Dynamic Voltage and Frequency Scaling for Scientific Applications*

Chung-Hsing Hsu and Ulrich Kremer

Department of Computer Science
Rutgers University
Piscataway, New Jersey, USA
{chunghsu, uli}@cs.rutgers.edu

DCS-TR-447

August 2001

Abstract. This paper discusses the benefit of dynamic voltage and frequency scaling for scientific applications under different optimization levels. The reported experiments show the tradeoffs between optimizations for performance in terms of execution time, power dissipation, and overall energy consumption.

1 Introduction

Modern architectures have a large gap between the speeds of the memory and the processor. Techniques exist to bridge this gap, including memory pipelines, 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, resulting in frequent stalls of the processor waiting for the memory subsystem to provide the data.

Minimizing the power/energy consumption of scientific computations leads to a reduction in heat dissipation and cooling requirements, which in turn reduces design and packaging costs of advanced architectures, and operating costs of such machines in computing and data centers. In particular, electric bills due to air conditioning of machine rooms have become an important concern. It is reasonable to assume that the amount of energy needed to drive a computer system is comparable to the amount of energy needed to remove the resulting dissipated heat from the physical environment.

Dynamic voltage and frequency scaling (DVFS) has been recognized as an effective technique to reduce power dissipation. Commercial processors that support DVFS have recently become available, including Transmeta's Crusoe, Intel's

* This research was partially supported by NSF CAREER award No. CCR-9985050.

XScale, and AMD's K6-IIIE+. To use the DVFS capability of such microprocessor, values in appropriate system registers or data structures have to be set that reflect the desired clock frequencies and voltage levels. These values may be changed during the course of program execution.

Current research on DVFS is mostly done in the context of real-time operating systems where the operating system determines the voltage and frequency levels with the lowest power/energy consumption, given the constant of a hard or soft program deadline. Almost all the proposed scheduling algorithms use the idea of eliminating CPU slacks to determine the desired frequencies. Given the frequencies, the lowest voltage is selected that is needed to support this frequency.

CPU slacks come from many sources. In this paper, we study those from the *imbalance* between CPU and memory activities. A recent study [14] has shown that even with an aggressive, next-generation memory system, a processor still spends over half of its time stalling for L2 cache misses. It is these stalls that we can exploit to save processor power.

Most advanced locality optimizations try to reduce the memory stalls to improve performance. This is typically done by reducing the number of memory references, or overlapping memory references with computation. In this paper, we investigate the impact of these transformations to the reduction of memory stalls, which are an indication of DVFS applicability and profitability. We analyzed five scientific codes that were compiled with Compaq's (formerly DEC's) V5.3-1155 Fortran compiler, arguably one of the best optimizing compilers available today. Energy and performance simulation results showed that

– There are still many opportunities to apply DVFS to the highly optimized codes, and the profitability is significant across the benchmarks;
– There are performance and energy consumption tradeoffs for different optimization levels in the presence of DVFS; given a fixed execution time deadline, choosing the highest optimization level, followed by DVFS, may not be the best strategy for maximal energy savings.

Whether to exploit this significant opportunity for DVFS through hardware, operating systems, or compilation techniques is an open problem. The best strategy will depend on the overhead of frequency and voltage adjustment operations, and the number of times the frequency and voltage needs to be changed during program execution. This paper tries to give a first assessment with respect to this issue.

The paper is organized as follows. We first present the methodology used to conduct the experiments in Section 2. Power, energy, and performance tradeoffs are discussed in Section 3, followed by a discussion of DVFS applicability and its profitability in terms of energy savings in Section 4. A comparison of the benefits of operating system support vs. compiler support for DVFS is presented in Section 5. Finally, Section 6 gives a brief summary of related work, and Section 7 concludes the paper.

2 Our Platform and Methodology

The *Wattch* CPU energy and performance simulator [1] was used for our experiments to determine execution times, power dissipation, and energy consumption. We configured the simulator to reflect most of the characteristics of DEC's Alpha 21264, a quad-issue dynamically-scheduled general-purpose processor. As suggested in [4], the combined LSQ (Load Store Queue) is set to 64 entries, and the RUU (Register Update Unit) uses 64 entries. The memory system in our study contains a 64KB, 2-way set associative instruction and data cache, and a 2MB direct-mapped L2 cache. All caches are write-back and have the block size of 64 bytes. The main memory runs at 1/4 of the processor speed and is connected to the CPU via a 8-byte memory bus. Disks are not modeled.

The default setting assumes a fully pipelined main memory, i.e., memory can service arbitrarily many access requests at any cycle. However, we found that the simulation results are inconsistent with the real measurements of programs optimized at different levels. We believe that the assumption that memory is fully pipelined was too optimistic, and therefore chose to use *non-pipelined* memory in our simulation. That is, memory can service at most one request at any given time. All the other requests are buffered until the memory is available. The memory latency was set to be 40 cycles for all experiments.

We selected five programs from the `SPECfp95` benchmarks, and compiled each program with Compaq's (formerly DEC's) Fortran V5.3-1155 `f90` compiler. All benchmarks were compiled with `-arch ev6 -non_shared` flags and different optimization levels. The default optimization level is `-O4`. Optimization level `-O5` performs all the optimizations in `-O4`, plus software pipelining and various loop transformations. It is the highest available optimization level in `-O[n]`. The resulting Alpha EV6 binaries were "executed" by our simulator to obtain performance, power, and energy results.

The `SPECfp95` benchmark programs have a common structure, consisting of an initialization phase followed by an execution phase which can be iterated multiple times. The number of iterations of the execution phase is part of an input file and can be adjust in order to reduce simulation time. For our experiments, all benchmarks took the provided reference data sets as input, with the number of iterations for the execution phase reduced to 10 iterations.

We used *Wattch*'s default manufacturing process parameter of $.35\mu$m at 600MHz. The simulator reported 98.86W peak power for our Alpha21264-style processor. Power is assumed to scale linearly with port or unit usage, and that unused units dissipate 10% of their maximum power. Besides the total energy and total execution time reported for the entire program, we extended *Wattch* to record all these values for intervals of 1,000,000 cycles. This was done to capture the time varying behavior of the programs, as described in [20].

3 Power/Energy/Performance Tradeoffs

In general, compiling for performance will also result in more power and energy efficient programs, at least for optimizations that reduce CPU workloads. For

Table 1. The effects of standard optimizations on power/energy/performance. All the reported energy values are in Joules, execution times in cycles, and average powers in watts (Joules/sec). **Insts** is the total number of instructions executed. **Misses** is the total number of L2 cache misses (memory accesses).

Benchmark	Opt Level	Energy	Exec Time	Avg Power	Insts	Misses
		Joules	10^6 cycles	Watts	10^6	10^6
swim	-O4	17.09	404.63	25.34	433.92	4.87
	-O5	16.23	387.74	25.12	429.09	4.43
tomcatv	-O4	69.86	1323.13	31.57	2159.69	9.39
	-O5	67.00	1226.37	32.78	2153.52	7.99
applu	-O4	62.79	1555.48	24.27	1606.90	16.61
	-O5	50.64	1420.04	21.40	1081.22	17.91
hydro2d	-O4	99.79	2710.61	22.09	2268.55	33.11
	-O5	101.75	2747.82	22.22	2365.39	33.31
turb3d	-O4	273.42	4402.67	37.26	9179.82	18.53
	-O5	273.41	4402.56	37.26	9179.63	18.53

advanced locality optimizations, compiling for performance tends to introduce considerable CPU workload with possibly marginal performance improvement. As a result, CPU power dissipation may increase. Kandemir et al. have shown that loop transformations such as loop tiling can reduce memory energy costs, but at the same time can increase CPU energy consumption, which may result in an overall energy increase [13].

Table 1 shows the simulation results obtained when five `SPECfp95` benchmarks are compiled with different optimization levels. The results show that, for each benchmark, the version that has the better performance always has the lower energy consumption. For most cases, this version has less CPU workload (approximated by **Insts**) and less memory workload (approximated by **Misses**). In the case of the `applu` benchmark, `-O5` version slightly increases memory workload. But it significantly reduces CPU workload and results in better performance and lower energy cost than `-O4` version.

To further illustrate the tradeoffs between power, energy, and performance, we performed simulation on several highly optimized versions of the `swim` benchmark. `Swim` is an iterative stencil computation. It has been a popular benchmark used by researchers in the optimizing compiler community. The first version was provided by Chen Ding from the University of Rochester, and uses *loop fusion* and *array regrouping* [5]. All the other versions were provided by Marta Jimenez from the Universitat Politecnica de Catalunya, Barcelona, Spain. Her optimizations include *loop tiling* at the cache level [22] and/or register level [11]. All these transformations tend to reduce the memory workload, and have been shown to be effective for `swim`. Table 2 presents the simulation results for the different versions of `swim`.

Loop tiling appears to improve performance better than simply fusing loops together. Tiling at the cache level significantly cuts down the number of memory

Table 2. The effects of various additional optimizations on power/energy/performance of the `swim` benchmark. All programs were optimized with `-O5`, with the exception of tiled versions where loop transformations and loop unrolling was disabled [10]. All the reported energy values are in Joules, execution times in cycles, and average power dissipation in watts (Joules/sec). **Insts** is the total number of instructions executed. **Misses** is the total number of L2 cache misses (memory accesses).

Opt Level	**Energy** Joules	**Exec Time** 10^6 cycles	**Avg Power** Watts	**Insts** 10^6	**Misses** 10^6
O5	16.23	387.74	25.12	429.09	4.43
O5+AR+LF	14.98	358.89	25.04	348.30	4.35
O5+AR+2dLT	18.58	338.47	32.94	597.46	2.16
O5+AR+1dLT	17.65	288.96	36.66	550.97	1.86
O5+AR+RT+2dLT	14.42	273.54	31.64	404.43	2.25
O5+AR+RT+1dLT	14.25	259.08	33.01	403.00	1.95

LF - loop fusion
AR - array regrouping
LT - 1-dim/2-dim loop tiling
RT - register tiling

accesses at the expense of increasing the CPU workload. Unlike the previous results in Table 1, performance is improved but the energy consumption goes up. It is one more evidence that compiling for performance and compiling for low power/energy may sometimes not be the same. With the help of register tiling, the CPU workload can be effectively reduced while similar memory workload as tiling at the cache level is preserved. Not only the program performance is further improved, the corresponding energy cost is reduced as well. As a matter of fact, the version `O5+AR+RT+2dLT` runs the fastest and consumes the least energy. But we want to point out that it has the second highest average power.

4 DVFS for Highly Optimized Codes

Extensive research on optimizing compilers has been carried out in the last decades, especially the loop and data layout transformations for scientific applications. All these high-level transformations try to improve the cache locality, and have been proved to be quite effective in reducing memory stalls. However, DVFS relies on exploiting memory stalls to save power and energy. In this section we will discuss the impact of loop transformations on the applicability and profitability of DVFS.

4.1 Quantify DVFS applicability

It is not yet clear how to capture the DVFS applicability. In earlier work, we introduced a metric, called slow-down factor δ [8], to capture DVFS opportunities. The slow-down factor is defined by how much the CPU can be slowed down without exceeding a prescribed performance penalty (soft deadline). The slow-down factor is never less than one, and larger factors represent slower clock frequencies.

The slow-down factor is defined as follows:

$$\delta \stackrel{\text{def}}{=} 1 + \min(\frac{r}{R_c}, \frac{R_m}{R_b})$$

where R_b represent the fraction of an interval in which CPU activity (including L1 and L2 activity) is overlapped with memory access, R_c is the fraction in which only CPU has activity, and R_m is the fraction in which CPU is stalled due to memory bandwidth and latency. The ratio r indicates the prescribed relative performance penalty. In this paper we set it to be 10%.

Figure 1 gives the DVFS applicability of the five benchmarks optimized with `-O5` over intervals of 1,000,000 cycles. All graphs show an initialization phase, followed by an execution phase consisting of repeated patterns. The regular patterns suggest that the compiler may be able to recognize them. If we consider the entire execution phase except the last few intervals, benchmarks such as `swim` and `tomcatv` have a minimum slow-down factor across all intervals that is significantly greater than one. This indicates that the default CPU speed is too high for the benchmark. As a result, choosing a single, fixed slow-down factor for the entire execution phase, namely the minimum value, will reduce energy consumption without incurring any switching overhead.

Inside each repeated pattern, there are regions that have slow-down factors much higher than the minimum value. These regions spend a considerable amount of time in each pattern. These are the places that the *dynamic* voltage and frequency scaling can be applied to further reduce the energy costs. In summary, there are still significant opportunities to apply DVFS to highly optimized scientific applications, a somewhat surprising result.

4.2 DVFS profitability

To quantify the benefit of applying DVFS to highly optimized codes, we introduced a simple analytical energy model. The model is based on associating with each CPU cycle an energy cost, and separating active cycles from idle cycles. The energy model is defined as follows.

$$E = \rho \cdot 10^6 \cdot [R_c + R_b + \frac{1}{10} \cdot R_m] \tag{1}$$

$$E' = \frac{\rho}{\delta^2} \cdot 10^6 \cdot [R_c + R_b + \frac{1}{10} \cdot \frac{1}{\delta} \cdot (\max\begin{pmatrix} \delta R_b \\ R_b + R_m \end{pmatrix} - \delta R_b)] \tag{2}$$

Energy measure E is for the interval without DVFS, which we had profiled in the simulation. An active CPU cycle consumes ρ Joules, while an idle one consumes only 10% of this figure. Energy measure E' is for the interval after applying DVFS using slow-down factor δ. The energy cost of an active CPU cycle is scaled down by a factor of δ^2. The CPU workload is kept the same after

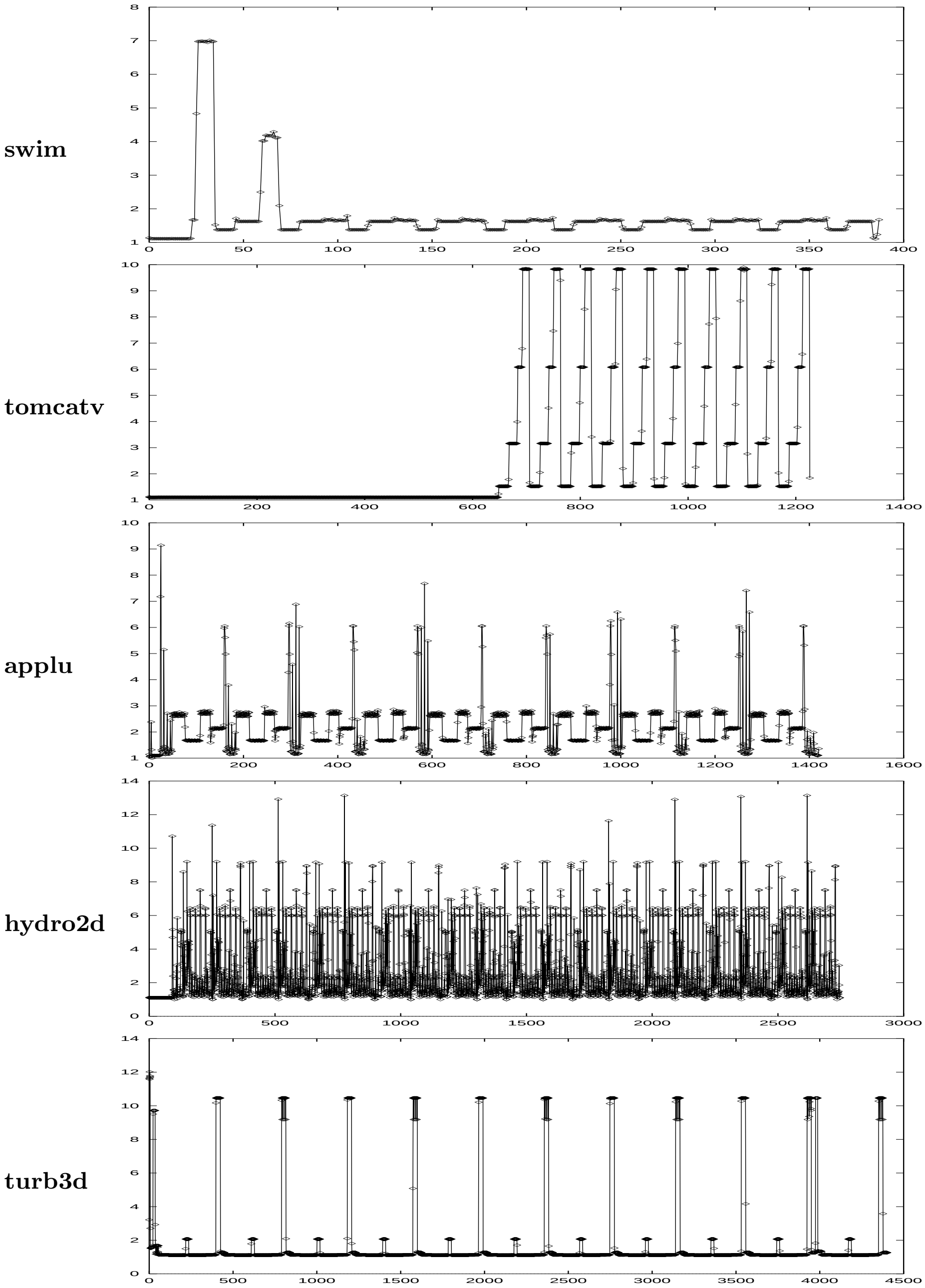

Fig. 1. DVFS applicability of the five `SPECfp95` benchmarks optimized with `-O5`. The x-axis represents intervals of 1,000,000 cycles and the y-axis represents the DVFS applicability in terms of the slow-down factor δ.

Table 3. DVFS profitability for benchmarks optimized with `-O5`. Energy savings and performance penalty are relative to the codes executed without DVFS. They are all represented as percentages.

Benchmark	Energy Saving (%)	Performance Penalty (%)
swim	58.54	9.50
tomcatv	39.09	8.42
applu	70.61	7.27
hydro2d	64.88	7.32
turb3d	26.36	9.27

Table 4. DVFS profitability on various version of the `swim` benchmark. All the reported energy values are in Joules and execution times in 10^6 cycles.

	w/o DVFS		w/ DVFS		Profitability	
Opt Level	Energy	Exec Time	Energy	Exec Time	Energy Saving (%)	Performance Penalty (%)
O5	16.23	387.74	6.73	424.57	58.54	9.50
O5+AR+LF	14.98	358.89	5.15	392.78	65.63	9.44
O5+AR+2dLT	18.58	338.47	12.17	370.88	34.49	9.58
O5+AR+1dLT	17.65	288.96	11.60	316.34	34.29	9.48
O5+AR+RT+2dLT	14.42	273.54	8.37	299.36	41.96	9.44
O5+AR+RT+1dLT	14.25	259.08	8.69	283.55	39.07	9.44

LF - loop fusion
AR - array regrouping
LT - 1-dim/2-dim loop tiling
RT - register tiling

applying DVFS [1], as reflected by $10^6 \cdot (R_c + R_b)$. However, the total execution time in *seconds* increases due to slower CPU speed. As a consequence, the number of idle CPU cycles for a DVFS'ed program is reduced. This can be reflected by the inequality $\frac{1}{\delta} \cdot (\max \begin{pmatrix} \delta R_b \\ R_b + R_m \end{pmatrix} - \delta R_b) \leq R_m$. Finally, the DVFS profitability can be estimated through the relative energy reduction $1 - E'/E$ using Equations (1) and (2).

Table 3 shows the possible energy savings and performance degradation if at every interval we slowed down the CPU according to the slow-down values shown in Figure 1. Energy may be saved up to 71% if the switching cost between different voltages and frequencies is negligible. Again, this is an optimistic approximation, but it clearly indicates the possibility of applying DVFS without significant performance penalties even for highly optimized codes.

[1] It is an ideal assumption and may not be the case in practice, for instance due to out-of-order instruction execution.

Table 5. The profitability of DVFS on various version of the `swim` benchmark with respect to a global deadline: the execution time of version `O5`. All the reported energy values are in Joules and execution times in 10^6 cycles. r is the relative performance penalty used to determine slow-down factor δ as discussed in Section 4.1.

Opt Level	w/o DVFS Energy	w/o DVFS Exec Time	r	w/ DVFS Energy	w/ DVFS Exec Time	Profitability Energy Saving (%)	Profitability Performance Penalty (%)
O5	16.23	387.74	0.00				
O5+AR+LF	14.98	358.89	0.08	5.97	386.39	63.23	-0.35
O5+AR+2dLT	18.58	338.47	0.15	10.62	385.95	34.56	-0.46
O5+AR+1dLT	17.65	288.96	0.34	7.30	368.46	55.03	-4.97
O5+AR+RT+2dLT	14.42	273.54	0.42	4.64	357.04	71.44	-7.92
O5+AR+RT+1dLT	14.25	259.08	0.50	4.52	349.39	72.14	-9.89

LF - loop fusion
AR - array regrouping
LT - 1-dim/2-dim loop tiling
RT - register tiling

5 Discussion

Compiling for low power and energy, and compiling for performance may require different optimization strategies. In the previous section we have shown that DVFS can be applied to codes which are already highly optimized for performance. In this section we would like to address a few other issues of using DVFS from the compiler's point of view.

As discussed in Section 3, sometimes the compiler optimizations introduce a considerable amount of processor workload, often only for marginal performance gains. In such situations, the CPU power dissipation goes up, and possibly the total energy consumption increases as well. Aggressive optimizations, if not applied carefully, may prohibit DVFS, and therefore will not benefit from this effective power/energy reducing technique.

Consider the various versions of the `swim` program with and without DVFS. First of all, as in the case of our five benchmark programs, there is significant opportunity and benefit for DVFS across the highly optimized versions of the benchmark. Table 4 gives the estimates of DVFS profitability. Notice that even though version `O5+AR+RT+1dLT` consumes the least energy without DVFS, it does not allow large energy reduction due to DVFS. On the other hand, version `O5+AR+LF` consumes slightly more energy but preserves the DVFS applicability, and in the final consumes the least amount of energy when DVFS is applied. The result shows that, while compiling for performance is in general good for compiling for low power, the DVFS applicability needs to be taken into account as well.

In a computation environment with limited energy resources, a compiler may trade-off execution speed for power and energy savings. In one possible optimization scenario, the compiler is presented with a fixed program execution deadline. The optimization goal is to minimize energy consumption while honoring this deadline. One possible compilation approach consists of applying advanced data locality optimizations, and using the resulting performance gain as the performance penalty r (see Section 4.1) that can be tolerated for choosing a slow-down factor δ. Table 5 shows the result of this approach for the `swim` benchmark with an execution deadline equal to the execution time of the O5 optimization level version. Although the fastest optimization turned out to be the most energy efficient one in this case (`O5+AR+RT+1dLR`), the ranking of the other versions switched, i.e., the less successful optimization lead to higher energy savings for the given deadline.

The previous experimental results assume that the cost of scaling to the desired voltage and frequency is negligible. In current technology, it is not the case. Right now the switching latency is as long as 75-520μs. If the switching overhead is significant, DVFS cannot be applied too frequently during the entire program execution. Fortunately, Figure 1 shows that four of the five benchmarks have stable regions that we can simply change the voltage and frequency at the beginning and end of these regions. We believe that DVFS at the compiler level, OS level, or microarchitecture level will perform equally well for these four programs.

Difficulties for effective DVFS at OS and microarchitecture levels arise in programs like `hydro2d`. There are too many peaks that prohibit effective application of OS or hardware guided DVFS approaches which rely on limited windows of observed past program behavior. In contrast, the compiler has a *more global* view of the program regions and their execution characteristics, and therefore can better tolerate frequent changes in CPU slack. For example, if we apply the technique in [9], the compiler is able to find a program region in the execution phase to slow down, as shown in Figure 2. The total execution time increases by 12.25%, but the energy is reduced by 64.62%. More importantly, there are only 20 voltage/frequency switches, compared to 2748 switches in the ideal case.

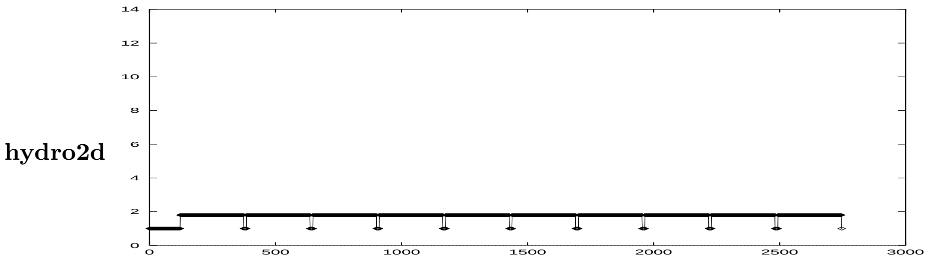

Fig. 2. The compiler-directed DVFS application of the `hydro2d` benchmarks optimized with `-O5`. The x-axis represents intervals of 1,000,000 cycles and the y-axis represents the DVFS applicability in terms of the slow-down factor δ.

There is an additional reason why compiler-directed DVFS is *complementary* to the OS-directed DVFS. In general, the operating system receives a given binary to execute, while the compiler has a version of the source code which may be translated to many different semantically equivalent binaries. If, for example, the compiler generates the tiled version of `swim`, OS-directed DVFS will not be able to reduce energy too much since the binary does not provide the opportunity. In contrast, the compiler has the choice of considering different DVFS opportunities with different power/energy/performance tradeoffs.

6 Related Work

There exist processors that support dynamic voltage and frequency scaling, such as Transmeta's Crusoe, Intel's Xscale, AMD's K6-IIIE+, and the prototypes from U.C., Berkeley [2] and from Delft University of Technology [19]. Current implementations involve draining instruction pipeline and waiting until the desired voltage is supplied through an external DC-DC regulator when DVFS is applied. It usually takes a long time to switch (75-520 μs). With such high latency the frequency of dynamically changing voltage may need to be taken extreme care.

Extensive research on optimizing compilers has been carried out in the last decade [18], mostly stressing on high performance. Until recently, power becomes an increasingly important issue and there is a growing interest in optimizing software for low power. While most of the researches focus on the back-end optimizations such as reducing bit switching activity in buses, we address in this paper the high-level optimizations, especially loop transformations for scientific applications.

Kandemir et. al. evaluated the influence of high-level compiler optimizations on the power/energy behavior of a program. They showed in [13] that optimizations such as loop tiling and data transformations may increase the CPU energy usage while reducing memory system energy. Valluri and John in [23] presented a quantitative study of the standard optimizations levels on power and energy of the processor. They concluded that optimizations that improve performance by reducing the workload (such as common subexpression elimination and copy propagation) are optimized for energy, and optimizations that improve performance by increasing the workload overlap (such as aggressive instruction scheduling) increase the average power dissipated in the processor.

It is interesting to note that several researchers have found that compiling for low power is quite different from compiling for performance. For example, in [12], Kandemir et al. found that the best tile size for the least energy consumed is different from that for the best performance. And in [16], Marculescu found that an optimal level of parallelism for energy consumption in modern processors is not necessarily the same as the one for performance.

To capture DVFS opportunity, most of the work has been done at the operating system level. Previous studies that adapted frequency to CPU activity based on intervals used ad hoc heuristics and "hand tuned" these heuristics. In each case, the goal was to reduce energy without compromising performance. A recent study, based on actual execution not on simulation, showed that this goal is not quite achieved [7].

Applying DVFS at the microarchitecture level is also advocated in recent years. For example, Ghiasi et al. in [6] and Childers et al. in [3] suggested using IPC (or IPS) to adapt the frequency to the current workload to save energy. Marculescu in [15] proposed to use cache misses as the scaling points.

To the best of our knowledge, compiler-directed DVFS has yet to be explored. In an early paper [8] we proposed a simple model to compute the CPU slow-down factor, and showed in a later paper [9] that it can save energy of 3.97%-23.75% for the `SPECfp95` benchmark suite with performance penalty of at most 2.53%. Other approaches include the ones proposed by [17, 21] to instrument program regions so as to exploit the CPU slacks missing when using the worst-case execution time assumption to determine the voltage schedule in the operating systems.

7 Conclusions

Dynamic voltage and frequency scaling (DVFS) has been recognized as an effective technique to reduce power dissipation. Since it relies on exploiting memory stalls and most advanced locality optimizations try to reduce the stalls, we investigate in this paper the impact of these transformations to the applicability and profitability of DVFS.

Five scientific codes were compiled with Compaq's Fortran compiler, arguably one of the best optimizing compilers available today. Energy and performance simulation results showed that there are still many opportunities to apply DVFS to the highly optimized codes, and the profitability is significant across the benchmarks. It is also observed that there are performance and energy consumption tradeoffs for different optimization levels in the presence of DVFS. A comparison of the benefits of operating system support vs. compiler support for DVFS is discussed as well.

Acknowledgements

The authors wish to thank Chen Ding from University of Rochester for a version of the `swim` code and the support of using their Alpha compiler. In addition, they want to thank Marta Jimenez from Universitat Politecnica de Catalunya (UPC) for tiled versions of the `swim` code.

References

1. 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.

2. T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*, July 2000.
3. B. Childers, H. Tang, and R. Melhem. Adapting processor supply voltage to instruction-level parallelism. In *Kool Chips 2000 Workshop*, December 2000.
4. R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in microprocessor simulation. In *the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, July 2001.
5. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of International Parallel and Distributed Processing Symposium*, April 2001.
6. S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity Effective Design*, June 2000.
7. D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI-2000)*, October 2000.
8. C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems (PACS)*, November 2000.
9. C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'01)*, August 2001.
10. M. Jimenez. Private communication.
11. M. Jimenez, J.M. Llaberia, A. Fernandez, and E. Morancho. A general algorithm for tiling the register level. In *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998.
12. 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.
13. M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference (DAC)*, June 2000.
14. W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proc. 7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, January 2001.
15. D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *Workshop on Complexity-Effective Design*, June 2000.
16. D. Marculescu. Profile-driven code execution for low power dissipation. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, July 2000.
17. D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compiler and Operating Systems for Low Power (COLP'00)*, October 2000.
18. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
19. J. Pouwelse, K. Langendoen, and H. Sips. Voltage scaling on a low-power microprocessor. In *International Symposium on Mobile Multimedia Systems & Applications (MMSA'2000)*, November 2000.

20. T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, Department of Computer Science and Engineering, University of California, San Diego, August 1999.
21. D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. In *To appear in IEEE Design and Test of Computers*, March 2001.
22. Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.
23. M. Valluri and L. John. Is compiling for performance == compiling for power? In *The 5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*, January 2001.