Predictive System Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation

Mani B. Srivastava, *Member, IEEE,* Anantha P. Chandrakasan, and Robert W. Brodersen, *Fellow, IEEE*

Abstract— **With the popularity of portable devices such as *personal digital assistants and personal communicators*, as well as with increasing awareness of the economic and environmental costs of power consumption by desktop computers, energy efficiency has emerged as an important issue in the design of electronic systems. While power efficient ASIC's with dedicated architectures have addressed the energy efficiency issue for niche applications such as DSP, much of the computation continues to be implemented as software running on programmable processors such as microprocessors, microcontrollers, and programmable DSP's. Not only is this true for general purpose computation on personal computers and workstations, but also for portable devices, application-specific systems etc. In fact, firmware and embedded software executing on RISC and DSP processor cores that are embedded in ASIC's has emerged as a leading implementation methodology for speech coding, modem functionality, video compression, communication protocol processing etc. This paper describes architectural techniques for energy efficient implementation of programmable computation, particularly focussing on the computation needed in portable devices where event-driven user interfaces, communication protocols, and signal processing play a dominant role. Two key approaches described here are predictive system shutdown and extended voltage scaling. Results indicate that a large reduction in power consumption can be achieved over current day solutions with little or no loss in system performance.**

I. Introduction
Computing Trends Driving Energy Efficiency in Programmable Computation

THROUGHOUT the history of general-purpose computers the emphasis, with few exceptions, has been on the development of faster computers. Recent advances in microprocessor design has resulted in clock rates approaching 300 MHz and caused power dissipation levels to rise above 30 W. However, two recent trends have resulted in the emergence of power consumption, or energy efficiency, as an important performance metric in general-purpose programmable computing systems. The first of these trends is the emphasis on saving energy in desktop computation, the reasons for which are both economic/environmental and technical. Studies [1] have shown that personal computers in USA waste \$2 billion of electricity, indirectly produce as much CO_2 as 5 million cars, and account for 5% of commercial electricity consumption (10% by the year 2000). These compelling economic and environmental reasons alone have resulted in the concept of "green computers." The technical reasons for emphasizing low power in desk top computers have primarily to do with the problem of heat dissipation which gets increasingly worse due to higher speeds and shrinking packages. Further, desk top computers of the future will no more be just plain data processing machines of the past—useful, but energy-hungry, features like DSP, multimedia, and communications are becoming integral parts of desktop computers, thus exacerbating the power consumption and heat dissipation problems.

The second trend in computing that has helped make energy efficiency an important problem is the increase in demand for portable computing and communication devices. Energy efficiency is a major design constraint in these portable devices, and not just an economic or technical factor. In particular, the need to extend battery life to a useful length of time for a given mass makes energy efficiency an important design constraint. It is this second trend that has been the motivation behind our work. Fig. 1 shows the architecture of one such portable computing/communication device—the *Infopad* wireless multimedia terminal from Berkeley [2]. All the application related computing is done on servers on a wired backbone network; the terminal itself is responsible only for the computing that is necessary for user I/O (speech, video, graphics, pen) and communications functionality. Using various recently developed techniques for low-power ASIC's [3] researchers at Berkeley have designed a set of low-power ASIC's [4] that implements parts of the terminal functionality, such as packet routing, speech and pen I/O, video framebuffer processing etc. Another example of a wireless terminal is Xerox PARC's *Tab* terminal [5] which too depends on network resources for most of its computing and storage needs. However, even with the minimalist approach to computation adopted by these wireless terminals, there are parts of functionality that are usually best implemented using general-purpose microprocessors, programmable DSP's, and programmable processor cores embedded in ASIC's. Examples include media access control and data link layer protocol processing for wireless RF communication between the portable terminal

Manuscript received May 23, 1994; revised December 14, 1994 and April 4, 1995.

M. B. Srivastava is with AT&T Bell Laboratories, Murray Hill, NJ 07974 USA.

A. P. Chandrakasan is with the Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139 USA.

R. W. Brodersen is with the EECS Department, University of California at Berkeley, Berkeley, CA 94720 USA.

Publisher Item Identifier S 1063-8210(96)01867-7.

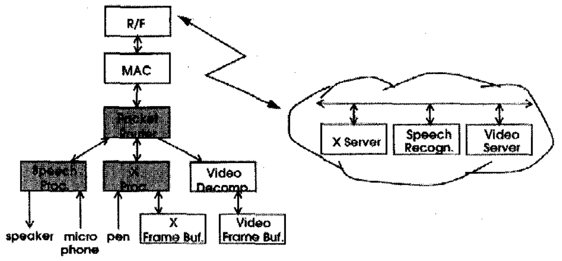


Fig. 1. Energy efficient programmable computation needed to extend battery life in portable computation.

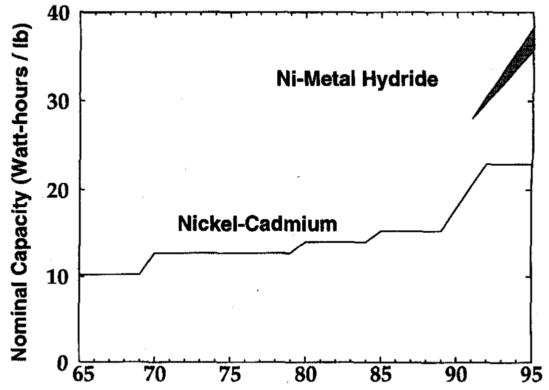


Fig. 2. Trends in battery technology.

and a basestation, the DSP processing for speech coding, and processing for parts of the graphics display server (X server, for example) that may reside on the terminal. The need for the use of software programmable components is even greater in other portable devices—such as independent laptop computers and PDA type devices—that do not adopt as extreme a partitioning of computation between the network servers and the portable device as is adopted in portable devices following a terminal-like architecture [6]. Implementation using software may be needed because 1) the algorithmic and logical (control) complexity of the application may preclude dedicated hardware and make software the only practical choice, or 2) the application may not operate continuously or different functionality may be needed at different times so that a time-multiplexed software implementation is more cost effective.

II. Architectural Approaches to Energy Efficiency

Our work is primarily focussed on architectural techniques to improve energy efficiency—or, minimizing the average power consumption—for the parts of a portable wireless terminal like Berkeley's Infopad or Xerox PARC's Tab that need to be implemented using software programmable components such as microprocessors, DSP's, and embedded processor cores. The goal is to maximize the run time for a given battery weight. The need for architectural techniques to this problem is justified by the observation that battery and semiconductor technology alone are not going to solve this problem. For example, Fig. 2 shows the trend in battery technology over the past 30 years—the slow improvement in battery technology is evident from the approximately factor of two improvement over three decades in the energy density (Watt-hours/lb) of nickel-cadmium cells to their present value of around 20 Wh/lb. The only likely new technology that will take over from NiCd over the next few years is Nickel-Metal Hydride, since it alone provides a combination of enhanced performance (30–35 Wh/lb) and environmental safety [7]. However, even for this new technology the projections are of at best another 30–40% improvement in energy density over the next five years.

Since technology alone will not provide the complete answer, one has to focus on other techniques for implementing the system such that the power consumption is reduced. So far the main approaches for lower power consumption and increased battery life have been restricted to using a limited amount of voltage scaling in the form of using 3.3 V parts instead of 5 V parts (often coupled with lower clock speed), and straightforward shutdown of the system power supply and/or clock—various notebook and laptop computers on the market illustrate these techniques. However, much higher reductions in power consumption are possible by using more sophisticated architectural and implementation strategies. For example, a proper addressing of the problems of *when* to shutdown, and *how* to scale the voltages can result in substantial improvement in energy efficiency with no or little loss in performance.

In CMOS technology there are three sources of power consumption: switching current (dynamic power), short-circuit current, and leakage currents. The switching component not only dominates in most designs, but is also the only one which cannot be made negligible even when proper circuit design techniques are used—architectural techniques therefore become important in reducing the switching component of power consumption. The average power consumption of a CMOS gate due to the switching component is given by

$$P = \alpha C_L V_{\mathrm{dd}}^2 f$$

where f is the system clock frequency, V_{dd} is the supply voltage, C_L is the load capacitance, and α is the switching activity (the probability of a $0 \rightarrow 1$ transition during a clock cycle).

The above expression suggests several strategies for increasing the energy efficiency (reducing the power consumption while maintaining the computation speed).

1) Activity-Based System Shutdown: Computations such as display severs, user interface functions, and communication interfaces are "event-drive" in nature with intermittent computation activity triggered by external events and separated by periods of inactivity. An obvious way to reduce average power consumption in such computations would be to shut the system down during periods of inactivity. It can be accomplished either by shutting off the clock ($f = 0$) or in certain cases by shutting off the power supply ($V_{\mathrm{dd}} = 0$).

2) Supply Voltage Reduction: Not all software computation is "event-driven"—data-flow systems such as DSP are "continuous" in nature. Obviously, shutdown is not an effective mechanism for these systems. An alternative strategy is to operate at the lowest possible supply voltage, as is suggested by the quadratic dependence of power on supply voltage V_{dd}.

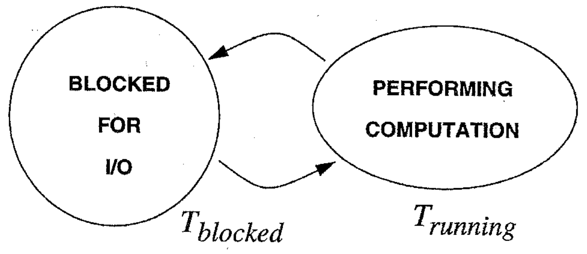


Fig. 3. Event-driven applications alternate between *blocked* and *ruuning* states.

Since only throughput, and not latency, is the metric of speed for most "continuous" applications, the loss in circuit speed that results from voltage reduction can be compensated for by architectural techniques like pipelining and parallelism that increase throughput.

3) Switching Activity Reduction: In addition to the above two strategies that are specific to event-driven and continuous computations, there is a range of architectural strategies that are applicable to all computation. Such strategies in general try to make the system more energy efficient by reducing the switching activity α. The reduction in α can be accomplished in a variety of ways—restructure the computation, restructure the communication, restructure the memory storage architecture and hierarchy, change the data encoding, etc.

The remainder of the paper describes specific architectural techniques that exploit the above strategies, focussing in particular on the first two.

III. "Shutdown" as an Energy Saving Technique for Event-Driven Computation

As shown in Fig. 3, event-driven computations are in one of two states: they are either blocked while waiting for an I/O event, or are performing computation. When running on a dedicated CPU, an event-driven application will alternate between a *blocked* state where it stalls while waiting for external events such as a key press or a mouse click, and a *running* state where it will execute instructions to perform computation. If $T_{blocked}$ and $T_{running}$ are the average time spent in the *blocked* and the *running* states, respectively, then one can improve the energy efficiency by as much as a factor of $1 + T_{blocked}/T_{running}$ provided the system is shutdown whenever it is in the *blocked* state.

There are two main problems in shutdown—*how* to shutdown, and *when* to shutdown. The first problem is addressed by mechanisms for stopping and restarting the clock ($f = 0$) or for turning off and on the power supply ($V_{\rm dd} = 0$). The second problem is addressed by policies such as "shut the system down if the user has been idle for five minutes." Although these two problems are not really independent because the decision about when to shutdown depends on the cost (in time and power) of shutting down and restarting the system, we focus primarily on the problem of deciding when to shutdown while being cognizant of the available shutdown mechanisms.

Simple shutdown techniques, for example shutting down after a few seconds of no keyboard or mouse activity, are already used to reduce power consumption in current notebook computers. However, the event-driven nature of modern window systems, together with efficient hardware shutdown mechanisms provided by newer microprocessors and system controllers, suggests the possibility of a more aggressive shutdown strategy where parts of the system may be shutdown for much smaller intervals of time while waiting for I/O events. We present such a shutdown mechanism for use on a portable X-terminal type device where a simple algorithm derived from analysis of actual traces is used to predict the length of the time spent in *blocked* state in order to minimize the impact on interactive speed, while the more frequent system shutdowns result in dramatic reductions in effective computation energy.

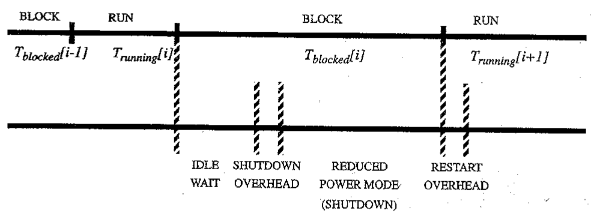


Fig. 4. Conventional shutdown approaches.

A. Conventional Shutdown Approaches

The portable computers available now use various shutdown techniques that are all variants of the following basic scheme: "Go to Reduced Power Mode after the user has been idle for a few seconds/minutes". Fig. 4 illustrates the philosophy underlying the conventional approaches to shutdown. A drawback of this straightforward policy is apparent—the system continues to waste energy while it idly waits to check for lack of user activity for a few seconds/minutes. Our experimental traces with an X server (Section III-B2) showed that while the X server spends 96–98% of its time in the *blocked* state, the average time spent on each visit to the *blocked* state is short ($\ll$ a second). The conventional shutdown schemes will therefore fail to exploit the large reduction in energy that is otherwise possible.

Typical of the conventional approaches to shutdown are the schemes used in Apple's popular Mac Powerbook series of portable computers, which have three different types of reduced power modes based on shutting down parts of the system [8]. A Power-Management IC controls the process of entering and exiting these shutdown modes by monitoring the input devices and the battery voltage, controlling the contrast of the LCD display.

1) Rest Mode: A technique termed *Power Cycling* is used in the rest mode on most models of PowerBooks. The computer enters the rest mode after 2 seconds of idle time upon which the processor registers are saved and the processor is powered down. However, the I/O devices remain on, the screen cursor continues to blink, and the keyboard continues to be scanned. After 1/60 s (16.7 ms) the power to the main processor is restored. If there has been no I/O activity, the processor is again shutdown for another 1/60 s.

The power consumption is reduced by up to 90%, i.e., by as much as $\times 10$, *while in the rest mode*. However, because the rest mode is entered only after 2 s of idle time, the effective reduction in power consumption is much less dramatic. Analysis of experimental battery life data reported in [9] for various usage scenarios of a PowerBook suggests that enabling the rest mode reduces the power consumption of the processor and logic section alone by $\times 6$ whereas the power consumption of the entire system, including the disk and display backlight, is improved by $\times 2$. Our experiments, described later, suggest improvements in the processor power consumption in the neighborhood of $\times 1.5$–$\times 2$ if power cycling is used for a processor running an X server. While $\times 1.5$–$\times 2$ is a good improvement, our shutdown technique described later achieves much larger improvement.

The impact of power cycling on the speed of interactive applications is negligible.

2) Sleep Mode: The sleep mode is entered after the computer has been idle for a user selected period of time, typically in the range of a few minutes. The computer exits the sleep mode and reverts back to the normal mode on the occurrence of an external event, such as a key press or a modem ring-detect. A decrease in computers responsiveness compared to the rest mode is traded off against an increase in energy conservation by shutting down the peripheral functions as well: the I/O ports, the disk ports, the display, the sound circuits. Power is retained only to the RAM and the Power-Manager IC.

Since the disk goes to sleep too, the effect on interactive applications is substantial—spinning the disk down and back up takes a substantial time. Further, the break-even point when it is worth spinning-down the disk and spinning it back up (because spinning-up the disk takes much increased power) is about 15 seconds [9], which implies that sleep mode is effective only for moderately long periods of idle time.

3) Shutdown Mode: This mode is typically entered on an explicit command from the user. It has the worst impact on the responsiveness of the computer but saves the maximum power—on most PowerBook models the entire computer, including the Power Manager IC, is turned off—only a tiny amount of power is drawn for a parameter RAM. The computer does not respond to external events at all while in this mode, and one needs to restart it.

B. Predictive Shutdown Approaches

The straightforward shutdown schemes described above either show only a moderate overall improvement in energy consumption with negligible loss of computer responsiveness as in the rest mode, or show a higher degree of improvement but at the cost of much decreased computer responsiveness, as in the case of sleep mode and shutdown mode. In this section we explore a shutdown mechanism where we try to predict the length of idle time based on the computation history, and then shut the processor down if the predicted length of idle time justifies the cost—in terms of both power and responsiveness—of shutting down. The basic philosophy behind the predictive approach can be summarized as follows: "Use computation history to predict whether $T_{blocked}$ will be large enough ($T_{blocked} > T_{cost}$) to justify a shutdown." Our analysis, which is based on real-life traces, suggests that our predictive shutdown approach leads to a much higher reduction in effective processor power, than is obtained with straightforward nonpredictive shutdown schemes, with only a small loss in responsiveness.

1) Helpful Trends in Computing and Communications: Two developments in computing and communication motivate more sophisticated shutdown mechanisms. First, newer microprocessors, such as Intel 486SL, and AT&T Hobbit, provide power management support integrated with the system control, thus enabling implementations of shutdown that are efficient both in terms of time and hardware cost. Power management features typically include the ability to shut down the clock and/or power supply to various parts of the system, and efficient mechanisms to store and restore the processor state for power down.

Second, the integration of computing and communication is resulting in a paradigm where increasingly the computer will become a device to access remote computation and information servers across a network. Unlike independent stand-alone PDA's and laptop computers with wireless communication capabilities, wireless terminals depend on network servers for storage and application-specific computation. The computation that is left on the personal device is largely related to the user interface and windowing system functionality. Such software is event-driven in nature, where the events arise because of user interaction. Except for extremely graphics intensive applications, a vast majority of time such software just idles while waiting for user input. Shutting down the hardware while the software is waiting for external events can give rise to potentially huge savings in computation energy.

2) Potential for Reduction in Computation Energy by Shutdown: Adopting the "X Terminal" model of computation described above, we studied the potential for energy reduction in the case of an X display server. Our analysis below, based on experimentally obtained traces of X server state, suggests that potential energy savings as high as $\times 30$ to $\times 60$ are possible.

Fig. 5 shows that the X server process running under a multitasking operating system, such as UNIX, is in one of the following three states.

a) It may be *blocked* or stalled while waiting for either an hardware event (key press or mouse activity) or requests from one of the existing client applications or a connection request from a new client.
b) It may be *running* on the processor doing some computation.
c) It may be *ready* to run but is waiting for the scheduler of a multi-tasking time-shared operating system to schedule it to run.

Let $T_{blocked}, T_{running}$, and T_{ready} be the time spent in the three states. If the X server were to run on a dedicated processor, as would be the case in our X Terminal model of computation, the server will never be in the *ready* state—it would always be either *blocked* or *running*—i.e., $T_{ready} = 0$. The hardware does nothing while the server is in the *blocked* state and can therefore be shut down by stopping the clock or by saving the state and then power down. The fraction $T_{blocked}/(T_{blocked} + T_{running})$ of the total time (in the case $T_{ready} = 0$) that the server spends in the

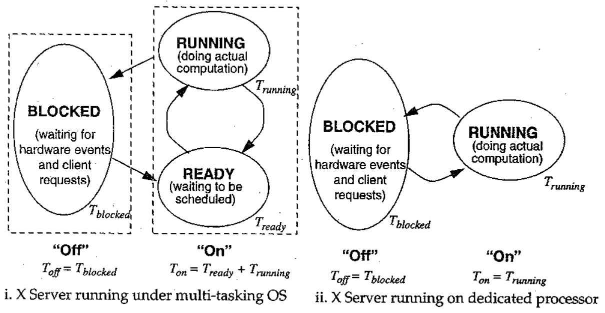

BLOCKED
(waiting for hardware events and client requests)
$T_{blocked}$
RUNNING
(doing actual computation)
$T_{running}$
READY
(waiting to be scheduled)
T_{ready}
"Off"
$T_{off} = T_{blocked}$
"On"
$T_{on} = T_{ready} + T_{running}$
i. X Server running under multi-tasking OS
BLOCKED
(waiting for hardware events and client requests)
$T_{blocked}$
RUNNING
(doing actual computation)
$T_{running}$
"Off"
$T_{off} = T_{blocked}$
"On"
$T_{on} = T_{running}$
ii. X Server running on dedicated processor

Fig. 5. States of X display server process.

blocked state corresponds to the maximum possible reduction in computation energy.

In order to estimate $T_{blocked}/(T_{blocked} + T_{running})$ under the condition that $T_{ready} = 0$, and thus the potential energy reduction in the X Terminal model of computation, we instrumented the X11R5 server from MIT, running under SunOS on a SPARCstation 2, to measure $T_{blocked}$ and $T_{running} + T_{ready}$. Unfortunately, T_{ready} is nonzero due to the multitasking time-shared nature of SunOS, and there is no easy way to measure $T_{running}$ and T_{ready} separately without modifying the kernel. Even though we ran our experiments on an unloaded workstation with no local X clients except for the console window and the window manager, there are background daemon processes over which we did not have any control. However, it is important to realize that the fraction $T_{blocked}/(T_{blocked} + T_{running} + T_{ready})$, which we can easily measure, is a lower bound on the fraction of energy that can be saved under ideal conditions. So our results err on the safe side—the estimate of the potential energy reduction that will result from using $T_{blocked}/(T_{blocked} + T_{running} + T_{ready})$, will thus be pessimistic.

From now on, we refer to the *blocked* state as the *off* state, and the *running* and *ready* states jointly as the *on* state. Further, $T_{off} = T_{blocked}$ and $T_{on} = T_{running} + T_{ready}$. Our instrumented X11R5 server measures T_{off} and T_{on}, and we use the ratio $T_{off}/(T_{off} + T_{on})$ as a lower-bound on the maximum possible reduction in energy under ideal shut down. Fig. 6 shows a sample trace of the time spent by the X server in the *off* and the *on* states over a small stretch of time, and Table I shows the results obtained by analyzing several traces obtained from real X sessions. These X sessions consisted of running our instrumented X11R5 server on a SPARCstation 2 together with the window manager *olvwm* and console window *contool* running locally, and several typical X clients, such as *xterms, FrameMaker, xclocks, mailtool, cm* (calendar manager), etc., running remotely. As is evident from these traces the X server spends most of its time, ranging from 96.5 to 98.4%, in the *off* state suggesting that energy reductions range from $\times 29$ to $\times 62$ under ideal shutdown conditions.

3) Shutdown Overhead: Although the X server trace analysis in the previous section suggests that there is a tremendous potential reduction in processor energy consumption, in practice it is much harder to realize this reduction. The chief reason for this is that the process of shutting down and restarting has a cost associated with it.

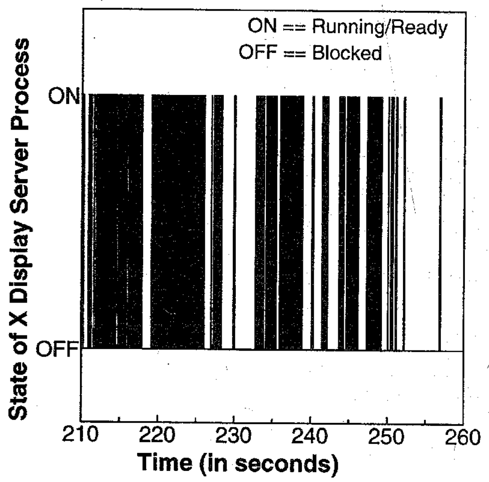


Fig. 6. Sample trace of X server state under UNIX.

TABLE I
ANALYSIS OF X SERVER TRACES FOR POTENTIAL ENERGY REDUCTION

	Trace		
	1	2	3
Trace Length (sec)	5182.48	26859.9	995.16
T_{off} (sec)	5047.47	26427.4	960.82
T_{on} (sec)	135.01	432.5	34.34
$T_{off}/(T_{off}+T_{on})$	0.9739	0.9839	0.9655
Maximum Energy Reduction (conservative estimate)	x 38.4	x 62.1	x 29.0

Enough processor state needs to be stored before shutting down so that the computation can be restarted, and the state needs to be restored to restart the computation. This process requires additional compute time and power although the precise numbers vary depending on the hardware and software organization. Some newer microprocessors, such as AT&T Hobbit and some versions of 80386 and 80486, use static CMOS logic which gives them the ability to shutdown by stopping the clock, thus reducing the power consumption to the microwatts range. Very little state needs to be saved for this as the processor registers retain their values even when the clock is stopped. This type of shutdown can be accomplished in a few microseconds. However, in other cases the entire processor state may need to be stored in the memory—for example if the processor uses dynamic logic (as most processors do) or if one wants to conserve even more energy by doing a power down instead of just stopping the clock. The overhead now increases substantially—to hundreds of microseconds to several milliseconds—as work similar to a context switch needs to be performed. At the penalty of more overhead, even more energy can be saved by storing the state on the disk as opposed to the main memory as it is no longer necessary to refresh the main memory which is typically DRAM. The overhead now increases to hundreds of milliseconds to several seconds.

The overhead due to shutdown creates problems. If the overhead is large enough, then there is problem of deciding *when* to shutdown. The time spent in the blocked state must be long enough to justify the overhead. If, after deciding to shut down, the blocking interval turns out to be too small, then one has to pay not only a power penalty, but more importantly, an effective slowing down of the computation speed because now the computer has to block for an interval higher than necessary. This slowing down translates into increased latency which, once it increases beyond a certain point, has an adverse impact on the interactive behavior of applications like X server.

4) Energy Reduction by Predictive Shutdown: The ideal situation, of course, would be to make the overhead zero or very small—but as the previous discussion suggested, overhead really is a function of the hardware and type of shutdown. The next best thing would be an *a priori* knowledge of the length of the blocking interval right at the beginning. Unfortunately, this is physically not possible.

One therefore has to resort to a heuristic to decide when to shut down. We present a novel approach where based on the recent computation history a prediction is made whether the idle time would be long enough to break-even with the shutdown overhead. Results demonstrate that for reasonable values of shutdown overhead, the predictive approach allows much higher energy savings to be achieved compared to the straightforward nonpredictive approach, while the degradation in interactive performance is negligible.

Restricting our analysis to X server running on a dedicated processor, the server process starts in the *on* state, and makes alternate transitions from *on* to *off*, and from *off* to *on* state.

Let $T_{on}[i]$ and $T_{off}[i]$ be the time spent by the X server in the ith visit to the *on* and the *off* state, respectively, for $i = 1, 2, 3, \cdots$. Relating to our previous terminology, T_{on} is the average of $T_{on}[i]$ over all i, and similarly T_{off} is the average of $T_{off}[i]$ over all i.

Further, let T_{cost} be the time overhead associated with the process of shutting down. We interpret this to mean that once we shutdown, it takes time at least equal to T_{cost} before the computation can begin. Thus, if it turns out that the event that wakes up the X server and hence restarts the computation occurs before the expiry of this T_{cost}, a penalty is paid in terms of increased latency over the case with no shutdown.

Conventional idle-time based shut-down mechanisms, such as Apple's *Power Cycling*, are nonpredictive in nature. In our model such a scheme involves making a decision to shutdown if the time spent in the *off* state after the most recent entry (say, the ith entry) has exceeded a certain idle-time threshold, which is 2 seconds in Apple's scheme. In other words we decide to shutdown on ith entry to the *off* state once $T_{off}[i]$ has exceeded 2 s. Of course, once we decide to do this, the computation cannot be restarted for at least T_{cost} time, so that if $T_{off}[i]$ would have been less than $2 + T_{cost}$, a penalty is paid in increased latency. The intuition behind such a scheme is that if the idle time has exceeded 2 s then most likely the user is going to remain idle for a long time. However, this scheme has two disadvantages. First, no shutdown is done for the first 2 s (or whatever is the idle time threshold that is chosen), and power is wasted during that period. Second, this scheme is able to take advantage of only relatively long idle periods—as our analysis showed, the average $T_{off}[i]$ is less than a second, and thus relatively short idle periods are more common.

a) Prediction of $T_{off}[i]$: To address the deficiencies of the conventional idle-time based shutdown scheme, we designed and studied a predictive scheme for deciding when to shut down. The idea is that a simple heuristic rule is used that, on the ith entry into the *off* state, predicts whether $T_{off}[i]$ is going to be long enough to justify shutting down. Specifically, the heuristic rule predicts whether $T_{off}[i] \geq T_{cost}$, and if so it is decided to shut the processor down.

The heuristic rule uses the computation history to make the prediction. In our case the previous values of $T_{on}[i]$ and $T_{off}[i]$ form the obvious computation history. In particular, $T_{on}[1] \ldots T_{on}[i]$, and $T_{off}[1] \ldots T_{off}[i-1]$ can be used to predict $T_{off}[i]$. Further, the prediction rule itself needs to be simple to evaluate and not require too much state information. Our intuition led to two approaches and the corresponding prediction rules:

- Our first approach was to use regression analysis to arrive at a model for predicting $T_{off}[i]$. We used *Mathematica* to analyze the traces obtained for the X11R5 server running on a SPARCstation 2 and arrived at the following model for $T_{off}[i]$ in terms of $T_{off}[i-1]$ and $T_{on}[i]$.

$$\begin{aligned} T_{off}[i] = {} & 0.074\,001\,8 + 0.553\,733 T_{off}[i-1] \\ & - 0.009\,473\,48 T_{off}[i-1]^2 + 1.422\,33 T_{on}[i] \\ & + 1.138\,83 T_{off}[i-1] T_{on}[i] - 1.491\,43 T_{on}[i]^2. \end{aligned}$$

  A quadratic model with cross terms was used because the scatter plot of $T_{off}[i]$ versus $T_{on}[i]$ in Fig. 7, and a similar plot of $T_{off}[i]$ versus $T_{off}[i-1]$ shown in Fig. 8, both suggested a hyperbolic behavior.
  If the value of $T_{off}[i]$ predicted by the above model is $\geq T_{cost}$, a decision is made to shut the processor down.
- The second approach is based on an even simpler intuition—$T_{on}[i]$ corresponds to the most recent history of $T_{off}[i]$, and the scatter plot of $T_{off}[i]$ versus $T_{on}[i]$ in Fig. 7 is a L-shaped plot with the points concentrated along the two axes. This suggests that a large value of $T_{on}[i]$ is followed by a small $T_{off}[i]$ with a very high probability, and that the $T_{off}[i]$ following a small value of $T_{on}[i]$ is fairly evenly distributed. This suggests the following simple filtering (or thresholding) rule as the prediction heuristic

$$T_{off}[i] \geq T_{cost} \Leftrightarrow T_{on}[i] \leq T_{on\_threshold}$$

  and, Fig. 7 suggests that for $T_{cost} = 10$ ms, a reasonable value of $T_{on\_threshold}$ is in the range of 10 ms to 15 ms. Note that $T_{cost} = 10$ ms is a very safe upper bound on the shutdown cost if the state is being saved in the main memory—in reality it is more likely to be $\times 10$–$\times 100$ smaller.

b) Hit ratios for prediction schemes: The above heuristic rules to predict whether $T_{off}[i] \geq T_{cost}$ are similar in nature to the replacement rules in caches. A good idea of their efficacy can therefore be obtained by measuring the probability with which our prediction is correct, i.e., the *hit*

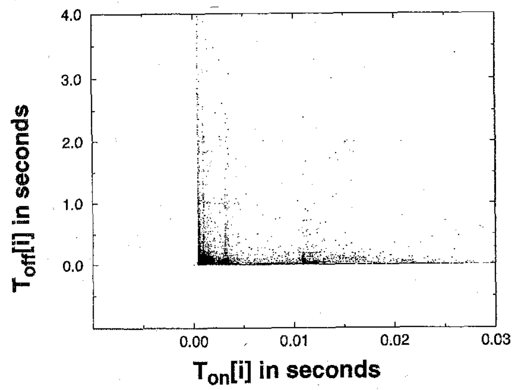


Fig. 7. L-shaped $T_{off}[i]$ versus $T_{on}[i]$ scatter plot.

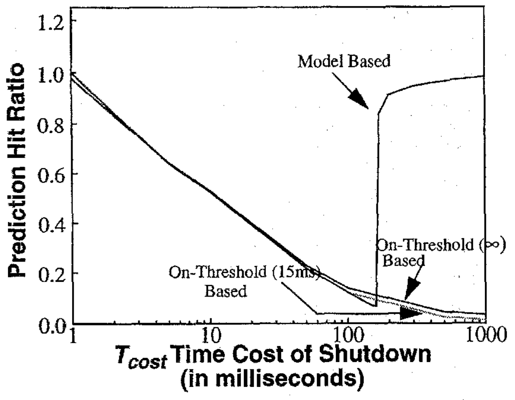


Fig. 9. Hit ratio curves.

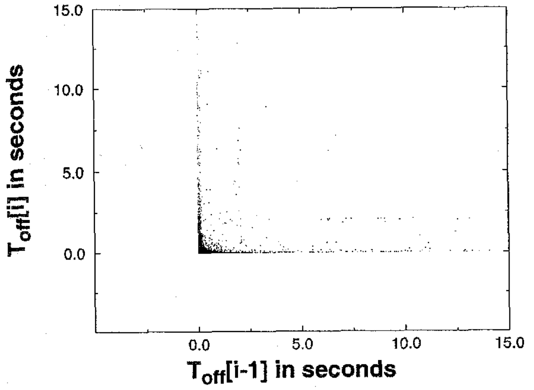


Fig. 8. $T_{off}[i]$ versus $T_{off}[i-1]$ scatter plot.

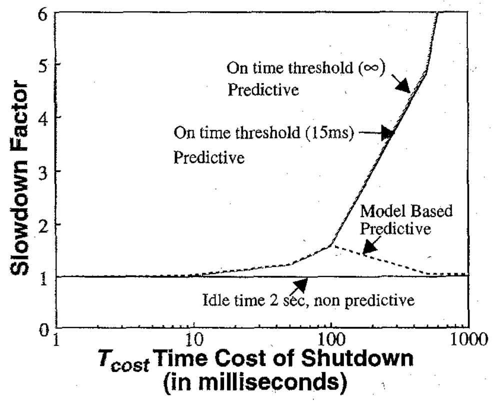


Fig. 10. *Slowdown* as a function of T_{cost}.

ratio. We used the experimentally obtained traces to simulate the X server running with predictive shut down. Fig. 9 shows the hit ratio for three schemes: the *model-based* prediction, the *on-threshold-based* prediction with $T_{on\_threshold} = 15$ ms, and the *on-threshold-based* prediction with $T_{on\_threshold} = \infty$. The last case corresponds to a scheme where we always decide to shutdown. The hit-ratio curve corresponding to the *model-based* prediction shows a sudden jump at around $T_{cost} = 160$ ms. This is an artifact of the quadratic model. Finally, note that the hit ratio for the *on-threshold-based* prediction scheme is relatively insensitive to $T_{on\_threshold}$ changing from 15 ms to ∞ (the two curves almost overlap), whereas it does degrade with increase in T_{cost}.

c) Impact on responsiveness—Slowdown: We mentioned earlier that the time overhead T_{cost} associated with shutdown can have a negative impact on the responsiveness of the computer due to increased latency. This *slowdown* is difficult to quantify as it involves ill-defined psychological and biological metrics—in fact some studies even suggest second order effects in interactive behavior such as an increase in the user think time as the computer responsiveness degrades [10]. In the absence of any well-defined metric, we used our own simple and intuitive measure of slowdown—it is the factor by which the total length of the X server session is increased due to shutdown being used. This increase occurs because many of the $T_{off}[i]$'s are longer than they would have been without shutdown because of the overhead T_{cost} associated with shutdown. Fig. 10 plots the slowdown resulting from the various schemes for different values of T_{cost}. For comparison we also plot the slowdown resulting from using a conventional *idle-time-threshold* based scheme with a threshold of 2 s, similar to Apple's Power Cycling Scheme.

As the curves demonstrate, the conventional schemes have almost no slowdown for all values of T_{cost}. However, the *on-threshold-based* prediction scheme performs reasonably well for values of T_{cost} smaller than 10–15 ms—the slowdown is less than 3 to 4% which is not noticeable at all. We verified this by creating a special version of the X11R5 server where an artificial delay, corresponding to T_{cost}, was added on entry to the *off* state. In fact, even a T_{cost} of 100 ms, which corresponds to a slowdown of 50 to 60% for *on-threshold-based* prediction scheme, resulted in a very usable, though noticeably sluggish, X server on the SPARCstation 2. As we mentioned earlier, T_{cost} corresponding to saving state in the main memory and restoring state from the main memory is typically going to range from just a few microseconds for processors with ability to stop the clock, to a few hundreds of microseconds or a few milliseconds for other processors—and for these range of values of T_{cost} the slowdown is negligible for the predictive schemes.

TABLE II
SUMMARY OF ENERGY REDUCTION

		Energy Reduction	% Slowdown
T_{cost} = 0(ideal)	Known $T_{off}[i]$	x 38.4	0 %
	Non-Predictive Idle-Time Threshold (2 seconds)	x 1.7	0 %
	On-Threshold-Based Predictive Scheme ($T_{on\_threshold}$ = 15 ms)	x 20.1	0 %
	On-Threshold-Based Predictive Scheme ($T_{on\_threshold} = \infty$)	x 38.4	0 %
	Model-Based Predictive Scheme	x 38.4	0 %
T_{cost} = 10 ms	Known $T_{off}[i]$	x 25.7	0 %
	Non-Predictive Idle-Time Threshold (2 seconds)	x 1.7	0 %
	On-Threshold-Based Predictive Scheme ($T_{on\_threshold}$ = 15 ms)	x 20.1	2.7 %
	On-Threshold-Based Predictive Scheme ($T_{on\_threshold} = \infty$)	x 38.4	2.8 %
	Model-Based Predictive Scheme	x 38.4	2.8 %
T_{cost} = 100 ms	Known $T_{off}[i]$	x 5.1	0 %
	Non-Predictive Idle-Time Threshold (2 seconds)	x 1.7	.025 %
	On-Threshold-Based Predictive Scheme ($T_{on\_threshold}$ = 1 ms)	x 2.1	21.1 %
	On-Threshold-Based Predictive Scheme ($T_{on\_threshold} = \infty$)	x 38.4	59 %
	Model-Based Predictive Scheme	x 38.4	59 %

d) Reduction in computational energy: Finally, we evaluated the various predictive, nonpredictive, and ideal shutdown schemes from the point of view of reduction in computational energy. The results are summarized in Table II for three values of T_{cost}—0 ms, 10 ms and 100 ms—the former being the ideal case, and the latter two being safe upper bounds for storing the processor state in main memory and on disk respectively. The results demonstrate that for $T_{cost} = 10$ ms, a conservative upper bound for most processors, the predictive schemes have much superior energy savings at negligible slowdown. However, for $T_{cost} = 100$ ms, which may correspond to the case where the processor state is saved on the disk, the predictive schemes have noticeable degradation of interactive performance.

IV. ARCHITECTURE-DRIVEN VOLTAGE REDUCTION

Not all programmable computation, even on user-interaction dominated portable devices, is event-oriented—many functions that are not event-driven and instead execute continuously also require implementation as software running on microprocessors or embedded core processors on an ASIC. In fact embedded software for DSP core processors on an ASIC has emerged as the dominant method of implementing speech coding, speech compression, text-to-speech synthesis, simple speech recognition, modem functionality, and many other signal processing functions in portable devices such as cellular phones, PDA's, etc. Given the "continuous" nature of signal processing functions, shut-down strategies such as the predictive shut-down technique of the previous section are not effective. The quadratic dependence of power on supply voltage V_{dd} however suggests another mechanism for energy efficiency—reduction in supply voltage. Unfortunately, as shown in Fig. 11(a), operation at reduced supply voltage comes at the expense of increased gate delays, and consequently slower clock frequency for a given circuit. However, certain nice attributes present in most DSP algorithms—their performance is determined by throughput alone and not latency, and their ample inherent concurrency—often make it possible for one to use architectural techniques such as parallelism and pipelining to increase the computation throughput which can then be traded off against the loss in speed due to voltage reduction for a net gain in energy efficiency for unchanged throughput.

This underlying approach of applying pipelining and parallelism to aggressively reduce the supply voltage (down to the 1–1.5 V range) and therefore increase the energy efficiency has been used to implement ASIC's with dedicated architectures for DSP algorithms [3], [4]. In this section, the goal is to apply architecture-driven voltage scaling to software programmable computation. While the supply voltages of many microprocessors have indeed come down to 3.3 V or 3 V from 5 V, the choice of 3.3 V has been driven by the fact that for this level of voltage scaling the degradation in system performance (clock rate) is not very significant. Architecture-driven voltage reduction, on the other hand, reduces voltage even further into the realm where there is a significant reduction in clock

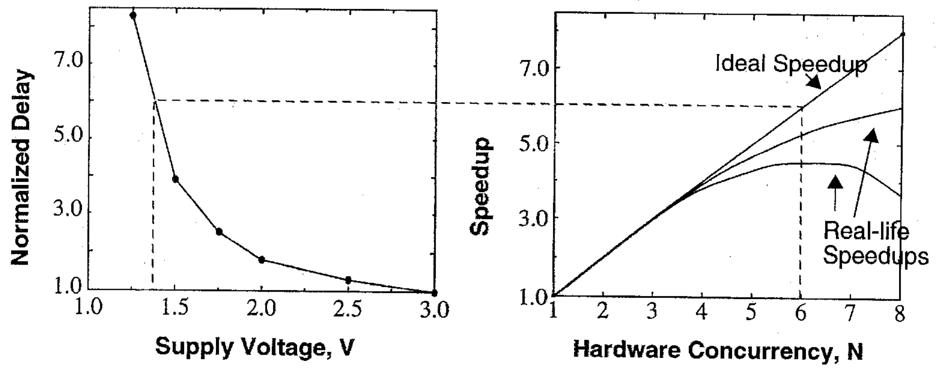


Fig. 11. Trade-off between voltage and hardware-concurrency. (a) Normalized gate delay versus supply voltage. (b) Speed-up versus hardware concurrency.

rate, but uses architectural concurrency to keep the system throughput at its original level.

A. Voltage-Concurrency Trade-Off and Architectural Bottlenecks

As already mentioned, the basic idea in architecture-driven voltage reduction is to compensate for the loss of speed due to increased gate delay when operating at a lower voltage, with the increase in speed due to increased hardware concurrency. The power consumption is reduced at a fixed computation speed (throughput). Hardware concurrency can be due to parallelism or pipelining or a mix of the two. For example, hardware that performs N operations in parallel has concurrency of N. Similarly, a hardware with N pipeline stages so that N successive operations are overlapped also has a concurrency of N. Compiler transformations such as parallelization and software pipelining play an important role in restructuring the computation so that the hardware concurrency is fully exploited.

Fig. 11 illustrates this trade-off between voltage and hardware concurrency by plotting the curve for increase in gate delay (normalized to gate delay at 3.0 V) next to hypothetical curves for increase in throughput (speedup) due to hardware concurrency, and using the same scale for normalized gate delay and speedup. If one had a hardware concurrency of 6—for example, by using a six processor parallel computer—then one can obtain an ideal speedup of $\times 6$ as shown by the plot in Fig. 11(b). Now if one were to reduce the voltage from 3.0 V to a level where the gate delay increases by $\times 6$, the clock frequency will have to be reduced by $\times 6$ as well. Each of the processor will therefore slow down by $\times 6$ so that the net speedup with the six processor machine operating at the reduced voltage will be 1 compared to the uniprocessor machine operating at 3.0 V. In general, the strategy would be to reduce the voltage to a level where the normalized gate delay increases by the same factor as the speedup. The throughput of the concurrent hardware operating at the reduced voltage level will then be the same as the throughput of the nonconcurrent hardware operating at the original voltage of 3.0 V. For our example the reduced voltage level at which the gate delay increases by $\times 6$ is 1.3 V, as the dotted lines in Fig. 11 show. Assuming that the switched capacitance for the six processor machine was $\times 6$ higher than for the uniprocessor machine, the power is reduced by a factor of $(1/6) * (3/1.3) * (3/1.3) * (6/1) = 5.3$.

Consider a more detailed analysis. Let $S(N)$ be the speedup for a hardware concurrency of N from Fig. 11(b). Obviously, for nonconcurrent hardware $N = 1$, and $S(1) = 1$. Let $V(d)$ be the voltage for a normalized gate delay d in Fig. 11(a), with $V(1) = 3.0$ V being the reference point. The initial hardware has $N = 1, S(1) = 1$, and $V(1) = 3.0$ V. Let $f(1)$ and $f(N)$ be the clock frequencies for the initial and the final hardware, let $C(1)$ and $C(N)$ be the switched capacitances, and let $P(1)$ and $P(N)$ be the power consumptions.

If the hardware concurrency is due to parallelism, then one can operate each of the N parallel hardware units at a frequency $f(N) = f(1)/S(N)$ so that the effective throughput is unchanged. Further, parallel hardware units will require some overhead circuit so that the total capacitance can be expressed as $C(N) = (N + \varepsilon(N)) * C(1)$ where $\varepsilon(N)$ is the overhead capacitance. Assuming unchanged switching activity, it follows that

$$\frac{P(1)}{P(N)} = \frac{C(1)V(1)^2 f(1)}{C(N)V(S(N))^2 f(N)} = \frac{S(N)}{N+\varepsilon(N)} \times \left(\frac{V(1)}{V(S(N))}\right)^2. \quad (1)$$

Similarly, if the hardware concurrency is due to N-level pipelining, then the hardware can be operated at a reduced voltage of $V(S(N))$ so that the clock frequency will be $f(N) = N * f(1)/S(N)$, which in turn gives an effective throughput that is unchanged. Let the overhead capacitance due to pipelining be $\varepsilon(N) * C(1)/N$, so that the total capacitance is $(1+\varepsilon(N)/N) * C(1)$. Assuming unchanged switching activity, the ratio of the power consumption is again given by an expression identical to that in (1).

An increase in energy efficiency is obtained whenever $P(1)/P(N)$ is > 1. In the ideal case of linear speedup $(S(N) = N)$ and no capacitive overhead $(\varepsilon(N) = 0)$, it is clear from (1) that $P(1)/P(N)$ is indeed > 1 because $V(1) > V(S(N))$. In the ideal case, one can get arbitrarily large improvements in energy efficiency by continuing to increase hardware concurrency and decrease voltage until the devices stop working.

However, (1) points to two fundamental architectural bottlenecks that prohibit an arbitrary increase in energy efficiency by reducing voltage. First, the speedup is not linear in most cases—as shown in Fig. 11(b), real examples typically show a speedup that starts saturating, and even decrease, as N is increased. In parallel hardware this may be due to lack of enough parallelism in the computation, or due to effects like bus contention. In pipelined hardware the speedup may not be linear because of granularity of pipelining, or because of the existence of pipeline interlocks and feedback cycles in the computation. The second architectural bottleneck occurs because $\varepsilon(N)$ is not zero in real world. In parallel hardware, capacitive overhead is contributed by data multiplexing/demultiplexing, bus arbitration, etc., while in pipelined hardware the capacitive overhead is contributed by pipeline registers.

Together these two architectural bottlenecks—nonlinear speedup $S(N)$ and capacitive overhead $\varepsilon(N)$—place a restriction on the increase in energy efficiency that can be obtained.

B. Exploiting Concurrency in Programmable Computation

Preceding discussion makes it clear that the energy efficiency can often be increased by trading voltage reduction with increase in concurrency. There are two independent issues in this trade-off: the available hardware concurrency, and the concurrency that is inherent in the computation algorithm. Compared to typical ASIC's it is harder to exploit concurrency in software computations. Three types of concurrency are inherent in software: i) Instruction level parallelism [11] which can be discovered by dataflow analysis and enhanced by software pipelining, loop unrolling, and branch prediction, ii) thread level parallelism as manifested, for example, in coarse grained parallelism of an X server handling multiple clients via separate threads to reduce unnecessary blocking, and iii) process level parallelism as exploited by time-sharing systems to improve throughput.

Complementary to algorithmic concurrency is hardware concurrency. There are three main ways of increasing the hardware concurrency in programmable computers: i) increase the number of processors to exploit process and thread level parallelism, with accompanying hardware and OS overhead, ii) increase the number of functional units to exploit instruction level parallelism, with accompanying overhead due to more complex instruction issue, and iii) increase the levels of pipelining to also exploit instruction level parallelism.

The success of a compiler in utilizing the intrinsic concurrency of an algorithm to exploit the available hardware concurrency is a major determinant in achieving energy efficiency.

C. A Power Consumption Model for MIMD

To evaluate the effect of hardware parallelism on power, a model to estimate the power consumption needs to be developed. For this we need to define the following quantities.

Let $S(N)$ = computation speed when the MIMD system has N processors. The speed metric $S()$ could represent either throughput or latency. Also, let $V(N)$ = Lowest supply voltage at which we can run the N processors while maintaining the same throughput as with the uniprocessor system. Assume that the uniprocessor is running at a reference voltage of 3 V (i.e., $V(1) = 3$ V). Given the speedup factor (based on the number of processors and algorithmic constraints), the supply voltage $V(N)$ can be determined from the function shown in Fig. 11(a) which is obtained by characterizing the semiconductor process. For example, if $N = 4$, and if $S(4) = 4$, then from Fig. 11(a), $V(4)$ is approximately 1.5 V.

Ignoring static power consumption due to leakage currents, the power consumed by CMOS circuits is given by $C * V^2 * f$. Therefore, the power consumption of a uniprocessor is

$$P(1) = (C_{processor} + C_{interconnect} + C_{memory})V(1)^2 f \quad (2)$$

where $C_{processor}, C_{interconnect}$ and C_{memory} are determined for a specified set of hardware modules (e.g., DSP32C or 386SL) and technology (PCB interconnect ≥3–4 pF/inch routing capacitance). For example, for a DSP32C processor, the average capacitance switched is given by $1.2\ \mathrm{W}/((5.25)^2 25\ \mathrm{MHz} = 1.75$ nF (the power of 1.2 W, obtained from data sheets, does not include the I/O power). For interconnect, considering both the bus and pin components, and assuming an average switching activity, the capacitance was determined to be 0.15 nF.

Now consider extending the model presented in (2) to multiple processors. Ideally, speedup grows linearly with the number of processors; however, due to interprocess communication overhead, the speedup is typically not linear. Similarly, parallelism will introduce capacitance overhead. The overhead is often attributed to an increase in interconnect capacitance (longer wires on the PC board/MCM), loading capacitance (more processor on the shared bus), control capacitance, interprocess communications overhead (e.g., more memory I/O transactions), etc. Taking these factors into account, the power consumption as a function of the number of processors is given by

$$P(N) = [NC_{processor} + NC_{interconnect}(N + \%C_{\text{over}}) + C_{memory}(N + \%C_{\text{over}})]\frac{fV(N)^2}{S(N)}$$

where $\%C_{\text{over}}$ is the interprocessor communications overhead. Note that the physical interconnect capacitance, the second term, increases linearly with the number of processors. This is a reasonable assumption since for PCB interconnect technology, the processors can be placed right next to each other (in a linear fashion) and routing can be performed on multiple layers (unlike ASIC's where the routing is confined typically to 2 metal layers).

Note that a minimum bound on the supply voltage, V, can be set by one of the following.

- Noise immunity requirements.
- $S(N)$ stops increasing with N. Using the curve in Fig. 11(a), a maximum bound on N can be translated into a minimum bound on V.
- There may be a maximum bound on N due to operating system and other software considerations. This too can be translated into a corresponding minimum bound on V.

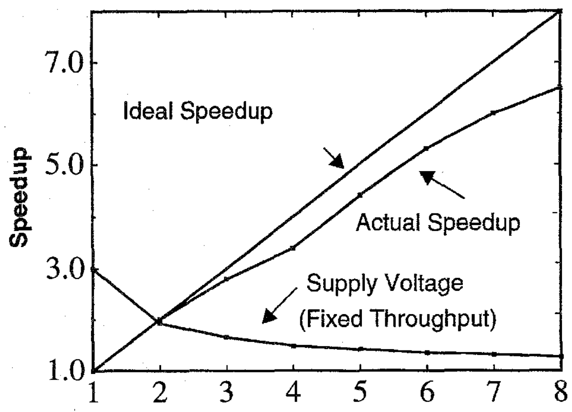


Fig. 12. Speedup and V_{dd} versus N.

Intuitively, a model of speedup in MIMD will have the following characteristics.

- For small values of N the speedup will be close to linear.
- At higher values of N, the speedup will start saturating and reach a peak.
- Finally, for even larger values of N, the speedup may actually start falling off. This may happen because, for example, the arbitration overhead for access to a shared bus may just grind the system to a halt.

Such a "hump" like speedup curve is characteristic of shared bus centralized memory systems. For our purposes, there is no rational reason to parallelize beyond the point of saturation. In other words, we are only interested in the initial part of the curve. One such model of speedup, using analytical results from [12], is given by the following

$$S(N) = \frac{N}{1 + \frac{N-1}{r}} \tag{3}$$

where r is a constant. As N gets large, the speedup will saturate to r. The value of r depends on, among other things, the ratio of computation to communication, the granularity and number of interacting tasks, and the communication network.

D. Example: CORDIC on a MIMD Using Thread Level Parallelism

First, we will consider signal processing applications running on programmable processors. For such applications, it is typically throughput (and sometimes latency) which is the design constraint rather than trying to compute as fast as possible. For example, a speech codec has to compress and decompress speech at a sample rate of 8 kHz, however, once this throughput requirement is met, there is NO advantage in making the computation faster. As described earlier, by making the computation faster (using more parallelism for example), the supply voltage can be dropped and hence the power can be reduced while still meeting the functional throughput. Typical applications include filters, speech processing, robotics and real-time image processing.

Most examples in this class exhibit a substantial amount of concurrency. For example, all DSP applications are executed in an infinite time loop, giving rise to temporal concurrency. In addition, several exhibit spatial concurrency. For low-power

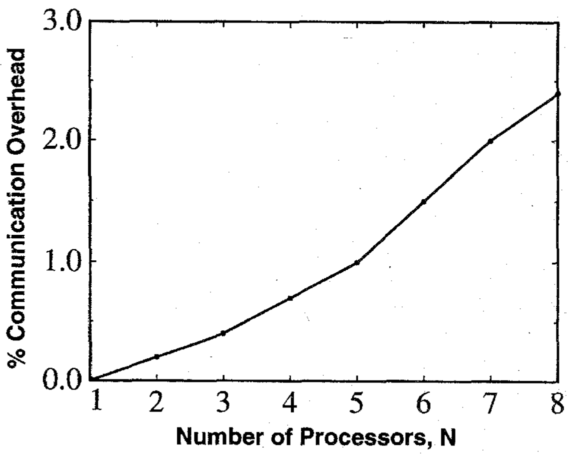


Fig. 13. %Communications overhead versus N.

operation (as we will see), it is very important to detect and exploit concurrency since we can run at reduced supply voltages. For example, temporal concurrency can often be exploited by pipelining, resulting in dramatic speedup and hence lower power for a fixed throughput. For application that have a lot of recursion (feedback loops), it often necessary to apply a series of transformations (like loop unrolling) to achieve speedup. Compilers have been developed to effectively exploit concurrency in DSP applications [13].

To study the power trade-offs of DSP multiprocessor implementations, a cordic algorithm (obtained from [13]) is studied that converts cartesian to polar coordinates iteratively in 20 steps. It takes as input an (X, Y) coordinate, as well as an array of correction angles. The loop iteratively calculates the corresponding amplitude and phase. Since each iteration is dependent on the results of the previous iteration, the computation is sequential in nature.

A scheduling algorithm which only exploits spatial concurrency would perform poorly on this example. However, optimizing using transformations like pipelining the loop and assigning successive loop iterations to successive processors, significant speedup can be achieved. The analysis is based on the DSP32C processor power numbers. Fig. 12 shows the ideal speedup (linear with processors) and the actual speedup obtained. Fig. 12 also shows the lowest supply voltage $(V(N))$ at which the various multiprocessor implementations can run while meeting the same functional throughput as the uniprocessor version. Fig. 13 shows the communications overhead as a function of the number of processors.

Fig. 14 shows the power consumption as a function of N. We see that a factor of 3.3 reduction in power can be accomplished by reducing the supply voltage from 3 to 1.5 V. The power starts to increase with more than 6 processors since the overhead circuitry (interconnect capacitance) starts to dominate, resulting an optimum number of processors for power.

E. Low Power Computers—Multiple Slow CPU's More Energy Efficient than a Single Fast CPU?

The discussion in this section shows that if energy efficiency is the consideration, it is better to use concurrent (parallel

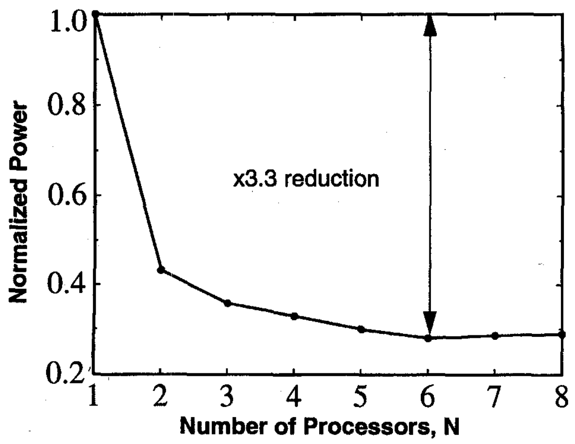


Fig. 14. Power versus N.

or pipelined) slower hardware operating at a lower voltage than to use nonconcurrent faster hardware at a higher voltage. To put it differently, multiple slower CPU's are more energy efficient than a single fast CPU for applications with enough algorithmic concurrency. This suggests that from a power perspective it may be better for future microprocessor chips to consist of multiple slow and simple CPU's operating at a slow clock frequency and a low voltage—a conclusion which is just the opposite of the current trend toward complex CPU's operating at extremely high frequencies (many 100's of MHz). However, since applications also have nonparallel components, it might make sense to have a fast processor with efficient shut-down circuitry for handling the serial portion of a computation, together with an array of multiple slow processors for the remainder.

V. Architecture Techniques for Reducing Switching Activity

While the predictive shutdown and architecture-driven voltage reduction techniques were specifically targeted at event-driven and continuous data-flow applications respectively, there is a range of architectural techniques that are more widely applicable because they try to restructure the computation, communication, and memory architecture so as to reduce the switching activity. One such technique is I/O bus switching activity reduction for which the various approaches can be classified into three categories: i) Reduction in data traffic, ii) Change in transmitted data format, and iii) Change in transmitted data encoding.

Approaches belonging to the first category rely on data traffic reduction to get reduction in switching activity. Examples include lossless data compression [14], variable length instruction encoding for low power (e.g., in AT&T's Hobbit), and on-chip caches to increase energy efficiency [15]. The second category approaches achieve switching activity reduction by selecting a suitable spatial and temporal format to transmit data bits, without changing the data bits themselves. The third category approaches rely on encoding the information bits to reduce the switching activity. For example, gray-coded program counters and auto-increment/decrement address modes [16] can reduce switching activity when accessing a sequence of memory addresses. Also in this category is our observation that the number of bit transitions is considerably reduced if sign-magnitude encoding is used instead of two's complement encoding to represent values of a data stream that has frequent zero crossings and a small dynamic range. Fig. 15, which shows the transition probabilities for various bit positions, makes it clear that the activity in higher order bits is significantly higher in the two's complement representation where the higher order bits are used for sign extension.

VI. Conclusion

We described three broad architectural approaches for energy efficient programmable computation: predictive shutdown, concurrency driven supply voltage reduction, and switching activity reduction. A significant reduction in power consumption can be obtained by employing these architectural techniques. For example, parallel processors operating at a reduced voltage can substantially improve the energy efficiency of a fixed throughput computation (by a factor of 4 in the example we described). For applications where continuous computation is not being performed (like X-server, etc.), we have shown that an aggressive shut down strategy based on a predictive technique can reduce the power consumption by a large factor compared to the straightforward conventional schemes where the power down decision in based solely on a predetermined idle time threshold.

In addition to the specific techniques described in the paper, our three architectural approaches—predictive shutdown, concurrency driven voltage scaling, and switching activity reduction—can also be applied to other parts of an architecture. For example, our predictive shutdown heuristic may be applied to manage the shutdown of peripherals such as disks. An on-line algorithm that makes the shutdown decision using a prediction of the time to next disk access may result in better power reduction compared to more conventional threshold based policies for disk shutdown [17], [18].

A second example where the ideas embodied in our techniques can be applied is the combination of parallelism-driven voltage reduction with switching activity reduction to increase the energy efficiency of memory operations when the access pattern is sequential in nature. Such sequential access patterns occur, for example, when fetching video data from a frame-buffer memory or when fetching cache lines from main processor memory or when writing a page of virtual memory back to the disk. Instead of accessing data from memory in a serial fashion, several words can be read from memory and the memory can be clocked at a lower rate for the same throughput. For example, reading four words in parallel implies that the memory can be clocked at 1/4 the rate of a serial implementation in which 1 word is read every cycle. The time available to read the memory for the parallel implementation is four times as long as for the serial version and therefore the supply voltage can be dropped for a fixed memory throughput. If the serial implementation runs at a supply of 3 V to meet a given throughput, then the parallel version can run at a supply voltage of 1.3 V while meeting

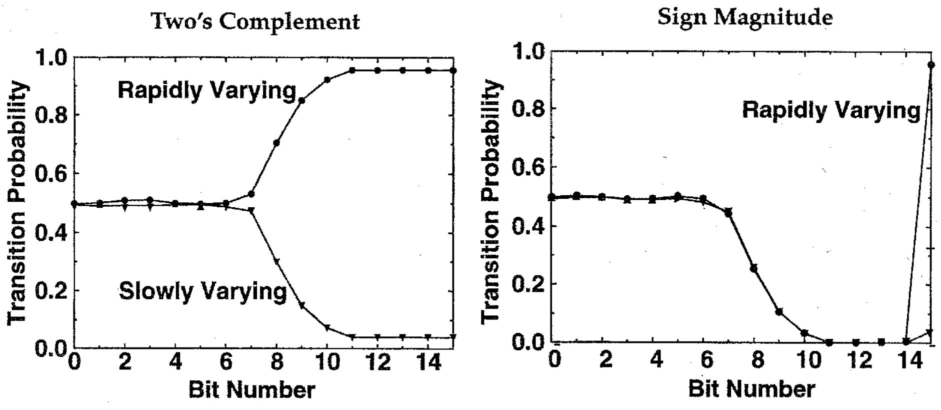


Fig. 15. Activity reduction using sign-magnitude representation.

throughput requirements (based on the delay versus V_{dd} for the SRAM in [4]).

In general purpose computation, data and instructions are not always accessed in a sequential fashion and therefore the above scheme of lowering voltage will result in lower memory throughput. However, parallel memory access can also be used to reduce, at a fixed voltage, the amount of capacitance that is switched when sequential memory locations are read. Since voltage is fixed, nonsequential reads pay no penalty. An example is the approach presented in [19] which uses parallel word access to reduce power in the sense amplifiers and the control circuitry. Four words are read from memory in a differential fashion and a 4:1 multiplexor (which also produces differential outputs) is used to select one of the words to be fed into a sense-amplifier. If the next word that is to be accessed from memory is sequential, then the memory is not precharged and evaluated; instead, the output is taken directly by switching the 4:1 multiplexor. In this case, the sense-amplifier is not needed since the memory output settles out in the first cycle and therefore the sense-amplifier is not enabled (which reduces power consumption). Also, the control overhead of a memory access (block select, address transitions etc.) is reduced. This cache architecture has also been studied under the names *Block Buffered Cache* and *Subblock Buffered Cache* by [15] who report power savings of 40–50% over nonbuffered cache designs for typical cache sizes (and as high as $\times 8$ savings for large cache arrays).

In conclusion, not only do our architecture techniques result in significant power reduction, but the underlying concepts of predictively shutdown, parallelism, and switching activity reduction are also applicable to power reduction in other parts of the system such as the storage (memory and disk) hierarchy.

References

[1] B. Nadel, "The green machine," *PC Mag.*, vol. 12, no. 10, p. 110, May 25, 1993.

[2] R. W. Brodersen, A. Chandrakasan, and S. Sheng, "Technologies for personal communications," in *Proc. VLSI Circuits Symp.*, Japan, 1991.

[3] A. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE J. Solid-State Circuits*, Apr. 1992.

[4] A. Chandrakasan, A. Burstein, and R. W. Brodersen, "A low-power chipset for portable multimedia applications," in *Proc. ISSCC*, Feb. 1994, pp. 82–83.

[5] M. Weiser, "Some computer science issues in ubiquitous computing," *Commun. ACM*, vol. 36, no. 7, pp. 75–85, July 1993.

[6] G. H. Forman and J. Zahorjan, "The challenges of mobile computing," *IEEE Comput.*, Apr. 1994.

[7] J. S. Eager, "Advances in rechargeable batteries spark product innovation," in *Proc. 1992 Silicon Valley Comput. Conf.*, Santa Clara, CA, Aug. 1992, pp. 243–253.

[8] Apple Computer Inc., "Power manager IC, and reduced power modes," in *Technical Introduction to the Macintosh Family*. Reading, MA: Addison-Wesley, Oct. 1992, ch. 20, 2nd ed.

[9] B. I. Berkoff, *Taking Charge: Powerbook-Battery Management*, MacUser, Feb. 1993.

[10] J. T. Brady, "A theory of productivity in the creative process," *IEEE CG&A*, May 1986.

[11] M. Butler *et al.*, "Single stream parallelism is greater than two," in *Proc. 18th Int. Symp. Comput. Architecture*, May, 1991.

[12] H. S. Stone, "Multiprocessor performance," in *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1987, ch. 6.

[13] P. D. Hoang, "Compiling real-time digital signal processing applications onto multiprocessor systems," Ph.D. dissertation, EECS Dep., Univ. California, Berkeley, June 1992.

[14] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. 25th Annu. Int. Symp. Microarchitecture*, Nov. 1992.

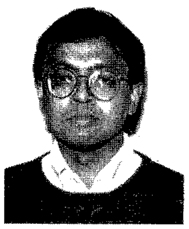
[15] J. Bunda, D. Fussell, and W. C. Athas, "Evaluating power implications of CMOS microprocessor design decisions," in *Int. Workshop on Low Power Design*, Apr. 1994.

[16] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Low-power architecture design and compilation techniques for high-performance processors," *Digest of Papers, IEEE COMPCON Spring '94*, Mar. 1994.

[17] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the power-hungry disk," in *Proc. 1994 Winter USENIX Conf.*, Jan. 1994, pp. 293–306.

[18] K. Li, R. Kumpf, P. Horton, and T. Anderson, "A quantitative analysis of disk drive power management in portable computers," in *Proc. 1994 Winter USENIX Conf.*, Jan. 1994, pp. 279–291.

[19] M. Muller, "The ARM6: Power efficiency & low cost," *Hot Chips Symp.*, Aug. 1992, pp. 3.3.1–3.3.11.

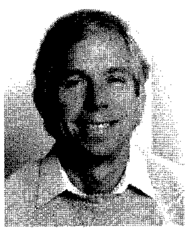


Mani B. Srivastava (S'87–M'92) received the B.Tech. degree from the Indian Institute of Technology, Kanpur, India, and the M.S. and Ph.D. degrees from the University of California, Berkeley.

He is currently a Member of Technical Staff in the Networked Computing Research Department at AT&T Bell Laboratories, Murray Hill, NJ. His primary research interests are in architecture and synthesis of network interfaces, system-level design automation issues such as hardware–software co-design for DSP and embedded systems, and networking issues in wireless computing. He also maintains interests in high-level synthesis for DSP and low-power computing.

Anantha P. Chandrakasan received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley in 1989, 1990, 1994, respectively.

Since September 1994, he has been Assistant Professor of Electrical Engineering at the Massachusetts Institute of Technology, Cambridge. He has been awarded the Analog Devices Career Development Chair. His research interests include the low-power implementation of custom and programmable digital signal processors, low-voltage circuit design, design of wireless systems, and computer-aided design tools for VLSI.



Robert W. Brodersen (M'76–SM'81–F'82) received the B.S. degrees in electrical engineering and in mathematics from the California State Polytechnic University, Pomona in 1966. He also received the Eng. and M.S. degrees in 1968 and the Ph.D. degree in 1972 from the Massachusetts Institute of Technology, Cambridge.

From 1972 to 1976, he was with the Central Research Laboratory at Texas Instruments, Dallas. In 1976, he joined the Electrical Engineering and Computer Science faculty at the University of California, Berkeley, where he is currently a Professor. In addition to teaching, he has been involved in research inclusive of new applications of integrated circuits, which is now focused in the areas of low-power design and wireless communications.

Dr. Brodersen has won conference best paper awards in Eascon (1973), International Solid State Circuits Conference (1975), and the European Solid State Circuits Conference (1978). He has received the 1979 W. G. Baker award for the outstanding paper of the IEEE Transactions and Journals, the 1985 Best Paper Award in the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN (1985), and the 1992 Best Tutorial Paper of the IEEE Communications Society. In 1978, he was named the outstanding engineering alumnus in California State Polytechnic University. In 1983, he was co-recipient of the IEEE Morris Libermann Award for "Outstanding Contributions to an Emerging Technology." In 1986, he received the Technical Achievement Award from the IEEE Circuits and Systems Society and in 1991 from the IEEE Signal Processing Society. In 1988, he was elected a member of the National Academy of Engineering.