TRANSFERRING PERFORMANCE GAIN FROM SOFTWARE PREFETCHING TO ENERGY REDUCTION

Deepak N. Agarwal*, Sumitkumar N. Pamnani*, Gang Qu, and Donald Yeung
Electrical and Computer Engineering Department and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742 USA
* Microprocessor Verification, AMD, M/S - 615, 5204 East Ben White Blvd Austin, TX 78741 USA

Abstract

Performance-enhancement techniques improve CPU speed, but at higher cost to other valuable system resources such as power and energy. We study this trade-off using *software prefetching* as the system performance-enhancement technique. We first demonstrate software prefetching provides an average 36% performance boost with 8% more energy consumption and 69% higher power on six memory-intensive benchmarks. However, when we combine prefetching with a (unrealistic) static voltage scaling technique, the performance gain afforded by prefetching can be traded off for savings in power/energy consumption. In particular, we observe a 48% energy saving when we slow down the system with prefetching so as to match the performance of the system without prefetching. This suggests a promising approach to build low power systems by transforming traditional performance-enhancement techniques into low power methods. We thus propose a real time dynamic voltage scaling (DVS) algorithm that monitors a system's performance and adapts the voltage level accordingly while maintaining the observed system performance. Our dynamic DVS algorithm achieves a 38% energy saving without any performance loss on our benchmark suite.

1 Introduction

Low power and low energy design is important for battery-operated portable systems such as personal computing devices, wireless communication and imaging systems. Interestingly, many such systems are not hungry for performance since they are already "fast enough" to satisfy end user's desire (e.g. the physical limitation of human visual and auditory systems). 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.* In this paper, we propose a generic approach that first improves system's performance and then transfers the performance enhancement to power/energy saving. We illustrate this by combining software prefetching and dynamic voltage scaling.

Prefetching is a latency tolerance technique, which is generally used to overcome the gap between processor cycle time and memory access latency. In prefetching, cache miss penalty is eliminated by generating prefetch requests to the memory system to bring the data into the cache before the data value is actually used. It can be triggered either by a hardware mechanism, or by a software instruction or by a combination of both. Several prefetching techniques have been proposed in the past solely to increase performance [3, 4, 5, 6, 7]. While hardware prefetching needs extra hardware resources, *software prefetching* approaches rely on data access patterns detected by static program analysis and allow the prefetching to be done selectively and effectively.

Dynamic voltage scaling (DVS) 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. We focus our discussion on the multiple voltage DVS systems where a set of predefined discrete voltages are available simultaneously due to their simplicity of implementation and effectiveness in power reduction. When the execution time information is available, there exist optimal algorithms to achieve the maximal energy saving on such multiple voltage system [10, 12]. Without knowing task's real execution time, voltage can also be scaled based on system level information such as current computation load and predicted future behavior [9, 11, 13].

We investigate the power/energy vs. performance tradeoff of software prefetching in this paper. In particular, we consider a multiple-voltage system that executes software programs and propose a framework that automatically transfers any "excess" performance enhancement (provided by prefetching) to energy reduction. At the heart of this approach is an on-line algorithm that monitors the system's performance and adjust the system voltage accordingly to save energy without causing noticeable performance degradation. Although we illustrate this by software prefetching and DVS, our approach is generic and applicable to any combination of high-performance and low-power techniques. In this sense, system designers can and should consider traditional performance-enhancement techniques when designing low power systems.

2 Software Prefetching for Energy Reduction

Our goal is to build a framework that automatically transfers any performance enhancement to energy reduction. Such transfer requires: (1) **No performance degradation**. We should use only the performance gain from whatever high-performance design technique for power/energy saving. The system's original performance should be maintained, if not enhanced. (2) **Minimum re-design effort**. The effort to integrate this transfer to existing system should be kept at the minimal level. In the remainder of this section, we discuss this in the context of prefetching (for high performance) and dynamic voltage scaling (for low power).

2.1 Software Prefetching

We have already mentioned software prefetching in the introduction. This section illustrates how application code is instrumented with the prefetch instructions. Several algorithms have been proposed for this purpose [5, 6], and their effectiveness depends on the type of memory references performed by the application code.

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

Figure 1: Affine array traversal code from the 2D Jacobi kernel.

Figure 1 illustrates the affine (linear) array traversal, the most common memory reference pattern, code from the 2D Jacobi kernel. Figure 2 illustrates the same code after instrumenting using Mowry's algorithm [6]. In addition to software prefetching for affine array accesses, we also consider applications with software prefetching for indexed array and pointer-chasing accesses in our experiments. We use the algorithm in [6] for prefetching indexed array references and the technique proposed in [5] for pointer-chasing accesses. Due to space limitations, we omit detailed explanation and examples of these techniques and refer the reader to the cited papers for more details.

```
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))
```

Figure 2: 2D Jacobi kernel from Figure 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.

2.2 Transferring Performance Gain to Energy Saving

Let t' and t be the times to run an application with and without prefetching, respectively. The ratio $\frac{t-t'}{t}$ measures the performance gain by prefetching. The optimal way to reduce energy is to use voltage v' such that the gate delay, which is proportional to $\frac{v_{dd}}{(v_{dd}-v_{th})^2}$, will be increased by a factor of $\frac{t}{t'}$. Therefore, the application can still finish at time t with software prefetching. On multiple voltage systems when v' is not available, the most energy-efficient way is to run at a slightly higher voltage v_1 for some time and then reduce the voltage to the next lower level v_2, where $v_1 > v' > v_2$, such that the execution terminates at t [10, 12].

However, the transfer from performance gain to energy saving never comes this easily. First, we may suffer user-perceivable performance loss although we will not delay the completion time of the application. For example, if most successful prefetchings occur in the second half of the execution, then running at a voltage lower than reference (plus the penalty for failed prefetchings) slows down the process and we will constantly fall behind during the first half of the execution. Although (successful) prefetchings will eventually help us to catch up, this slow down may be noticeable and becomes unacceptable. Moreover, this is not practical for real-time applications because selecting the proper voltage scaling strategies requires knowledge about t or t'.

Figure 3 illustrates our online DVS algorithm, which guides the selection of operating voltage to simultaneously achieve low power consumption and performance guarantee. The algorithm periodically conducts real time profiling to estimate the performance gain by prefetching and transfers it to energy reduction by DVS.

For each N instructions, we execute the first M instructions with prefetching and the next M instructions without prefetching. The latter can be achieved by treating prefetch instructions as NOPs. Assuming that they take (c_p) and (c_{np}) cycles respectively, the prefetching gain can be calculated in step 6. If prefetching does not provide us any performance gain, we use the maximum voltage to execute the next W instructions (without prefetching) before we start profiling again (step 9). Note that we turn off prefetching because it does not help and we use the maximum voltage to avoid performance loss.

When we identify performance gain, we keep prefetching on for the rest instructions while scaling the operating voltage (steps

```
1.  repeat for each $N$ instructions till the completion of the application {
2.    execute $M$ instructions with prefetching;
3.    $c_p$ = number of cycles for the execution of these $M$ instructions;
4.    execute the next $M$ instructions without prefetching;
5.    $c_{np}$ = number of cycles for the execution of these $M$ instructions;
6.    $gain = \frac{c_{np}-c_p}{M}$;
7.    if ($gain < 0$)
8.       execute W instructions at the maximum voltage;
9.       goto step 2;
10.   repeat for the rest instructions {                /* profiling done. */
11.     update $present_{saving}$ and $cumulative_{saving}$;
12.     if ($present_{saving} \geq c_{th}$)   voltage_down();
13.     if ($present_{saving} < 0$) {
14.        if ($cumulative_{saving} \geq c_{th}$)    voltage_down();
15.        else voltage_up();
16.     }
17.   }
18. }
```

Figure 3: Pseudo code for the real time profiling DVS algorithm.

10-17). We compute how much ahead we are as compared to no-prefetching counterpart, which equals to the difference between prefetching gain and the slow down due to running at a lower frequency. ($present_{savings}$) keeps track the performance gain (CPU saving) since last update and is added to the ($cumulative_{savings}$). We scale voltage down when we are saving fast (step 12) or we still have lots of savings (step 14). We scale voltage up only when we are saving currently (step 13) and do not have sufficient savings (step 15). Otherwise, the current voltage is kept.

We use the following parameters during our simulation: N = 100k instructions (profiling frequency); M = 5k instructions (profiling period); c_{th} = 50 us (threshold savings); W = 5k instructions (waiting time before re-profiling when there is no prefetching gain). They are set to prevent the application from spending more than a small percentage of its total execution time in profiling. We claim the features of low-power and performance-guarantee of the proposed algorithm. The word "almost" in Claim 1 can be removed if prefetching has no negative impact in system's performance. Detailed proofs are omitted due to space limitations, but both claims are nicely validated by the simulation.

Claim 1. The proposed algorithm guarantees (almost) no performance loss compared to the execution at the reference voltage without prefetching.

Claim 2. The proposed algorithm converges to the optimal voltage setting when prefetching's gain is estimated accurately.

3 Experimental Methodology and Evaluation

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

Application	Problem Size	Memory Access Pattern
IRREG	14K node mesh	Indexed array
MOLDYN	13K molecules	Indexed array
NBF	14K node mesh	Indexed array
MATMULT	200x200 matrices	Affine array
JACOBI	200x200x8 grid	Affine array
HEALTH	5 levels, 500 iters	Pointer-chasing

Table 1: Summary of benchmark applications.

3.1 Experiment Setup

Table 1 lists the six benchmarks, representing three classes of data-intensive applications, along with their problem sizes and memory access patterns. Irreg is an iterative PDE solver for an irregular

Processor Model	Issue Width	8	Integer Latency	1 cycle
(600 Mhz)	Instruction Window Size	64	Floating Add/Mult/Div Latency	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	L1/L2 Cache Size	16K-split/512K-unified	L1/L2 Associativity	2/4 cycles
(1 cycle = 1.25 ns)	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

Table 2: Simulation parameters for the processor, cache, and memory sub-system models. Latencies are reported either in processor cycles or in nanoseconds.

Application	No Prefetch (1.6V)		Prefetch (1.6V)		Prefetch (1.5V)		Prefetch (1.4V)		Prefetch (1.25V)		Prefetch (1.1V)	
	time	energy	time	energy	time	energy	time	energy	time	energy	time	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

Table 3: Static voltage scaling: numbers in bold indicate the performance and power when there is no significant loss in performance. (execution time is in the unit of ms and energy in the unit of 10^7 *Watts*cycle*).

mesh; Moldyn is abstracted from the non-bonded force calculation in an NIH system; NBF is from the GROMOS molecular dynamics code; The next two applications are from the SPEC/NAS benchmark suite; and Health is taken from the OLDEN benchmark suite.

We use Wattch, an architectural level power analysis tool [1], and SimpleScalar sim-outorder, a detailed simulator supporting out-of-order issue and execution [2], to track different units accessed per cycle and hence record the total energy consumed for a given application. Wattch power model is based on 0.35um process technology parameters with clock gating. Prefetching frequently issues memory requests and thus increases memory activity. Therefore, it is crucial to have a detailed memory model. Table 2 gives the baseline memory sub-system model we build to replace the simply memory model in the SimpleScalar sim-outorder simulator.

We use Transmeta's Crusoe processor, running at 600MHz and 1.60V, as the baseline processor. For voltage scaling, we adopt five different voltages: $1.60V$, $1.50V$, $1.40V$, $1.25V$ and $1.10V$, which provide frequencies of 600MHz, 500MHz, 400MHz, 300MHz, and 200MHz respectively [8]. Finally, our technique requires only a few counters and their power dissipation is also considered during the simulation by Wattch.

3.2 Experimental Evaluation

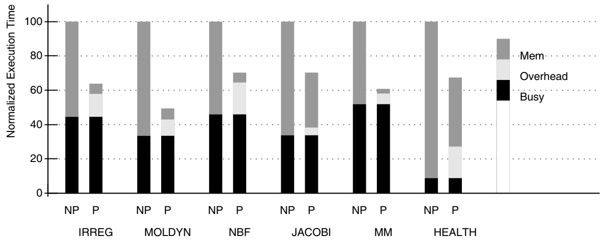

Figure 4: Execution time breakdown for annotated memory instructions.

Prefetching Performance. Figure 4 plots, for each application, the execution time with prefetching (labeled "P") normalized to that without prefetching (labeled "NP"). "Busy" is the execution time (without prefetching) on a perfect memory system where all memory accesses are completed in one cycle; "Mem" represents the additional memory access time on our non-perfect memory system model (Table 2). "Overhead" is the prefetching overhead in execution time. The real numbers of the total execution times are also shown in Table 3, the 2nd and 3rd column under *"time"*. One can see that on average software prefetching is capable of boosting the performance by 36.04%.

Static Voltage Scaling. We run each application at each of the five available voltages with their corresponding frequencies. For each run, the voltage is fixed and the prefetching is used throughout the entire execution. Table 3 reports the execution time (in ms) and energy reported by Wattch (in 10^7 *Watts*cycle*).

First, comparing columns 2 and 3, we see that at the same voltage and speed, prefetching completes earlier than no prefetching for all applications, but consumes more energy in general (**8%** on average, the average power overhead is **69%**) even after we use clock gating to turn off all the idle hardware units. This is because 1) prefetching improves performance and reduces the processor stalls and thus keep most hardware units active; 2) prefetching has overhead in the form of extra instructions and thus wastes energy when the prefetched data is not used.

However, as we reduce voltage from 1.6V to 1.1V, the completion time increases but the energy (and power) consumption decreases. For each application, the entry in bold identifies the voltage setting for static voltage scaling (SVS) to achieve the similar performance as no-prefetching. On average, we have a **48.72%** energy saving over the non-prefetching version.

We conclude that prefetching enhances application performance at the cost of higher energy consumption (and much higher on power which is the ratio of energy over time). However, prefetching combined with SVS can reduce the energy (and power) while maintaining a given performance level. Finally, SVS is not practical for real time systems as it requires the knowledge of actual execution time to determine the optimal static voltage.

Application	DVS-Power Savings %	DVS-Gain in Performance %	SVS-Power Savings %	SVS-Gain 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

Table 4: Energy saving and performance gain achieved by DVS and SVS over the non-prefetching version.

Dynamic Voltage Scaling. Table 4 gives the energy savings and performance gains achieved by the proposed online DVS approach over the non-prefetching version (the 2nd column of Table 3). Due to the discrete nature of the five available voltages and frequencies of the baseline Crusoe processor, we are unable to transfer all the performance gain to energy/power reduction. But our DVS algorithm is very competitive with the unrealistic SVS approach across the six applications, where it does manage to transfer all except 2.0% of the performance gain by prefetching to a significant **38.11%** energy saving.

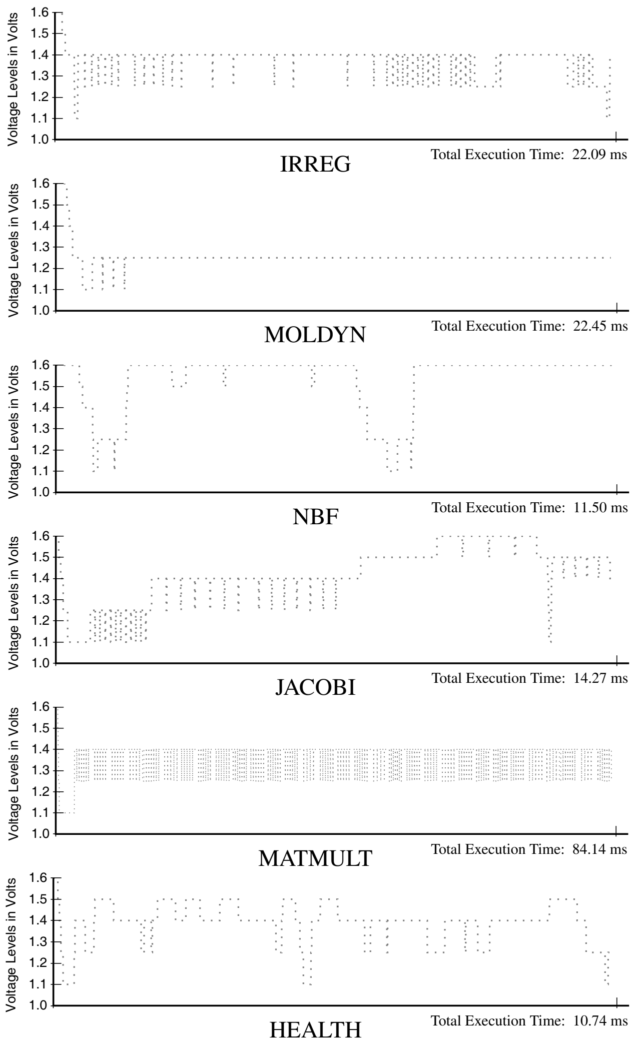

Figure 5: Voltage profiles of the online DVS algorithm.

Online DVS algorithm. Figure 5 gives the operating voltage level throughout a complete run of each application. It reveals some interesting insight of the proposed online DVS algorithm and the application's 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 exist 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 1.25V (21.98 ms) is very close to that in the non-prefetching version (22.23 ms). Therefore, after some initial toggling, DVS algorithm finds 1.25V as the optimal voltage and stays there. The small performance gain is canceled by the prefetching overhead.

However, none of the other applications has 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. Moreover, the application's dynamic behavior can also lead to a more irregular shape of the voltage profile. For example, considering the Health application, Figure 5 clearly shows that the online DVS algorithm first tries to stabilize at a voltage level between 1.5V and 1.4V, and then decides to find one between 1.4V and 1.25V in the second part of execution, where prefetching has reported more performance gain.

4 Conclusions

We propose a low power system design methodology where DVS and performance-enhancement techniques are coupled to simultaneously reduce energy consumption and provide performance guarantees. The developed online DVS algorithm periodically measures the performance gain delivered by software prefetching, and automatically adapts the voltage level to minimize power while maintaining the performance level of the original system. Simulation on real life benchmarks shows that significant energy/power reduction can be achieved without any performance loss compared to the system without prefetching.

This approach gives the traditional performance-enhancement techniques new meanings, namely offsetting or minimizing the performance loss caused by DVS on real time low power systems. We believe that this is promising and important for high-performance and low-power computing based on our encouraging results.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.

[3] T. Chen and J. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, Vol. 44, No. 5, pages 609-623, May 1995.

[4] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990. ACM.

[5] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, Toulouse, France, January 2000.

[6] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems*, 16(1):55–92, February 1998.

[7] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[8] Transmeta-corporation. Tm5400 processor specifications.

[9] K. Govil, E. Chan, and H. Wasserman. "Comparing algorithms for dynamic speed-setting of a low-power CPU", *ACM International Conference on Mobile Computing and Networking*, pp. 13-25, 1995.

[10] T. Ishihara and H. Yasuura. "Voltage Scheduling Problem for Dynamically Variable Voltage Processors," *ISLPED'98: International Symposium on Low Power Electronics and Design,* pp. 197-202, 1998.

[11] T. Pering, T.D. Burd, and R.W. Brodersen. "Voltage Scheduling in the IpARM Microprocessor System," *ISLPED'00: International Symposium on Low Power Electronics and Design,* pp. 96-101, July 2000.

[12] G. Qu. "What is the Limit of Energy Saving by Dynamic Voltage Scaling?" *IEEE/ACM International Conference on Computer-Aided Design*, pp. 560-563, November 2001.

[13] M. Weiser, B. Welch, A. Demers, and S. Shenker. "Scheduling for reduced CPU energy", *USENIX Symposium on Operating Systems Design and Implementation*, pp. 13-23, November 1994.