Virtual-Machine Driven Dynamic Voltage Scaling

Vivek Haldar, Christian W. Probst, Vasanth Venkatachalam and Michael Franz

{vhaldar,cprobst,vvenkata,franz}@uci.edu
Department of Computer Science
University of California
Irvine, CA 92697-3425

ABSTRACT

In current DVS approaches, voltage scaling decisions are made statically at compile time, and/or dynamically at the OS level. While this has yielded excellent results for a wide range of applications, there is an even better solution for platform independent code (such as Java bytecode) that executes on virtual machines. Such virtual machines have fine-grained execution information about the actual workloads that run on them, as opposed to static compilers that at best have off-line profiling data from previous workloads. Based on their high-level model of the actual workload, virtual machines can make DVS decisions with high precision.

1. INTRODUCTION

Rapidly increasing chip densities and processor speeds have made energy dissipation a leading concern in computer design. The growing number of transistors in a chip causes significant heat increases that affect a chip's reliability and lifetime. Cooling mechanisms add to the packaging cost. Recent trends suggest that processor power consumption doubles every four years and cooling costs rise exponentially with heat increases [12]. At this alarming rate, a Pentium processor will have over a billion transistors and consume several hundred watts in the next decade. The future processors will be hotter than light bulbs and require energy management solutions more cost effective than the cooling fans processors use today.

The power consumption of a circuit depends on four factors: capacitance, switching activity, clock frequency and supply voltage. Accordingly, one can reduce power by reducing one or more of these variables. Power is linear in capacitance, activity and frequency, but quadratic in voltage. By reducing the supply voltage, one increases a circuit's delay linearly. Thus, supply voltage places an upper limit on clock frequency. Dynamic voltage scaling is based on this relationship between frequency and voltage. By reducing both of these quantities in tandem, it achieves a cubic power reduction. The motivation behind dynamic voltage scaling is that running tasks slower finishes the same amount of work as running them faster, but dissipates less energy (c.f. Figure 1). DVS algorithms must account for the performance overhead of slowing down the processor, including the overhead of switching between frequencies and voltages. As Figure 2 shows, these overheads increase execution time.

A number of processors support dynamic voltage scaling. Examples include Intel Speedstep [24], and Transmeta's Crusoe [6, 7]. Speedstep switches between two clock frequencies and automatically sets the voltage for each frequency. It uses a high frequency when relying on AC power, and a low frequency when relying on batteries. In contrast, Transmeta's Crusoe modulates frequency and voltage during the execution of applications through a software controlled feedback loop.

The rest of the paper is organized as follows: Section 2 motivates our approach of virtual machine based dynamic voltage scaling. Section 3 describes our online algorithm for driving dynamic voltage scaling; Section 4 gives details of our implementation of this algorithm in a Java virtual machine. Section 5 discusses our benchmarking methodology and presents our results. Section 6 surveys related work in the field and Section 7 discusses how our work differs from previous work. Finally, Section 8 summarizes our conclusions as well as plans for future work.

2. RATIONALE

Dynamic voltage scaling has recently attracted widespread attention in the lowpower community and can be done at a number of levels. These include the hardware level, operating system level, compiler level, virtual machine level, and application level. Nearly all DVS research has focused on the first three levels. Though the hardware level provides mechanisms for reducing frequency and voltage, it also needs information about program behavior to decide when to apply these mechanisms. Techniques for deriving this information are too expensive to implement in bare hardware. For these reasons, the hardware level lacks information on when to make DVS decisions.

Operating systems have more information, namely, about what programs are running and what resources they use. Thus, they can make DVS decisions based on CPU usage patterns. However, operating systems lack forward looking information about program behavior and are hence limited

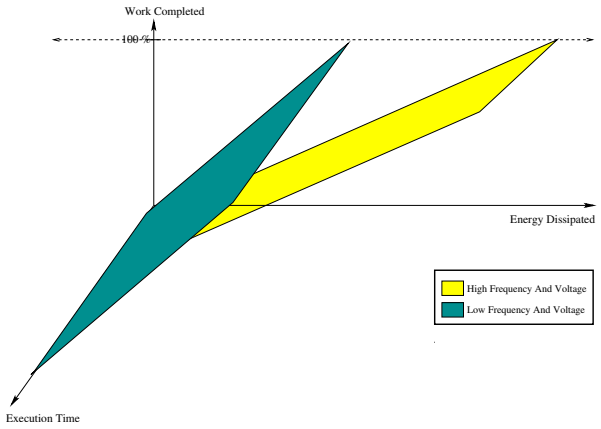

Figure 1: Completion of work vs execution time and dissipated energy

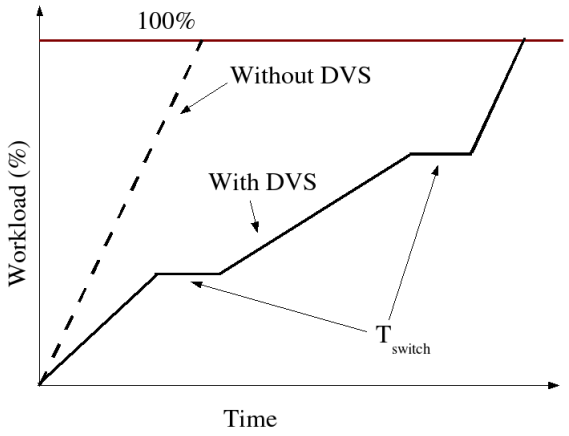

Figure 2: Time vs fraction of workload completed with and without DVS

to extrapolating future behavior from past behavior.

Compilers, however, receive an entire program as input. Thus, they can predict with greater accuracy the paths a program's execution will take. Compilers can make DVS decisions at a finer granularity than operating systems by inserting DVS instructions into program regions such as basic blocks. Nevertheless, statically optimizing compilers lack runtime information and often resort to exhaustive simulation or previously collected offline profiles to decide what program regions should slow down and how much they should slow down. Once made, these decisions remain fixed for a program's execution.

These static approaches suffice for application specific embedded devices, since the specialized software for these devices is burnt into the chip and unlikely to change its execution behavior. However, a plethora of emerging Java enabled mobile devices have less tightly coupled hardware and software. These devices include high end servers, desktop machines, laptops, set-top boxes, PDAs, pagers as well as cellphones. They present a new challenge not addressed by traditional static compilers. First, they require a level of indirection allowing software to be written once but run on any of them. Moreover, the applications running on these multipurpose devices vary continuously in execution behavior. Static analysis is less effective for managing the power consumption of these applications since it lacks runtime information and couples power management policies with specific program binaries. Whenever the program behavior changes, as it would, for example, when a program's inputs change at runtime, the analysis underlying these policies must be repeated. This is an infeasible task given the growing numbers of diverse architectures and runtime configurations.

Working at a higher abstraction level than compilers, virtual machines provide a layer of indirection between platform independent code and diverse architectures. Like compilers, virtual machines have a model of future program behavior and can thus make more accurate power management decisions than operating systems or bare hardware. However, unlike static compilers, virtual machines have an infrastructure allowing them to profile and reoptimize programs *in execution*. This dynamic optimization infrastructure allows virtual machines to continuously adapt power management decisions to varying execution behavior.

Finally, at the application level, programmers can make design decisions that reduce execution time and create opportunities for slowing down the processor. However, doing all of the analysis for DVS at the application level may place too much of a burden on programmers.

For these reasons, we propose a novel virtual machine based DVS algorithm that profiles a program online and adapts its decisions at runtime. To our knowledge, this is the first implementation of DVS in a virtual machine setting.

3. RUNTIME-PROFILE BASED DYNAMIC VOLTAGE SCALING

In this section we present our algorithm, both on an informal and a formal level. The approach will be clarified by an example.

Our algorithm is based on *runtime profiling* of bytecodes executing in a virtual machine. The goal is to *reduce* energy consumption by means of scaling down frequency, while at the same time *minimizing* performance loss.

3.1 An Example

Figure 3 shows the code of our running example. It contains a `main` method, calling several methods, which in turn call again methods.

In order to decide which methods should be executed at a lower frequency, the virtual machine constructs an *invocation tree*. An invocation tree is a *method callgraph* constructed at runtime, as methods are called. Each node represents a method, and its children are the callees of that method. Thus, at any instant, the invocation tree contains at least one node for every method that has already been called during execution. Figure 4 shows the final tree for our example program. As can be seen, methods `mB` and `mC` occur

```
main() {
   mA();
   mB();
   mA();
}

mA() {
   mB();
}

mB() {
   mC();
}

mC() {
}
```

Figure 3: The example program

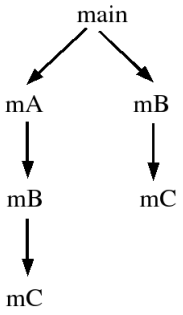

Figure 4: Invocation tree for the example program

time [ms]	method	comment	invocation tree	freq. [MHz]
0.00	main	invoke		1300
15.00	mA	invoke		1300
20.00	mB	invoke		1300
22.00	mC	invoke		1300
32.00	mC	return	$t_{mC_1} = 10$	1300
36.00	mB	return	$t_{mB_1} = 16$	1300
57.00	mA	return	$t_{mA} = 42$	1300
58.00	mB	invoke		
switching to 800MHz				
58.00	mB	invoke		800
61.25	mC	invoke		800
66.12	mC	return	$t_{mC_2} = 3$	800
67.75	mB	return	$t_{mB_2} = 6$	800
switching to 1300MHz				
70.75	mA	invoke		
switching to 1000MHz				
70.75	mA	invoke		1000
74.65	mA	return	$t_{mA} = 22.5$	1000
switching to 1300MHz				
75.00	main	return		1300

Figure 5: Execution of the example program

twice, since they are called by two different methods, namely `main` and `mA`. By differentiating calls to a method in different contexts, our approach avoids mixing timing decisions that actually do not correlate. Each node n is associated with the cumulative time already spent in the method m_n represented by n and all methods transitively called by m_n. Initially, these times are all set to 0.

In order to measure the execution time of a method, the virtual machine wraps timing constructs around every method invocation. Thus, the VM records the cumulative time spent in the method and all methods called by it. Once a method has been called, its execution may be scaled down at the next invocation in the same context. Additionally, the first execution time of a method is cached for scaling decisions if the same method is called in a different context.

We will now describe the execution of the program in Figure 3 and show the decisions of our approach. Figure 5 shows the execution with hypothetical execution times. We assume the program to execute on a Pentium-M 1.3GHz processor. The supported frequencies are given in Table 7.

The execution of the example program starts by calling methods `main`, `mA`, `mB`, and `mC`. Since none of these methods has been called before, the frequency is left unchanged and the execution times are cached and stored in the invocation tree. After 58ms, method `mB` is invoked the second time. While the first invocation was in method `main`, the execution time of its previous invocation been cached and is used for prediction in the current context. After the execution of `mB`, the cached time is discarded. Let t_m denote time spent in method `m`. How should the VM decide whether to scale down the frequency for the next invocation of this method? When scaling frequency, we need to take into account that the system will require some time T_{switch} to stabilize after switching the frequency. Additionally, the system must be prevented from switching too often, since this stabilizing penalty is going to sum up. Thus, one requirement is that the expected execution time of the method. must be higher than $T_{switch} \times \alpha$, where α is a parameter in our framework. For the Pentium-M, the actual value is $T_{switch} \approx 10\mu$s. For the example we assume $\alpha = 5$ and $T_{switch} = 1$ms. Since $t_{mB} = 16\text{ms} > 5 \times 1\text{ms}$, the method `mB` qualifies for scaling.

Next, we need to calculate the frequency that the method shall be executed at. To do so, we must first predict the expected runtime of the application without scaling. We adopt the heuristic used for optimizations in the Jalapeno VM [5]: our algorithm assumes the projected time (T_{proj})to be twice the runtime of the application up to the invocation plus the expected execution time of the method.

For method `mB`, $T_{proj} = 2 \times 58\text{ms} + 16\text{ms} = 132\text{ms}$. The decision whether or not to scale down frequency is made by looping over the possible frequencies supported by the underlying architecture from *slow* to *fast*. For each frequency f we compute the expected execution time T_f^{app} of the program assuming the method is executed at f.

For example, T_{600}^{app} predicts the expected execution time of mB if run at 600MHz. It is computed as $T_{600}^{app} = 2 \times 58\text{ms} + 2 \times T_{switch} + T_{600}^{mB} = 152.67ms$, where $T_{600}^{mB} = t_{mB} \times 1300\text{MHz}/600\text{MHz} = 34.67\text{ms}$. Since we want to minimize the actual performance impact of frequency scaling, the VM demands that the predicted execution time of the application be at most $T_{proj} \times (1 + \epsilon/100)$, where ϵ is the tolerable performance loss in percent. For the example we assume $\epsilon = 10$, thus the maximal acceptable time is 145.2ms.

Similarly, the execution times for higher frequencies are computed to be $T_{800}^{app} = 144\texttt{ms}$ and $T_{1000}^{app} = 138.8\texttt{ms}$. Thus, the VM will scale down the frequency to 800MHz, resulting in the expected execution time of mB to be $T_{800}^{mB} = 26\texttt{ms}$.

After mB returns, the frequency is reset to 1300MHz and execution of main is continued. After 70.75ms, mA is called for the second time. Based on $T_{proj} = 2 \times 70.75\texttt{ms} + 42\texttt{ms}$, the VM scales the execution down to 1000MHz, resulting in an expected execution time $T_{1000}^{mA} = 54.6\texttt{ms}$. Since this call to mA already terminates after 3.9ms, the time t_{mA} in the invocation tree is updated to $(3.9\texttt{ms} * 1000\texttt{MHz}/1300\texttt{MHz} + 42\texttt{ms})/2$. Thus, the times in the invocation tree resemble the actual execution time of the method if the processor had run at the highest possible frequency.

3.2 Algorithm

Based on past runtime for each method information, the VM decides whether to slow down the processor in future invocations of each method. The decision process for each method splits total runtime into runtime for the method and runtime for the rest of the application. When a method is slowed down, the total execution time increases due to two factors. The method's slowed down runtime replaces its original runtime. An additional overhead arises from changing processor speed before and after the method execution. Two constraints must be met for the slowdown to count as advantageous. First, the program's new runtime must be within a threshold of the original runtime. Moreover, the method's execution time must be significantly larger than the switching overhead. If these two conditions are met, the VM slows down the method on its next invocation.

Figure 6 gives a pseudo code description of our algorithm, describing how the information in the invocation tree is used. Before every method invocation, procedure *decide_switch* is called. There are two cases in which the processor frequency will definitely not be scaled down:

- the frequency has already been scaled down, or
- the penalty imposed by switching the frequency is of the same order as the expected runtime of the method.

The purpose of the first constraint is to avoid switching frequencies too often, while that of the second is to prevent the VM from slowing down methods with very short expected execution times. These restrictions also address the case when a method is initially executed, and its previous execution time is unknown.

Below is a formal explanation of the above algorithm. We use the following notation: for a method m, let

- T_{orig}^{m} be its original average execution time,
- T_{f}^{m} be its *projected* execution time if run at frequency f,
- T_{switch} be the time it takes to switch the frequency of the CPU and stabilize the system,

```
method current_method;
powerstate current_state;

procedure powertree_call(method callee)
    ts = timestamp();
    current_method → called = callee;
    current_method = callee;
    decide_switch(callee);
end

procedure decide_switch(method callee)
    if (already in power-saving state)
        return;
    if (time spent in current method < T_switch × α)
        return;
    T_orig^callee = callee → num_ticks;
    T_app = runtime of the application;
    T_proj = 2 × T_app + T_orig^callee;
    for (i = 0;i < (number of power-states);i++)
        T_powersave = T_freq_i^callee + 2 × T_app + 2 × T_switch
        if (T_powersave < T_proj * (1 + ε/100))
            switchto(i);
end
```

Figure 6: Performing Dynamic Voltage Scaling

- T_{app} be the total time spent executing the application so far, and
- ε be the slowdown we are willing to accept.

In order to estimate the overall application performance loss of scaling down a method, we need to predict how much of the application's future execution time will be taken up by a method, and for how long the application will run. We use a very simple heuristic for these predictions – we assume that the application's future execution time will be the same as the time it spent executing so far, and similarly, that the future execution time of a method will be the same as it's average execution time so far. Dynamic optimization systems such as Jalapeno[5] use similar prediction heuristics.

Then, the total projected application run time, T_{proj}, assuming no voltage and frequency scaling will be

$$T_{proj} = 2 \times T_{app} + T_{orig}^{m}$$

We make the simplifying assumption that execution time scales linearly with frequency. So, for example, if we halve the frequency f, then T_{f}^{m} will be twice of T_{orig}^{m}. This assumption is not very accurate, since, e.g. for memory bounded computations, the frequency could be reduced *without* observing linear slowdowns. However, it is conservative, and thus will not affect our performance measure adversely.

Thus, if the voltage and frequency were scaled down for this method, the expected total runtime of the application would be

$$T_{powersave} = T_{f}^{m} + 2 \times T_{app} + 2 \times T_{switch}$$

The term $2 \times T_{switch}$ accounts for the overhead of *two* voltage

scalings — scaling down at method invocation, and scaling back up when the method returns.

Then, we can switch the frequency down if the projected execution time of the whole application at the slower frequency is within the acceptable limit. That is, if

$$T_{powersave} < (1 + \varepsilon)T_{proj}$$

Also, we require

$$\frac{T_f^m}{T_{switch}} > \alpha$$

This means that the execution time of a method must be relatively large compared to the switching overhead.

4. IMPLEMENTATION

We have modified version 1.1 of the KVM [18], a Java virtual machine targeted for resource-constrained devices, to use our online voltage scaling algorithm. The KVM is the reference implementation of the Java Connected Limited Device Configuration (CLDC). It is a pure interpreter.

As explained in the previous section, runtime profiles of methods are maintained in a invocation tree structure, where each node represents a method, and the node's children represent its callees. The node maintains the cumulative time spent in that method, including its callees. This invocation tree is constructed at runtime. There is always a direct pointer to the node representing the currently executing method.

The modifications to the KVM are minor. At every method invocation and return, we needed to insert calls to appropriately manage the invocation tree and update its profiles – precisely one call for each invocation and return site. In the case of the Java bytecode instruction set, the relevant call instructions are `invokevirtual, invokespecial, invokestatic` and `invokeinterface`. The relevant return instructions are the variants of the `return` bytecode – `ireturn, lreturn, freturn, dreturn, areturn` and `return`. The code for profile management and voltage-scaling decisions was 446 lines of code (including comments), with 187 semicolons.

The implementation is neatly modular. Future work on the heuristic, the online algorithm, or profiling techniques will only involve changes to the invocation-tree and profile-management code, and not the KVM. [1] Since the modifications to the VM are minor, we expect to be able to port our online algorithm to other virtual machines easily.

For controlling voltage scaling, we use the `cpufreq` module of the Linux kernel version 2.6.0-test2. As is, it uses a file system interface for setting the frequency states of the processor. The procedure for changing the frequency is to open a special file and write a value (reflecting the new desired frequency) into it. This is inefficient for a large number of frequency switches. Therefore, we extended the module by a simple system call that switches the frequency.

[1]Unless we profile at a granularity smaller than methods.

Freq. (MHz)	Voltage (V)
600	0.956
800	1.260
1000	1.292
1200	1.356
1300	1.388

Figure 7: Voltage and frequency levels for the Pentium-M 1.3GHzprocessor

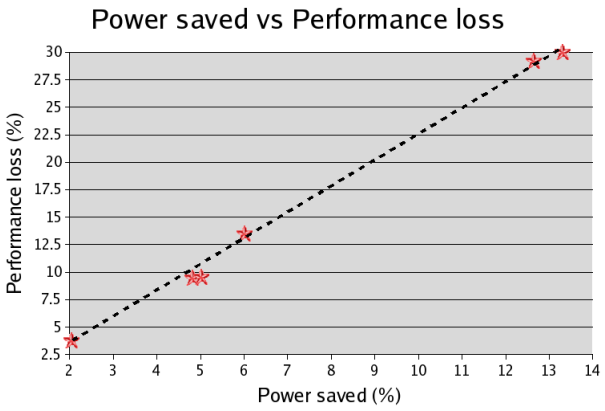

Figure 8: Plot of power saved vs. performance loss for benchmarks

5. RESULTS

We have evaluated our online voltage-scaling algorithm using a software-simulation methodology. All our benchmarks were run on a Dell Inspiron 600m laptop, with a Pentium-M 1.3GHz processor and 384MB of memory. The machine was running the Linux kernel version 2.6.0-test2. Using the hardware time-stamp counter on Intel x86, we keep track of the number of cycles (including the overhead for voltage switching) spent in each voltage and frequency level. From this, we derive the expected relative power dissipation (which is proportional to $f \times v^2$, for frequency f and voltage v), and the total execution time. The voltage and frequency levels of the Intel Pentium-M 1.3GHz processor are shown in figure 7. Thus, our performance numbers are real measurements, but power numbers are simulated.

We have used the Java Grande benchmark set. Table 9 contains our detailed results. Due to the limited class library of the KVM, we were unable to run all the benchmarks.

Currently, the tolerable slowdown (ε) and ratio of method runtime to switching overhead (α) are specified for each run, and remain fixed. Ideally, these would also be adjusted dynamically. Also, sometimes energy spent during longer execution times overcomes the gains of scaling down the voltage and frequency. We need a *bailout* strategy to dynamically detect when this is happening.

DVS reduces both frequency and voltage for the CPU, and thus provides a *cubic* energy reduction. Even though a program may run longer and hence consume more energy, the

Name	Power			Performance	Time (sec)		Switches	ε	α
	saved (%)	peak	scaled	loss	full speed	scaled			
Crypt	0.000072	131194.976801	131194.881793	0.000223	52.383549	52.383666	15	2	10
Crypt	2.054173	130780.209734	128093.758309	3.755870	52.217940	54.179178	17	5	10
Crypt	6.020241	130799.333938	122924.898643	13.524157	52.225576	59.288645	17	10	10
Crypt	6.020113	130800.685499	122926.336957	13.523869	52.226116	59.289107	17	15	10
Crypt	-0.000020	130771.124998	130771.150712	0.000020	52.214313	52.214323	1	2	50
Crypt	2.054167	130773.436400	128087.131957	3.755821	52.215236	54.176347	3	5	50
Crypt	6.020222	130802.199502	122927.616600	13.524117	52.226720	59.289923	3	15	50
Crypt	-0.000020	130771.594811	130771.620991	0.000020	52.214501	52.214511	1	2	100
Crypt	2.054101	130775.260000	128089.003911	3.755701	52.215964	54.177039	3	5	100
Crypt	6.020257	130803.299521	122928.604743	13.524195	52.227160	59.290463	3	15	100
LUFact	0.767535	142269.791409	141177.820356	1.676721	56.805502	57.757972	259229	2	10
LUFact	5.034122	143277.712293	136064.936750	9.552970	57.207945	62.673003	504479	5	10
LUFact	13.308955	143227.552746	124165.462235	29.998583	57.187917	74.343482	504479	10	10
LUFact	13.464788	143251.932831	123963.363760	30.640640	57.197652	74.723378	504481	15	10
LUFact	0.833207	143239.310957	142045.830872	1.808891	57.192612	58.227164	257169	2	100
LUFact	5.029016	143254.680388	136050.379929	9.541529	57.198749	62.656384	500259	5	100
LUFact	13.304407	143238.333392	124181.322666	29.987809	57.192222	74.342916	500265	10	100
LUFact	13.459671	143229.306315	123951.113603	30.627224	57.188617	74.703903	500287	15	100
HeapSort	1.196691	187556.183383	185311.715143	2.384677	74.887459	76.673283	3283727	2	100
HeapSort	4.828266	191235.208838	182001.864063	9.453061	76.356421	83.574440	4998369	5	100
HeapSort	12.675495	189048.205135	165085.409631	29.196093	75.483194	97.521337	4998369	10	100
HeapSort	13.663165	188915.633150	163103.778703	33.328114	75.430260	100.569743	4998365	15	100
HeapSort	1.226347	188076.072604	185769.607415	2.447878	75.095040	76.933275	3279601	2	1000
HeapSort	4.820182	191170.433856	181955.670841	9.433954	76.330557	83.531547	4983133	5	1000
HeapSort	12.668818	188979.720044	165038.223953	29.179061	75.455849	97.473157	4983385	10	1000
HeapSort	13.652001	188867.935241	163083.682191	33.296476	75.411215	100.520493	4983179	15	1000
FFT	0.264985	888018.539381	885665.424576	0.487534	354.568166	356.296805	16775979	2	100
FFT	0.530186	877730.433587	873076.830438	1.127928	350.460330	354.413271	16775979	5	100
FFT	0.951397	873662.176483	865350.178160	2.788181	348.835955	358.562132	16775981	10	100
FFT	0.955557	873758.172966	865408.918005	2.795901	348.874285	358.628465	16775987	15	100
FFT	0.260865	887040.427232	884726.446460	0.477003	354.177625	355.867063	16764633	2	1000
FFT	0.529118	877628.295721	872984.604172	1.125388	350.419548	354.363127	16764593	5	1000
FFT	0.950054	873550.765769	865251.559823	2.785106	348.791471	358.505683	16764599	10	1000
FFT	0.954384	873644.029434	865306.109102	2.793647	348.828710	358.573754	16764651	15	1000

Figure 9: Benchmark results

cubic saving due to DVS usually overcomes this, resulting in a net energy saving.

As expected, the results show that power saved is proportional to the performance loss (c.f. Figure 8). They also indicate up to a 13% saving in simulated processor power consumption with a performance loss of up to 33%.

Varying the ratio α of method runtime to switching overhead has little effect on the results. However, this ratio must be large enough to prevent the frequency from switching too often. During our benchmark runs we faced CPU crashes when this ratio was too low. This suggests a hardware limit on how often the frequency can be switched. To our knowledge, Intel has not documented this limit for the Pentium-M processor.

6. RELATED WORK

Dynamic voltage scaling has been explored at different granularity levels. These include the interval level, intertask level and intratask level.

6.1 Interval Level

At the largest granularity are *interval-based* policies that regularly adjust processor speed based on prior workloads. The simplest algorithm of this kind is PAST [26]. PAST adjusts CPU speed at fixed length intervals based on the idle and active cycles of the previous interval. If the idle cycles exceed a threshold, it slows down the processor. Else if the active cycles are higher, it speeds it up. As Govi et al. [10] point out, PAST uses a narrow window of past information to predict future workloads and changes the clock speed at every interval, increasing energy dissipation and cycles. Variations of PAST [10] address these shortcomings. Examples include *Aged Averages* which estimates future processor usage as a weighted average of usage in prior intervals and *Pattern*, which predicts future CPU usage to follow a past usage pattern.

The main similarity between interval based approaches and our approach is the use of online information to make predictions about future behavior. However, these predictions occur at a coarser granularity in interval based approaches. They are based on CPU usage alone and assume similar

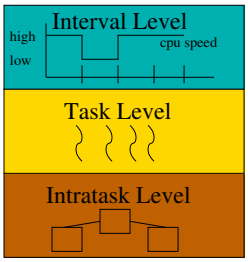

Figure 10: Granularities of Scheduling

workloads in every interval. Though our approach uses past information, it gathers this information at the method level. Thus, our approach can adapt its decisions to the varying execution profiles of individual methods, allowing substantially greater scheduling accuracy than interval based approaches.

6.2 Intertask Level

At a higher granularity are *intertask* policies that determine execution rates of *individual* tasks. The simplest example of an intertask policy is *Energy-priority scheduling* [21]. This policy maintains an even workload distribution as new tasks enter a system, to minimize battery drain rate. In every iteration, EPS schedules the task with furthest deadline and fewest overlapping tasks. It computes the minimum workload increase due to the new task and speeds up already scheduled tasks to make room and fill up slack. The real-time OS literature abounds with more intertask policies, many of which combine frequency scaling with traditional policies such as *Earliest Deadline First* (EDF) and *Rate Monotonic* (RM). Pillay and Shin [20] compare five such policies. The first two are static versions of EDF and RM. These policies run all tasks chosen by an EDF or RM scheduler at the single lowest frequency meeting all deadlines. In contrast, dynamic versions of these policies vary clock frequency for individual tasks and fall into two categories, *cycle conserving* and *lookahead*. Cycle conserving policies minimize total cycles by shifting most of a task's workload earlier in the schedule.

Intertask approaches achieve greater scheduling accuracy than interval based approaches by exploiting task level information. However, they fix the execution rates of individual tasks. Thus, even these approaches fail to achieve the granularity of our approach.

6.3 Intratask Level

Intratask approaches vary clock frequency and voltage within individual tasks and are most closely related to our approach. These approaches have been implemented in operating systems and compilers. OS-assisted intratask policies are in Dudani et al. [8], Zhu and Mueller [28] and Gruian [11]. To combine EDF scheduling with frequency scaling, Dudani et al. split each task the scheduler chooses into two subtasks, later running at full speed and the earlier running slower. They choose the earlier subtask's speed to keep the combined execution time of both subtasks below the average execution time for the whole task. Zhu and Mueller [28] examine preemption handling in the context of this scheme. Gruian proposes a more granular intratask policy that estimates an optimal clock speed for every cycle of a task's execution. His approach records the probability distribution of cycle times for a task. Given this distribution, measured over successive task invocations, it expresses the expected energy dissipation for a task in terms of the speed chosen for each execution cycle. It then derives the optimal speeds for each cycle analytically, allowing runtime modulation of task speed.

The above intratask policies are typical of OS assisted policies in using coarse-grain information to predict future execution patterns. Compilers have a model of future program behavior allowing more accurate scheduling decisions in critical program regions such as basic blocks. However, prior work in compiler assisted DVS, in contrast to our work, ignores runtime information that would yield even better decisions.

One of the earliest compiler-assisted DVS approaches is by Lee and Sakurai [17]. It chooses a target execution time for each of a task's (statically determined) timeslots that allows the task to finish within its worst case runtime. It then assigns the timeslot a clock frequency whose maximum runtime is within the target. While it does frequency and voltage scaling at runtime, it does all previous steps at compile time. Thus, the DVS decisions, once made, remain fixed, in sharp contrast to the dynamically varying decisions of our approach.

Related work by Hsu and Kremer [13] discusses how to select regions where DVS decisions should be made. The idea is to instrument a program with profiling code and execute the program to build a table of execution frequencies and average cycles for each region under all possible clock frequencies. Using this exhaustive approach, Hsu and Kremer select the region whose slowdown minimizes energy dissipation and incurs the smallest increase in runtime. In a separate work, Hsu et al. [14] use a similar exhaustive heuristic to determine how slow to run each selected region. Like Lee and Sakurai's work, this work also fixes DVS decisions. Moreover, it is highly input and architecture specific, as well as being too time consuming to implement in nontrivial programs.

In contrast, Shin et al. [23] propose an elegant approach exploiting a program's varying slack at runtime. They statically determine the worst case remaining execution cycles at each basic block and initially run the program fast enough to complete total worst case cycles within a deadline. When the program diverges to basic blocks whose remaining worst case cycles are significantly fewer than those on the worst case path, the clock frequency can be reduced. Shin et al. statically instrument these blocks with frequency reduction code. Though their algorithm allows frequency changes at runtime according to control flow, its analysis for these changes still happens at compile time. Moreover, it bases its decisions only on worst case execution cycles. Runtime information would allow more aggressive scaling of frequency and voltage based on live execution behavior as well as resource levels.

Most intratask approaches are implemented exclusively in a compiler or an operating system. A few researchers [19, 3, 2, 1]. have attempted to combine OS and compiler interaction.

Their approach statically splits a program into fixed length intervals, each beginning with a *power management point*. Prior to each point, the compiler inserts an instruction that saves the worst case remaining cycles into a register. At runtime, the power management points invoke an interrupt service routine that reads this register and adjusts processor speed accordingly. Like the previous approaches, this approach is dynamic only in the sense of allowing frequency and voltage to vary throughout a program's execution. It is static in performing the analysis for inserting power management points at compile time.

As well as combining operating systems and compilers in voltage scaling decisions, researchers are exploring how to combine scaling with traditional performance oriented compiler optimizations. Saputra et al.[22] discuss these issues. They first propose a static scheme that initially applies performance optimizations to reduce CPU and memory cycles and then scales the CPU voltage to lengthen execution time back to the original while reducing energy quadratically. Saputra et al. then generalize this scheme to an ILP formulation that allows runtime voltage modulation in different program regions. Their approach depends on energy cost tables constructed at compile time and is thus static, like all the others discussed above.

7. DISCUSSION

Prior techniques for compiler assisted voltage scaling are static since their analysis of when and how much to slow down the CPU occurs offline and remains fixed for a program's duration, even if it allows modulation of voltage and frequency at runtime. Power management techniques based on static analysis are suitable for application specific embedded systems (e.g., toaster oven).

However, mobile code is rapidly becoming ubiquitous in consumer electronics, already appearing in a proliferation of mobile devices including cellphones, set-top boxes and PDAs. These mobile code enabled devices pose a new challenge of developing power management policies that adapt to the continuously varying execution patterns of programs.

Our work addresses this challenge by performing dynamic voltage scaling at runtime in a Java virtual machine. As well as having a model of future program behavior, virtual machines have an infrastructure allowing runtime program profiling and reoptimization. This infrastructure allows virtual machines to make more fine-grained DVS decisions than hardware, operating systems and compilers.

We have already demonstrated that a dynamic optimization infrastructure can significantly improve cache performance [15]. We are now exploring the use of it for continuous power management. In doing so, we also aim to provide power management support for a wide variety of emerging mobile devices. Our policy complements previous policies that target hardware, compilers, and operating systems, to the exclusion of virtual machines.

8. CONCLUSIONS AND FUTURE WORK

We have presented a first step towards runtime dynamic voltage scaling in a Java virtual machine.

- Our algorithm is **based on lightweight profiles collected at runtime**. It records the execution time for every method called and uses this information to estimate performance loss from scaling the method. It scales the method on future invocations as long as this loss, including switching overhead, remains acceptably small.
- We have demonstrated the advantages of runtime DVS over traditional static policies. Our technique is **architecture independent, avoids time consuming offline simulation** and **adapts to varying execution patterns**.
- It is **easily implemented in existing virtual machines**
- Results obtained through simulation suggest up to a 13% power savings with no more than a 33% performance loss.

Given our initial prediction heuristic's simplicity, these numbers are promising. However, many avenues for further work remain. Our first agenda is to measure power on real systems. We also plan to explore more sophisticated statistical prediction models that use multiple records of past information. Although time is currently our only factor for voltage scaling decisions, other criteria can be used. Most modern processors, including the Intel x86 series, offer a detailed array of performance measurement counters. By measuring cache and TLB misses through these counters, we can quantify a program region's memory boundedness and make better voltage scaling decisions. Ideally, hardware should directly expose power consumption information to a compiler. As many have argued[27], this would allow feedback-directed online algorithms to drive power consumption with explicitly stated energy and performance requirements. Until current systems offer a direct way of measuring energy consumption, energy can only be measured in terms of the factors it depends on. Thus, low energy compilation techniques, including ours, integrate these factors rather than energy *per se* into their cost models.

A third avenue is to explore how *code annotations* can enable more accurate online predictions for dynamic voltage scaling. Traditional online algorithms suffer from a learning lag at the start of program execution, when insufficient information exists for predictions. Annotations derived from offline profiling or static analysis can act as hints to the prediction model and assuage performance loss in this initial learning period. Krintz [16] has shown such an approach to be profitable for dynamic optimization in virtual machines. The problem of finding suitable annotations for dynamic voltage scaling remains unsolved. For portability, such annotations must be expressed in an architecture neutral language. Another requirement is that they can be verified prior to execution. Malicious annotations could mislead DVS algorithms to make erroneous decisions that increase power consumption. These malicious annotations could then be exploited in a denial of service attack.

Our current implementation is built on a purely interpreted virtual machine. We expect the major issues to remain the same when implementing our online algorithm for a

just-in-time dynamic compiler with multiple optimization levels. However, an interesting area of investigation is to explore the interaction between the dynamic optimizer and dynamic voltage scaling algorithm. Different optimizers and optimized versions of code will likely have different trade-offs when power is introduced as a constraint.

Likewise, devices will introduce tradeoffs due to their differing computational power and resource constraints. We are currently porting our algorithm to the Sharp Zaurus handheld, which has a StrongArm 1100 processor. The processor offers a wide range of frequency settings. This will enable us to evaluate our algorithm for smaller, more resource-constrained devices. For portability, we also plan to investigate DVS issues in more powerful machines such as high-end servers as well as nontraditional architectures such as FPGAs, dataflow machines, and reconfigurable memories [4].

Finally, we plan to develop algorithms for offloading parts of the DVS decision process to a powerful server infrastructure [25, 9]. Such an infrastructure would support DVS analyses that are more computationally intensive than those that a resource constrained mobile device can support. It would allow us to increase the accuracy of DVS decisions as well as the speed at which these decisions are made. Speeding up the decision process is essential, since analysis time adds to a program's runtime in a dynamic optimization framework.

We now have a framework in place that allows us to experiment with a variety of runtime voltage scaling algorithms. Moreover, due to the portability of the virtual machine we are using, we can experiment across a wide variety of processor and hardware architectures.

Acknowledgements

Parts of this effort are sponsored by the National Science Foundation under ITR grant CCR-0205712 and by the Office of Naval Research under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

9. REFERENCES

[1] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven. Energy Management for Real-Time Embedded Applications with Compiler Support. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 284–293. ACM Press, 2003.

[2] N. AbouGhazaleh, D. Mosse, B. Childers, and R. Melhem. Toward the Placement of Power Management Points in Real Time Applications. In *Workshop on Compilers and Operating Systems for Low Power*. ACM Press, 2001.

[3] N. AbouGhazaleh, D. Mosse, B. Childers, R. Melhem, and M. Craven. Collaborative Operating System and Compiler Power Management for Real-Time Applications. In *IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, 2003.

[4] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259. IEEE Computer Society, 1999.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeno JVM: The Controller's Analytical Model. In *The 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[6] Crusoe Processor Product Brief: Model tm5800, transmeta corporation, 2003.

[7] LongRun Power Management: Dynamic Power Management for Crusoe Processors, White Paper, Transmeta Corporation, 2001.

[8] A. Dudani, F. Mueller, and Y. Zhu. Energy-Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints. In *Proceedings of the joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 213–222. ACM Press, 2002.

[9] M. Franz. *Compilers and Operating Systems for Low Power*, chapter A Fresh Look At Low-Power Mobile Computing. Kluver Academic Publishers, 2001.

[10] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithm for Dynamic Speed-Setting of a Low-Power CPU. In *Mobile Computing and Networking*, pages 13–25, 1995.

[11] F. Gruian. Hard Real-Time Scheduling for Low-Energy using Stochastic Data and DVS Processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 46–51. ACM Press, 2001.

[12] S. Gunther, F. Binns, D. Carmean, and J. Hall. Managing the Impact of Increasing Power Consumption. *Intel Technology Journal*, 1st quarter 2001.

[13] C.-H. Hsu and U. Kremer. The Design, Implementation and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, 2003.

[14] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scheduling. In *Workshop on Power-Aware Computer Systems (PACS'00)*, 2000.

[15] T. Kistler and M. Franz. Continuous Program Optimization: A Case Study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.

[16] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization*, Mar. 2003.

[17] S. Lee and T. Sakurai. Run-Time Voltage Hopping for Low-Power Real-Time Systems. In *Proceedings of the 37th Conference on Design Automation*, pages 806–809. ACM Press, 2000.

[18] S. Microsystems. *KVM - Kilobyte Virtual Machine White Paper.* `http://java.sun.com/products/kvm/wp/`. Palo Alto, CA, USA, 1999.

[19] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications. In *Workshop on Compilers and Operating Systems for Low Power*. ACM Press, 2000.

[20] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102. ACM Press, 2001.

[21] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design ISLPED'01*, Aug. 2001.

[22] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer. Energy-Conscious Compilation based on Voltage Scaling. In *Proceedings of the joint conference on Languages, Compilers and Tools for Embedded Systems*, pages 2–11. ACM Press, 2002.

[23] D. Shin, J. Kim, and S. Lee. Low-Energy Intra-Task Voltage Scheduling using Static Timing Analysis. In *Proceedings of the 38th Conference on Design Automation*, pages 438–443. ACM Press, 2001.

[24] Mobile Intel Pentium III Processors. http://www.intel.com/support/processors/mobile/pentiumiii/ss.htm.

[25] V. Venkatachalam, L. Wang, A. Gal, C. Probst, and M. Franz. ProxyVM: A Network-based Compilation Infrastructure for Resource-Constrained Devices. Technical Report 03-13, University of California, Irvine, March 2003.

[26] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[27] A. Weissel and F. Bellosa. Process Cruise Control: Event-driven Clock Scaling for Dynamic Power Management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 238–246. ACM Press, 2002.

[28] Y. Zhu and F. Mueller. Preemption Handling and Scalability of Feedback DVS-EDF. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, 2002.