Compiler-controlled and Compiler-hinted Voltage Scaling Approaches

Dmitry Zhurikhin[1], Andrey Belevantsev[1], Kirill Batuzov[1], Valery Ignatiev[1], Roman Zhuykov[1], and Semun Lee[2]

[1] Institute for System Programming, Russian Academy of Sciences
{`zhur,abel,batuzovk,rook,zhroma`}`@ispras.ru`
[2] Samsung Corp.
`semun.lee@samsung.com`

Abstract. This paper reports on the two approaches to dynamic voltage and frequency scaling (DVS) hinted by GCC and controlled by Linux kernel. The first approach uses profiling information for marking DVS regions which should be executed with lower frequency, while the kernel does the actual switching on entry and exit of those regions, taking into account possible switching requests from multiple processes. In the second approaches, the kernel itself decides when and to what value the frequency would be switched, and the compiler provides simple information on behaviour of program regions. For both approaches, a light-weight sampling-based profiling technique is developed. Results show some CPU energy savings with both approaches, but not whole-system energy savings on the test boards used.

1 Introduction

In our previous work [14] we have evaluated some of the compiler techniques for lowering the CPU energy consumption using GCC compiler, including compiler-controlled voltage scaling (DVS), bit-switching minimization, and memory optimizations. The most promising technique was found to be DVS, which provided several per cent CPU energy savings in our testing and very small overall system energy savings. However, the initial implementation of our DVS technique had limitations, mainly a) being an intraprocedural transformation (and thus considering neither interprocedural program regions nor any regions containing function calls as candidates for DVS) and b) not taking the multiprocess environment into account.

We have conducted a research project that aimed at removing those limitations. We have developed two DVS algorithms. One is a fully static interprocedural DVS approach (that is, a compiler controls the points of changing frequency and the values on which it should be changed) that uses a kernel manager for handling conflicts of queries for frequency changes between different processes. The other DVS approach is implemented in the kernel as a `cpufreq` governor that uses compiler-provided information for making decisions on DVS (a so-called "mixed" approach). We have also developed a light-weight sampling-based profiling mechanism that is used in both approaches for devising the needed information.

The rest of the paper is organized as follows. Section 2 describes the static DVS approach together with the devised profile support. Section 3 reports on the mixed DVS approach. Section 4 provides experimental results. Section 5 concludes.

2 Static Interprocedural DVS Operating in Multiprocess Environment

The developed static DVS algorithm is based on the algorithm in [3] and its implementation in our previous work [14]. We will shortly highlight the main stages of the algorithm for clarity. The basic idea of the algorithm is to divide a program into single-entry/single-exit (SESE) regions and to estimate their execution time on each frequency from profiling data. Then, given that we know CPU power consumption on each frequency from hardware specifications, and we know frequency switching latency from experiments, we can assign any execution frequency for each region and still we will be able to estimate the total time and energy needed to execute the program. Therefore, this data is enough to solve the optimization problem of finding the set of regions that provides the lowest energy consumption while meeting some deadline on execution time.

Our implementation in [14] operates on a single function at a time and consists of the following stages:

1. Construct basic regions and combined regions. A basic region is just a basic block or a loop, while a combined region is a SESE region made from basic regions.
2. For every basic region, profile overall execution time at each available processor frequency, $T(R, f)$, and the number of times a region is executed, $N(R)$.
3. Estimate $T(R, f)$ and $N(R)$ for combined regions[3].
4. Find the best region (basic or combined) and its execution frequency using which minimizes CPU energy consumption and does not increase running time above given threshold (controlled by user).
5. Insert frequency switching commands at the entry and the exit of the selected region.

Compared to [14], we have improved the following parts of the algorithm: building program regions suitable for DVS (more combined regions are considered suitable); finding the regions which will be executed with lower frequency (a set of regions is considered instead of a single region); profiling mechanism (a light-weight sampling-based profiling is used); and working in multiprocess environment (by handling queries for switching frequency from different processes in the kernel). We will expand on these improvements in the following subsections.

[3] $N(R)$ is taken from the basic region that is at the entry of the combined region R. $T(R, f)$ is computed as sum of $T(BR, f)$ over all basic regions BR that form the combined region R.

2.1 Building Program Regions Suitable for DVS

The original intraprocedural implementation did not allow us to handle regions with calls, such regions were not considered for optimization. In this paper, static DVS optimization is implemented as an interprocedural pass in GCC, so we have removed this restriction.

We build regions in two stages. The basic regions for each function are built during early local passes and stored in `struct function` (GCC per-function data). The reason for this is that the basic regions are actually profiled, so their construction should happen at the moment when profiling information is read. The actual optimization needs to see regions from all functions, so it happens in an interprocedural pass later in the compiler pipeline.

In this pass, the combined regions are built for each function as SESE regions (i.e. region body should be dominated by entry and postdominated by exit of the region) whose blocks all belong to previously constructed basic regions. These basic regions are allowed to contain calls. A combined region still may not cross function boundaries though, as this was not implemented. However, combined regions from all functions are merged in a single array, so that the solving part of the DVS optimization can consider all regions at once.

It can be noted that constructing basic regions at one point of the compiler pipeline and using them for optimization at another point creates the problem of keeping the regions consistent between the two passes. This problem is solved similarly to the problem of updating profile information: when control flow is modified through the GCC `cfghooks.c` API, that is, when a basic block is created, deleted, added to or removed from a loop, or merged with another block, the information about basic region is updated accordingly. We have also written a verifier to check the region consistency that is called when control flow is modified.

2.2 Finding The Best Set of Regions to Perform DVS

We choose the set of regions on which we need to change frequency in an interprocedural pass, after constructing combined regions of all functions in a translation unit. The problem we solve is as follows. We consider two available frequencies for program execution, maximum and minimum. Let us consider T as the extra time we can use for program execution (calculated as $p\%$ slowdown on execution time using the maximum frequency) and n program regions. Each region has weight w_r, calculated as the extra time needed to execute the program using the minimum frequency and the latency of switching frequency, and value v_r, calculated as the energy saved when the region is executed on the minimum frequency[4]. Now the problem of choosing the optimal set of regions can be formulated as a 0-1 knapsack problem. However, there is additional restriction on regions when operating in interprocedural mode: some regions may intersect with each other (e.g., a region containing a call and a region in the callee), and the regions we are choosing may not intersect.

[4] We only consider regions with $v_r > 0$ and $w_r < T$.

We have implemented two algorithms for solving our problem. The first algorithm is a simple greedy algorithm. We sort the regions based on v_r/w_r ratio and we choose regions starting from the one with the lowest ratio. We add a region to the set when it does not intersect with the regions that are already in the set and when their total weight does not exceed T. If a region does not meet these conditions, we consider the region with the next best ratio. The final set is found when we process all regions.

The second algorithm is a backtracking algorithm that is used in case of small number of regions to find the optimum solution. We sort the regions in ascending order by weights and then in descending order by values, and we assign indexes to the regions according to the resulting order. On each step, we try to add the next region to the candidate set considering only regions with indexes greater than the ones already in the set. When the current region intersects with some of the candidate set regions, we process the next region in the sort order. When adding the current region to the set the total set weight will exceed T, we don't process next regions as due to the sort order they will be too costly. In this case, and also when the last region is processed, we backtrack by removing the region with the largest index from the set and proceeding with adding the next region. When we have succeeded in adding the region, we remember the current best solution and proceed to the next region. In addition, we prune the search space by backtracking immediately when adding all remaining regions to the set will not provide better solution than the current best solution.

We can also model the intersecting regions by considering the graph G whose vertexes correspond to program regions and whose edges connect a pair of regions which do not intersect. Then a set of regions eligible for switching frequency will form a clique in G, and our goal will be to find a clique whose vertices have the maximum sum of values while having the sum of weights not greater than T. One of the possible solutions to this problem can be found in [5]. We did not implement this approach, as it is not obvious it would provide much better solutions for our tasks.

2.3 Handling Recursive Functions

We need to make some additional efforts to handle regions that contain calls to recursive functions. As the call graph may be incomplete, and it also may have indirect calls, we may fail to detect recursive calls (i.e., loops in the call graph) during compilation time. To solve this issue, we have created wrappers around instructions for changing frequency, which are added to `libgcc`. For each region, we compute its hash based on the source file name, the function name, and the region number. When calling the wrapper, we store the region hash and the call depth. The frequency is lowered when the depth equals to zero, then with each subsequent request having the region hash equal to the stored one the depth is increased. The frequency is raised back when the depth equals to one and the region hash equals to the stored one.

2.4 A Sampling-based Profiler

The initial profile mechanism for DVS implemented in [14] turned out to be too heavyweight, so the DVS optimization basically worked using incorrect data. We have developed a lighter profiler based on kernel timer interrupts. The instrumented program and the kernel communicate via shared memory. When the program starts, it requests a shared memory region from the kernel via the `ioctl` call. The shared memory holds a stack of currently executing DVS regions. When entering region R, its ID is pushed to the stack. When leaving R, region IDs are popped from the stack until the ID of R is removed. This allows gathering correct statistics in case we haven't tracked some region exits. When a timer interrupt occurs, all region counters that are currently stored on the stack are updated. When the program is finished, the resulting sample data is written on disk so that GCC can parse it later.

There is a problem of constructing a proper region ID. As we noted, a region is identified by a source file name, a function name, and a region number. We would like to fit the ID in a 32-bit number. However, as we need some space for the function name and the region number, it would be hard to hash the file names so that the hash function values will fit in the remaining space, so with a large program (several hundred files) the collisions in detecting the regions could very probably happen. To avoid this situation, we use a counter of compiled translation units as a hash, stored in a separate file and incremented once per compilation. Of course, this cannot be used in a production GCC, as this leads to differences in code generation of the same file compiled several times.

Last thing to note here is that the profiling info for basic regions is updated together with them between the early local pass of constructing basic regions/reading profile information and the interprocedural pass of performing the DVS optimization.

2.5 Handling DVS Requests from Multiple Processes

In a multiprocess environment, it is possible that the requests on frequency change from several DVS processes will conflict. For this case, we have implemented the mechanism of changing frequency as a modification of `ondemand` governor of `cpufreq` instead of directly calling the wrappers added to `libgcc`. The `ondemand` governor measures the periods of idle CPU and the periods of executing useful code. When the ratio of these values exceeds certain threshold, the CPU frequency is raised; when it is below another threshold, the frequency is lowered, otherwise it is left unchanged.

We have modified the `ondemand` governor so that when a DVS program executes the region that should work on the lower frequency, the manager considers its execution time as idle CPU, so that the decision of lowering frequency becomes more probable. The communication between the program and the kernel happens through shared memory similarly to the profiling mechanism. The choice of communicating with the kernel or using direct frequency changing calls (i.e. pure static DVS) is controlled by a GCC option.

3 A Compiler-hinted Mixed DVS Approach

A mixed DVS approach is based on the idea that the CPU frequency control should be done in an OS kernel dynamically, while using information about program execution that can be gathered statically by a compiler. We have begun our work by studying a number of state-in-the-art mixed DVS approaches. Since most of these approaches are aimed at real-time systems, we couldn't follow them exactly as we don't expect to have so much information about the running application. Instead we have decided to enhance one of the pure online algorithms with using additional information from the compiler.

The following subsections will provide more details about the related work in mixed DVS approaches and Linux power management, our Linux kernel algorithm, and our compiler algorithm.

3.1 Related Work on Mixed DVS

We will present two mixed DVS approaches that are most interesting to us. The approach proposed by Azevedo et al. [1] is based around *checkpoints*. A checkpoint is a special place in the program's code marked with a label. It serves as a point where the calculations of the needed CPU frequency are done. During program compilation the program time constraints are set for the execution of the code regions between checkpoints, in terms of acceptable lower and upper bounds. Such information is stored in a special *checkpoint database*, along with the possible checkpoint transitions derived from the program control flow. Similarly to our approach, the program is profiled and run in order to get representative data on its power consumption.

The actual CPU frequency scaling takes place during program execution and can be done either by OS or by the code inserted at the checkpoints by the compiler. The main idea of the scaling phase is to generate at each checkpoint a list of *events*[5] that may arise later when executing the program. Based on this list, the upper frequency bound[6] and the optimum frequency[7] are computed, and the frequency found is set accordingly. The drawbacks of this approach are that it doesn't take into account the additional power and time that are needed to calculate the new CPU frequency and to set it, and that the approach was only tested on a simulator. At the same time, the algorithm in [1] provides the flexible way of specifying desired time and power properties of the compiled program.

The other approach by AbouGhazaleh et al. [2] is similarly divided into two stages, offline and run-time. Initially, the data on program execution is collected during profiling. At the offline stage this data is used to compute when

[5] An event contains a list of next possible checkpoints.

[6] The bound shows the maximum CPU frequency that should be set in order to save any power.

[7] The optimal frequency is the CPU frequency that should be set in order to satisfy time constraints of all next possible checkpoints (i.e., the event list).

and how frequently the *power management points*[8] (PMPs) will be called. Also, during the offline phase the program is instrumented with *power management hints*[9] (PMHs). Each PMH consists of the compiler inserted code that computes the worst-case remaining cycles starting from the current PMH location to the program end. This value may vary dynamically based on the executed path for each run. For example, the remaining cycles at a PMH inside the function body are dependent on the path from which the function is called.

During run-time, a PMH computes and passes dynamic timing information to the OS in a predetermined memory location which holds the most recent value of the estimated worst-case remaining cycles. Periodically, a timer interrupt invokes the OS to execute the PMP code, so the OS adjusts the CPU frequency based on the latest value and the remaining time to the program deadline. The drawback of this approach, common with the previous one, is that they are aimed at real-time systems, so they could not be used directly as such.

3.2 Power Management in Linux

There are three widespread approaches to Linux power management:

– implemented as a stand-alone application or a daemon, e.g. CPUSpeed [6] or Open Hardware Monitor [7] projects;
– implemented as a stand-alone module or a patch for the kernel, e.g. the Dynamic Power Management project [8] or the approaches based on pre-2.6 Linux kernels;
– implemented as a governor of `cpufreq`, which is the Linux kernel module that controls the CPU frequency added in the 2.6 kernel version.

At present, the Linux kernel contains five simple CPU frequency governors: `userspace`, which allows the user to set any desired supported frequency; `powersave`, which automatically sets minimum supported CPU frequency; `performance`, which sets maximum supported CPU frequency; `ondemand`, which is an interval-based dynamic CPU frequency scheduler[10], increasing the CPU frequency when the calculated load is more than 80% and decreasing it when the load is less than 20%; and `conservative`, which is very close to the `ondemand` governor, but it makes its decision also looking at the load of previous intervals. All these governors are implemented as modules and depend on the `cpufreq` module. This allows loading/unloading the governor modules and switching between them at runtime.

There are three canonical DVS algorithms proposed in [10], which are OPT, FUTURE, and PAST. The first two are impractical as they are able to look into

[8] A power management point shows the moments of time when the OS makes a decision on the new CPU frequency.

[9] The hints allow the OS to estimate the time remained until program ends; they are used by the CPU frequency manager.

[10] The `ondemand` governor calculates the CPU load on the last time interval as the sum of the run times of all tasks from the last interval divided by the interval length.

the future of the trace data and are used for reference purposes only. The latter is a practical variant formulated as a result of experiments with the former two. OPT is an unbounded-delay perfect-future algorithm that uses available energy in an optimal way by stretching the run times in a trace to fill all available idle time. While the algorithm is simple, it is unfeasible as it doesn't care when a specific job completes as long as it does so before the end of the total time span of the trace. As a result, OPT can produce large delays in jobs' run time and cannot give adequate response to real-time events. FUTURE is a modification of OPT that can only look into the future by a small window of the next allocated time interval. Energy consumption is optimized within the window while making sure no work is delayed past the end of the window. FUTURE approaches OPT in terms of energy savings for large windows, while for small ones its energy savings are small as well. The other advantage of this algorithm is that no response is delayed past the end of the window giving good real-time response in the case of small window sizes. The PAST algorithm uses a window in the past instead of looking into the future. PAST assumes that the workload in the next window will be the same as the previous one. As with FUTURE, the window size can be adjusted to give different performance results. Its performance has subsequently been evaluated as relatively good even compared to newer and more sophisticated algorithms [9].

The AVGn algorithm by Pering et al. [11] computes an exponential moving average of the previous windows. Again, the idea is that the workload of the next time interval is expected to be similar to the previous ones. AVGn improves on the three similar algorithms that predict workload by searching for patterns in the past CPU utilization, hoping that more intelligent heuristics will lead to larger energy savings. The CYCLE algorithm is based on the idea that the CPU utilization may be structured in a cyclical pattern of interleaved peaks and valleys, a phenomenon that is observed often on CPU utilization graphs. When the algorithm finds such cycles in the past intervals, it sets the CPU speed such that the amount of work needed with any excess cycles still left can be completed within the next window. When no pattern can be found, the load is predicted to be constant[11]. PATTERN is a generalization of the CYCLE method meant to detect any kind of pattern in the load level data of the previous intervals. When a match is found, the load of the next interval is predicted to be the same as that of the interval following the previous occurrence of the sequence. Finally, the PEAK algorithm looks in the past for high peaks of activity interspersed with more stable plains. Its prediction uses several heuristics based on the expectation of narrow peaks.

The other more advanced approaches for controlling the CPU frequency are usually aimed at special types of system load, e.g. when practically just media applications are run. Such approaches present good results on corresponding workflows, hence it is useful to know which types of load and kinds of applications are expected for the target system.

[11] The constant is given as a parameter to the algorithm.

Surprisingly, performance results of the described algorithms do not indicate that the more complex heuristics outperform the simpler algorithms. According to simulations in [9] and [12], the AVGn algorithm with its simple averaging performs best of all, while CYCLE, PATTERN, and PEAK approaches are only slightly better than PAST. Most of the described algorithms are implemented and evaluated for the Linux kernel during the project YADDA [13].

3.3 Linux Kernel Part of Our Mixed DVS Approach

We have implemented our mixed DVS approach as a new `cpufreq` governor called `mixeddvs` using performance measurement counters similarly to [4]. When the governor is active, programs compiled with the support of this mixed DVS will provide it with their status to guide its decisions. When such program is executed, first it communicates with the governor via the `ioctl` call. The kernel then allocates a hunk of shared memory for the program and registers its process in the list of so-called *controlled* tasks, also starting the hardware counters if this is the first controlled process.

The majority of the work of the kernel manager is done at the periods of process scheduling, when the previous process (which just left the CPU) and the next process (which is going to be executed) are known. If the previous process is the controlled one, the kernel reads the hardware counters' values and the shared memory values. The hardware counters' events are `CCNT`[12], `INSTR`[13], and `DCM`[14]. The shared memory value is $\mathtt{REG}_{\mathtt{TPI}}$, an average time for executing a single instruction on the maximum CPU frequency. It is profiled at the compilation stage and then is written to the shared memory via the instrumentation code.

Based on these values and on available slack time for slowdown, the kernel manager selects the new CPU frequency to be used when the previous controlled process will be executed next time. First, the slack time is modified as follows: *slack-time* $\mathtt{-=}\ \mathtt{CCNT}/\mathtt{F}_{\mathtt{cur}}\ \mathtt{-}\ \mathtt{INSTR}\cdot\mathtt{REG}_{\mathtt{TPI}}/\mathtt{F}_{\mathtt{max}}\cdot(100+\mathtt{pf}_{\mathtt{loss}})/100$, where $\mathtt{F}_{\mathtt{cur}}$ is the current CPU frequency, $\mathtt{F}_{\mathtt{max}}$ is the maximum CPU frequency, and $\mathtt{pf}_{\mathtt{loss}}$ is the percent of allowed slowdown. The first part of the right-hand side of the formula shows the CPU time actually spent by a program, while the second part shows the CPU time the program would have spent (on the maximum CPU frequency), also increased by the slowdown percent. The difference shows the CPU time[15] used by the task that is above the slowdown percent. When the resulting slack time is too low or too high, the manager selects the maximum or the minimum CPU frequency accordingly, otherwise it looks at the hardware counters' values to find whether the currently executing region is good for DVS or not. We have tried several heuristics on this step, and we have found out that the relative number of data cache misses, `DCM`, works good. So, the lower or the higher CPU frequency is selected when `DCM` is high or low accordingly.

[12] A number of CPU cycles passed from the last counter reset.

[13] A number of ARM instructions executed from the last counter reset.

[14] A number of data cache misses.

[15] This time is positive when the process has used lower CPU frequency previously and negative otherwise.

When the new CPU frequency value is found, it will be used next time when the previous controlled process will be executed. The maximum CPU frequency is used for the previous process otherwise and also for the next process when it is not controlled. The only exception is when the process execution time is low, which is usual for daemons or short processes. The current CPU frequency will be kept then so that the number of the CPU frequency transitions will be lower. When the controlled process finishes or otherwise ends unexpectedly, it signals the kernel (again via `ioctl`) to free its data and the allocated shared memory.

The performance slowdown threshold is controlled in several ways. When selected, the `mixeddvs` governor creates a file called `performance_loss` in the `cpufreq` root directory. The value stored in the file represents the slowdown threshold that the new processes will have. It is also possible to set this value for the given program during compilation via the special option and the parameter.

3.4 GCC Part of Our Mixed DVS Approach

The GCC compiler part we used for the mixed DVS approach implementation is based on our previous work in Section 2. The profiling workflow is practically the same except that the single run on the maximum CPU frequency is needed for gathering data. This information includes $\texttt{REG}_{\texttt{TPI}}$ values for each region. The region $\texttt{REG}_{\texttt{TPI}}$ is low when it is CPU bounded and no memory stalls are present; when $\texttt{REG}_{\texttt{TPI}}$ is relatively high, then large number of memory stalls were encountered during region execution, so it is probably efficient to lower the CPU frequency on the region. $\texttt{REG}_{\texttt{TPI}}$ values are calculated by the kernel using `CCNT` and `INSTR` events pretty much the same way as with sampling-based profiling described in Section 2.4.

When the profiling results are used on the second compilation, GCC inserts the instrumentation code into the program, including the `ioctl` calls to allocate/free the kernel shared memory and the storing of $\texttt{REG}_{\texttt{TPI}}$ values of the DVS regions to the shared memory. As it is hard to predict the time of the next process scheduling, and as several DVS regions could be executed since the last process scheduling, the compiler also inserts code that averages $\texttt{REG}_{\texttt{TPI}}$'s over all executed DVS regions from the previous process scheduling.

4 Experimental Results

We have implemented both our approaches in GCC version 4.3.1 and Linux kernel version 2.6.24. We have evaluated our DVS optimizations (both static and mixed) together with the common `cpufreq` governors on the Aburto test suite, which represents many common scientific applications. The following measurement scheme was used. The CPU governor was set to the needed one before the test was run and then the `powersave` governor was set right after the test finish. Measurements were done during the same equal time for each run of the test. The average static power consumption of the test board was subtracted from the resulting power meter value.

Test results are shown in Table 1. Here, `time` is the actual run time of the test and is shown in seconds; `energy` is the CPU energy consumed during the measurement and is shown in mWh; the percent values show the difference between the given value and the performance column value. `Performance`, `powersave`, `ondemand`, `mixed`, and `offline` columns represent the runs with the maximum/minimum CPU frequency set, the kernel DVS approach, and our mixed and offline (with kernel manager) DVS approaches accordingly. The last two approaches had a slowdown threshold of 20%. Only some of the Aburto test programs are shown; other tests showed no significant differences in their run-time behavior.

		`performance`		`powersave`		`ondemand`		`offline`		`mixed`	
		time	energy	time	energy	time	energy	time	energy	time	energy
heapsort	val	63	6.7	87	4.3	63	7	67	6.7	69	5.8
	%	0	0	-38.1	35.82	0	-4.48	-6.35	0	-9.52	13.43
nsieve	val	56	5.8	65	3.3	58	5.8	66	3.4	59	5.1
	%	0	0	-16.07	43.1	-3.57	0	-17.86	41.38	-5.36	12.07
sim	val	132	15.1	219	9.7	133	15.4	212	14.3	180	12.8
	%	0	0	-65.91	35.76	-0.76	-1.99	-60.61	5.3	-36.36	15.23
tfftdp	val	82	10.12	132	6.77	82	10.16	82	10.15	97	9.42
	%	0	0	-60.98	33.1	0	-0.4	0	-0.3	-18.29	6.92
whets	val	164	21.1	323	15.2	164	21.7	194	19.5	202	20.4
	%	0	0	-96.95	27.96	0	-2.84	-18.29	7.58	-23.17	3.32
madmp3	val	169	9.2	169	7.7	169	7.7	169	7.4	169	7.5
	%	0	0	0	16.3	0	16.3	0	19.57	0	18.48
Total	val	666	68.02	995	46.97	669	67.76	790	61.45	776	61.02
	%	0	0	-49.4	30.95	-0.45	0.38	-18.62	9.66	-16.52	10.29

Table 1. Evaluation results.

The results show that the mixed DVS approach is able to save 10.3% CPU energy on average while slowing down execution on 16.5% in average, while the offline DVS approach is able to save 9.7% CPU energy at the cost of 18.6% slowdown.

5 Conclusions

We have completed our work by improving the static DVS approach over the one proposed in [14] by making it an interprocedural pass in GCC and via implementing a light-weight sampling-based profiling for receiving its data; and by making it interprocess via implementing a kernel manager for handling its requests. We have also implemented the mixed DVS approach as the new `cpufreq` governor using the data gathered via the similar sampling-based profiling and the shared memory for communication between the program and the Linux kernel.

We have evaluated both static and mixed DVS approaches. The results show that the mixed DVS approach is able to save 10.3% CPU energy on average while slowing down execution on 16.5% on average, while the offline DVS approach is able to save 9.7% energy at the cost of 18.6% slowdown. These results show some CPU energy consumption reduction, but not the whole-system energy consumption reduction. We believe that either the approaches should be evaluated on the real mobile devices, where the ratio of the CPU energy consumption to the whole system energy consumption will be larger, or they will be useful for the future devices with multicore CPUs that will again have the above ratio increased compared to the current one.

References

1. A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints. In *Proceedings of the Conference on Design, Automation and Test in Europe*, March 2002, IEEE Computing Society.
2. N. AbouGhazaleh, D. Moss, B. R. Childers, and R. Melhem. Collaborative Operating System and Compiler Power Management for Real-Time Applications. In *ACM Trans. Embed. Comput. Syst.*, vol 5, 1, Feb. 2006, pp. 82-115.
3. C. Hsu. Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction. Doctoral Thesis, Rutgers University, 2003.
4. K. Choi, R. Soma, and M. Pedram. Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Volume 1, February 2004.
5. A. Massaro, M. Pelillo, and I. M. Bomze. A Complementary Pivoting Approach to the Maximum Weight Clique Problem. SIAM J. on Optimization 12, 4 (Apr. 2002), pp. 928-948.
6. CPUSpeed kernel module. `http://www.carlthompson.net/Software/CPUSpeed`
7. Open Hardware Module. `http://ohm.freedesktop.org`
8. Dynamic Power. `http://dynamicpower.sourceforge.net`
9. D. Grunwald, C. B. Morrey, P. Levis, M. Neufeld, and K. I. Farkas. Policies for Dynamic Clock Scheduling. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation*, Volume 4, San Diego, California, October 2000.
10. M. Weiser, B. Welch, A. Demers, and S. Shenker, S. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, Monterey, California, November, 1994.
11. T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pp. 76-81, 1998.
12. K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-setting of a Low-power CPU. In *Mobile Computing and Networking*, pp.13-25, 1995.
13. The YADDA project. `http://www.eecg.toronto.edu/~tamda/csc2228`
14. D. Zhurikhin, A. Belevantsev, A. Avetisyan, K. Batuzov, and S. Lee. Evaluating power-aware optimizations within GCC compiler. Presented on GROW'09 workshop, January 2009, `http://www.doc.ic.ac.uk/~phjk/GROW09/papers/06-PowerBelevantsev.pdf`.