

# Power Conservation Algorithms and Scheduling Techniques

Bill Cowell, Matthew Reed, and Ben Tribelhorn

cowellb@onid.orst.edu, reed@eecs.oregonstate.edu, tribelbe@enr.orst.edu

## 1 Introduction

Power conservation as a means towards environmental, economic, and political improvement is certainly a dominant theme in modern life today. At the same time, the availability and use of computers throughout the world has increased dramatically. Hence, “green” computing should be an area of research in computer architecture that is gaining more focus. Additionally, the “convenience factor” associated with more efficient architecture (i.e. battery life) adds motivation on a productivity level. Along this thread, we have undertaken a survey of power reduction techniques looking at software based methodologies with a discussion of complementary hardware strategies.

After an investigation of potential hardware power optimization strategies, we show an analysis of the power consumption at the instruction level which motivates compiler strategies as well as on chip design patterns. This will then lead us to a discussion of algorithms for power conservation during computation (assuming variable voltage/frequency processors) beginning by defining Earliest Deadline First (EDF) as a baseline algorithm. Comparing the trade-off of power and performance we will then address the following questions. What are the possible improvements over EDF and can these improvements be combined for an even greater power savings? Where is the point of no return between saving energy from intelligent schedul-

ing and using too much energy (aka O-time) to schedule tasks? What improvements can be achieved only in soft real time systems? Finally, how large of an impact can application aware scheduling have on power consumption? The answers to these questions then will be revisited in light of hardware-level restrictions and optimizations to consider potentially synergistic options for power reduction.

## 2 The Tradeoff

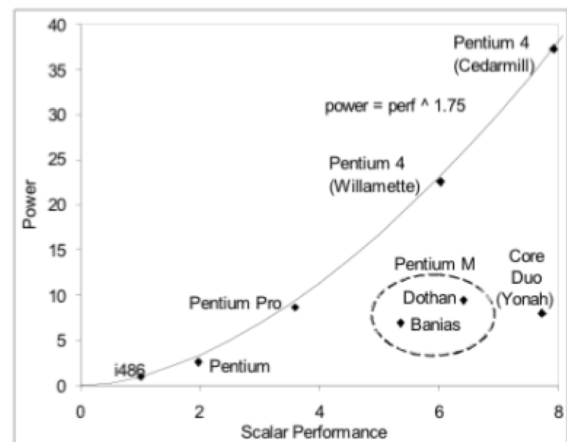


Figure 1: Power vs. performance for various Intel processors. This shows the exponential relationship in single-core processors.

The basis of “green computing” in the context of this report is underscored by the tradeoff between power and performance. Increasing clock speeds increases power utilization ex-

ponentially. Figure 1 shows that for Intel processors the relationship is roughly  $Power = f^{1.75}$ . Another point is that the dual core architecture uses cores that are roughly 15% slower than single processors, but the power savings is on the order of 50% or more! Clearly, sacrificing some amount of performance, especially in unsaturated systems, can have very meaningful reductions in power use.

The complexity of micro-architecture is at the threshold where slight increases in any functionality or size drastically increase the power consumption. This is due to the additional wire length and components on the processor and the need to speed up the clock to increase performance. Chip design needs to refocus to consider power utilization in future architecture. We will address specific improvements that will work in tandem with power conservation algorithms to decrease the total power consumption of the processor. Some of these hardware side improvements are directly motivated by the algorithms that are commonly used for scheduling.

### 3 Hardware Strategies

The implementation of software based power conservation algorithms with respect to superscalar computing is strongly coupled to hardware modifications. These hardware features are needed in multiple ways: 1) the register/buffers that are needed to monitor signals required by the algorithms to determine energy efficient paths, 2) to change frequency and voltage levels to lower power consumption when appropriate in multiple clock domain or variable voltage designs or 3) to selectively reconfigure a processor to reduce power by disabling functional blocks of the processor when appropriate. In addition, there are simple hardware implementations that can be incorporated in generic processor design that can supplant savings gained algorithmically. The following dis-

ussion will investigate several power savings strategies that have been researched in the area of superscalar computing.

- Introduction to Microprocessor Hardware Power Consumption
- Cross Talk Power Consumption
- Dynamic processor reconfiguration
- Multiple clock domains/voltage scaling
- Register file strategies

These strategies will be discussed in the context needed with respect to hardware interactions with algorithmic scheduling power management.

#### 3.1 Microprocessor Hardware Power Consumption

Power consumption in a semiconductor based integrated device is an avoidable cost of use. The components of the power used are primarily composed of two mechanisms: transistors switching on and off, and interconnect line capacitance. Through the evolution of the semiconductor manufacturing process, the power used by microprocessors has changed in many ways. The primary driver of these changes have been Moores law, which has driven the device size down, the device speed up, and the number of devices on a microprocessor up. Whereas the decreased dimensions of the modern transistor has driven the current dissipation per transistor down, the speed of switching and the number of devices has more than counteracted that savings. This can be seen through the following equation[1]:

$$P = (\sum_{i=1} a_i C_i V_i) V_{DD} F_{clk} \quad (1)$$

In this equation  $V_{DD}$  is the supply voltage,  $F_{clk}$  is the clock frequency,  $V_i$  is the voltage at which the node capacitance will charge,

$C_i$  is the total capacitance of the node, and  $a_i$  is the activity factor of the node (the average number of charging voltage transitions that occurs on the node per clock cycle).  $V_{DD}$  has decreased less than an order of magnitude, while the number of nodes (proportional to the number of transistors) and the clock frequency has increased by several orders of magnitude. Hence it is clear that this simplified view of power consumption would indicate that the modern microprocessors use significantly more power. The strategies that have been employed to reduce the power with respect to transistor switching will be discussed in following sections. At a high level, efficient use of transistors is the central theme of hardware power conservation here. In terms of this paper, this translates into 1) minimizing the number of transistors used (dynamic processor configuration), 2) lowering voltages and frequencies (multiple clock domain/voltage scaling), and 3) looking at the number of transistors used in the registers (Register file strategies).

The second component of power dissipation is created by the interconnections of transistors on an integrated circuit. Historically, this power consumption had a component from the resistivity of the interconnect lines. The introduction of multilayer Copper interconnects in microprocessor manufacturing has significantly reduced this component in two ways. First by shortening the average length of interconnects (an effect of multilayered approach) and secondly by reducing the bulk resistivity of the interconnect (an effect of using Copper). The remaining power consumption is therefore mainly an effect of capacitive effects. The reduction of power consumed by capacitive effects is the first hardware strategy that will be discussed in the next section.

### 3.2 Bus Conductor Crosstalk

Nieuwland et al.[2] propose a power reduction strategy that is not specific to superscalar ar-

chitectures but is appropriate in that superscalar microprocessors are typically complex and have significant interconnect density. The capacitance between interconnects that is discussed is illustrated in Figure 2.

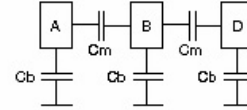


Figure 2: Three line (A, B, and C) capacitance.

In this case, the power dissipated due to signals running in the lines is given as:

$$P_{avgB} = \frac{1}{4}(C_B + \frac{3}{2}C_m)V^2 \quad (2)$$

This represents a potential power savings of up to 25%. This discussion is not meant to be technically in depth or specific to superscalar architectures. It is, however, meant to show an example of how the inherent material components of power consumption can be minimized. The following sections will describe strategies more specific to superscalar architectures.

### 3.3 Dynamic Processor Configuration

Modern superscalar design has been built around the exploitation of instruction level parallelism (ILP). The premise of ILP is that multiple instructions with no (or limited) interdependence can be executed at the same time, thereby decreasing the net time of execution for the software code. This concept implies higher performance, but when one looks at the tradeoffs associated with it, the performance gains become less clear. This is driven by the fact that the opportunities for ILP are limited by the dependencies of instructions in the code coupled with the energy overhead associated with the hardware involved with ILP.

There are several proposed strategies to minimize the power consumption of microprocessors when ILP is limited.

Maro et al.[3] have suggested that monitoring the hardware usage associated with ILP is a means to determine when energy savings are present. The premise of their proposed savings is that Instructions Per Cycle vary over a program run time (Figure 3) and can give indications of program segments in which power can be saved. They proposed several monitoring schema that could be used in this regard that allow the processor to be put into one of three low power configurations. The low power configuration disable parts or all of the integer and floating point clusters. Maro et al. have suggested that monitoring the hardware usage associated with ILP is a means to determine when energy savings are present. The premise of their proposed savings is that Instructions Per Cycle vary over a program run time (Figure 3) and can give indications of program segments in which power can be saved. They proposed several monitoring schema that could be used in this regard that allow the processor to be put into one of three low power configurations. The low power configuration disable parts or all of the integer and floating point clusters.

First, integer and floating point functional unit usages are tracked to determine the possibility of energy savings by use of a shift register. When a predefined number of units are not being used, a 1 is shifted into the register. When the number of 1s in the register reaches a certain level, functional units are selectively disabled. Monitoring the number of dependencies in the instruction window also gives a picture of the amount of ILP available. If the dependency count in the instruction window is high, the amount of ILP available is low and selective disabling of functional units is less costly.

Restricting the amount of instruction issue by instruction queue is a second technique. This is monitored by looking at Instructions

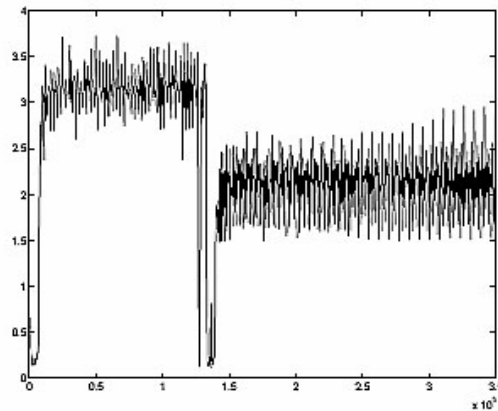


Figure 3: IPC variation for the hydro2d benchmark. The X axis represents the execution timeline while the Y axis is IPC.

Per Cycle (IPC) that are issued and committed. Maro et al. set a lower limit to the ratio of committed instructions to issued instructions to determine when a low power mode could be activated. A situation where the issue rate is significantly higher than the commit rate is indicative of low ILP. Hence, if limited resources are being used, a low power mode is activated and portions of the processor are disabled.

Folegnani et al.[4] similarly proposed monitoring the ratio of committed to issued instructions, but instead of disabling functional units for power savings, they suggested limiting size of the instruction queue. They discovered that the issued and committed instructions in many benchmarks were typically from the first quarter of the instruction queue. Figure 4 shows the placement in the instruction queue for both issued and committed instructions across 6 benchmarks.

With this in mind, disabling portions of the instruction queue was surmised to have little effect on the overall IPC due to the fact that instructions residing in the earliest portions of the instruction queue were rarely issued. Hence during periods where committed instructions were low relative to those being issued, the instruction queue does not need to be as deep.

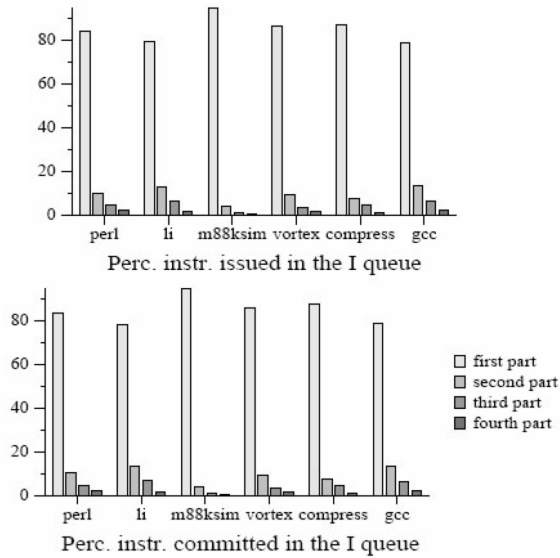


Figure 4: Instruction queue location of issued and committed instructions.

Seeing as the suggested modifications are hypothesized to not affect IPC, and would also not involve frequency change, the scheduling estimations would not be affected.

Another strategy proposed by Pokam et al.[5] is to modulate the bit width of the register files in a dynamic fashion. They base their technique on the fact that many operands in a superscalar pipeline do not use the full bit-width (Figure 5).

The strategy that is proposed here is that the compiler is exposed to areas of narrow bitwidth so that it is able to schedule them together while the register bitwidth is reduced. On the hardware side, a “bitslice” register design would be employed that would allow the register file to resize between 8, 16, and 32 bits. Hence, there is both hardware and software components to this scenario. Generic compiler strategies will be discussed later in the paper.

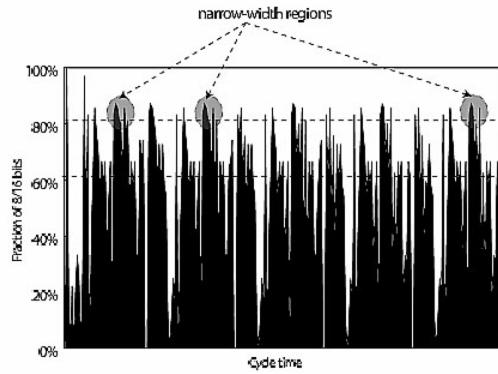


Figure 5: Bitwidth by cycle time for adpcm benchmark. The Y axis represents the percentage of operands that are 8 versus 16 bits (datapath is 32 bits).

### 3.4 Dynamic Voltage Scaling and Multiple Clock Domains

These techniques rely on being able to modulate the voltage and frequency to various parts of the processor. The main effort here is to exploit energy savings by selectively lowering the operating voltages and frequencies on the processor while attempting to preserve performance. The driving force to the MCD that is most attractive is the fact that it is necessary due to clock skew anyway.

Semeraro et al.[6] simulated a multiple clock domain (MCD) architecture with globally-asynchronous, locally-synchronous (GALS) clock to look into the potential energy savings. They specifically investigated dynamic frequency and voltage scaling effects over a wide range of benchmarks. The clock domains that they define are: front end (L1-Instruction cache, branch prediction, rename and dispatch), integer core, floating point core, load/store (L1D cache and L2 cache), and the main memory (external to processor). The main memory is not considered controllable by the processor and is always run at full speed. They presented an algorithm that controlled the processor frequencies and voltages (they are

Algorithm	Performance Degradation	Energy Savings	Energy-Delay Product Improvement	Power / Performance Ratio
<i>Attack/Decay</i>	3.2%	19.0%	16.7%	4.6
<i>Dynamic-1%</i>	3.4%	21.9%	19.6%	5.1
<i>Dynamic-5%</i>	8.7%	33.0%	27.5%	3.8
<i>Global (Attack/Decay)</i>	3.2%	6.5%	7.8%	2.0
<i>Global (Dynamic-1%)</i>	3.4%	6.6%	3.6%	2.0
<i>Global (Dynamic-5%)</i>	8.7%	12.4%	5.0%	1.9

Figure 6: Comparison of Attack/Delay vs. other MCD and Globally Clocked (no domains) strategies.

scaled together) through the analysis of the instruction issue queue in each of the domains (not the memory). If the queue is underutilized (used below a defined level) in a domain, the frequency of the clock in that domain is lowered. When utilization increases, the frequency in that domain is likewise increased. Therefore, the algorithm attempts to minimize the energy used in program windows where resources are not needed. There simulated results for this attack/delay” strategy are shown relative to a other MCD strategies in Figure 6.

The findings of their simulations illustrated that the Attack/Decay strategy had the least performance degradation. Unfortunately, the hardware associated with this strategy is quite extensive with respect to changing the frequency of the clock, so the area cost of this technique is high. Their investigations, however, elucidated the fact that the variation of savings as well as performance degradation was quite large with respect to the benchmark being tested.

Interestingly, Snowdon et al.[7] have indicated that while processor savings can be achieved through these techniques, the power associated with the memory of the system dominates so the savings are incrementally small. This is consistent with other findings, only worded in a more pessimistic fashion.

### 3.5 Register Based Power Reduction Strategies

The registers in a superscalar processor are critical to the performance attained. Unfortunately, a significant portion of the power consumed by the processor is driven by the registers. There has been significant research associated with energy savings involving register strategy. Zyuban et al. [8] gives a good high level description of the components energy use in register files and how to best design the register hardware infrastructure with power consumption in mind.

Register file power consumption is driven by different physical components depending on the size of the file. Large register files (defined by Zyuban as having > 128 entries) have their power consumption dominated by bit line energy. This is due to the fact that larger register files have longer bit lines, hence the energy required to charge the bit line across the file is high. Smaller register files have shorter bit lines that contribute less to total power consumption. The energy used by the sense amplifier becomes the dominant energy component in this scenario.

Increasing ILP leads to a needed increase in the number of read and write ports in superscalar processors. This is the second driver of energy consumption with respect to register file design. The logical presumption that en-

ergy increases with the number of ports holds true, which needs to be considered when balancing performance with power consumption. Figure 7 illustrates the power consumption correlation to register file size and number of ports in Zyubans paper.

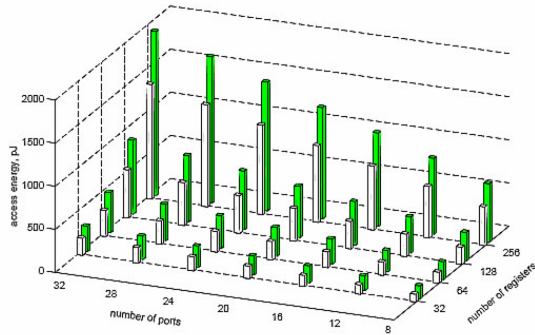


Figure 7: Energy consumption relative to the number of ports and register files. The number of ports includes both read and write ports.

Tseng et al.[9] discuss several hardware designs to improve the energy dissipation of register files at a design level. The analysis of the benchmarks used in their study indicates that 82% of the bits fetched from the register file are zeros. This leads them to a proposed storage cell design that takes advantage of this fact and minimizes the energy used during cell discharge. The tradeoff here is in the physical size of the register file which increases by 9%. A more real estate efficient proposal is to bypass the R0 register. The R0 location is always 0, so there is an energy use associated with the bitline discharge when this register is read. By providing a separate zero input to the processor hardware, this register bit can be deleted which reduces overall energy. Unlike the modified storage cell design, this change comes at a lower overall layout area increase.

Moshovos [10] reports that the register renaming unit in the Pentium Pro design dissipates approximately 4% of the power of the chip. He proposes reductions in read and write ports associated with the register alias table

(RAT) under the premises that 1) most instructions do not use the maximum number of source and destination operands and 2) the maximum number of renamed instructions is not always necessary. He goes on to investigate savings that can be achieved by hardware modifications to the circuitry used in branch prediction. Their simulations across benchmarks show the results in Figure 7 for reduced read and write ports. This indicates that reducing write ports from 8 to 6 has low (<1%) performance impact, while reducing read ports from 24 to either 20 or 16 has low impact.

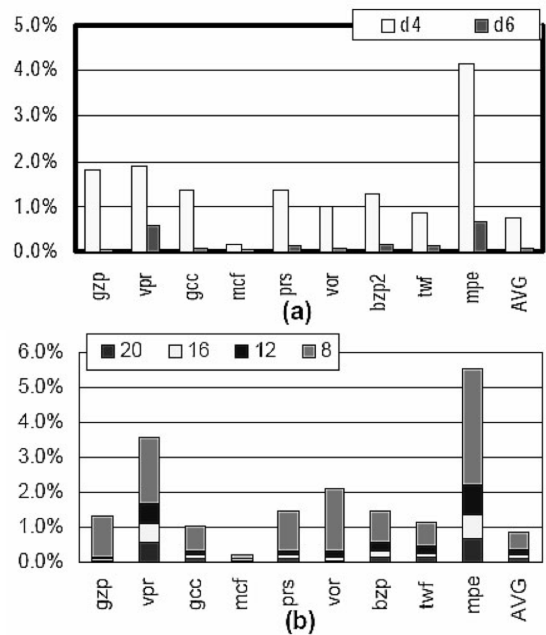


Figure 8: Performance slowdown for (a) RAT write port reduction and (b) RAT read port reduction.

The innovation in his approach (it is common knowledge that reducing read and write ports saves power) is to use an out of order branch resolution schema as apposed to the normally accepted in order resolution. This schema allows a branch checkpoint to be returned to the checkpoint pool (and hence allowing a new branch to start) as soon as the branch in question is resolved. The justification

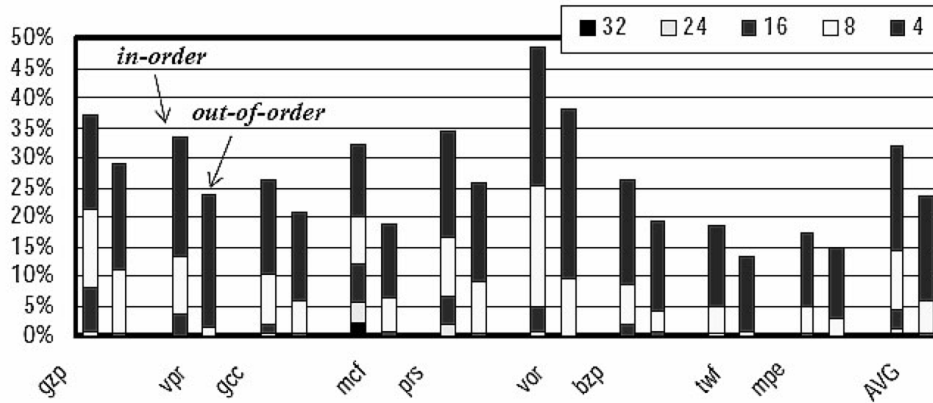


Figure 9: Performance slowdown for a number of branch checkpoints for in and out of order designs. (NOTE: AVG, right-side, for in order has 24 checkpoints at the bottom of the bar whereas OoO has 8 checkpoints.)

here is made by looking at the potential outcome scenarios with out of order branch resolution (this assumes the branch in question was predicted correctly). The first scenario occurs when all previous branches resolve correctly. In this case, there is no need for the checkpoint held by the released branch, so the checkpoint release caused no impact. The second case occurs when the branch checkpoint in question is released and a prior branch is mispredicted. In this case the checkpoint for the mispredicted branch is used to return the RAT to the correct state, so once again, the checkpoint in question is not needed.

Moshovos goes on to claim that there might even be performance improvements due to the fact that the checkpoints in his scenario are no longer implemented via a circular queue, but rather as check-pointing cells embedded with each row in the RAT. In the circular queue design, the time to return the RAT to the appropriate state after a misprediction is dependent on shifting through the circular queue until the correct branch is found. On the contrary, the checkpointing bit in their design can be accessed immediately. Figure 9 shows the impact of number of branch checkpoints for both in order (common practice, left bar) and out of

order (Moshovos scheme, right bar) scenarios. This graph suggests that 16 checkpoints in an out of order scheme performs better than 24 checkpoints in an in order scheme.

### 3.6 Summary

The preceding discussions illustrate a variety of methodologies that may be employed in the sphere of superscalar architectures to reduce the power consumption of the microprocessor. What has been shown is that power reduction can be achieved through careful layout as well as intelligent utilization and design of the building blocks used in superscalar design. In the end, the main lesson learned is that any power savings has some performance impact. This performance impact is to varying degrees depending on which benchmark is being used to assess performance. Therefore, one must ponder their own need for speed” when balancing performance against power savings.

## 4 Software Power Analysis

### 4.1 Software Power Consumption

Analysis of power consumption traditionally focuses on the hardware. When we do focus on the software, it is usually to determine how to make software run faster, not how to make it consume less power. Although many techniques that make software run faster can also make it consume less power, this is not the primary focus. In this section, we focus on a few facets of power consumption from a software perspective.

First, why would we want to know how much power software consumes? Consider a project where we are building a consumer embedded device, such as a mobile phone. Suppose we want to know how big of a battery to put into the device, and suppose we have a limit on battery size. If the device just sits idle and not doing anything, it consumes a minimal amount of power. Once we start loading and running software on the device, the power consumption will increase. If it was possible to assign a specific power cost for each software application we load onto the device, we can estimate how much total power will be consumed by adding up the cost of each application times the estimated amount of times the application will be run. We can also control the total power consumption by limiting the applications that go on the device, and balancing the applications such that the total power consumption does not exceed out budget. It would even be possible to limit the number of applications so that we could use smaller batteries and save money. If we could save one dollar per device, and if we are making 250,000 devices, we would save \$250,000.

If we could profile power consumption of applications on different processors, we might find that different processors use different amounts of power to run the same applications. This

would allow us to choose more efficient processors. Likewise, we could profile compilers in order to determine which ones give more power efficient code.

How can software consume power? It is not a physical object; you cannot hold it in your hands. It has no voltage applied to it, no current flows through it. And yet, like the driver of a car, the software directs the activities of the CPU, and therefore directly influences the amount of power the CPU consumes.

In order to estimate how much power a software program will consume, we need to understand how the CPU consumes power, and what influences this. Roy et al.[11] suggested that system memory consumes from 10-25% of the power. Cache access also consumes power, but less than memory access. System busses (address, control, data) consume power, and how much is largely determined by the software being executed. Various functional units consume power, and the power consumption of some of the units is also determined by the software. Other components of the processor that have more constant power consumption, such as clock circuitry, is also relevant because the longer it takes to run a program, the more of this type of power will be consumed.

### 4.2 Instruction Level Power Analysis

One method of determining power cost is called Instruction Level Power Analysis. The purpose of this is to analyze the power being consumed by different instructions or combinations of instructions. If we can determine a difference between different instructions, it may be possible to modify a compiler to favor more efficient instructions.

Tiwari et al.[12] performed a series of experiments to determine how the software being executed can influence power consumption. They used specially modified mother boards that al-

lowed an ammeter to be placed in series with the power line to the processor. By doing so, they could monitor the power being consumed by the cpu at any time.

They then developed and executed various programs that executed the same instruction over and over again, and measured the current consumption while the program was running. Their programs were designed so as to minimize stalls and cache misses, and to minimize the cost of the jump instructions.

When they did this, they discovered three different categories of power consumption. The first category consists of the base cost of a single instruction, executed over and over again. This specific cost requires that the test program be constructed such that there are no cache misses, no pipeline stalls, and minimal jump/branch penalties. Table 1 gives an example of the results[13].

They noticed that the base cost of an instruction could vary somewhat according to the address and value of the operands used. Because the variation is less than 10%, and because it is not usually known in advance what these values are, it is difficult to take advantage of this at compile time.

They also observed that instructions with similar functionality tended to consume the same amount of power. This is not surprising, considering that similar functionality should use the same functional components of the processor.

The second category of power consumption comes from what is called circuit state overhead for instruction pairs. After a specific instruction executes, the various gates in the processor and functional units are left in a specific state such that when another specific instruction executes, the power consumption of the second instruction is higher than when the second instruction executes over and over again. In other words, it was found that when certain instructions followed other instructions, the to-

tal power consumption of the two instructions was greater than the sum of the power consumption of the instructions when they were not executed one after the other.

They observed that this circuit state overhead was smaller in larger processors. A possible reason for this is that in larger processors, a lot of the circuit activity is common to all instructions. This common overhead tends to minimize overhead that is unique to certain instructions.

The third category of costs is other inter-instruction effects, such as unavoidable stalls, cache pre-fetches, etc. If the processor stalls for a few cycles, it is still consuming power.

Once we know what the different categories of power consumption, we can estimate the cost of a software application. For any given program, its power cost,  $E_p$ , would be as follows:

$$E_p = \sum_i (Base_i \cdot N_i) + \sum_{ij} (Inter_{ij} \cdot N_{ij}) + \sum_k E_k \quad (3)$$

Where  $Base_i$  is the base cost of instruction  $i$ ,  $N_i$  is the number of times instruction  $i$  executes,  $Inter_{ij}$  is the inter instruction cost when instruction  $j$  follows instruction  $i$ ,  $N_{ij}$  is the number of times instruction  $j$  follows instruction  $i$ , and  $E_k$  is all other inter instructions costs.

Some of the results were not what you would expect. For example, it was found in some cases that an OR instruction consumed as much power as a MULTIPLY did. You would think that a simple OR instruction would consume less cycles and therefore less power. This suggests inefficiency in the hardware design.

Another discovery was that instructions that accessed memory consumed much more power than those that did not. For example, in the 486DX2, register instructions cost about 300mA, memory reads cost 400mA, and memory writes cost more than 530mA. It becomes apparent that if we can reduce the number of times we read and write memory, we can sig-

Intel 486DX2				Fujitsu SPARClite '934			
Instruction	Current (mA)	Cycles	Energy	Instruction	Current (mA)	Cycles	Energy
nop	276	1	2.27	nop	198	1	3.26
mov dx,[bx]	428	1	3.53	ld [10],i0	213	1	3.51
mov dx,bx	302	1	2.49	or g0,i0,10	198	1	3.26
mov [bx],dx	522	1	4.30	st i0,[10]	346	2	11.4
add dx,bx	314	1	2.59	add i0,o0,10	199	1	3.28
add dx,[bx]	400	2	6.60	mul g0,r1,r2	198	1	3.26
jmp	373	3	9.23	srl i0,1,10	197	1	3.25

Table 1: Comparing instruction costs between two processors. Energy units are  $10^{-8}$ J.

nificantly reduce our power consumption. If we can more efficiently use registers so as to reduce the need to access memory, we can reduce power. If we have more registers, we can likewise reduce dependency on memory. It is possible that the power savings for having a larger register file can offset the additional power consumption required for accessing a larger register file.

Roy et al.[11] conducted some tests on instruction reordering for reduced power consumption. They found that it was possible to achieve a power savings of 10% to 30%.

### 4.3 Reduction of Memory Access

Tiwara et al.[14] observed that instructions that access main memory consume greater amounts of power. They felt that by reducing the usage of these instructions they could reduce overall power consumption. In their tests, they accomplished this through more efficient usage of registers.

As an example, consider Table 2: `h1cc.asm` is a heapsort program produced by the C compiler `lcc`. When it was executed with no modifications, it ran in about 11 microseconds and consumed about 525.7 milliamps of current. The column for `hht1.asm` represents the code after it was modified for faster performance, but not for more efficient power consumption. When executed, it actually consumed more

current—534.2 milliamps compared to 525.7 milliamps. Because it executed faster, the overall power savings was about 13.5%.

The fourth column, `hht2.asm`, represents the code modified for reduced memory access. Three variables that were stored in memory were moved to registers. The code did execute faster and also consumed less current while it was running. The overall power savings was 23.5%.

The last column, `hht3.asm`, represents the code modified for greater reduction in memory access. Five variables were moved to registers. Again, less average current was consumed, the program ran faster, and the energy savings was 40.6%.

This is a very dramatic savings in power. It is likely that in real world situations, one would not realize this much of a savings. It is also worth mentioning that these tests were run over 12 years ago, and that modern compilers today will automatically optimize the code for greater efficiency. While modern compilers are not specifically tuned to give lower power consumption, they will significantly enhance performance. Clearly reducing running time is linked to lowering power consumption.

### 4.4 Loop Fusion

Roy et al.[11] discussed a technique we covered in class that we call loop fusion. Consider the following code snippets:

Program	hlcc.asm	hht1.asm	hht2.asm	hht3.asm
Average Current (mA)	525.7	534.2	507.6	486.6
Execution Time ( $\mu\text{sec}$ )	11.02	9.37	8.73	7.07
(mA $\cdot\mu\text{sec}$ ) $\cdot 10^{-6}$	5.79	5.01	4.43	3.44
% Faster		14.9	20.8	35.8
% Current Change		1.6	-3.4	-7.4
% Savings		<b>13.5%</b>	<b>23.5%</b>	<b>40.6%!</b>

Table 2: A comparison of register movement for the heapsort algorithm.

Program A	Program B
<pre>for(int i = 0; i &lt; j ; i++ ) {     B[i] = A[i]; } for(int i = 0; i &lt; j ; i++ ) {     C[i] = B[i]; } </pre>	<pre>for(int i = 0; i &lt; j ; i++ ) {     B[i] = A[i];     C[i] = B[i]; } </pre>

Program A moves the contents of array A into array B during the first loop, and then from B to C during the second loop. Each loop iterates  $j$  times, for a total of  $2j$  iterations. Program B moves A to B to C in one step, and only one loop iterates. When I ran the above code snippets, I observed the following results shown in Figure 10.

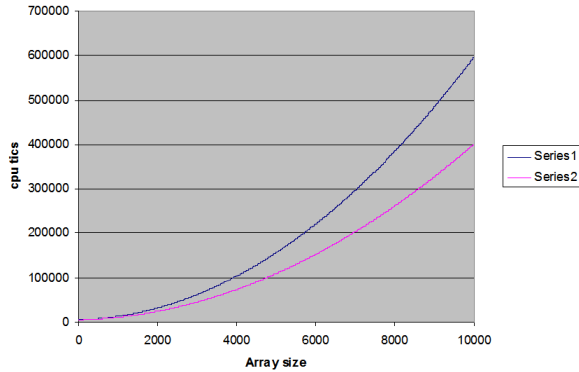


Figure 10: Series 1 is the code from Program A. Series 2 is the modified code from Program B.

Series 1 is the code from Program A. Series 2 is the modified code from Program B. As you can see, the modified code ran in  $2/3$  of the time the original code ran. We suspect that it is due to cache hits in the second case where the value had to be accessed from main memory twice in Program A. This simple instance

of loop fusion gives significantly faster performance, and therefore a significant reduction of power consumed.

## 5 Power Conservation Algorithms

The definition of a power conservation algorithm (as used in this paper) is a scheduling algorithm that dynamically adjusts the voltage, and therefore clock speed, of a processor. These algorithms should attempt to meet any deadlines, but as a caveat with this sort of algorithm we will consider all deadlines “soft” rather than “hard” (explained below). This is because switching clock speeds takes a certain activation energy (computationally) so we don’t want state changes all the time and also we may end up running too slow for too long. The other motivation is that even if the average processor speed is low it doesn’t mean the algorithm minimized power consumption. Recall that the performance vs. power is an exponential function. So in many cases it is actually better to average a higher speed. Most processors that support scaling the voltage have specific predefined voltage/frequency combinations that it can perform at. Research has shown that having more than a small number of domains is not significantly better than a continuous scale. Figure 11 shows AMD’s report on how far cutting back can help. There is a certain point where the functional units, registers, and other components become the dominant term of pro-

cessor consumption. So, often times the minimum speed that a processor can be set to is greater than half its speed.

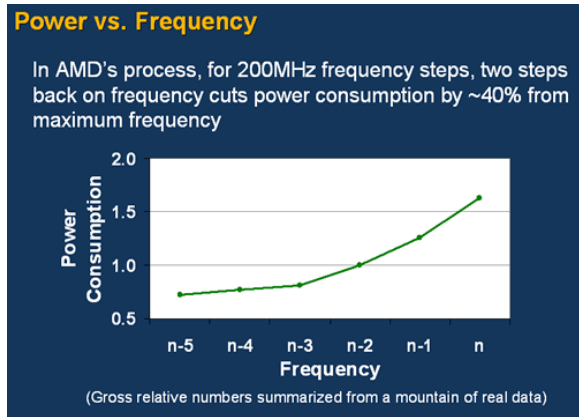


Figure 11: AMD shows that after a certain point reducing the speed does not have a continued effect on power reduction.

As mentioned above, hard deadlines are mission critical. A simple example of this is that the embedded system in your car needs to be able to finish the job that determines whether or not you are crashing in time to deploy the airbags. However, if the system takes a bit longer to switch the tracks on your CD then the system still continues to function acceptably. That would be a soft deadline. The reason we have to consider this distinction is that many embedded systems have hard deadlines, while deadlines on a desktop computer are generally considered soft because we want an average performance that is high.

## 5.1 A baseline

The Earliest Deadline First (EDF) algorithm is the baseline algorithm for comparing scheduling algorithms because it is optimal for preemptive single-processors. This means that if it is possible to schedule every task then EDF will schedule them so that they all complete prior to their deadlines. For periodic tasks this means that so long as cpu utilization is upper

bounded by 100% then all jobs will be scheduled to complete on time. The disadvantage to this algorithm is that in systems that have higher than 100% utilization the behavior of missed deadlines is unpredictable so it is not typically used in real-time systems. Relating this back to power conservation, there is a notion of priority and “soft” deadlines that this algorithm clearly does not include. However, it is a very good baseline for comparing maximum power usage as it schedules everything as soon as possible.

## 5.2 Tricks

The basic goal of most scheduling algorithms is to take advantage of slack times. The slack time is the idle time between jobs and their deadlines as well as any potential idle time that could be gained by delaying execution of jobs with later deadlines.

Additionally, application specific optimization is common in many embedded devices because most tasks are periodic and easily profiled for specific optimizations. We won't delve deeply here as generic algorithms are of more interest.

## 5.3 Fixed Priority Scheduling

For general power savings there is no *a priori* knowledge of the workload. This motivates forming scheduling policies using dynamic voltage scaling to minimize power consumption in real time in light of soft deadlines. As DVS can reduce power usage quadratically relative to performance decrease in the processor, it is an important thing to consider.

### 5.3.1 Hard Real-Time

A method presented by Quan and Hu[15] improves on the methods investigated in [16]. We can compare this method to EDF as it is less

common in real-time systems because fixed-priority scheduling is more practical due to its low overhead and predicability. Two algorithms are presented by this work given the assumption that  $N$  is the number of jobs queued to be scheduled. The basis of this work is by the following statement of optimality:

**THEOREM:** *Given a set of jobs starting at  $t_0$  and to be completed by  $t_1$ , the voltage schedule that employs a constant voltage in  $[t_0, t_1]$  is necessarily and optimal schedule in the sense that no other schedule consumes less energy to complete the jobs in time.*

They present an obvious method for determining this minimum constant voltage with an  $O(N^2)$  algorithm. Although that sounds pretty slow, remember that in a real-time system the number of jobs queued up ideally is a single digit number and so our required computation is not extreme. However, this algorithm does not produce a voltage schedule because it simplifies deadlines by finding the critical minimum interval. They extend the first algorithm to find a schedule within  $O(N^3)$ . They show that this schedule always saves more energy than one that applies the minimum constant speed when the processor is busy and shuts down when idle (basic EDF). The results of this algorithm show a modest improvement over existing algorithms. This is clearly a software approach and as such makes some assumptions that are not necessarily true. First, most variable voltage processors have only a small number of discrete speeds rather than ability to adjust on a continuous scale. Second, a processor can not be adjusted to “zero” consumption during an idle time. Also, we must consider the design complexity of scheduling code within the OS. In worst case scenarios the overhead induced by an  $O(N^3)$  algorithm could be excessive compared to EDF. Unfortunately algorithms that adjust the processor speed are exclusive, so when considering these algorithms we can not get a synergy by combining two. The following algorithms presented are simpler

implementations that can be done within the hardware and have a potentially smaller overhead overall.

### 5.3.2 Soft Real-Time

The algorithm presented in [17] by Rusu, et al. schedules jobs first come first serve (FCFS), without preemption. It assumes that jobs are presented to the system (or processor) as events of a certain type with a deadline,  $D$ . In order to eliminate over-computation of a voltage schedule there are two cases where the processor speed may be changed: job completion and job arrival. With these considerations, there are a number of potentially effective policies. Prediction based schemes are dependent on how well the workload can be predicted. An “application oblivious” policy would only monitor the system utilization, e.g. the CPU itself. As a very simple single predictor this is an effective method that can be extended to monitor other resources. Application aware prediction monitors request inter-arrivals and processing requirements. As many workloads are unpredictable in general computing this method is actually slightly less effective than the previous policy due to either lag or overly aggressive response. It also has a higher complexity factor. The third prediction policy is reinforcement learning, however the investigation done shows that this is far too complex (since it requires many variables to be tracked and processed) to be as effective as the simpler prediction methods. The other option in contrast to prediction based policies is stochastic algorithms. Without reporting all the details, a summary of this method is to generate the probability distribution of request CPU cycles classified by some granularity term. The benefit to this algorithm is that it runs in  $O(B + M^2)$  time where  $B$  is the number of bins (range of cycle times) and  $M$  is the number of speeds the processor can be adjusted to (which is typically only 2-5). This is a highly efficient algorithm because the

number of bins needed to be highly effective is only on the order of 100. This method improves over the application oblivious prediction method by 15-30% on the applications tested given a prediction period for updating of one second. The stochastic method presented can achieve up to a 20x savings in power compared to no power management which is about an average of 50% better than the best prediction method. The following two graphs show these savings. Figure 12 shows around a 50% improvement in total power consumption by the stochastic method over the application oblivious method. Figure 13 compares no policy management (assume EDF at max speed) versus the three algorithms: Application Oblivious, Application Aware, and Stochastic.

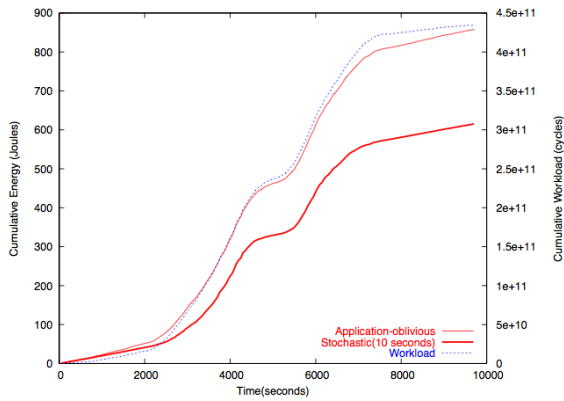


Figure 12: Energy consumed in the *Event Extraction* 81 minute trace (example program).

## 5.4 Dynamic Critical Path

A critical path of execution is one that if delayed will slow the progress of the application. The following algorithm uses a critical path predictor as presented in [18].

While it may seem intuitive that dynamic information about instruction criticality can be used for improved processor performance, it may not be as clear that critical path information can also be used to decrease power con-

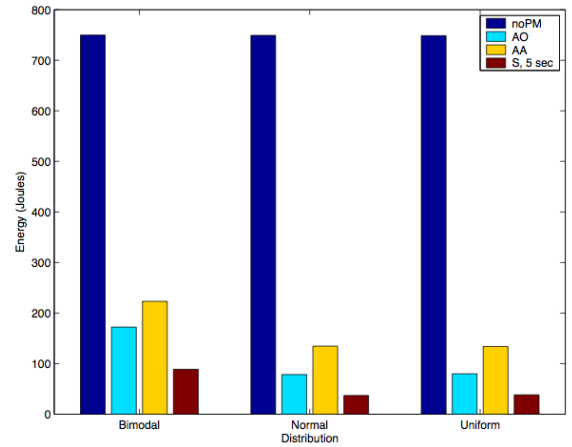


Figure 13: Comparison of the Soft Real-Time DVS policies based on randomly generated distributions of request cycles (job times).

sumption within specific parts of the processor without a decrease in performance.[19] Research demonstrates that you can effectively “nice” non-critical instructions by executing them through resources that are designed for lower power rather than performance. Therefore a section of the processor that uses less dynamic power and higher threshold voltage transistors can result in a tremendous power reduction. The first reduction stems from the fact that single cycle integer functional units (FU) require a lot of circuitry to attain this high performance. By reducing the single cycle requirement on these units the power reduction is not insignificant. The key is to then only execute non-critical instructions on these “slow” functional units. Second, they recommend altering the instruction queue to reduce its size and to consider critical and non-critical instructions separately.

To assess the power savings a number of assumptions were made about the ratio of power consumption between fast and slow functional units as well as their idle usage. The faster unit is assumed to consume 10% of its operational power when idle and the slow unit 5% of its non-idle. The last detail is that there needs to be a critical path predictor table which will

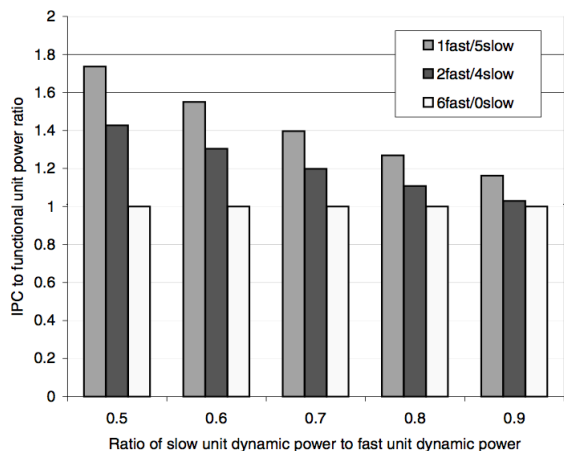


Figure 14: The ratio of IPC to functional unit power for various FU configurations and various assumptions about the power dissipation of the slow units.

consume some amount of power. However, they have shown that this table does not need to be very large (1Kb is virtually as good as 64Kb). This allows for each instruction to have a 1-bit CP-predictor. Issue is then determined by that predictor whether it will go to a fast or slow functional unit. On a system with 6 functional units it was found that the performance difference between zero and one fast functional unit is more than half the difference between zero and six fast units. The power savings of just one fast unit versus all fast units is considerable, see Figure 14. This reduction to only one fast unit is estimated to have an average performance decrease of about 8%, but the power savings can be from 25-75% depending on the difference in consumption of a slower unit. The only caveat to the performance is that if you do not have a simple load balancing algorithm then it is possible to degrade performance heavily. A method presented for this uses a shift register and improves performance on the critical predictor by 7% alone. They conclude that a one fast 5 slow ratio is the best choice.

Allowing for a two queue system the complexity and therefore power is reduced. How-

ever, to avoid complexities in this method of scheduling the processor needs to issue the critical queue in order. This is a great power savings since the single issue (due to a single fast unit) is very simple. This queue separation has a miniscule performance reduction overall. In conclusion, this critical path prediction has the best impact on high-performance processors. However, we must remember that this requires a change in the hardware which could interfere with prior optimizations.

## 6 Green Computing Conclusions: Hardware, software, or both?

The literature surveyed to investigate power saving strategies with respect to modern microprocessors showed a very consistent theme. Namely, hardware and software solutions are usually intertwined. We have shown that there are hardware only solutions that can optimize the physicality of interconnects with respect to power consumptions. We have also shown that by optimizing code, there are opportunities to reduce the amount power consumed by a program. In either of these scenarios, the true realized power conservation is most likely not as great as isolated simulations or theoretical analysis predicts. Additionally, we should hope that the isolated hardware or software technocrat is leveraging this common knowledge regarding power consumption to offer efficient end products to the largest extent possible. The most technically intriguing realm of discovery in our survey was where software and hardware interacted in concert towards the end of power savings. In the end this showed that hardware and software must play off each other to maximize power savings. To illustrate this, reconsider three power conservation strategies with respect to the interplay between hardware and software: the reduced memory access com-

piler strategy, the critical path strategy and multiple clock domains.

The very nature of the reduced memory access compiler strategy takes advantage of hardware. Namely the efficient use of registers in the hope of minimizing the number of cache misses. It was shown that this strategy can save power not only through less use of main memory and the long bus lengths therein, but through the speed up of a program. This is a strong example of how a software scheme is actually an exploitation of hardware based power savings.

The critical path functional unit strategy that was discussed also has strong interactions between software and hardware. The algorithmic determination through scheduling analysis of which instructions go to which functional unit is clearly in the software regime. The fact that we are discussing functional units with different speeds is clearly in the hardware regime. This shows that we are, in reality, discussing a strategy that transcends the software/hardware divide.

Lastly, reconsider multiple clock domains. MCD strategies are widespread in many of the commercially available modern microprocessors. This strategy depends heavily on algorithmic analysis of domain activities to determine when and how much a domains clock/voltage can be scaled down without impact to performance. This condition certainly points out an interplay between software (algorithmic analysis and feedback) and hardware (domain clock frequency/voltage control). Without one side of the paradigm, the other would be ineffective.

It can therefore be surmised that any effort to truly minimize microprocessor power consumption should consider both hardware and software angles. Not only is an effort in both arenas more likely to succeed, but it also allows for the paradigm of a fresh set of eyes proposing better solutions to a problem". By this we mean that software centric engineers might have valuable insights into hardware design and

visa versa. It can certainly be argued that having collaborative efforts between engineers with diverse training can lead to great innovations!

## References

- [1] Y. Taur and T. Ning, *Fundamentals of modern VLSI devices*. Cambridge University Press, 1998.
- [2] A. Nieuwland, A. Katoch, and M. Meijer, “Reducing cross-talk induced power consumption and delay,” pp. 179–199, 2004.
- [3] R. Maro, Y. Bai, and R. I. Bahar, “Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors,” *Lecture Notes in Computer Science*, vol. 2008, pp. 97–??, 2001. [citeseer.ist.psu.edu/maro00dynamically.html](http://citeseer.ist.psu.edu/maro00dynamically.html)
- [4] D. Folegnani and A. Gonzalez, “Reducing power consumption of the issue logic,” 2000. [citeseer.ist.psu.edu/folegnani00reducing.html](http://citeseer.ist.psu.edu/folegnani00reducing.html)
- [5] G. Pokam, O. Rochecouste, A. Sez nec, and F. Bodin, “Speculative software management of datapath-width for eenergy optimization.” [citeseer.ist.psu.edu/pokam04speculative.html](http://citeseer.ist.psu.edu/pokam04speculative.html)
- [6] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, “Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling,” in *HPCA*, 2002, pp. 29–42. [citeseer.ist.psu.edu/semeraro02energyefficient.html](http://citeseer.ist.psu.edu/semeraro02energyefficient.html)
- [7] D. C. Snowdon, S. Ruocco, and G. Heiser, “Power management and dynamic voltage scaling: Myths and facts,” Sept. 2005. [citeseer.ist.psu.edu/733379.html](http://citeseer.ist.psu.edu/733379.html)
- [8] V. Zyuban and P. Kogge, “The energy complexity of register files,” pp. 305–310. [citeseer.ist.psu.edu/zyuban97energy.html](http://citeseer.ist.psu.edu/zyuban97energy.html)
- [9] J. Tseng and K. Asanovic, “Energy-efficient register access,” 2000. [citeseer.ist.psu.edu/article/tseng00energyefficient.html](http://citeseer.ist.psu.edu/article/tseng00energyefficient.html)
- [10] A. Moshovos, “Power-aware register renaming.” [citeseer.ist.psu.edu/540106.html](http://citeseer.ist.psu.edu/540106.html)
- [11] K. Roy and M. Johnson, “Software design for low power,” 1996. [citeseer.ist.psu.edu/roy96software.html](http://citeseer.ist.psu.edu/roy96software.html)
- [12] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: a first step towards software power minimization,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 437–445, 1994. [citeseer.ist.psu.edu/tiwari94power.html](http://citeseer.ist.psu.edu/tiwari94power.html)
- [13] V. Tiwari, S. Malik, A. Wolfe, and M. Lee, “Instruction level power analysis and optimization of software,” 1996. [citeseer.ist.psu.edu/tiwari96instruction.html](http://citeseer.ist.psu.edu/tiwari96instruction.html)
- [14] V. Tiwari, S. Malik, and A. Wolfe, “Compilation techniques for low energy: An overview,” 1994. [citeseer.ist.psu.edu/tiwari94compilation.html](http://citeseer.ist.psu.edu/tiwari94compilation.html)
- [15] G. Quan and X. Hu, “Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors,” in *Design Automation Conference*, 2001, pp. 828–833. [citeseer.ist.psu.edu/quan01energy.html](http://citeseer.ist.psu.edu/quan01energy.html)

- [16] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Design Automation Conference*, 1999, pp. 134–139. [citeseer.ist.psu.edu/shin99power.html](http://citeseer.ist.psu.edu/shin99power.html)
- [17] C. Rusu, R. Xu, R. Melhem, and D. Moss, "Energy-efficient policies for request-driven soft real-time systems." [citeseer.ist.psu.edu/690714.html](http://citeseer.ist.psu.edu/690714.html)
- [18] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *HPCA*, 2001, pp. 185–196. [citeseer.ist.psu.edu/tune01dynamic.html](http://citeseer.ist.psu.edu/tune01dynamic.html)
- [19] J. Seng, E. Tune, and D. Tullsen, "Reducing power with dynamic critical path information." [citeseer.ist.psu.edu/seng01reducing.html](http://citeseer.ist.psu.edu/seng01reducing.html)