Journal of Circuits, Systems, and Computers
Vol. 16, No. 4 (2007) 1–23
© World Scientific Publishing Company

LOW POWER SYSTEM DESIGN BY COMBINING SOFTWARE PREFETCHING AND DYNAMIC VOLTAGE SCALING

SUMITKUMAR N. PAMNANI* and DEEPAK N. AGARWAL†

Microprocessor Verification, AMD, M/S — 615,
5204 East Ben White Blvd, Austin, TX, 78741, USA
**sumitkumar.pamnani@amd.com*
†deepak.agarwal@amd.com

GANG QU‡ and DONALD YEUNG§

Electrical and Computer Engineering Department
and Institute for Advanced Computer Study,
University of Maryland, College Park, MD, 20742, USA
‡gangqu@eng.umd.edu
§yeung@eng.umd.edu

Received 13 December 2004
Revised 28 October 2005
Accepted 21 September 2006

Performance-enhancement techniques improve CPU speed at the cost of other valuable system resources such as power and energy. Software prefetching is one such technique, tolerating memory latency for high performance. In this article, we quantitatively study this technique's impact on system performance and power/energy consumption. First, we demonstrate that software prefetching achieves an average of 36% performance improvement with 8% additional energy consumption and 69% higher power consumption on six memory-intensive benchmarks. Then we combine software prefetching with a (unrealistic) static voltage scaling technique to show that this performance gain can be converted to an average of 48% energy saving. This suggests that it is promising to build low power systems with techniques traditionally known for performance enhancement. We thus propose a practical on-line profiling based dynamic voltage scaling (DVS) algorithm. The algorithm monitors system's performance and adapts the voltage level accordingly to save energy while maintaining the observed system performance. Our proposed on-line profiling DVS algorithm achieves 38% energy saving without any significant performance loss.

Keywords: Dynamic voltage scaling; software prefetch; energy efficiency; performance.

1. Introduction

Low power and low energy design is important for battery-operated portable systems such as personal computing devices, wireless communication, and imaging

‡Corresponding author.

systems. Interestingly, many such systems are not hungry for performance since they are already "fast enough" to satisfy end-user's desire. For example, the physical limitation of human visual and auditory systems implies that a DVD player does not necessarily need to be implemented using the fastest processor. One natural question is *what will be the role, if any, of traditional high-performance techniques such as pipelining, caches, prefetching, and branch prediction in low power system design?* We believe that the performance gain achieved by these techniques can be converted into power and energy reduction. We propose a general approach that monitors the performance gain and transfers it into power and energy saving at runtime by dynamic voltage scaling (DVS). In this article, we use software prefetching as an example to demonstrate this idea of utilizing high-performance techniques for low power system design.

1.1. *Prefetching*

Prefetching is a latency tolerance technique, which is generally used to overcome the gap between the (short) processor cycle time and the (long) memory access latency. In prefetching, data is brought to a level of the memory hierarchy that is closer to the processor in advance rather than on demand. This will reduce or eliminate the cache miss penalty and improve performance. More important, it has been regarded as a promising approach for solving the memory wall problem.[1]

Prefetches can be triggered either by a hardware mechanism, or by a software instruction, or by a combination of both. While hardware prefetching requires extra hardware resources to determine what to prefetch, *software prefetching* approaches detect data access patterns by static program analysis and allow prefetching to be done selectively and effectively. Several prefetching techniques have been proposed in the past to improve performance.[2–6] However, there has been very little reported on the power and energy overhead associated with prefetching.

1.2. *Dynamic voltage scaling*

Dynamic voltage scaling is a technique that varies the supply voltage and clock frequency based on the computation load to provide desired performance with the minimal amount of energy consumption. The most practical and well-studied DVS system is the multiple voltage DVS system where a set of predefined discrete voltages are available simultaneously due to their simplicity of implementation and effectiveness in power reduction.[7–9] In fact, the International Technology Roadmap for Semiconductors has predicted such systems as the trend of future low power systems.[10]

DVS reduces the operating voltage and thus clock frequency. It saves power and energy at the cost of longer execution time. Therefore, it is applicable only when there exists slack time. That is, the system goes back and forth between "busy" and "idle". When the execution time information is available, there exist optimal algorithms to achieve the maximal energy saving.[8,9,11] Without knowing a task's

real execution time, voltage can also be scaled based on system level information such as current computation load and predicted future behavior.[12–14] Normally, voltage scaling is performed on the arrival and completion of a task, and periodically during task execution.

1.3. *Our approach*

In this article, we investigate the power/energy vs. performance tradeoff of software prefetching on a multiple-voltage system. We first make two observations from simulation: software prefetching is a very power- and energy-consuming technique, but voltage scaling has the potential to balance the software prefetching's performance gain and power/energy overhead. Therefore, we propose a framework that automatically transfers any "excess" performance enhancement (provided by prefetching or any other techniques) to energy reduction. At the heart of this approach is an on-line profiling based algorithm that monitors the system's performance and adjusts the system voltage accordingly to save energy without causing noticeable performance degradation. Although we illustrate this for software prefetching, our approach is general and applicable to other high-performance techniques. In sum, we claim that system designers can and should consider traditional performance-enhancement techniques for low power system design.

The rest of this article is organized as follows: In Sec. 2, we review the previous work on prefetching for high-performance and the low power DVS technique. In Sec. 3, we elaborate on the proposed on-line profiling based DVS algorithm that transfers the performance gain of prefetching automatically to energy saving via careful voltage scaling. We report our experimental methodology in Sec. 4 and evaluate the simulation results in Sec. 5. We conclude in Sec. 6 with discussion on future research directions.

2. Related Work

In this section, we elaborate on the previous work in hardware and software prefetching as well as how slack time is utilized in DVS to reduce power and energy.

2.1. *Prefetching for high-performance*

The hardware prefetching approach detects accesses with regular patterns and issues prefetches at run-time. The main advantages of the hardware-based approach are that prefetches are handled dynamically without compiler intervention and that code compatibility is preserved. However, it suffers from the requirement of extra hardware and also that unwanted data could be prefetched. Jouppi[3] introduced stream buffers to improve direct mapped cache performance. Chen and Baer[2] investigated a mechanism for prefetching data references characterized by regular strides. Mowry and Gupta[6] showed that prefetching can be effective for tolerating large memory latencies in a shared memory multiprocessor environment.

On the other hand, software-directed prefetching approaches rely on data access patterns detected by static program analysis and allow the prefetching to be done selectively and effectively. An intelligent compiler inserts data prefetch instructions so that they execute several cycles before their corresponding memory instructions. These prefetch instructions are explicitly executed by the processor to initiate prefetch requests. Porterfield[15,16] presented a compiler algorithm which prefetched all array references in inner loops one iteration ahead. Klaiber and Levy[17] extended this work by recognizing the need to prefetch more than a single iteration ahead. Gornish *et al.*[18] presented an algorithm for determining the earliest time when it is safe to prefetch an entire subarray. The software scheme has some drawbacks. First, some useful prefetching may not be uncovered so cache misses may not be eliminated completely. Second, there is an execution overhead due to the extra prefetch instructions.

Both hardware-based techniques and software-directed prefetching approaches are successful in identifying prefetches of data with simple constant stride patterns. These techniques have been studied solely for improving performance. Little is known about the power and energy overhead of prefetching.

2.2. *Dynamic voltage scaling for low power*

We have already mentioned that DVS is applicable in the presence of slack time. We now survey the existing efforts on DVS in two categories. The first set of work assumes slack comes from the difference among a task's deadline, worst case execution time, and real execution time. The scheduler must know these information to scale voltage accordingly. The second set of work does not require such application information and scales voltage based on system level information such as current computation load and predicted future behavior.

Yao *et al.*[19] were among the first to study scheduling policies for energy reduction on variable speed CPUs. They built a model for the tasks to be executed, where each task is associated with its arrival time, deadline, and worst case execution time. They discussed how to schedule the tasks and determine the CPU's speed, controlled by varying voltage to complete all tasks with the minimum amount of energy. There have been rich follow-up works addressing similar problems for periodic tasks, aperiodic tasks, or mixed tasks; for priority tasks; for ideal voltage scaling system where there is no overhead for voltage changing; for practical voltage scaling system where there are physical constraints on maximal/minimal voltage and how fast voltage can be changed; etc.[7–9,11,20]

Weiser *et al.*[14] proposed to divide time into intervals of 10–50 ms and adjust the CPU clock speed by the task-level scheduler based on the processor utilization over the preceding interval. Govil *et al.*[12] enhanced this by looking for recurring patterns of processor utilization and setting processor speed accordingly. Pillai and Shin[21] presented a class of algorithms that modify the operating system's real-time scheduler and task management service to save energy while maintaining real-time

deadline guarantee. Most recently, Hua *et al.*[22] proposed the probabilistic design for soft real-time systems, such as multimedia applications, where occasional execution failure can be tolerated. They intentionally dropped tasks to reduce energy while providing a statistical performance guarantee.

The novelty of our approach is that we utilize performance-enhancement techniques (such as software prefetching) to reduce the actual execution time of the task, and thus create slack that may not exist otherwise. Such slack will provide us opportunities to apply DVS in order to reduce both power and energy consumption.

2.3. *Multiple-voltage DVS system*

Most DVS systems mentioned above assume on-chip DC–DC converters to provide continuous and flexible operating voltage at run-time to achieve the maximal power saving.[9,13,19] This inevitably introduces time and energy overhead, most notably the time for voltage to stabilize at the new level. Moreover, such overhead cannot be treated as constant because the *transient response time* and consequently, the energy consumption for voltage switch depend on the difference between the source voltage and the desired voltage.

Unlike these systems, we consider DVS systems with multiple levels of supply voltage and threshold voltage physically implemented on the chip. Such systems are referred to as multiple-voltage DVS systems and can be implemented by using multiple-voltage regulators, each of which is dedicated to generating one specific voltage and clock frequency. This eliminates the use of DC–DC converters or other auxiliary devices (such as buffers, delay line, and/or charge pump) to implement DVS. By doing so, although we sacrifice the ability to generate voltage at any level, we minimize the time and energy overhead during voltage switches.

First, the delay overhead for each voltage switch will be a constant. The operating system controls the clock frequency at run-time by writing to a register in the system control state the same way as in Ref. 13. This takes one clock cycle and there is a fixed transient response time (normally a few cycles as reported in Refs. 24 and 25).

Second, the power dissipation on the voltage regulators will be the sum of one regulator's dynamic power and the static power for other regulators. This is because the system, when active, uses only one regulator at any time and can shut down the rest for energy conservation. Note that recent advances in linear voltage regulator design have led us to low dropout (LDO) regulators with high efficiency, low power, and low transient response time. The LDO regulator's quiescent current is in the order of μA's[25] and its dynamic power can be well below the mW mark. For example, Ref. 24 reports an adaptive voltage scaling controller that consists of a voltage regulator, a charge pump, a resettable delay-line, level shifters, and a clock generator among other devices. This controller consumes 2 mW at 3 V and 20 MHz, with most energy dissipation spent on devices other than the regulator.

Third, using multiple LDO regulators may not cause more area overhead than a DC–DC converter. This is because we do not require the auxiliary devices to produce flexible voltage and, as it has been reported,[26] two or three regulators will be sufficient for energy efficiency for most application-specific embedded systems.

3. Software Prefetching for Energy Reduction

In this section, we first layout our framework of transferring processor's performance improvement to power/energy saving. Then we use software prefetching as a performance-enhancing technique and apply the proposed framework to convert the performance gain from prefetching to energy saving.

3.1. *Bridging performance enhancement and energy reduction*

The continuing push for high performance has brought us to a point where the microprocessor is sufficiently fast to handle specific applications. Such improved performance comes, in general, with larger amount of transistors and higher power consumption. For example, our simulation shows that software prefetching is able to improve the performance of our benchmarks by 36%. However, this takes 8% more energy consumption and the average power consumption is 69% higher. (The power is much higher because with prefetching, more energy is consumed in a shorter period. See Sec. 5.1 for the detailed report.)

This experimental finding clearly suggests that it is important to study the power and energy behavior when evaluating any performance-enhancement technique. A less obvious issue is whether such techniques should be considered at all in the design of low power systems where only moderate performance is required. Consider the same example as above. With the 36% performance gain from prefetching, we are able to slow the system down without suffering any performance loss compared to the system without prefetching. We enable the clock slow down by scaling voltage down and this results in a 48% power and energy saving! (We have the same average power and energy saving because we slow the clock such that we complete the execution at the same time that the system without prefetching would. See Sec. 5.2 for the detailed report.)

This suggests that it is possible to use performance-enhancement techniques for low power system design. We propose a framework that automatically transfers the performance gain to energy and power reduction. Such transfer requires:

- *Balanced performance degradation and energy reduction.* Ideally, we should maintain the user specified performance level and convert all the "excess" performance beyond that level to power and energy saving.
- *Minimum re-design effort.* It should be easy to integrate this transfer in existing systems. This implies that we should not assume information, which requires extensive offline profiling, is known *a priori*.

```
A(N,N,N),B(N,N,N)
do j=2,N-1
  do i=2,N-1
    A(i,j) = 0.25 * (B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1))
```

Fig. 1. Example affine array traversal code from the 2D Jacobi kernel.

In the remainder of this section, we elaborate on this proposed framework by presenting an algorithm that transfers "excess" performance, which is provided by software prefetching, to energy saving by DVS. Our algorithm does not require the unrealistic information such as a task's real execution time, the success ratio and distribution of prefetches, and the performance gain by each successful prefetch.

3.2. *Software prefetching as a performance-enhancing technique*

Software prefetching is a promising technique for overcoming the speed gap between processor and memory. It relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. At run-time, the inserted prefetch instructions bring the data into the processor's cache in advance of its use, thus overlapping the cost of the memory access with useful work in the processor. This section discusses how application code is instrumented with the prefetch instructions.

Several algorithms have been previously proposed to instrument software prefetching.[4,5,23] The best algorithm to apply depends on the type of memory references performed by the application code. The most basic memory reference pattern is affine (linear) accesses to multidimensional arrays such as the 2D Jacobi kernel shown in Fig. 1. This benchmark performs a grid computation in which the array A elements are computed as the average of neighboring values in both dimensions from the array B elements. Prefetches for affine accesses like those in the 2D Jacobi kernel can be instrumented using Mowry's algorithm.[5] Figure 2 illustrates the 2D Jacobi kernel after Mowry's algorithm has been applied.

Mowry's prefetch algorithm involves three steps. First, the affine array references within inner-most loops are identified as prefetching candidates. To permit prefetching of only the missing dynamic instances of identified references, loop unrolling is performed to create multiple static instances of each memory reference within the loop body; the degree of loop unrolling is set to the number of back-to-back memory references co-located in the same cache block. By unrolling the loop this number of iterations, the dynamic instances that miss in the cache are isolated — the leading static memory reference in the unrolled loop always misses, while the remaining unrolled static memory references always hit. Hence, a single prefetch for the leading memory reference can be inserted into the unrolled loop, prefetching exactly the missing dynamic instances, thus avoiding unnecessary prefetches and reducing prefetch overhead.

```
A(N,N,N),B(N,N,N)

do j=2,N-1
  do i=2,N-1                            // Prologue Loop
    prefetch(&B[i][j])
    prefetch(&B[i][j+1])
    prefetch(&B[i][j-1])
    prefetch(&A[i][j])

  do i=2,N-PD-1,step=4                  // Unrolled Loop
    prefetch(&B[i+PD][j])
    prefetch(&B[i+PD][j+1])
    prefetch(&B[i+PD][j-1])
    prefetch(&A[i+PD][j])

    A(i,j) = 0.25 * (B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1))
    A(i+1,j) = 0.25 * (B(i,j) + B(i+2,j) + B(i+1,j-1) + B(i+1,j+1))
    A(i+2,j) = 0.25 * (B(i+1,j) + B(i+3,j) + B(i+2,j-1) + B(i+2,j+1))
    A(i+3,j) = 0.25 * (B(i+2,j) + B(i+4,j) + B(i+3,j-1) + B(i+3,j+1))

  do i=N-PD,N-1                        // Epilogue Loop
    A(i,j) = 0.25 * (B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1))
```

Fig. 2. 2D Jacobi kernel from Fig. 1 instrumented with software prefetches using Mowry's algorithm. The instrumented code contains a prologue loop, an unrolled "steady-state" loop, and an epilogue loop.

Figure 2 illustrates the loop unrolling and prefetch insertion transformations for the 2D Jacobi kernel. In this loop, there are five affine references, four for the B array, and one for the A array. Assuming each reference accesses 8 bytes and assuming a 32-byte cache block, the loop should be unrolled by a factor of 4. After loop unrolling, four prefetches are inserted for the five leading memory references: $A(i, j)$, $B(i-1, j)$, $B(i+1, j)$, $B(i, j-1)$, and $B(i, j+1)$. Notice the $B(i-1, j)$ and $B(i+1, j)$ references lie on the same cache block, thus saving 1 prefetch.

The next step in Mowry's algorithm is to perform *prefetch scheduling*. Given the high memory latency of most modern memory systems, a single loop iteration normally contains insufficient work under which the cost of memory accesses is hidden. To ensure that data arrive in time, the prefetches inserted in the first step of the algorithm must be initiated multiple iterations in advance. The minimum number of loop iterations needed to fully overlap a memory access is known as the *prefetch distance*. Prefetch scheduling determines the prefetch distance and applies it to the inserted prefetches. Assuming the memory latency is l cycles and the work per loop iteration is w cycles, the prefetch distance, PD, is simply $\lceil \frac{l}{w} \rceil$. Figure 2 illustrates the indices of prefetched array elements contain a PD term, providing the early prefetch initiation required.

Finally, the last step is to "fix up" inefficiencies in prefetching created by prefetch scheduling. By modifying prefetches to initiate PD iterations in advance as

illustrated in Fig. 2, the first PD iterations do not get prefetched, and the last PD iterations prefetch past the end of each array. These inefficiencies can be addressed by performing loop peeling to handle the first and last PD iterations of the loop separately. The loop peeling transformation creates a *prologue loop* to execute PD prefetches for the first PD array elements, and an *epilogue loop* to execute the last PD iterations without prefetching. Figure 2 illustrates the prologue and epilogue loops created by loop peeling.

In addition to software prefetching for affine array accesses, in Sec. 5, we also consider software prefetching for indexed array and pointer-chasing accesses. We use the algorithm in Ref. 5 for prefetching indexed array references which is very similar to the algorithm described above except for a small extension to the prefetch scheduling step. For pointer-chasing accesses, we use the prefetch arrays technique.[4] This approach requires creating and managing *prefetch pointers*, in addition to instrumenting software prefetches, to overcome the serialization effects of pointer-chasing memory references. We omit the detailed explanation of these techniques and refer the reader to the cited papers for more information.

3.3. *Transferring performance gain to energy saving via DVS*

Let t' and t be the time to run an application with and without prefetching, respectively. When prefetching is successful, the system will complete the application $t-t'$ earlier than required. That is, prefetching improves performance and creates slack. This slack allows the system to slow down (to conserve power and energy) while still managing to complete at time t. The ratio $\frac{t-t'}{t}$ measures the performance gain and now we discuss how to transfer such gain into energy saving.

The optimal way for voltage scaling to reduce energy is by switching the voltage level from v to v' such that the gate delay, which is proportional to $\frac{v_{dd}}{(v_{dd}-v_{\mathrm{th}})^2}$, will be increased by a factor of $\frac{t}{t'}$. In this case, the application will finish at time t if the system applies prefetching and operates at the constant voltage v'. If, however, the system can only operate at certain pre-designed voltage levels (such a system is referred to as *a multiple-voltage system*) that do not include v', then the most energy-efficient way is to run at the voltage v_1 which is slightly higher than v' for some time and then reduce to the next lower level v_2 (where $v_1 > v' > v_2$) such that the execution terminates at t.[8,9] Note that power dissipation is approximately proportional to $Cv_{dd}^2 f$, where C is the capacitance and f is the clock frequency. The energy consumption will be reduced because the application is operated at lower voltages and frequencies.

Unfortunately, the transfer from performance gain to energy saving is not so simple and straightforward. First, for most real-time applications, we do not have the luxury to know both t and t' and therefore will be unable to determine the optimal voltage scaling strategy. Off-line profiling will not provide a complete solution because the success of prefetching depends on the application's run-time behavior and the performance gain from each successful prefetch may be different and is,

in general, unpredictable. Second, even if we know both t and t', each prefetch is independent and successful prefetches may not be uniformly distributed throughout the application's execution. Consider that prefetching helps to improve the performance of a multimedia application (e.g., by reducing the frame decoding time) and assume that most of the successful prefetches occur in the second half of the application. The above voltage scaling strategy suggests to run at a lower voltage. However, most of the prefetchings fail in the first half of the execution and we will constantly fall behind the execution at the reference voltage without prefetches. Although (successful) prefetchings will eventually help us to catch up (by completing the multimedia application at the same time as the run at reference voltage), the slow down in the first half may cause unacceptable user-perceivable performance degradation.

We propose an on-line profiling DVS algorithm that automatically selects voltage at run-time based on the accumulated performance gain to reduce energy and provide performance guarantees (if prefetching does not have any negative impact on system performance. See Theorem 1 and its proof for detail). Figure 3 depicts our algorithm.

We periodically conduct real-time profiling to estimate the performance gain of prefetching. For each N instructions, we execute the first M instructions with prefetching and the next M instructions without prefetching (which can be easily controlled by executing prefetch instructions as NOPs). We then compute c_p and c_{np}, the cycles for these two sets of instructions as shown in steps 2–5. Next, we calculate the prefetching gain in step 6. If there is no gain due to prefetching, then we pull up to the reference voltage for execution. This is because prefetching has

```
1. repeat each cycle for each N instructions {
2.    for M instructions
3.       compute c_p, number of cycles to execute M instructions with prefetching;
4.    for M instructions
5.       compute c_np, number of cycles to execute M instructions without prefetching;
6.    gain = (c_np − c_p) / M;
7.    if (gain < 0)
8.       pull to the reference voltage and execute W instructions;
9.       goto step 2;
10.   repeat each cycle for N - 2M instructions {
11.      compute present_saving;
12.      if (present_saving ≥ c_th)      voltage_down();
13.      if (present_saving < 0) {
14.         if (present_saving + cumulative_saving ≥ 2c_th)      voltage_down();
15.         if (present_saving + cumulative_saving < c_th)       voltage_up();
16.      }
17.   }
18. }
```

Fig. 3. Pseudo-code for real-time profiling DVS algorithm.

not been helpful and executing at any voltage lower than reference would result in performance loss. After W instructions, we start profiling again (step 9).

If prefetching does provide gain, we repeat steps 10–17 for the next $N - 2M$ instructions. We compute $present_{savings}$, which keeps track of how far ahead we are as compared to execution at reference voltage without prefetching. It can be quantitatively defined as the difference between prefetching gain and slow down of running at lower voltage. We then scale voltage up and down by instructions *voltage_up*() and *voltage_down*() based on both the current and accumulated performance gain. The algorithm is designed in a manner such that *voltage_up*() is called only when the accumulated performance saving is below a pre-determined threshold c_{th} (step 15).

We use the following parameter values for this algorithm in our simulation:

(1) N : 100 K instructions (how often do we profile).
(2) M : 5 K instructions (profile period).
(3) c_{th} : 50 μs (threshold against which the savings are compared).
(4) W : 5 K instructions (waiting time before re-profiling).

The setting of these parameters is application dependent and directly affects the performance of the algorithm. In order to guarantee no loss of performance, the values of N, M, and c_{th} have to be selected carefully. Although there is no systematic way to determine the optimal values of these parameters to achieve the most energy saving, they can be set rather precisely for application-specific systems by studying the bahavior of the application. For general purpose systems, we can use the following basic guidelines and as we will show in the experimental evaluation section, these guidelines lead to the above parameter settings that give significant energy saving for a variety of data-intensive applications.

First, the profiling period M needs to be sufficiently long to distinguish the system performance with and without prefetching. On the other hand, our basic assumption is that prefetching will improve performance and we will use prefetching whenever possible. This implies that M should be relatively small compared to N so only a small portion of the instructions will be executed without prefetching (in step 3). Next, when we execute with prefetching for the $N - 2M$ instructions (steps 10–17), we are conservative in converting the performance saving to energy saving with the use of threshold c_{th}. The selection of c_{th} is based on the setting of the voltages. The larger the gap is between two voltage levels, the larger c_{th} is.

Finally, we mention that the proposed method is only applicable when the prefetching technique is effective in giving performance gain. This includes many real life applications that have regular behavior as we will elaborate in the next section. If the application has irregular behavior, for example, when its computation loads in the first M instructions (when we do not prefetch), in the second M instructions (when we do prefetch), and the next $N - 2M$ instructions are dramatically different, our estimated performance gain will not be accurate and our algorithm may fail to provide the performance guarantee. This is because when we re-evaluate

1 the performance gain from executing the $N-2M$ instructions with prefetching, we use the performance of the first M instructions without prefetching as the baseline.
3 In such case, varying the setting of above parameters can help.

To summarize this section, we give two theorems about the low power and
5 performance-guarantee features of the proposed algorithm.

Theorem 1. *Compared to the execution at the reference voltage without*
7 *prefetching, the proposed algorithm guarantees that there is no performance degradation due to voltage scaling.*

9 **Proof.** We start with prefetching at the reference voltage (steps 1–3). If we assume that prefetching does not have any negative impact on system performance, we will
11 not fall behind the non-prefetching execution for the first M instructions. The next M instructions will be executed without prefetching, so we maintain our perfor-
13 mance gain, if any, from the successful prefetches in the first M instructions. For the rest of the execution, when we call *voltage_down*(), the system's speed will slow
15 down and we will lose the cycle savings that we have accumulated. However, when we set the parameters (profiling frequency N, period M, and threshold c_{th}) care-
17 fully such that the *voltage_up*() calls will pull up voltage back to reference before we consume all the performance gain by prefetching. □

19 **Theorem 2.** *On multiple-voltage system, the proposed algorithm converges to the optimal voltage setting.*

21 **Proof.** For multiple-voltage system, if the optimal v_{opt} is available, the optimal voltage scaling strategy is to run at v_{opt}; otherwise, the system will run only at
23 the two voltages v_1 and v_2 such that $v_1 > v_{\text{opt}} > v_2$ and there is no other voltage available between v_1 and v_2. Our algorithm starts with the reference voltage and
25 reduces voltage gradually via *voltage_down*() calls. If the prefetching performance gain is estimated accurately, the $present_{saving}$ computed in step 11 will become
27 negative when the current voltage is below v_{opt}. However, our algorithm may still lower voltage if the accumulated performance saving is high and will not raise
29 voltage until the performance saving is below the threshold c_{th}. The $present_{saving}$ starts to be positive when the voltage level passes v_{opt} and reaches v_1. Now, our
31 algorithm will not raise voltage any higher (one condition for *voltage_up*() call in step 13 is false). Once the accumulated performance saving exceeds c_{th}, the voltage
33 goes down to v_2 (step 12). At this level, we are running slower than v_{opt} (i.e., $present_{saving} < 0$), so the voltage will not go below v_2 (step 14). If the accumulated
35 performance gain is below c_{th}, we will raise voltage back to v_1 (step 15). This implies that we have reached the optimal voltage setting: toggling between the two voltages
37 v_1 and v_2 next to v_{opt}. This claim is also confirmed by a couple of applications in our simulation. □

4. Experimental Methodology

We use software prefetching to improve application performance. All the benchmarks used are instrumented with software prefetching by hand, following the algorithms discussed in Sec. 3.2. The performance of these optimized codes was then measured on a detailed architectural simulator. The performance boost achieved was later traded for power savings by voltage scaling.

4.1. *Application overview*

Our experimental evaluation employs six benchmarks, representing three classes of data-intensive applications. Table 1 lists the benchmarks along with their problem sizes and memory access patterns. The first two applications perform affine array accesses. MatMult multiplies two matrices and Jacobi performs a 3D Jacobi relaxation, which is frequently found in multigrid PDE solvers such as MGrid from the SPEC/NAS benchmark suite. The next three applications perform indexed array accesses. Irreg is an iterative PDE solver for an irregular mesh. Moldyn is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems. NBF is abstracted from the GROMOS molecular dynamics code. Finally, the last application, Health, performs pointer-chasing accesses. It simulates the Columbian health care system and is taken from the olden benchmark suite.[27]

4.2. *Simulation environment*

We use Wattch, an architecture level power analysis tool,[28] in our simulation. Wattch provides a framework for analyzing and optimizing microprocessor power dissipation and is an order of magnitude faster than the existing layout-level power tools, and at the same time maintains accuracy within 10%. Wattch models main processor units as one of the following four structures: array structures, fully associative content-addressable memories, combinational logic and wires and clocking.

The power consumption of the units modeled very much depends on their specific implementation, particularly on the internal capacitances for the circuits that make up the processor. Wattch models the internal capacitance using assumptions similar to those made by Wilton and Jouppi[29] and Palacharla *et al.*[30] Current CPU

Table 1. Description of the benchmark applications.

Application	Problem Size	Access Pattern
MATMULT	200×200 matrices	Affine array
JACOBI	$200 \times 200 \times 8$ grid	Affine array
IRREG	14 K node mesh	Indexed array
MOLDYN	13 K molecules	Indexed array
NBF	14 K node mesh	Indexed array
HEALTH	5 levels, 500 iters	Pointer-chasing

designs increasingly use conditional clocking to disable all or part of a hardware unit to reduce power consumption when it is not needed. We use Conditional clocking-3 as described in Ref. 28 for our simulation. In this clock gating, power is scaled linearly with port or unit usage; unused units dissipate 10% of their maximum power.

The Wattch power model is interfaced with Simplescalar sim-outorder,[31] which is used to keep track of different units accessed per cycle and hence record the total energy consumed for a given application. Sim-outorder is a detailed simulator supporting out-of-order issue and execution based on Tomasulo's approach for dynamic instruction scheduling. The processor's memory system is simple and does not accurately model DRAMs or the memory bus contention. We have replaced this by a cycle accurate memory controller and DRAM memory sub-system. Each L2 request to the memory controller simulates several actions: queuing of the request in the memory controller, Row Access Strobe (RAS) and Column Access Strobe (CAS) cycles, and data transfer across the memory system bus. We simulate concurrency between DRAM banks, but bank conflicts require back-to-back DRAM accesses to perform serially. Also memory requests are serialized at the memory bus, i.e, we model bus contention, but we assume infinite bandwidth between the L1 and L2 caches. The bottom portion of Table 2 lists the parameters for our baseline memory sub-system model.

The reason for going to a detailed memory model is to get realistic performance numbers for prefetching. Prefetching increases memory activity by issuing memory requests frequently. Thus, modeling memory contention is crucial for accurately simulating prefetching performance. For our power study, we focus on the processor because in most systems, the processor consumes the most energy. This is true for small hand-held devices like PDAs[32] which have very few components, but also

Table 2. Simulation parameters for the processor, cache, and memory sub-system models. Latencies are reported either in processor cycles or in nanoseconds. We assume a 1.25-ns processor cycle time. ([a]latency is for floating operations.)

Processor model: 600 MHz			
Issue width	8	Integer latency	1 cycle
Instruction window size	64	Add/mult/Div latency[a]	2/4/12 cycles
Load-store queue size	32	Branch predictor	gshare
Fetch queue size	32	Branch predictor size	2048 entries
Integer/floating point units	4/4	BTB size	2048 entries
Cache model: $1 cycle = 1.25\,ns$			
L1/L2 Cache size	16K-split/512K-unified	L1/L2 associativity	2/4 cycles
L1/L2 Cache block size	32/64 bytes	L1/L2 Latency	1/10 cycles
L1/L2 MSHRs	8/16	L1/L2 write buffers	8/16
Memory sub-system model			
DRAM banks	32	Row access strobe	22.5 ns
Memory system bus width	64 bytes	Column access strobe	22.5 ns
Address send	7.5 ns	Data transfer (per 8 bytes)	7.5 ns

on large laptop computers[33] that have many components, including large displays with backlighting. As a result, the design problem generally boils down to a trade-off between the computational power of the processor and the system's battery life.

Our baseline simulations assume a SimpleScalar processor model running at 600 MHz and at 1.6 V. For voltage scaling, we use the same five different voltage levels, 1.60 V, 1.50 V, 1.40 V, 1.25 V, and 1.10 V, as those in Transmeta's Crusoe processor. They provide frequencies of 600 MHz, 500 MHz, 400 MHz, 300 MHz, and 200 MHz, respectively.[34] As we have discussed in Sec. 2.3, we assume that the time and power overhead for such multiple-voltage DVS system to switch voltage from one level to another is negligible. In our proposed technique to convert performance gain into power reduction, only a few counters are required for the real-time profiling. Their power dissipation and the execution time of the profiling algorithm are both considered in the simulation.

Finally, we mention that memory has become one of the major energy consumers in modern systems. In the memory hierarchy, the memory's power consumption goes up as its speed goes up (or as it gets closer to the processor). Our simulator models both L1 and L2 cache and hence their energy consumption is included in the simulation. In addition, when the processor slows down due to voltage scaling, the memory speed can also be reduced accordingly without affecting the system performance. Therefore, in our simulation, we scale the voltage for both L1 and L2 caches whenever the processor speed changes. This reduces the energy consumption in the memory hierarchy without affecting the system performance. However, peak power may increase due to heavy memory activity during prefetching. In our experimental results, we focus only on average power or the energy consumption in a given period of time.

5. Experimental Evaluation

This section describes the experimental result and evaluates the performance of software prefetching and power savings achievable through voltage scaling.

5.1. *Prefetching performance*

We begin by evaluating the performance of software prefetching over no prefetching. Figure 4 plots the normalized execution time along y-axis for the prefetching and no-prefetching versions of different applications. The detailed execution time information is also reported in the second and third columns of Table 3.

Each execution-time bar has been broken down into three components: useful computation, prefetch-related software overhead, and memory stall, labeled "Busy," "Overhead," and "Mem," respectively. "Busy" is the execution time of the "NP" version assuming a perfect memory system (e.g., all memory accesses complete in one cycle). "Overhead" is the incremental increase in execution time of the "P" version over "Busy," again on a perfect memory system. "Mem" is the

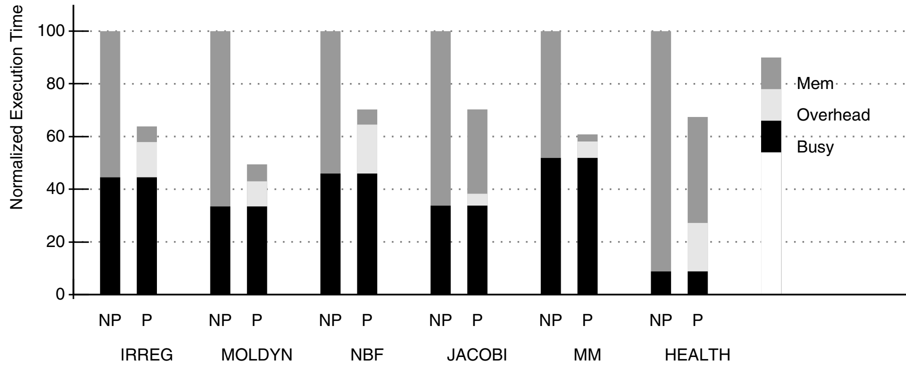

Fig. 4. Execution time breakdown for annotated memory instructions. Individual bars show performance without prefetching labeled "NP" and with prefetching labeled "P".

Table 3. Simulation results on static voltage scaling. The numbers in bold indicate the voltage setting such that there is no significant increase in the execution time.

	NP(1.6 V)		P(1.6 V)		P(1.5 V)		P(1.4 V)		P(1.25 V)		P(1.1 V)	
Application	ms	Energy	ms	Energy	ms	Energy	ms	Energy	ms	Energy	ms	Energy
IRREG	22.38	8.45	14.29	9.09	17.20	6.60	**21.65**	**4.59**	28.58	2.80	43.30	1.49
MOLDYN	22.23	7.36	10.99	6.50	13.24	4.70	16.65	3.29	**21.98**	**2.01**	33.30	1.07
NBF	13.21	5.46	09.28	6.45	11.10	4.69	**14.07**	**3.25**	18.57	1.99	28.14	1.05
JACOBI	14.04	4.99	09.87	4.96	11.89	3.60	**14.96**	**2.49**	19.75	1.51	29.90	0.80
MATMULT	86.74	40.5	52.75	39.4	63.55	28.6	**79.93**	**19.8**	105.5	12.1	159.8	6.40
HEALTH	10.45	2.32	07.05	3.13	08.49	2.27	**10.68**	**1.57**	14.10	9.60	21.36	5.09

incremental increase in execution time over "Busy"+"Overhead" assuming a real memory system. All times are normalized against the "NP" bars.

Comparing NP and P bars we find prefetching enhances the performance of all the applications significantly. For example, in MOLDYN, prefetching reduces the execution time by 50.56%. On an average, software prefetching is able to boost the performance by 36.04%. We now see the system's power and energy consumption with and without prefetching.

In Table 3, the second and the third columns give the execution time and energy consumption for the runs at the same reference 1.6 V voltage without ("NP" column) and with ("P" column) prefetching, respectively. We see that the run with prefetching does not necessarily save energy consumption. We mention that we have used clock gating to turn off hardware units whenever they are idle. There are three sources for prefetching's energy overhead. First, it utilizes the parallelism among different hardware units and therefore keeps them active, dissipating energy. Second, the execution of prefetching instructions consumes energy. Third, unsuccessful prefetches happen, further increasing energy consumption. Only when the performance gain from prefetching is sufficiently high (such as in the case of Moldyn and MatMult), can we observe energy saving. On an average, prefetching requires 7.67% more energy to achieve the above 36.04% speed-up.

Although prefetching may still result in some small energy saving, its average power consumption will be much higher than that of the system running at reference voltage without prefetching. Intuitively, prefetching reduces the execution time but consumes slightly more energy; therefore, the average power (which is the ratio of energy consumption over execution time) will be high. Based on the values reported in Table 3, prefetching consumes 69% more power on average. The reason for this drastic increase is that, as we have mentioned above, more hardware units will be active at the same time due to the prefetching instructions. In the next two sections, we will show how to transfer the performance improvement from prefetching into power and energy saving.

5.2. *Static voltage scaling*

In order to transfer the prefetching gains into power and energy savings, we look into both static and DVS. For static voltage scaling, the no prefetching version (NP) was run at the highest voltage, i.e., 1.6 V, with its corresponding frequency of 600 MHz. The prefetching version (P) was run at each different voltage level with the corresponding frequency while we kept the voltage fixed for each run. Table 3 reports the time (in ms) and energy (in $10e + 7\,\mathrm{W} \cdot cycle$) to complete each application at different voltage levels.

From the last five columns, we see that as we reduce the voltage from 1.6 V to 1.1 V, the system with prefetching requires more time to complete the application, but consumes less energy (and power). For each application, the column in bold identifies the voltage setting for static voltage scaling that achieves the same (or close) performance as NP, the run at reference 1.6 V voltage without prefetching. These are the closest that we can get with the five different settings of (voltage, frequency) pair on the Crusoe processor. Consider the Moldyn application, which has the maximum prefetching gain of 50.56% (see Fig. 4). We are able to reduce the voltage to 1.25 V and still have an execution time 21.98 ms shorter than the reference (22.23 ms). At this low voltage level, the energy consumption drops to $2.01 \times 10^7\,\mathrm{W} \cdot$ cycle from $7.36 \times 10^7\,\mathrm{W} \cdot$ cycle, a 72.69% saving. The average power reduction is about the same because we have kept the execution time close (21.98 ms and 22.23 ms). For the six applications, on an average, prefetching with static voltage scaling converts the 36.04% performance gain to 48.72% energy and average power saving. The power saving and performance change for each application are reported in the last two columns of Table 4.

The significance of this static voltage scaling experiment is that it shows the potential of combining voltage scaling and prefetching techniques to reduce system's power and energy consumption. In fact, due to the quadratic relationship between energy consumption and voltage, one can predict that the energy saving converted from "excess" performance by slowing the system down is more than linear. For instance, on the above six benchmarks, 36.04% speed-up becomes 48.72% energy saving. This promising energy and power reduction without performance loss is the incentive behind our study of the on-line DVS algorithm.

Table 4. Simulation results on on-line profiling based DVS. The third column indicates the performance loss (if negative) or gain in terms of the execution time change. The last two columns give the comparison with the power saving by static voltage scaling as a reference.

Application	DVS-Power Savings %	DVS-Gain in Performance %	SVS-Power Savings %	SVS-Gain in in Performance %
IRREG	39.2	+1.30	45.68	+3.26
MOLDYN	65.0	−1.00	72.69	+1.12
NBF	8.00	+13.0	40.00	−6.51
JACOBI	38.5	−1.57	50.01	−6.55
MATMULT	52.2	+3.00	53.11	+7.85
HEALTH	25.8	−2.70	32.32	−2.20

5.3. *Dynamic voltage scaling*

Although static voltage scaling can give excellent results, it requires the hardware to be designed with the right voltage to run the particular application. This requirement of profiling means that the application is to be run at different voltage levels and the optimum voltage level determined and stored somehow for each application. For a general purpose environment, this is impractical. We propose the on-line profiling DVS algorithm (Fig. 3) to solve this problem.

Table 4 gives the power savings and performance change for different applications using the proposed on-line profiling DVS algorithm. Both are compared with the system running at reference voltage without prefetching. On an average across the six applications, DVS has a 38.11% power savings and is 2.00% faster. Ideally, we want to convert all the performance gains to energy and power saving. This may not be possible due to the discrete nature of the five available voltages and frequencies of the baseline Crusoe processor. In addition, in three of the applications (MOLDYN, JACOBI, and HEALTH), we have a slight performance loss (1.0%–2.7%). This is caused by the irregular behavior of these applications as we have discussed in Sec. 3.3 before Theorem 1. There is also a 13.0% performance gain in NBF that our algorithm fails to convert to further energy reduction. This is mainly because of the voltage setting. A detailed explanation of these results is given next when we analyze the voltage profile of each application. Finally, we mention that compared with the power saving potential provided by the impractical static voltage scaling (the fourth column in Table 4), we see that the proposed on-line profiling based DVS algorithm is effective. In fact, the power saving by DVS is within 11% on average to the saving by static voltage scaling. This saving is impressive particularly when one considers the fact that the energy and execution time overhead for on-line profiling and voltage adjustment has been included in our simulation.

Figure 5 gives the operating voltage level throughout a complete run of each application following the on-line profiling DVS algorithm. It reveals some interesting insights of the proposed algorithm and the application's run-time behavior.

As claimed earlier, we expect the DVS algorithm to reach the optimal voltage level such that the modified code with prefetching will not run slower than the original code without prefetching. This is possible only when there exists a voltage level at which the system can slow down to offset completely the prefetching gain. Moldyn experiences this steady-state behavior. Table 3 indicates that the completion time of Moldyn at 1.25 V (21.98 ms) is very close to that in the non-prefetching version (22.23 ms). Therefore, after some initial toggling, DVS algorithm finds 1.25 V as the optimal voltage and stays there. The small performance gain is canceled by the prefetching overhead.

However, none of the other applications have the same "steady-state" behavior. The main reason is that their required optimal voltage levels lie between the available "discrete voltage levels." This leads to the toggling behavior as one can see in Irreg and MatMult (see Fig. 5). For example, Table 3 suggests that the optimal voltages for Irreg and MatMult should lie between 1.4 and 1.25 V.

Another reason for the toggling behavior is the dynamic behavior of an application, like that of Health. Compared to the earlier regular applications (Irreg, Moldyn, and MatMult) that execute the same loop over and over again, Health has irregular behavior, spending time in lots of function calls and different loops with different prefetching gains. Different prefetching gains lead to a more dynamic behavior from our DVS algorithm's standpoint. From Fig. 5, it is clear that in the first half of the execution of the Health application, our algorithm tries to stabilize at a voltage level between 1.5 V and 1.4 V, while in the second half, it goes down to

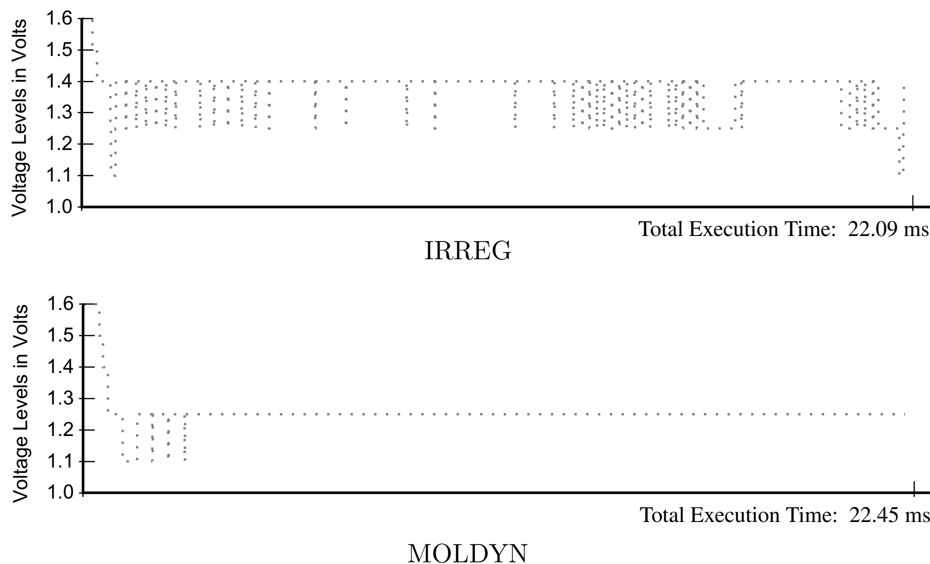

Fig. 5. DVS for different applications, with voltage levels in volts on x-axis and time in ms along the y-axis.

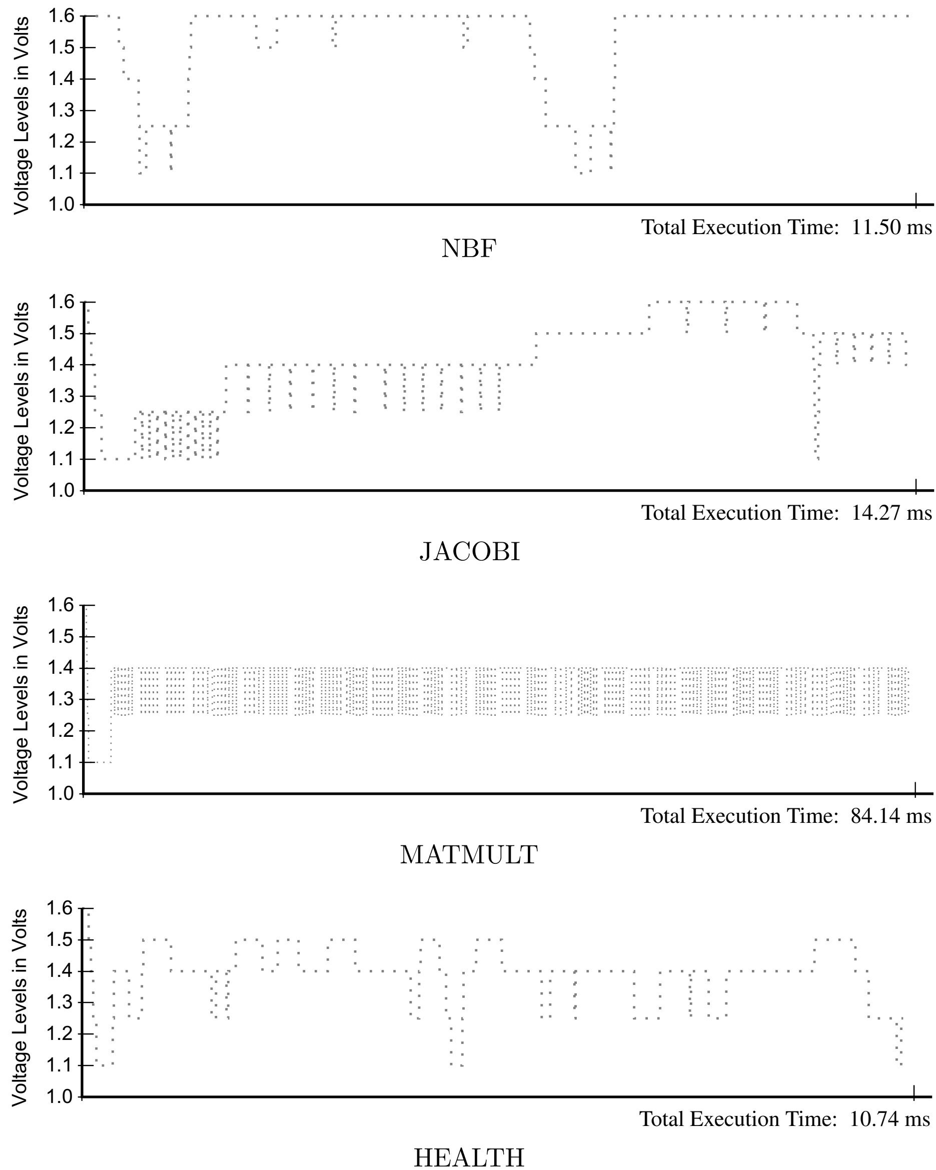

Fig. 5. (*Continued*)

between 1.4 V and 1.25 V. We find the main reason is that prefetching has a higher performance gain in the second half.

Similarly, the application Jacobi has a few different "for loops" in the source code. Our algorithm is able to track the optimum voltage for each "for loop" (by toggling about it) as the application behavior changes. Finally, NBF has anomalous behavior which we are currently investigating.

6. Conclusions and Future Work

We propose a low power system design method that couples DVS and performance-enhancement techniques to simultaneously reduce energy consumption and provide performance guarantees. Specifically, our approach exploits the quadratic power savings of voltage scaling, using the "excess" performance provided by software prefetching to absorb the increased gate delays resulting from voltage scaling. We present an on-line profiling based DVS algorithm that periodically measures the performance gain, and automatically adapts the voltage level to convert such gain to energy reduction. The proposed algorithm is general and applicable to other performance-enhancement techniques.

Our simulation results show that this approach is promising in building low power systems. On six memory-intensive applications, software prefetching provides a 36% performance boost on average. Using the impractical static voltage scaling, this gain has the potential to be converted to 48% average power reduction while still maintaining the same level of performance as the system without software prefetching. Furthermore, we show that our practical on-line DVS algorithm achieves similar energy reduction as the static scheme (within 11% on average) while maintaining approximately the same performance level as the original system (within 6%).

Based on this encouraging result, we believe that the idea of coupling DVS with performance-enhancement techniques is attractive for simultaneously reducing energy consumption and minimizing performance impact in low power systems, and deserves further investigation. Important future work includes coupling DVS with other performance-enhancement techniques, improving our on-line profiling based DVS algorithm to provide further power/energy savings and better performance guarantees, and studying in-depth the cache memory's energy consumption when the processor is slowed down.

References

1. W. Wulf and S. McKee, Hitting the memory wall: Implications of the obvious. *Computer Archit. News*, **23**(1) (1995) 20–24.
2. T. Chen and J. Baer, Effective hardware-based data prefetching for high-performance processors, *Trans. Comput.* **44**(5) (1995) 609–623.
3. N. P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in *Proc. 17th Ann. Int. Symp. Comput. Archit.* pp. 364–373, Seattle, WA (ACM, 1990).
4. M. Karlsson, F. Dahlgren and P. Stenstrom, A prefetching technique for irregular accesses to linked data structures, in *Proc. 6th Int. Conf. High Performance Comput. Archit.* Toulouse, France, January 2000.
5. T. Mowry, Tolerating latency in multiprocessors through compiler-inserted prefetching, *Trans. Comput. Syst.* **16**(1) (1998) 55–92.
6. T. Mowry and A. Gupta, Tolerating latency through software-controlled prefetching in shared-memory multiprocessors, *J. Parallel Distributed Comput.* **12**(2) (1991) 87–106.

7. J. Chang and M. Pedram, Energy minimization using multiple supply voltages, *IEEE Trans. Very Large Scale Integration Syst.* **5**(4) (1997) 436–443.
8. T. Ishihara and H. Yasuura, Voltage scheduling problem for dynamically variable voltage processors, *ISLPED'98: Int. Symp. Low Power Electron. Des.* August 1998, pp. 197–202.
9. G. Qu, What is the limit of energy saving by dynamic voltage scaling? *IEEE/ACM Int. Conf. Computer-Aided Des.* November 2001, pp. 560–563.
10. http://public.itrs.net, 2001.
11. C. Chen and M. Sarrafzadeh, Probably good algorithm for low power consumption with dual supply voltages, *IEEE/ACM Int. Conf. Computer Aided Des.* November 1999, pp. 76–79.
12. K. Govil, E. Chan and H. Wasserman, Comparing algorithms for dynamic speed-setting of a low-power CPU, *ACM Int. Conf. Mobile Comput. Network.* November 1995, pp. 13–25.
13. T. Pering, T. D. Burd and R. W. Brodersen, Voltage scheduling in the IpARM microprocessor system, *ISLPED'00: Int. Symp. Low Power Electron. Des.* July 2000, pp. 96–101.
14. M. Weiser, B. Welch, A. Demers and S. Shenker, Scheduling for reduced CPU energy, *USENIX Symp. Operating Syst. Des. Implemen.* November 1994, pp. 13–23.
15. D. Callahan, K. Kennedy and A. Porterfield Software methods for improvement of cache performance on supercomputer applications, *PhD* thesis, Department of Computer Science, Rice University (May 1989).
16. D. Callahan, K. Kennedy and A. Porterfield, Software prefetching, in *Proc. 4th Int. Conf. Archit. Support Program. Languages Operating Syst.* April 1991, pp. 40–52.
17. A. C. Klaiber and H. M. Levy, An architecture for software-controlled data prefetching, in *Proc. 18th Int. Symp. Comput. Archit.* Toronto, Canada (ACM, 1991), pp. 43–53.
18. E. Gornish, E. Granston and A. Veidenbaum, Compiler-directed data prefetching in multiprocessors with memory hierarchies, in *Proc. Int. Conf. Supercomputing* (ACM, 1990).
19. F. Yao, A. Demers and S. Shenker, A scheduling model for reduced CPU energy, *FOCS'95: IEEE Ann. Foundations Comput. Sci.* 1995, pp. 374–382.
20. G. Quan and X. Hu, Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors, *Des. Automat. Conf.* 2001, pp. 828–833.
21. P. Pillai and G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, *Proc. 18th ACM Symp. Operating Syst. Principles* 2001, pp. 89–102.
22. S. Hua, G. Qu and S. S. Bhattacharyya, Energy reduction techniques for multimedia applications with tolerance to deadline misses, *Design Automation Conf.* 2003, pp. 131–136.
23. C.-K. Luk and T. C. Mowry, Compiler-based prefetching for recursive data structures, in *Proc. Seventh Int. Conf. Archit. Support for Programming Languages Operating Syst.* Cambridge, MA (ACM, 1996), pp. 222–233.
24. S. Dhar, D. Maksimovic and B. Kranzen, Closed-loop adaptive voltage scaling controller for standardcell ASICs, *Int. Symp. Low Power Electron. Des.* 2002, pp. 103–107.
25. C. Simpson, Linear and switching voltage regulator fundamentals, National Semiconductor, www.national.com/appinfo/power/files/f4.pdf.
26. S. Hua and G. Qu, Voltage set-up problem for embedded systems with multiple voltages, in *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* **13**(7) (2005) 869–872.

27. M. C. Carlisle and A. Rogers, Software caching and computation migration in olden, in *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, July 1995, pp. 29–38.
28. D. Brooks, V. Tiwari and M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in *ISCA*, 2000, pp. 83–94.
29. S. J. E. Wilton and N. P. Jouppi, An enhanced access and cycle time model for on-chip caches, *WRL Research Report*, June 1994.
30. S. Palacharla, N. P. Jouppi and J. E. Smith, Complexity-effective superscalar processors, in *ISCA*, 1997, pp. 206–218.
31. D. Burger and T. M. Austin, The SimpleScalar tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.
32. C. S. Ellis, The case for higher-level power management, in *Proc. 7th IEEE Workshop Hot Topics Operating Syst.* 1999, pp. 162–167.
33. J. Lorch and A. J. Smith, Apple Macintosh's energy consumption, *Micro* **18**(6) (1998) 54–63.
34. Transmeta-corporation. Tm5400 processor specifications.