.

In order to save power effectively, it's important that the UI on P_{LITTLE} itself is capable of updating the screen whenever the calculator is being used. This has two consequences. First, it requires the use of an external display controller. Secondly, it requires the ability of several UI managers to coordinate access and share information about screen content and contexts. This may be accomplished through the shared file or buffer system.

3 The Target Device

In this section, we introduce the SensorWatch, a small wearable device with several sensors intended to help it infer its wearer's condition. For example, our hypothetical SensorWatch is able to measure it's wearer's s body temperature and pulse. When the

user takes the SensorWatch off, putting it on his bedside table, the device infers, from the lack of a pulse and temperature, that the user is not wearing the device. This is used to enter a power saving mode, disabling interfaces and tasks which are not required when the watch is not worn. On the other hand, it maintains the integrity of the watch, keeping the time with the most power efficient processor. Note that, in this case, time is not the only task that must be run in detached mode, since the watch will want to sample sensors periodically to determine when the wearer puts the watch on again. For simplicity in exposition and analysis, we simplify the analysis to consider just static scheduling of tasks. In other words a task's characteristics, such as the processor on which it should run, are established at task creation time and do not vary. A task, whenever it is invoked, will always run on the same processor.

Assumptions. The SensorWatch has time and sensor monitoring functions which must take place continuously, and at the lowest possible power. It also has on-demand functions requested in various ways by the user, which have response time requirements that may make it necessary to run them on higher powered processors. SensorWatch is a hypothetical device. For clarity we ignore other power consuming devices, such as sensors, memory, network interface and display. It is assumed that:

1. We have a wearable device with multiple processors.
2. The device has multiple sensors it must monitor at the lowest possible power.
3. Wearers create a predictable mix of events.
4. Each processor is maximally duty cycled.
5. The CPU cycles required to enter and exit SLEEP mode are negligible, relative to the task CPU cycles.
6. The CPU cycles required by the first level interrupt handlers are negligible, relative to the task CPU cycles.
7. The power consumed by GOV is negligible.
8. Each processor is able to accommodate the worst case combination of tasks which run concurrently under a multi-tasking operating system on the processor.

We consider SensorWatches with one, two and three processors. We first describe our hypothetical task characteristics and review certain task scheduling issues. Next we present a processor energy model and define energy related task parameters. We then give a suite of tasks to be considered for analysis. Finally, we give the results.

Task Characteristics. We characterize tasks as either *CPU-bound* or *I/O-bound*. CPU-bound tasks run to completion as quickly as their CPU can process them, never entering SLEEP mode. I/O-bound tasks run until they must issue an I/O request through some interface, which they then wait for by putting the processor in SLEEP mode. When the I/O completes, the SLEEP mode is interrupted and the task resumes execution. Task events arrive in two ways: either randomly or predictably. A *randomly* scheduled task is characterized by how many times per day, *NIPD*, the user, or some other random-like process, triggers the task. A *predictably* scheduled task is characterized by an interrupt frequency, $F=1/T$, where T is the scheduling interval. Thus, we can now define the following 4 task types.

Type A: CPU Bound, randomly scheduled.
Type B: CPU Bound, predictably scheduled.
Type C: I/O Bound, randomly scheduled.
Type D: I/O Bound, predictably scheduled.

Scheduling. As a first approximation to the scheduling algorithm outlined earlier, we are going to assume a static distribution of tasks. Thus, for any interrupt received by GOV, it always results in the same processor selection for task execution.

4 Processor Energy Model

We will assume a simple energy model for the processors. It is assumed that a processor dissipates the same amount of energy in each cycle. This enables us to represent a processor's energy efficiency by its Energy Per clock Cycle, *EPC*.

The energy model is based on the assumption that the energy efficiency improves as the processor clock frequency decreases. This may be achieved by voltage scaling, and by optimizing transistor design parameters [3]. Further, the overall size of the chip may be reduced by making the caches smaller, by shrinking register and bus widths, and by ensuring that the tasks that run a low-end processor limit themselves to the native register widths. These constraints would further reduce *EPC* by lowering the number of switching elements per operation and by reducing wiring capacitance.

Finally, we mention the technique of *adiabatic switching* [3]. The notion of adiabatic switching is to charge up the switching capacitor slowly by ramping up the supply voltage in synchronization with the change in output bit value, thus effectively minimizing heat loss in the resistive path. In conventional CMOS switching technology, a transistor state is changed by instantaneously applying or removing the supply voltage, V_{dd}, across the RC element. Adiabatic switching promises $EPC \propto f_{clk}$. In other words, the slower the processor is running the better the energy efficiency. However, to implement adiabatic switching requires additional control circuitry which increases capacitance and complexity. Thus, some of the advantage is lost. Adiabatic switching circuits appears to be most promising in low-speed circuits with clocking frequencies smaller than 10 MHz or so [9,10].

By lumping together all the techniques mentioned above, and being somewhat conservative about the net result, we assume that a processors energy efficiency may be characterized by the equation

$$EPC = K \bullet \sqrt{f_{clk}} \ . \qquad (1)$$

where *K* is a proportionality constant.

Task Related Energy Parameters. Next, we define the following parameters:

NC: Number of Cycles to complete task.
CPS: Cycles Per Second [Hz] required to complete task in time, *T*.
NIPD: Number of Interrupts Per Day

For periodic tasks, i.e. type B, *NIPD* may be calculated as $NIPD_i = 86,400\text{s}/T_i$ where T_i is the maximum duration the task may take to complete, which in case of a type B task is identical to the interrupt interval, and 86,400 is the number of seconds in a day. For type A and type C tasks, *NIPD* is based on the User Activity Level, *UAL*, or how frequently he uses his PvD. When discussing specific tasks later, we are going to assign typical values of *NIPD* to these tasks and then consider what happens if the user is either a more active or less active user.

The Number of Cycles Per Day, *NCPD*, for task *i* on processor *j* may be calculated as

$$NCPD_{i,j} = NC_{i,j} \bullet NIPD_i \ . \tag{2}$$

The Energy Per Day, *EPD*, for task *i* on processor *j* may be calculated as

$$EPD_{i,j} = NCPD_{i,j} \bullet EPC_j \ . \tag{3}$$

The Energy Per Day, *EPD*, for processor *j* may be calculated as

$$EPD_j = \sum_{i=1}^{NT_j} EPD_{i,j} \ . \tag{4}$$

NT_j is the number of tasks on processor *j*. The total Energy Per Day, EPD_{tot}, for all *NP* processors may be calculated as

$$EPD_{TOT} = \sum_{j=1}^{NP} EPD_j \ . \tag{5}$$

5 Task Suite

We now create a hypothetical mix of tasks, categorized into three categories depending on their requirements for processor performance. The task mix and their associated computational characteristics are listed in Table 1-3. The names of the tasks should be self-explanatory. The second column accounts for the basic demands to response time (or periodicity), *T*, the number of cycles, *NC*, to run to completion (if applicable) and the number of times per day the task is toggled (user dependent). The third column contains the demand to processor bandwidth required to sustain the task. At the very bottom (in bold font) is the total demand to processor bandwidth, CPS_{TOT}, assuming the worst case mix of tasks executing simultaneously. (Note, some tasks may be mutually exclusive.) The fourth, and last, column contains the total number of cycles per day for each task. At the very bottom (in bold font) is the accumulated total number of cycles per day, $NCPD_{TOT}$, for the particular task suite.

The low-performance tasks (shown in Table 1) are all CPU bound and periodic (type

B) in that they are timer interrupted tasks which in turn poll a sensor interface (except *TimeDate* which just updates time and date), update some variables and then determine if the new values of the variables have exceeded a threshold value, or if the evolution of the values signifies some interesting change. The purpose of the low-performance tasks is largely to determine whether to initiate/enable or disable other tasks and hardware components for the sake of power management and to infer about the state of the user and users surroundings. All tasks may run concurrently.

Table 1. Characteristics of *low*-performance tasks (NT_{low}=8).

Task Name	Basic Demands and Task Properties	*CPS* [Hz]	*NCPD* [×10^6]
TimeDate	*T*=1s, *NC*=500	500	43
UserTemp	*T*=60s, *NC*=500	8	0.7
UserPulse	*T*=10s, *NC*=500	50	4.3
UserAudio	*T*=100ms, *NC*=500	5 k	433
AmbTemp	*T*=1s, *NC*=500	500	43
AmbHumid	*T*=100ms, *NC*=500	5 k	433
DeviceOrient	*T*=50ms, *NC*=500	10 k	864
DeviceAccel	*T*=50ms, *NC*=500	10 k	864
		31 k	**2,685**

Table 2. Characteristics of *medium*-performance tasks (NT_{med}=5).

Task Name	Basic Demands and Task Properties	*CPS* [Hz]	*NCPD* [×10^6]
EvaluateWorld	*NIPD*=250/day, *T*=50ms, *NC*=10000	0.2 M	2.5
UpdateDisplay	*NIPD*=1000/day, *T*=50ms, *NC*=50000	1 M	50
FetchDbRec	*NIPD*=250/day, *T*=50ms, *NC*=10000	0.2 M	2.5
UINavigation	*NIPD*=500/day, *T*=50ms, *NC*=5000	0.1 M	2.5
SyncDb	*NIPD*=10/day, *T*=10ms/rec·1000rec=10s *NC*=1000cycles/rec·1000rec=10^6	0.1 M	10
		1.5 M	**67.5**

Table 3. Characteristics of *high*-performance tasks (NT_{high}=2).

Task Name	Basic Demands and Task Properties	*CPS* [Hz]	*NCPD* [×10^6]
VoiceCommand	*NIPD*=100/day, *T*=1s, *NC*=7.5·10^6	7.5 M	750
AudioMemo	*NIPD*=20/day, *T*=5s, *NC*=12.5·10^6	2.5 M	250
		7.5 M	**1,000**

The middle-performance tasks (shown in Table 2) have user-centric real-time requirements: acceptable behavior for them is governed by a reaction time requirements based on user experience considerations. *EvaluateWorld*, *UpdateDisplay*, *FetchDbRec* and *UINavigation* are of type A since they have to run

completion within a time acceptable to a user. On the other hand, *SynchDb* (synchronize database) is a task that may incorporate network resources. It will typically send out requests and information and then sit and wait for a reply of sorts. Thus, it is of type C. The reply, once it arrives, may not be continuous but rather arrive in multiple chunks. This task may put the processor into the SLEEP state while waiting for the network interface to generate an interrupt. All tasks may run concurrently.

The high-performance tasks (shown in Table 3) are similar to most of the medium-performance tasks in that they are randomly interrupted and, once interrupted, run as fast as they need to sustain their function. Their tasks are both of type A. The two tasks are mutually exclusive.

User Activity Level. As mentioned earlier, the User Activity Level, *UAL*, will impact the total energy performance, thus *UAL* must be included in the analysis. If *UAL*=1, then the user is assumed to use the system exactly as described above. For example, he would issue 100 voice commands per day, where each command lasts 1sec and he would synchronize his databases 10 times per day. Now, if the user is twice as active, i.e. *UAL*=2, he will issue 200 voice commands per day and synchronize his databases 20 times per day. More generally, we'll assume that the users Activity Level only affects middle- and high-performance tasks.

Processor Speeds and Energy Efficiencies. First, we need to make an assumption about energy efficiency at some given clock frequency. So let's assume a good mobile processor, such as the StrongARM SA-1110, as our reference candidate. This processor dissipates 240mW@133MHz. Thus, we can calculate the Energy Per Cycle for this reference point $EPC(f_{clk}=133\text{Mhz}) = 0.24\text{W}/133\text{MHz} = 1.8\text{nJ}$ from which the proportionality constant in Eq. 1 can be calculated, and thus Equation 1 now becomes

$$EPC = 1.8\text{nJ}/\sqrt{133\text{MHz}} \bullet \sqrt{f_{clk}} = 156\text{fJ}\cdot\text{s}^{1/2} \bullet \sqrt{f_{clk}}\,. \qquad (6)$$

As mentioned earlier, we assume that each processor is designed to support exactly the worst case combination of tasks that may conceivably run on each processor. The requirement to processor *j*'s clock frequency, $f_{clk,j}$ is found by appropriately summing CPS_{TOT} from Table 1-3 according to how many processors are considered and on which processor each task suite is executing. In turn, we can then calculate EPC_j . The results follow.

1-processor system: All tasks run on P1.
hP1: $f_{clk,1} = 9.131\text{MHz}$ => $EPC_1 = 0.471\text{nJ}$

2-processor system: Low-performance tasks run on P1 and other tasks on P2.
P1: $f_{clk,1} = 0.031\text{MHz}$ => $EPC_1 = 0.028\text{nJ}$
P2: $f_{clk,2} = 9.1\text{MHz}$ => $EPC_2 = 0.471\text{nJ}$

3-processor system: Low-, medium- and high-performance tasks run on P1, P2 and P3, respectively.

P1: $f_{clk,1}$ = 0.031MHz => EPC_1 = 0.028nJ
P2: $f_{clk,2}$ = 1.6MHz => EPC_2 = 0.197nJ
P3: $f_{clk,3}$ = 7.5MHz => EPC_3 = 0.427nJ

Results: Assuming the user activity level varies from very inactive (*UAL*=0.001) to very active (*UAL*=10), we calculated the energy performance, EPD_{TOT}, for 1-, 2- and 3-processor systems. The results are shown in Figure 3. When comparing the 2- and 3-processor case as with the 1-processor case, it may be seen that the processor energy consumption is reduced by a factor of 18 when the user is very inactive (*UAL*=0.001), by a factor of 3 when the user activity is average (*UAL*=1) and by a factor of 1.25-1.4 when the user is very active. The reason why the 3-processor system does not offer much improvement is that the task suite that runs on P2 does not significantly contribute to the total amount of task cycles consumed by the entire system.

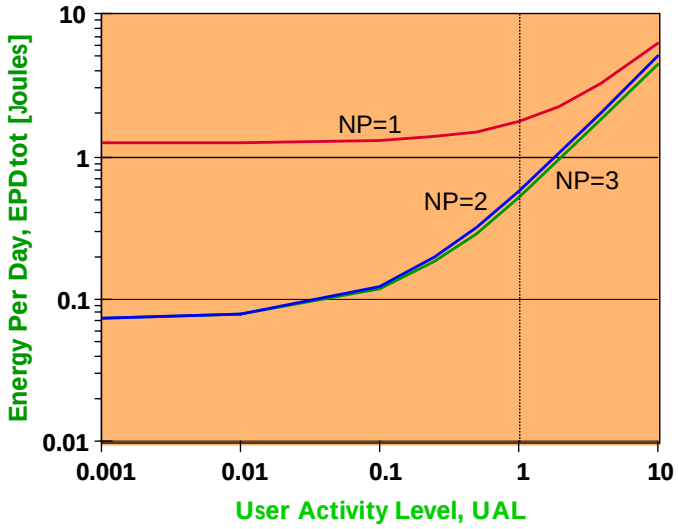


Fig. 3. Total Energy Per Day versus User Activity Level for 1-, 2- and 3-processor systems. *UAL* only affects medium- and high-performance tasks.

Keeping processor design constant in the 3 cases, and considering a usage model in which the user does not use the high-performance task suite at all (say, if he uses the device in a noisy environment) and in which he's toggles the medium-performance tasks at a five times higher rate, the 3-processor case would clearly improve the efficiency over the 2-processor case in the *UAL*=0.1-10 range where the energy consumption is dominated by the medium- and high-speed processors.

6 The Big Picture: A Discussion

Hardware Systems Perspective. The above results look very promising. But there are several factors that currently make it difficult, if not impossible, to reap the benefits of a low-power multi-processor system. Most importantly, energy-efficient low-speed (say <5 MHz) processors do not exist and there is no ongoing commercial effort to fabricate them.

Furthermore, the multi-processor system is a part of a bigger system which may have several peripheral components and several analog interfaces. Current state-of-the-art peripheral components and interfaces such as memory, network interface, sensors, etc and their associated power management operation may limit the minimally achievable energy consumption by the whole system. At least for the low-speed system considered above. To put the minimum EPD_{TOT} = 75 mJ in Fig. 3 in perspective, consider that a state-of-the-art 1.8V 2MB SRAM [7] consumes around 2 uW in its Standby mode. Keeping 2MB data alive for the whole day would then consume 173 mJ = 2.3·EPD_{TOT}. Next, let's put the EPC_1 = 0.028 nJ of the most power efficient processor in the 2-processor system in some perspective. Consider that the same SRAM consumes around 45 mW during a 70 ns read operation, then a single read operation alone would consume around 3 nJ = 113·EPC_1. With respect to retrieving code, this is best performed out of FLASH (at low speeds). A read operation in a state-of-the-art 1.8V 8MB FLASH [8] dissipates around 0.9 nJ = 31·EPC_1.

From the above perspective it may appear there is not much power savings to be gained with the multi-processor system. After all, a useful computer system needs to read/write memory and keep the data alive. However, it turns out the same techniques that may be used to reduce the EPC in processors, in particular transistor device scaling, voltage scaling and adiabatic switching, are also applicable to the design of low-power memory architectures [3,11,12] as well as for driving output pads and busses [3]. What really may limit power consumption in the end in low-speed systems, is not the central components (e.g. memory and processors) but rather the analog interfaces, such as sensor, visual, audio and wireless interfaces.

With respect to visual, audio and wireless interfaces, these may be strategically power managed by trading off usability/performance for reduced power consumption. How effectively this can be done depends on how frequently and for how long they are toggled, and on their power consumption which will depend on their position relative to the receiver (i.e. the eye, ear and wireless receiver). Their relative impact on battery life will increase as the energy efficiency of the central components decrease.

With respect to sensor interfaces, their power consumption, size and complexity is quite dependent on the type of sensor [13]. Furthermore, sensors have to be strategically positioned to perform optimally. Obviously, supplying power to these sensors from a central energy source (e.g. battery) is impractical. With the recent advancement in MEMS technology (micro electromechanical systems) [13,14], many of these sensors will become much more power efficient as well as smaller and

cheaper. Also, certain remote sensors may be self-powered. For example, any sensor or I/O device that is in use when placed on the skin (e.g. heart rate monitor, mini-speaker in ear) may in principle be self-powered (at least partially) from the heat differential between the skin and the ambient air which can be converted into electrical power by means of the thermoelectric Seebeck effect. This effect was recently used to self-power a watch [15].

Systems Issues. Due to the probably different instruction set in each processor, each task associated with an interrupt must be compiled for each of those processors that it makes sense, from a power savings perspective, to execute the task on. One consequence of this is that both more ROM and RAM is needed to hold the extra task, OS and library binaries. In turn, this may result in increased cost and increased footprint. Another thing that will impact cost is the longer time required to develop and profile mature code for multiple processors. The fact that there is more than one processor in the system will also boost cost and physical size. Finally, in order for the programmer to properly code and profile the application code, a significant understanding and appreciation for the power and performance characteristics of the device is needed.

We have proposed a system in which GOV is able to schedule tasks dynamically and power efficiently by selecting the most optimal processor from a range of processors. A static system is the special case of the dynamic system in which each task only has one processor entry in the DAT/TAT. The dynamic approach allows for more freedom to expand and configure the system. Though the static approach is simpler and may be a quite sufficient solution for devices that are designed to offer zero to limited expandability and/or which are designed to perform only a well known set of functions. In a static system the user/programmer would have to determine on which processor a new application should run on, whereas in a dynamic system, the user would just load the software and the system would figure this out by itself.

7 Summary

We have proposed a low-power multi-processor computer system. It was shown how such a system may be constructed and operated to exhibit much lower power consumption than a single processor system by taking advantage of the energy-per-cycle differential between high-end and properly-designed low-end processors. It was pointed out that energy-efficient low-speed (say <5 MHz) processors do not exist and that there is no ongoing commercial effort to fabricate them either. The same may be said about memory components. This is not surprising as there hasn't been an incentive to develop such products; there hasn't been an application which demands such parts, and which would merit the significant investment required to design, test and mature them. We believe, however, that the applications and thus the motivation are presently emerging in that there is a continuing effort to make pervasive devices smaller, lighter, more wearable and/or more distributed. These devices accordingly will have limited energy capacity. Smart

Dust [16] is an extreme example of this trend. If the application is important enough, the excess investment required to develop and mature a low-power multi-processor system may be justified.

Acknowledgements

We would like to thank Chandra Narayanaswami and Mandayam Raghunath of IBM Research for support and helpful discussions. We also thank Jaime Moreno, David J. Frank, Gheorghe Almasi and Jose Castanos of IBM Research for useful discussions.

References

1. W.R. Hamburgen et al, “Itsy: Stretching the Bounds of Mobile Computing,” IEEE Computer, 4/2001.
2. N. Kamijoh et al, “Energy trade-offs in the IBM Wristwatch computer,” Int’l Symp. Wearable Computing (ISWC2001), Zurich, 10/8-9/2001.
3. A.P. Chandrakasan, R.W. Brodersen, “Low Power Digital CMOS Design,” Kluwer Press, 1995.
4. F. Inagami, “Mobile telephone terminal having selectively used processor unit for low power consumption,” US Patent #5,058,203, 1991.
5. T.E. DAiley, “Low power architecture for portable and mobile two-way radios,” US Patent #5,487,181, 1996.
6. T. Makimoto et al, “The Cooler the Better: New Directions in the Nomadic Age,” IEEE Computer, 4/2001.
7. Samsung K6F1616R6M 16Mbit SRAM data sheet, 6/2001.
8. Intel 28F320W18 32Mbit FLASH data sheet, 8/2001.
9. J.-H. Kwon et al, “A three-port nRERL register file for ultra-low-energy applications,” Int’l Symp. Low Power Electronics Devices (ISLPED’00), 7/2000.
10. D.J. Frank and P.M. Solomon, “Electroid-oriented adiabatic switching circuits,” Int’l Symp. Low Power Electronics & Devices (ISLPED’95), 4/1995.
11. R.H. Dennard and D.J. Frank, “Memory with adiabatically switched bit lines,” US Patent 5,526,319, 1996.
12. S. Avery and M. Jabri, “A three-port adiabatic register file suitable for embedded applications,” Int’l Symp. Low Power Electronics Devices (ISLPED’98), 8/1998.
13. R. Frank, “Understanding smart sensors,”2nd Ed., Artech House, 2000.
14. R. Allan, “MEMS designs gear up for greater commercialization,”Electronic Design, 6/2000.
15. http://jin.jcic.or.jp/trends98/honbun/ntj990207.html, 2/1999.
16. J.M. Kahn et al, “Next century challenges: Mobile networking for Smart Dust,” ACM Int’l Conf. Mobile Computing & Networking (MOBICOM’99), 8/1999.