# Compiler-Directed Dynamic Frequency and Voltage Scheduling*

Chung-Hsing Hsu[1], Ulrich Kremer[1], and Michael Hsiao[2]

[1] Department of Computer Science, Rutgers University
Piscataway, New Jersey, USA
{chunghsu, uli}@cs.rutgers.edu
[2] Department of Electrical and Computer Engineering, Rutgers University
Piscataway, New Jersey, USA
mhsiao@ece.rutgers.edu

**Abstract.** Dynamic voltage and frequency scaling has been identified as one of the most effective ways to reduce power dissipation. This paper discusses a compilation strategy that identifies opportunities for dynamic voltage and frequency scaling of the CPU without significant increase in overall program execution time. The paper introduces a simple, yet effective performance model to determine an efficient CPU slow-down factor for memory bound loop computations. Simulation results of a superscalar target architecture and a program kernel compiled at different optimizations levels show the potential benefit of the proposed compiler optimization. The energy savings are reported for a hypothetical target machine with power dissipation characteristics similar to Transmeta's Crusoe TM5400 processor.

## 1 Introduction

Modern architectures have a large gap between the speeds of the memory and the processor. Several techniques exist to bridge this gap, including memory pipelines (outstanding reads/writes), cache hierarchies, and large register sets. Most of these architectural features exploit the fact that computations have temporal and/or spatial locality. However, many computations have limited locality, or even no locality at all. In addition, the degree of locality may be different for different program regions. Such computations may lead to a significant mismatch between the actual machine balance and computation balance, typically resulting in long stalls of the processor waiting for the memory subsystem to provide the data.

We will discuss the benefits of compile-time voltage and frequency scaling for single loop nests. The compiler not only generates code for the input loop, but also assigns a clock-frequency and voltage level for its execution. The goal of this new compilation techniques is to provide close to the same overall execution

* This research was partially supported by NSF CAREER award CCR-9985050 and a Rutgers University ISC Pilot Project grant.

time while significantly reducing the power dissipation of the processor and/or memory. The basic idea behind the compilation strategy is to slow down the CPU that otherwise would stall or be idle. Frequency reduction and voltage reduction may lead to a linear and quadratic decrease of power consumption, respectively. In addition, recent work by Martin et al. has shown that reducing peak power consumption can substantially prolong battery life [24].

### 1.1 The Cost Model

The dominant source of power consumption in digital CMOS circuits is the dynamic power dissipation ($P$), characterized by

$$P \propto CV^2 f$$

where $C$ is the effective switching capacitance, $V$ is the supply voltage, and $f$ is the clock speed [6]. Since power varies linearly with the clock speed and the square of the voltage, adjusting both can produce cubic power reductions, at least in theory. However, reducing the supply voltage requires a corresponding decrease in clock speed. The maximum clock speed for a supply voltage can be estimated as

$$f_{max} \propto \frac{(V - V_T)^\alpha}{V}$$

where $V_T$ is the threshold voltage ($0 < V_T < V$), and $\alpha$ is a technology dependent factor ($1 \leq \alpha \leq 2$). Despite the non-linearity between clock speed and supply voltage, scaling both supply voltage and clock speed will produce at least quadratic power savings, and as a result quadratic energy savings. Figure 1 gives the relation between clock speed, supply voltage, and power dissipation for Transmeta's Crusoe TM5400 microprocessor as reported in its data sheet [33]. For a program running for a period of $T$ seconds, its total energy consumption ($E$) is approximately equal to

$$E = P_{avg} * T$$

where $P_{avg}$ is the average power consumption.

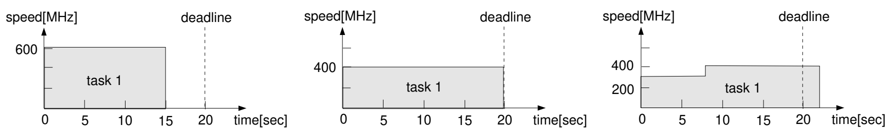
| Frequency (MHz) | 70 | 100 | 200 | 300 | 350 | 400 | 500 | 600 | 700 |
|---|---|---|---|---|---|---|---|---|---|
| Voltage (V) | 0.90 | 0.95 | 1.10 | 1.25 | 1.33 | 1.40 | 1.50 | 1.60 | 1.65 |
| Power (%) | 3.0% | 4.7% | 12.7% | 24.6% | 32.5% | 41.1% | 59.0% | 80.6% | 100% |

**Fig. 1.** The relation between clock frequency, supply voltage, and power dissipation of Transmeta's Crusoe TM5400 microprocessor. The voltage figures for frequencies 100MHz and 70MHz are interpolations and are not supported by the chip.

### 1.2 Voltage Scheduling

In the context of dynamic voltage scaled microprocessors, *voltage scheduling* is a problem that assigns appropriate clock speeds to a set of tasks, and adjusts the voltage accordingly such that no task misses its predefined deadline while the total energy consumed is minimized. Researchers have proposed many ways of determining "appropriate" clock speeds through on-line and off-line algorithms [34, 14, 13, 16, 28]. The basic idea behind these approaches is to slow down the tasks as much as possible without violating the deadline.

This "just-in-time" strategy can be illustrated through a voltage scheduling graph [27]. In a voltage scheduling graph, the X-axis represents time and the Y-axis represents processor speed. The total amount of work for a task is defined by the area of the task "box". For example, task 1 in Figure 2 has a total workload of 8,000 cycles. By "stretching" it out all the way to the deadline without change of the area, we are able to decrease the CPU speed from 600MHz down to 400MHz. As a result, 23.4% of total (CPU) energy may be saved on a Crusoe TM5400 processor.




(a) original schedule. (b) voltage scaled schedule. (c) energy-performance tradeoffs.

**Fig. 2.** The essence of voltage scheduling.

In case of a soft deadline, energy can be saved by trading off program performance for power savings. For example, if the voltage schedule starts with a 300MHz clock speed for 8 seconds and then switches to 400MHz for 14 seconds, the resulting execution time for task 1 is 22 seconds. The schedule pays 10% of performance penalty (with respect to the deadline), but it saves 29.1% of total energy as compared to the 600MHz case. These estimates assume that there is no performance penalty for the frequency and voltage switching itself.

### 1.3 Our Contributions

We propose a simple compile-time model to identify and estimate the maximum possible energy savings of dynamic voltage and frequency scaling under the constraint that overall program execution times may only be slightly increased, or not increased at all. In many cases, a compiler is able to predict and shape the future behavior of a program and the interaction between large program segments, giving compilers an advantage over operating systems techniques. Typically, operating system techniques rely on the observed past program behavior within a restricted time window to predict future behavior. Preliminary simulation results show the effectiveness of our model for optimized and unoptimized loops. The impact of various compiler optimizations on energy savings is discussed. In summary, we propose a simple model and new compilation strategy for dynamic voltage and frequency scaling.

The rest of the paper is organized as follows: Section 2 presents the simple model. Using a single simple benchmark, the effect of various compiler optimizations is illustrated in Section 3. Section 5 gives a brief summary of related work, and Section 6 concludes the paper.

## 2 Compiler-Directed Frequency Scaling

Consider the simple C program kernel in Figure 3(a). The loop scans a two-dimensional array in column-major order, and has no temporal locality (i.e., each array element is referred only once). Array size `n` is carefully chosen so that no spatial locality is present across iterations of the outermost `j`-loop. The loop will have spatial locality (i.e., successively accessed array elements reside in the same cache block) only if the array is scanned in row-major order.

Suppose the program is executed on a hypothetical superscalar machine with out-of-order execution, non-blocking loads/stores, and a multi-level memory hierarchy.[1]

The graphs shown in Figures 3(b) and (c) illustrate the opportunities for dynamic frequency scaling for our program kernel. The unoptimized version is heavily memory-bound, allowing a potential processor slow-down of up to a factor of 20 without a significant performance penalty. Figure 3(b) shows several scaled clock speeds whose relative performance is very close to 100%. These scaled speeds are 1/2, 1/5, 1/10, and 1/20 and result in performance penalties of less than 1%. If we are able to identify these scaled speeds, CPU energy consumption can be reduced by more than one order of magnitude for our target architecture, assuming it has an energy characteristics similar to that of a Crusoe TM5400 processor.

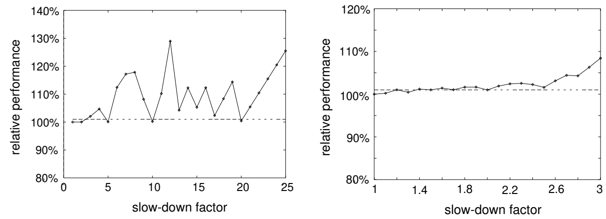
Using advance optimizations such as loop interchange, loop unrolling, and software prefetching, the computation/memory balance of the example loop can be significantly improved. Given that our target architecture has an L2 block size of 64 bytes and a single bank memory with a latency of 100 cycles, Figure 3(c) shows the best performance possible for the code, i.e., the performance is only limited by the physical memory bandwidth of the architecture.[2] Even for this best case scenario, there is still significant opportunity for voltage and frequency scaling with a performance penalty of less than 1%.

Both examples show that choosing the right slow-down factor is crucial to achieve energy savings with minimal performance penalties. In fact, increasing the slow-down factor may actually result in overall performance improvements,

[1] More details regarding our target machine can be found in Section 4.
[2] More details regarding the performed optimizations can be found in Section 3.

```
float A[n][n], accu

for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    accu += A[i][j];
```




(a) A simple loop. (b) No loop optimizations, and no data locality. (c) Highly optimized: loop interchange & unrolling, software prefetching.

**Fig. 3.** A simple C program and its unoptimized and optimized performance under different CPU clock frequencies. The horizontal lines indicate the threshold for a 1% performance degradation.

a somewhat non-intuitive result. This behavior can be attributed to synchronization effects between memory and CPU.

### 2.1 A Simple Model

We divide the total program execution time ($T$) into three portions:

$$T = cpuBusy + memBusy + bothBusy$$

with the right-hand side entities defined as follows:

- the CPU is busy while the memory is idle (*cpuBusy*); this includes CPU pipeline stalls due to hazards,
- the memory is busy and the CPU is stalled while waiting for data from memory (*memBusy*), and
- CPU and memory are both active at the same time, i.e., are working in parallel (*bothBusy*).

Consider now the CPU speed is reduced by a factor of $\delta$. Assume that the program in the reduced clock speed behaves exactly the same for every program step as the program in the normal speed, but only executed in "slow motion".[3] The new, slowed-down execution time becomes

$$T_{new}(\delta) = \delta * cpuBusy + \max \begin{pmatrix} bothBusy + memBusy \\ \delta * bothBusy \end{pmatrix}$$

In order to have the new execution time very close to the original one, for instance $T_{new}(\delta)/T \leq 101\%$, $\delta * cpuBusy$ needs to be very close to $cpuBusy$,

[3] This may not be the case in practice, for instance due to out-of-order instruction execution.

and also $\delta * bothBusy$ cannot be too large. Based on the model, we propose two conditions

$$(\delta - 1) * cpuBusy \leq 1\% \tag{1}$$

$$1 \quad \leq \quad \delta \leq 1 + \frac{memBusy}{bothBusy} \tag{2}$$

such that once they are satisfied, the new execution time is within 1% of the original execution time.

Condition (1) indicates that *cpuBusy* can be used as a measure of performance penalty. When it is relatively large, the clock speed cannot be slowed down without hurting the performance significantly. In addition, Condition (2) states that the slow-down factor cannot be arbitrarily large. How much it can be slowed down is highly dependent on how much the CPU is stalling due to memory requests. The more CPU stalls, the slower we can set the CPU speed. For the unoptimized loop in Figure 3(b), simulation produces the following figures:

| *cpuBusy* | *memBusy* | *bothBusy* |
|---:|---:|---:|
| 0.01% | 93.99% | 6.00% |

From Condition (1) and (2), it can be derived that $\delta \leq 1 + 1\%/cpuBusy = 1 + 1\%/0.01\% = 101$ and $\delta \leq 1 + 93.99\%/6.00\% = 16.67$, respectively. Since both conditions need to be satisfied, the maximum slow-down factor suggested by the simple model is 17. In other words, the clock speed can be reduced to as much as 1/17 without more than 1% performance penalty.

However, as observed in Figure 3(b), not all CPU speed reductions ($\leq 1/17$) result in a $\leq$1% or less performance slow-down. For example, execution time increases by 30.0% when the clock speed is set to 1/12 of the original speed. The reason for this significant performance decrease is the mismatch of the memory and CPU cycle times, resulting in clock skew effects during synchronization. Our model takes this effect into account by introducing a third condition:

$$\text{memory latency is divisible by } \delta \tag{3}$$

Finally, in order to simplify the analysis, we require the forth condition:

$$\delta \text{ has an integral value} \tag{4}$$

As a result, the model correctly identifies speed reductions 1/2, 1/4, 1/5, and 1/10 that satisfy the deadline constraint. However, the speed reduction by 1/20 is not suggested by our model. Possible reasons include the imprecision of CPU and memory workload prediction and the "ideal-world" assumption that program behavior remains the same under different clock speeds. For the optimized code of Figure 3(c), our model selects the slow-down factor $\delta = 2$.

(1) Identify program regions as scheduling candidates

(2) Model expected performance
   ($a$) Determine *cpuBusy*, *memBusy*, and *bothBusy*
   ($b$) Compute slow-down factor $\delta$ using model discussed in Section 2.1

(3) Generate voltage/frequency scheduling instructions for each scheduling candidate; adjust performance optimizations, if necessary

**Fig. 4.** Outline of basic compilation approach

### 2.2 Basic Compilation Strategy

The basic compilation strategy is show in Figure 4. The granularity of scheduling candidates has to be large enough to compensate for the overhead of voltage and frequency adjustments. Each scheduling candidate will be assigned a single voltage and frequency, allowing dynamic changes of voltage and frequency only between scheduling candidates. Initially, we will consider loop nests as scheduling candidates that will be analyzed and assigned a frequency and voltage. Possible identification of such candidate loop nests include the phase definition introduced by Kennedy and Kremer in the context of automatic data layout [21].

Different strategies can be used to determine values for the our model parameters *cpuBusy*, *memBusy*, and *bothBusy*. Static compile-time analysis, on and off line performance monitoring, or a combination of static and dynamic techniques are currently under investigation. For this discussion, we assume that the values of the three model parameters are available. The main focus of this paper is the discussion of a model that is able to select a suitable slow-down factor $\delta$ for a deadline constraint $d_s$, given values for *cpuBusy*, *memBusy*, and *bothBusy*.

The third and last compilation step will insert frequency and voltage adjustment instructions before each scheduling candidate, i.e., before each candidate loop nest. Assuming that the overhead of switching relative to the computation within a single loop nest is so small that it can be ignored, the collection of solutions for individual loop nests will represent an optimal solution for the entire program. We are currently investigating scheduling candidates of a finer granularity where the switching overhead is significant. In this case, an optimal frequency and voltage assignment requires multiple scheduling candidates to be considered at the same time.

## 3 The Impact of Compiler Optimizations

In the following, some of the advanced memory hierarchy optimizations will be used to demonstrate the impact of performance-oriented optimizations on the possibility of slowing down the CPU speed without noticeable penalty. Such optimizations can either reduce the number of memory references (e.g.: loop interchange, loop tiling), or hide the memory latency (e.g.: loop unrolling, software prefetching).

### 3.1 Techniques to Optimize Workloads

At the beginning of Section 2, it is mentioned that the array is scanned in column-major order while C uses row-major order. As a consequence, the program does not have any data locality. Loop interchange can be used to access consecutive rows of the array, resulting in spatial locality and a reduction in the total amount of memory accesses. Figure 5(a) gives the transformed program through loop interchange and its model parameters.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    accu += A[i][j];
```

(a) Transformed loop.

| | |
|---|---|
| *cpuBusy* | 18.93% |
| *memBusy* | 73.66% |
| *bothBusy* | 7.41% |

(b) Its workload.

**Fig. 5.** The impact of loop interchange.

Loop interchange effectively reduces workload of both CPU and memory. Since spatial locality is exploited, total memory work is reduced to 1/16 (every 16 `j`-iteration will generate a memory access). At the same time, sequential memory access pattern simplifies the address computations, and, as a result, 20% of pure computations (in instructions) are eliminated. In addition, instructions can be grouped more efficiently, and total CPU work (in cycles) is reduced to 1/2. As a result, the transformed program speeds up by a factor of 10.73.

However, according to our model, the transformed program is not a good candidate for slowing down the CPU without noticeable performance impact; *cpuBusy* (18.93%) is too large to satisfy Condition (1). The source of the problem is that the work of the CPU and memory has little overlap. A careful examination of the execution trace reveals that since the resources (RUU units) are used up very quickly, only a few `j`-iterations are issued, and then the fetch process is stalled until data arrives from memory. Once these `j`-iterations are executed, a few more `j`-iterations need to be executed before a new memory request is made.

Resources are used up very quickly because of the associated overheads in every `j`-iteration. Loop unrolling may be able to alleviate the problem by reducing the iteration overheads. Figure 6(a) gives the transformed program with the `j`-loop unrolled 16 times. As a result, many more `j`-iterations (24 to be exact) can be issued before the memory access is completed. New memory requests are made before the old memory access is done. In other words, loop unrolling has the effect of "implicit" data prefetching in our hypothetical machine. This explains why *cpuBusy* is so small (0.67%).

As discussed in the previous section, loop unrolling is designed to reduce CPU workload but it has the side-effect of workload overlapping through "implicit" data prefetching. Data prefetching can be done explicitly as well. The intention

```
for (i=0; i<n; i++)
  for (j=0; j<n-15; j+=16) {
    accu += A[i][j];
    ...
    accu += A[i][j+15]; }
```

(a) Transformed loop.

| | |
|---|---|
| *cpuBusy* | 0.67% |
| *memBusy* | 65.60% |
| *bothBusy* | 33.73% |

(b) Its workload.

**Fig. 6.** The impact of loop unrolling.

is to prefetch needed data to avoid CPU interlocks. Figure 7(a) shows a version of the transformed code.

```
for (i=0; i<n; i++) {
  prefetch A[i][0]
  for(j=0; j<n-16; j+=16) {
      prefetch A[i][j+16]
      accu += A[i][j];
      ...
      accu += A[i][j+15]; } }
```

(a) Transformed loop.

| | |
|---|---|
| *cpuBusy* | 0.67% |
| *memBusy* | 74.04% |
| *bothBusy* | 25.29% |

(b) Its workload.

**Fig. 7.** The impact of software data prefetch.

The explicitly data-prefetched program has similar workload distribution as the implicit version. Both allow the CPU speed to be reduced to 1/2 with only less than 1% of performance penalty. Figure 8 gives a summary of the relative performance, possible slow-down factors, and potential energy consumption for different versions of the optimized code.

The results show that even at the highest optimization levels, dynamic voltage and frequency scaling can achieve energy savings of 35% over the fastest, fully optimized version without any significant performance penalty ($< 1\%$). For the unoptimized case, the 70% energy savings are obtained without any performance degradation.

### 3.2 Relationship with Program Balance

The concept of *balance* ($\beta$) has been defined in a number of studies (e.g., [11, 12, 25, 17, 2]) as a ratio of the number of memory operations $M$ to the number of floating-point operations $F$, i.e., $\beta = M/F$. When applied to a particular machine, $\beta$ can indicate either peak [11, 12] or "average" [25] machine performance. Similarly, every program (or loop) has its own balance value $\beta_p$, which may take

| | $\delta$ | $T$ | $f$ | $V$ | $E$ |
|---|---|---|---|---|---|
| unoptimized | 1 | 100.00% | 700 | 1.65 | 100.00% |
| | **10** | 100.00% | 70 | 0.90 | 29.75% |
| inter- | **1** | 9.32% | 700 | 1.65 | 9.32% |
| change | 2 | 11.84% | 350 | 1.33 | 7.69% |
| unroll | 1 | 6.37% | 700 | 1.65 | 6.37% |
| | **2** | 6.43 % | 350 | 1.33 | 4.18% |
| prefetch | 1 | 6.37% | 700 | 1.65 | 6.37% |
| | **2** | 6.44% | 350 | 1.33 | 4.18% |

**Fig. 8.** The impact of optimizations on performance and energy consumption: $T$ is the relative execution time performance over the original, unoptimized code; $\delta$ is the $f$ and $V$ are the corresponding adjustment if running on an architecture with energy characteristics similar to the Crusoe TM5400 processor; $E \propto V^2T$ is the relative energy consumption over the unoptimized code. The values for $\delta$ selected by our model are shown in bold typeface.

into account pipeline interlock [11]. Optimization techniques are proposed to restructure a program so that its balance be closer to the underlying machine balance.

In terms of our model, program balance is a relationship between the work of the CPU and the memory, and their overlaps. When there is no overlap, slowing down CPU without significant performance penalties is not possible (*cpuBusy* is too large). This situation is already illustrated in Figure 5. On the other hand, when the work of CPU and memory overlaps almost perfectly, the program can be either cpu-bound or memory-bound.

The loops in Figure 6 and 7 are considered memory-bound. Simulation results show that 16 `j`-iterations take 32 cycles in total, assuming a perfect cache. Since the memory latency is 100 cycles, it cannot be be fully hidden in 16 `j`-iterations. On the other hand, CPU work is almost "embedded" in memory work. Memory-boundness may create opportunities for reducing CPU clock speed with negligible performance impact.

Some workload reduction transformations change the program from memory-bound to cpu-bound. For instance, loop fusion combines the bodies of multiple loops into a single loop. It not only reduces loop overheads and memory accesses, but also increases CPU work relatively more than memory work per iteration. According to our model, such cpu-bound programs cannot be slowed down without significant performance penalty. On the other hand, memory accesses can be slowed down without affecting the total performance.

## 4 Experiments

All simulations are done through the SimpleScalar tool set [9], version 3.0a, with memory hierarchy extensions [10]. SimpleScalar provides a cycle-accurate simulation environment for modern out-of-order superscalar processors with 5-stage pipelines and fairly accurate branch prediction mechanism. Speculative execution is also supported. The processor core contains a Register Update Unit (RUU) [30] which acts as a unified reorder buffer, issue window, and physical register file. Separate banks of 32 integer and floating point registers make up the architected register file and are only written on commit.

The processor's memory system employs a load/store queue (LSQ). It supports multi-level cache hierarchies and non-blocking caches. The extensions add a one-level page table, finite miss status holding registers (MSHRs) [23], and simulation of bus contention at all levels, but not simulation of page hits, precharging overhead, refresh cycles, or bank contention.

### 4.1 Simulation Parameters

The baseline processor core includes the following: a four-way issue superscalar processor with a 64-entry issue window for both integer and floating point operations, a 32-entry load/store queue, commit bandwidth of four instructions per cycle, a 256-entry return address stack, and an extra branch misprediction of 3 cycles.

In the memory system, we use separate 32KB, direct-mapped level-one instruction and data caches, with a 512KB, direct-mapped, unified level-two cache. The L1 caches have 32-byte blocks, and the L2 cache has 64-byte blocks. The L1/L2 bus is 256 bits wide, requires one cycle for arbitration, and runs at the same speed as the processor core. Each cache contains eight MSHRs with four combining targets per MSHR. The L2/memory bus is 128 bits wide, requires one bus cycle for arbitration, and runs 1/4 of the processor core speed. Figure 9 summarizes the simulation parameters used in the paper.

### 4.2 Dynamic Voltage Scaling Capability

The current implementation of the SimpleScalar tool set does not support dynamic frequency scaling. Our simulation is done by multiplying the total number of CPU cycles and the slow-down factor. For example, if the clock speed of the baseline processor is reduced by half, the latencies of the memory and L2/bus are reduced by half. The total number of CPU cycles is then multiplied by two to get the absolute performance. We are in the process of extending the SimpleScalar instruction et to support dynamic speed setting.

The energy estimation of the program is not yet incorporated into our version of the SimpleScalar simulator. In the future we will implement energy accounting as suggested by Wattch [5], which is based on SimpleScalar version 3.0 and publicly available.

### 4.3 Experimental Results

In addition to measurements for the accumulator loop shown in Figure 3(a) with results shown in Figure 8, we evaluated our model for the two BLAS1

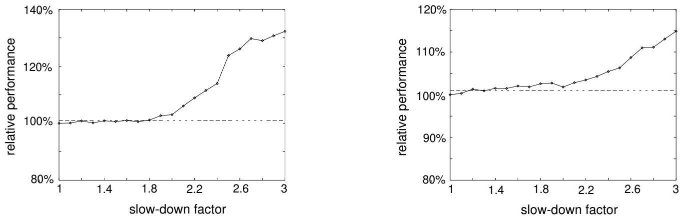
| Simulation parameters | Value |
|---:|:---|
| fetch width | 4 instructions/cycle |
| decode width | 4 instructions/cycle |
| issue width | 4 instructions/cycle, out-of-order |
| commit width | 4 instructions/cycle |
| RUU size | 64 instructions |
| LSQ size | 32 instructions |
| FUs | 4 intALUs, 1 intMULT, 4 fpALUs, 1 fpMULT, 2 memports |
| branch predictor | gshare, 17-bit wide history |
| L1 D-cache | 32KB, 1024-set, direct-mapped, 32-byte blocks, LRU, 1-cycle hit, 8 MSHRs, 4 targets |
| L1 I-cache | as above |
| L1/L2 bus | 256-bit wide, 1-cycle access, 1-cycle arbitration |
| L2 cache | 512KB, 8192-set, direct-mapped, 64-byte blocks, LRU, 10-cycle hit, 8 MSHRs, 4 targets |
| L2/mem bus | 128-bit wide, 4-cycle access, 1-cycle arbitration |
| memory | 100-cycle hit, single bank, 64-byte/access |
| TLBs | 128-entry, 4096-byte page |
| compiler | gcc 2.7.2.3 -O3 |

**Fig. 9.** System simulation parameters.

kernels `sdot` and `saxpy`. Both codes were optimized by hand using advanced transformations such as loop unrolling, loop splitting, software pipelining and software prefetching. The resulting code versions were compiled using `gcc -O3`. The following table lists the measured values for the three model parameters *cpuBusy*, *memBusy*, and *bothBusy*, and the resulting slow-down factor $\delta$ as computed by our model.

| | `sdot` | `saxpy` |
|---|---:|---:|
| *cpuBusy* | 0.19% | 0.92% |
| *memBusy* | 73.53% | 85.88% |
| *bothBusy* | 26.27% | 13.20% |
| $\delta$ | 2 | 2 |

The performance of the optimized versions of the two kernels under different slow-down factors is reported in Figure 10. As in the case of the optimized accumulator kernel, the graphs show a performance behavior that is nearly constant for small values of $\delta$, until the performs starts to degrade close to linearly with the slow-down factor. Figure 11 shows the performance characteristics of the optimized codes and the computed CPU slow-down factor $\delta$. In both cases, the model determines $\delta = 2$. For `sdot`, the resulting energy saving is 33% with a 3% performance penalty relative to the optimized version running at full CPU speed. For `saxpy`, these figures are 34% energy savings and 1.9% performance




(a) Highly optimized `sdot`: loop splitting and unrolling, software pipelining & prefetching.

(b) Highly optimized `saxpy`: loop unrolling, software pipelining & prefetching.

**Fig. 10.** Two simple BLAS1 kernels and their optimized performance under different CPU clock frequencies. The horizontal lines indicate the threshold for a 1% performance degradation.

penalty. For the unoptimized code versions, the model was not able to determine slow-down factors greater than 1.

| | $\delta$ | $T$ | $f$ | $V$ | $E$ |
|---|---|---|---|---|---|
| unoptimized | **1** | 100.00% | 700 | 1.65 | 100.00% |
| | | `sdot` | | | |
| optimized | 1 | 75.18% | 700 | 1.65 | 75.18% |
| | **2** | 77.45% | 350 | 1.33 | 50.32% |
| | | `saxpy` | | | |
| optimized | 1 | 93.18% | 700 | 1.65 | 93.18% |
| | **2** | 94.92% | 350 | 1.33 | 61.67% |

**Fig. 11.** The impact of optimizations and computed CPU slow-down on performance and energy consumption for `sdot` and `saxpy`. The values for $\delta$ selected by our model are shown in bold typeface.

The experiments show that our model was not able to precisely predict the performance penalty as a result of the CPU slow-down. In our measured cases, this imprecision has no significant impact. However, we are currently investigating refinements to our model that will include the basic computation and memory access patterns of program regions, in particular loop nests.

## 5 Related Work

Extensive research on optimizing compilers has been carried out in the last few years [35, 26], mostly execution time oriented and for high-performance processors. Since battery-powered mobile computers are getting more popular, there is a growing interest in optimizing software for low power.

In general, most performance-oriented transformations will also improve the overall energy consumption of an application [32]. However, recent results indicate that optimization techniques such as loop tiling and data transformations may increase the energy usage of the datapath while reducing memory system energy, which leads to challenging trade-offs between prolonging battery life and limiting dissipated energy within a package [19, 20]. In the context of loop tiling, it has been found that the best tile size for the least energy consumed is different from that for the best performance [19].

Other recent research has shown that power-aware register relabeling algorithm can reduce the overall power consumption of the register file by up to 12% without any performance impact [36]. For handheld battery-powered devices, compiler-directed remote task execution can be an effective technique to save energy on the mobile device [22].

### 5.1 Energy Models

Just as performance-oriented compiler optimizations need performance models to evaluate various coding schemes, power-aware optimizations need power models.

Tiwari et al. [32] proposed to assign each instruction an energy cost and estimate total energy consumption of a software based on instructions. Along the same way, [4] proposed a functional decomposition of the activities accomplished by a generic microprocessor and exhibited generalization capabilities. In [29], a function-level power estimation methodology is proposed. With this method, microprocessor vendors can provide users the "power data bank" without releasing details of the core to help users get early power estimates and eventually guide power optimization.

A lot of attention has been given to the memory subsystem for its energy consumption. For example, [31, 15, 18] all proposed analytical models for the memory subsystem with various precision. A more precise model may possibly take into account the run-time access statistics, which can be derived analytically (as many classical optimizations already do) or through simulation [1]. The analytical energy models for buses are also proposed recently [37]. In addition, many simulators were built in the past few years to more precisely capture the energy consumption of the processor core. Wattch [5] and SimplePower [36] are two such examples. More details can be found in [3].

### 5.2 Dynamic Voltage Scaling

Recently, methods have been developed to dynamically control supply voltage to adopt to the program's execution behavior. For example, operating frequency can be set to the lowest possible for the program execution, and dynamically vary the voltage accordingly. This approach is used by Transmeta [33] and researchers at the University of California at Berkeley [8]. Another approach is to dynamically adjust a transistor's threshold voltage. The chip can also be divided into blocks, with independent supply voltage control for each block. If a block is not in use, its supply can be cut to save energy.

Dynamic voltage scaling does not come without overheads. For example, for a large voltage change, the transition can take as long as 520$\mu$s and consumes energy 130$\mu$J [7]. Such a long transition time suggests the coarse speed control and gradual speed settings.

## 6 Conclusion and Future Work

Dynamic frequency and voltage scaling is an effective way to reduce power dissipation and energy consumption of memory bound loops. Choosing a maximal CPU slow-down factor is a difficult problem if deadlines have to be met. This paper discussed a simple performance model that allows the selection of efficient slow-down factors . Experiments based on three numerical kernels and a simulator for an advanced superscalar architecture indicate the effectiveness of the new model. Assuming the power characteristic of Transmeta's Crusoe processor, the resulting energy savings of our compilation strategy are in the range of 33% to 70%. The results show that even for highly optimized code there is still significant room for additional energy savings by applying our power optimization strategy. More experiments will be needed to further validate our proposed compilation strategy.

The implementation of the proposed models and compilation techniques are currently underway. In addition, we are extending our model to deal with computation-bound loops that allow energy savings by slowing down the memory subsystem. Algorithms for partitioning the program into regions of fixed frequency and voltage assignments, with voltage and frequency transitions between them need to be developed. In this paper, we have concentrated on single loop nests with single voltage and frequency assignments. Future work will address global, whole program solutions that will consider the execution time and energy overheads of voltage and frequency scaling.

## Acknowledgements

The authors wish to thank Professor Doug Burger from the University of Texas at Austin for providing the SimpleScalar tool set with his memory extensions.

## References

1. R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design (ISLPED)*, pages 70–75, August 1998.
2. F. Bassetti. A lower bound for qualtifying overlap effects: An empirical approach. In *Submitted to International Performance, Computing, and Communications Conference*, 1998.

3. L. Benini and G. Micheli. System-level power optimization: Techniques and tools. *ACM Transactions on Design Automation of Electronic Systems*, 5:115–192, April 2000.
4. C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. An instruction-level functionality-based energy estimation model for 32-bits microprocessors. In *37th IEEE-Design Automation Conference (DAC)*, pages 346–351, June 2000.
5. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture (ISCA)*, June 2000.
6. T. Burd and R. Brodersen. Processor design for portable systems. *Journal of VLSI Signal Processing*, 13(2–3):203–222, 1996.
7. T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *International Symposium on Low Power Electronics and Design (ISLPED)*, July 2000.
8. T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage-scaled microprocessor system. In *IEEE International Solid-State Circuits Conference (ISSCC-2000)*, February 2000.
9. D. Burger and T. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin, June 1997.
10. D. Burger, A. Kägi, and M. Hrishikesh. Memory hierarchy extensions to SimpleScalar 3.0. Technical Report TR99-25, Department of Computer Science, University of Texas at Austin, April 1999.
11. D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.
12. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, November 1994.
13. J. Chang and M. Pedram. Energy minimization using multiple supply voltages. In *International Symposium on Low Power Electronics and Design (ISLPED-96)*, pages 157–162, August 1996. also published in IEEE Transaction on VLSI Systems 5(4): Dec 1997.
14. K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *the 1st ACM International Conference on Mobile Computing and Networking (MOBICOM-95)*, pages 13–25, November 1995.
15. P. Hicks, M. Walnock, and R. Owens. Analysis of power consumption in memory hierarchies. In *International Symposium on Low Power Electronics and Design (ISLPED-97)*, pages 239–242, 1997.
16. T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 197–202, August 1998.
17. L. John, V. Reddy, P. Hulina, and L. Coraor. Program balance and its impact on high performance RISC architectures. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 370–379, January 1995.
18. M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *Proceedings of 1997 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 143–148, 1997.
19. M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and H.S. Kim. Experimental evaluation of energy behavior of iteration space tiling. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 2000.

20. M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference (DAC)*, June 2000.
21. K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, July 1998.
22. U. Kremer, J. Hicks, and J. Rehg. Compiler-directed remote task execution for power management. In *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, October 2000.
23. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-81)*, pages 81–87, May 1981.
24. T. Martin and D. Siewiorek. The impact of battery capacity and memory bandwidth on cpu speed-setting: a case study. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'99)*, August 1999.
25. J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, December 1995.
26. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
27. T. Pering and R. Brodersen. Energy efficient voltage scheduling for real-time operating systems. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS-98), Working in Progress Section*, 1998.
28. T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 76–81, New York, August 1998. ACM Press.
29. G. Qu, N. Kawabe, K. Usami, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *Design Automation Conference (DAC)*, June 2000.
30. G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.
31. C. Su and A. Despain. Cache design trade-offs for power and performance optimization: A case study. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 63–68, 1995.
32. V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2/3):1–18, 1996.
33. http://www.transmeta.com Transmeta corporation.
34. M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *the 1st Symposium on Operating Systems Design and Implementation (OSDI-94)*, pages 13–23, November 1994.
35. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Co., 1996.
36. W. Ye, N. Vijaykrishna, M. Kandemir, and M.J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Design Automation Conference (DAC)*, June 2000.
37. Y. Zhang, R. Chen, and M.J. Irwin. System level interconnect modeling. In *Proceedings of the 11th IEEE International ASIC Conference*, September 1998.