Optimizing Power-Performance Trade-off for Parallel Applications through Dynamic Core and Frequency Scaling

Satoshi Imamura
Graduate School of Information Science and Electrical Engineering
Kyushu University
s-imamura@soc.ait.kyushu-u.ac.jp

Hiroshi Sasaki
Faculty of Information Science and Electrical Engineering
Kyushu University
sasaki@soc.ait.kyushu-u.ac.jp

Naoto Fukumoto
Graduate School of Information Science and Electrical Engineering
Kyushu University
fukumoto@soc.ait.kyushu-u.ac.jp

Koji Inoue
Faculty of Information Science and Electrical Engineering
Kyushu University
inoue@ait.kyushu-u.ac.jp

Kazuaki Murakami
Faculty of Information Science and Electrical Engineering
Kyushu University
murakami@ait.kyushu-u.ac.jp

Abstract

As power consumption being the first order constraint to build microprocessors, they are required to achieve high performance within the strictly limited power budget. For example, capping the peak power consumption is a strongly desired feature in large scale data centers or massive HPC machines. Future many-core processors are expected to host a variety of workloads which have different characteristics and requirements. Therefore, a novel runtime environment which can manage these applications in an energy-efficient manner needs to be developed.

Traditional dynamic voltage and frequency scaling (DVFS) which optimizes the trade-off between performance and power consumption offers an efficient execution for single-threaded applications. However, this is not always the case for multi-threaded applications executed on many-core processors depending on their parallelisms. We propose dynamic core and frequency scaling (DCFS) technique to optimize the power-performance trade-off for multi-threaded applications. Our proposed technique adjusts core counts and CPU frequency depending on the parallelism of applications under the power consumption constraint. DCFS dynamically controls the settings to optimize against the behavior within programs. The evaluation results show that we can achieve 6% performance improvement on average across ten applications from PARSEC benchmarks and up to 35% for `dedup`.

1. Introduction

Recently, multi-core became the mainstream architecture to build microprocessors because the performance of single-core processors is now limited by power consumption, which therefore prevents the traditional performance improvement techniques such as increasing the operating frequency or implementing complex out-of-order wide-issue superscalar processors. The number of cores equipped on a single chip tends to increase as the technology shrinks, and the many-core era is expected to arrive in the near future [5, 7, 8]. The key to achieve both high performance and low power in multi-core processors is efficient parallel processing, and the importance becomes more significant for many-core processors.

As power consumption has become the first order concern in today's microprocessors, future processors are required to achieve high performance within the strictly limited power budget. Capping the peak power consumption is a strongly desired feature in large scale data centers where server consolidation is becoming a main technique to manage them, or massive HPC machines with heavily multi-threaded applications take place. Future many-core processors need to host a variety of workloads which have different characteristics and requirements. Therefore, we need to have a runtime environment which can manage these applications in an energy-efficient manner.

A traditional approach for achieving energy-efficient execution within a power consumption constraint is to apply dynamic voltage and frequency scaling (DVFS), which optimizes the trade-off between performance and power consumption [9]. DVFS offers an efficient execution for single-threaded applications and multi-threaded applications running on CMPs (chip multiprocessors) [4]. However, this is not always the case for heavily multi-threaded programs being executed on many-core processors. CPU frequency along with supply voltage is an efficient lever to control both performance and power consumption by setting it to the maximum available value considering the power budget. However, there exists another principle factor which determines the performance for multi-threaded applications: scalability or parallelism. Multi-threaded programs being executed on many-core processors tend not to guarantee that neither the highest performance nor the most energy-efficient execution realized by utilizing all the underlying core counts.

In this paper, we propose dynamic core and frequency scaling (DCFS) technique to optimize the power-performance trade-off for multi-threaded applications. We believe that there are two main knobs to control the power-performance trade-off in many-core processors: CPU core throttling and frequency scaling. The

Table 1. Configuration of the system for experiment (AMD Opteron)

Processor	AMD Opteron 6136
Number of processors	4
Number of cores per processor	8
Total number of available cores	32 (4 x 8)
L1 I/D cache	128 KB x 32
L2 cache	512 KB x 32
Shared L3 cache	12 MB x 4
Main memory	16 GB (DDR3-1333)
Bus speed	6.4 GT/s
Technology Size	45 nm

key idea is to distribute the precious power budget to either additional number of cores or increasing the operating frequency, depending on the characteristics of the program. The characteristics differ among and within programs, and we propose a dynamic technique which can apply the core throttling and frequency scaling during runtime. While the traditional execution only relied on frequency and voltage scaling for energy-efficient execution, DCFS is possible to produce additional power to be distributed by appropriately controlling the number of cores to be activated. Our evaluation under a fixed amount of power budget shows that the proposed technique improves the performance by up to 35% compared to the conventional execution running with the maximum available core counts and the minimum available frequency.

The rest of the paper is organized as follows: Section 2 shows the experimental environment and motivates our work by showing variety of characteristics of applications which have different parallelism and different sensitivity with CPU frequency among and within applications. Section 3 explains the overview and implementation of the proposed technique. Section 4 shows evaluation environment and results of the proposed DCFS technique. Section 5 discusses the results of evaluation to understand the effect of DCFS. Section 6 introduces related work and Section 7 concludes our study.

2. Performance Characteristics with respect to Core Counts and CPU Frequency

This section shows that performance characteristics with regard to core counts and CPU frequency differ depending on the kind of applications or executed phases under power constraint. Note that we execute applications from PARSEC benchmark suite [2] on an AMD Opteron based real system in this experiment.

2.1 Experimental environment

First, we explain the experimental environment. The configuration of system used in the experiment is shown in Table 1. The system is symmetric multi-processor (SMP) machine which includes four processors with each processor having eight cores. Therefore, the system has 32 cores in total. We select three applications (`blackscholes`, `dedup` and `x264`) from PARSEC and use the "native" input set.

2.2 Assumption of power consumption constraint

We assume that the maximum CPU frequency is decided by the core counts so as not to exceed the power consumption constraint. We set the constraint to the power consumption when all cores (32 cores) run on the minimum available CPU frequency according to equation (1). Let a be the switching activity of the circuit, $N_{allcores}$ be the total number of cores on a chip, C be the load capacitance per core, f_{min} be the minimum CPU frequency, and V_{min} be the minimum supply voltage. We assume that the capacitance of processor is proportional to the number of cores.

$$P_{constraint} = a \cdot N_{allcores} \cdot C \cdot f_{min} \cdot V_{min}^2 \quad (1)$$

We calculated the maximum available CPU frequency for each number of cores so that their power consumption does not exceed $P_{constraint}$ [1]. Table 2 shows the maximum CPU frequency and supply voltage we assumed in this study depending on the number of cores.

Table 2. Maximum CPU frequency and supply voltage under power constraint for each core count (AMD Opteron)

Number of cores	CPU frequency [GHz]	Supply voltage [V]
1 – 5	2.400	1.300
6 – 8	1.900	1.213
9 – 12	1.500	1.125
13 – 19	1.100	1.038
20 – 32	0.800	0.950

2.3 Variety of Characteristics among Programs

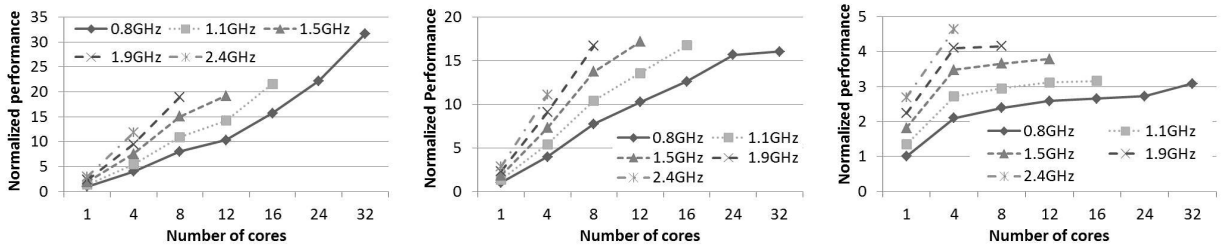
Figure 1 shows the parallelism or scalability of three applications executed on the AMD Opteron platform. The x-axis represents the number of cores to be assigned to each program, and the y-axis represents the normalized performance where the value one denotes the performance of a single core execution with the lowest frequency which is 0.8 GHz in our experimental environment[2]. We show five lines in each figure that each corresponds to execution with different frequencies (0.8, 1.1, 1.5, 1.9, and 2.4 GHz). The maximum number of cores that can be assigned to each frequency is restricted by their power consumption constraint as shown in Table 2.

Figure 1(a) shows the performance result of `blackscholes` which shows almost an ideal performance increase in proportion to the number of cores and CPU frequency. In such an application, it is generally more energy-efficient to increase the core counts rather than increasing the CPU frequency. This is quite intuitive because the power consumption of a CPU is proportional to the Vdd square and the CPU frequency. For example, when we compare doubling the CPU frequency and doubling the core counts, power consumption doubles from doubling each factor in both cases, however, increasing the CPU frequency consumes more power because it needs to raise its Vdd. This can be clearly seen from the figure that the 0.8 GHz execution with 32 cores achieves the highest performance within the power constraint.

For applications which saturate the performance by increasing the core counts such as `x264` (Figure 1(b)) or `dedup` (Figure 1(c)), we can achieve higher performance within the power envelope by restricting the number of cores and convert the surplus power into performance by increasing the CPU frequency. As seen from the figure, `x264` performs the highest performance by 1.5 GHz execution with 12 cores. The result of `dedup` is quite interesting that the increase in CPU frequency gives much better improvement in performance rather than increasing the core counts, which result in 2.4 GHz execution with 4 cores achieving the highest performance.

[1] Our maximum CPU frequency assumption is conservative in the sense that we consider $P_{constraint}$ as a hard limit that can never be exceeded.

[2] All the execution create 32 threads, and threads are packed to the number of cores which is represented on the x-axis [3]. We use this "Thread Packing" technique for the rest of our study.



(a) blackscholes (b) x264 (c) dedup

Figure 1. Performance of three programs from PARSEC benchmark suite with different core counts and CPU frequencies

2.4 Variety of Characteristics within a Program

Figure 2 shows the detailed performance characteristics of `x264` application which varies time after time. The figure shows performance results of pairs of five bars on the x-axis for different loops. Each pair consists of bars of performance execution with five different core counts (4, 8, 12, 16, and 32). The x-axis expresses the time flow by showing result for five consecutive loops handling different frames of the input video and the y-axis shows the committed IPS (instructions per second) which corresponds to their performance. IPS is a good indicator to measure performance because the total number of dynamic instructions is stable among the execution with different number of cores under Thread Packing execution [3].

In the first loop, 1.5 GHz execution with 12 cores achieves the highest IPS among all combinations. In contrast, in the second, the third, and the fourth loops, the best performing configuration changes to 0.8 GHz execution with 32 cores. In the fifth loop, 1.5 GHz execution with 12 cores achieves the highest performance again. We can see from the figure that each loop has different characteristics with respect to its core counts and operating CPU frequency. This suggests us that we need a runtime technique to deal with such a variety within programs.

As seen from the discussions, Figures 1 and 2 motivate us of developing a dynamic optimization technique which tries to maximize the performance by controlling the number of cores and the operating CPU frequency within a fixed amount of power budget. The key idea is to (1) detect the performance characteristics which are the scalability and the performance sensitivity to CPU frequency during runtime and (2) select the best configuration appropriately. We describe the details of our proposed technique in the next Section.

3. Dynamic Core and Frequency Scaling

3.1 Overview

The objective of our technique is to maximize the performance of parallel programs executed on many-core processors under power consumption constraint. In case of a traditional execution, we tend to execute multi-threaded applications by creating equal or larger number of threads than the underlying logical core counts to fully utilize the system, and the OS allocates each thread to all the cores. However, as we have seen in the previous section, this does not necessarily give us the maximum performance nor the most energy-efficient execution.

Therefore, we propose a sophisticated technique to dynamically control the number of cores and the CPU frequency according to the characteristics of the application which we call dynamic core and frequency scaling (DCFS). For fully parallelized applications such as `blackscholes` in Figure 1(a), we should allocate greater number of cores as possible. However, for applications with middle or low scalability, there is room for better optimization as we have discussed earlier. We have to appropriately provide the limited power consumption to control the two knobs, which are CPU core throttling and frequency scaling, to achieve efficient execution.

3.2 Determination of core counts and CPU frequency

Our DCFS technique is composed of two phases which are "Training" and "Execution". In the Training phase, we execute the program with different configurations (a set of combinations with different number of allocated cores and CPU frequency) for a short period of time each to identify the characteristics. In this study, IPS is used for this purpose. After the Training phase, the optimal number of cores and CPU frequency which maximize the performance are estimated from the measured IPS values. The execution phase simply applies this configuration to the program. This Training and Execution phases are repeated iteratively until the end of the program so that we can follow the dynamic behavior within a program. Note that this is a totally dynamic technique which requires no static information nor modifications to the application binary. Figure 3 shows the overview of this technique and we explain the detail.

- **Training**
  Training phase dynamically measures the IPS by changing the configuration to find the optimal pair of core counts and CPU frequency. The key idea is to find the best performing number of cores for each possible frequency and compare them to find the best pair. It works as follows: first, the program is executed with all the cores and the maximum possible frequency (for example, 32 cores and 0.8 GHz as seen in Figure 1, and we will refer the numbers from this Figure for this explanation) and IPS is measured for a short period of time (called "Training period"). Next, we decrease the number of cores while keeping the CPU frequency (24 cores and 0.8 GHz) and measure the IPS. We keep decreasing the number of cores until the IPS decreases. At that point, we are able to find the best core counts for that frequency (0.8 GHz). This works because we assume a convex curve for the number of cores versus performance, which is quite general, and it is true for almost all of the evaluated benchmarks in our environments. Next, we increase the frequency (1.1 GHz) and measure the IPS by starting from the maximum available core counts considering the power cap to the point we see a performance degradation. We repeat this process for all the possible frequencies (0.8, 1.1, 1.5, 1.9, and 2.4 GHz) and the optimal configuration which achieves the highest IPS is determined by comparing the IPS against each other.

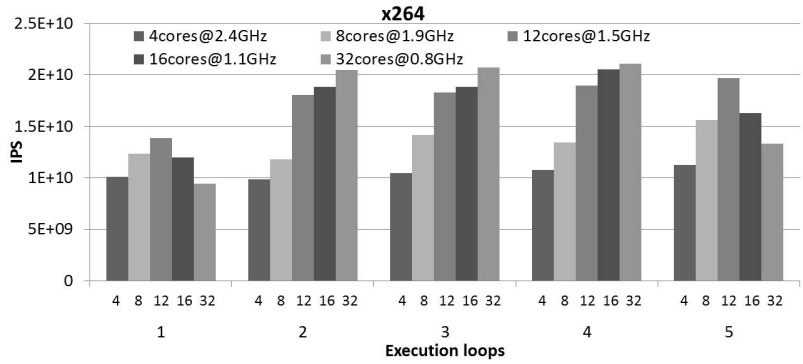


Figure 2. Performance characteristics of `x264` for different phases

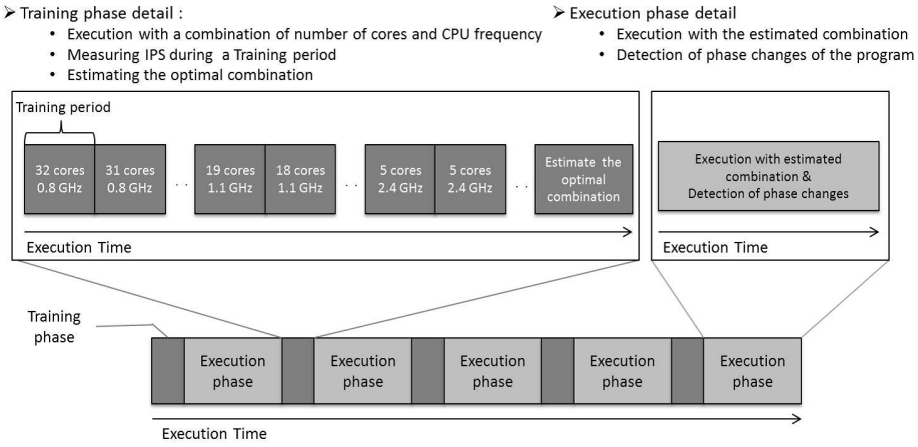


Figure 3. The overview of the proposed technique

- **Execution**
  After the Training phase, the selected combination of core counts and the frequency is applied to the program and is executed with the combination. As we can see from Figure 2, the behavior of the program changes during runtime. Therefore, IPS is measured periodically (every one second in our work) to detect the change and DCFS switches to Training phase again if the current IPS increases or decreases by a certain range compared to the previous IPS.

3.3 Implementation

We have built a prototype user level runtime system to implement the proposed DCFS technique. The system is built on top of the Linux `perf-tools` toolset to allow periodical access to the performance counters in order to measure the IPS. Additionally, we use `sched_setaffinity(2)`, a standard Linux API to control the CPU affinity of the evaluated program to bind the processes to a specific number of cores.

An important parameter for our proposed technique is how long to set the Training phase. We measured the time it takes to stabilize the behavior after changing both the affinity and CPU frequency. The maximum time it took in the worst case by changing the number of cores and the frequency from the minimum to the maximum was 30 ms. Therefore, we ignore the first 30 ms after changing the configuration and use the next 30 ms as the length of the "Training period". In the Execution phase, behavior changes of the executed program are detected if the current IPS increases or decreases by more than 10 % compared to the previous IPS, and our technique switches to the Training phase.

4. Evaluation

4.1 Evaluation environment

We evaluate our proposed DCFS technique with two kinds of platforms: AMD Opteron and Intel Xeon. The configuration of Opteron is already shown in Table 1 and that of Xeon is shown in Table 3. In common with the Opteron platform, we set the dynamic power consumption constraint as the value when all 12 cores run at the minimum available CPU frequency (1.596 GHz) in Xeon platform. The maximum available CPU frequency for each number of cores is calculated similarly as shown in Table 4. Note that we invalidate both Turbo Boost technology and Hyper-threading technology to obtain stable results for Xeon platform.

We chose ten benchmarks from the PARSEC benchmark suite 2.1 [1] and used the "native" input set for evaluation. We measured the scalability of the benchmarks using our Opteron platform in order to classify them into three types. Table 5 shows the classification of the evaluated benchmarks according to their parallelisms by showing the speedup against a single core execution to execution with maximum number of cores.

Table 5. Classification of the evaluated benchmarks according to the parallelisms

Parallelism	Benchmark	Speedup against 1 core (Opteron)	Speedup against 1 core (Xeon)
High	blackscholes	31.6x	11.9x
	swaptions	31.6x	7.1x
	vips	29.7x	11.6x
	ferret	21.4x	9.4x
Middle	freqmine	18.4x	10.1x
	x264	16.3x	10.0x
	canneal	13.0x	10.1x
	bodytrack	12.4x	5.4x
	streamcluster	10.4x	7.7x
Low	dedup	3.1x	3.5x

Table 3. Configuration of the evaluation system (Intel Xeon)

Processor	Intel Xeon X5670
Number of processors	2
Number of cores per processor	6
Total number of available cores	12 (2 x 6)
L1 I/D cache	32 KB x 12
L2 cache	256 KB x 12
Shared L3 cache	12 MB x 2
Main memory	96 GB (DDR3-1333)
Bus speed	6.4 GT/s
Technology Size	32 nm

Table 4. Maximum CPU frequency and supply voltage under power constraint for each core count (Intel Xeon)

Number of cores	CPU frequency [GHz]	Supply voltage [V]
1, 2	2.927	1.350
3	2.527	1.132
4	2.261	1.023
5	2.128	0.968
6	1.995	0.914
7	1.862	0.859
8, 9	1.729	0.805
10 – 12	1.596	0.750

4.2 Evaluation Results

We compare our DCFS technique against the traditional execution using all the cores with minimum frequency. For DCFS, we evaluate three cases with different implementations. DCFS-3 and DCFS-10 are our proposed technique without detecting the behavior changes within a program. The values 3 and 10 indicate the length of the Execution phase, which is constant during the evaluation. DCFS-WD (with detection) is our proposed technique which dynamically detects the behavior changes within a program and switch back to the Training phase.

4.2.1 Results of the Xeon platform

Figure 4(a) shows the performance results of the evaluation on the Xeon platform. The x-axis shows the benchmarks and the y-axis shows the performance normalized to the traditional execution with 12 cores at 1.596 GHz. In this platform, DCFS cannot achieve any speedup for all the evaluated benchmarks except `swaptions`. The reason of this is that the performance of these benchmarks are not saturated even if they are executed with all 12 cores, which suggests that the traditional execution is close to the ideal case. Moreover, DCFS has an additional overhead of the Training phase which slows down the execution. `swaptions` has a characteristic that achieves a higher performance when executed with core counts which is a power of two. Therefore, DCFS detects the best number of cores which is eight and improves the performance for 18% by our DCFS technique. DCFS-WD achieves the highest performance among the DCFS techniques for all benchmarks except `swaptions`. This result indicates that DCFS-WD successfully reduces the number of unnecessary Training phase by detecting the behavior changes of the executed programs.

4.2.2 Results of the Opteron platform

Figure 4(b) shows the performance results of the evaluation on the Opteron platform. The structure of the figure is the same as Figure 4(a) except that the result of DCFS technique is much more interesting. For four benchmarks with high parallelism (`blackscholes`, `swaptions`, `vips`, and `ferret`), DCFS cannot improve performance as expected. The reason is the same as the Xeon platform. `canneal` and `streamcluster` also show almost no or slight performance improvement, and the reason is discussed in the next subsection.

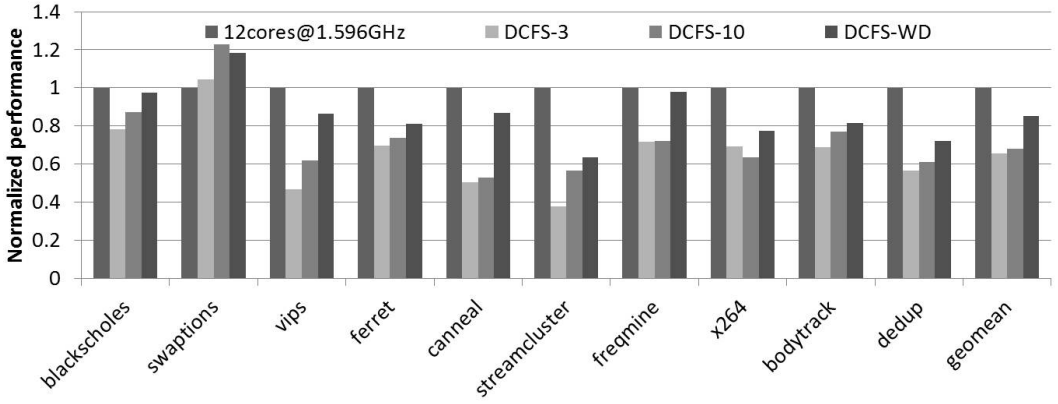
DCFS improves performance of four benchmarks: `freqmine`, `x264`, `bodytrack`, and `dedup`. As can be seen from Table 5, all of these benchmarks does not have high parallelism. Especially, the performance improvement of `dedup` is the highest among all the applications and show a 35% speed up. Further analysis of the reason of this improvement is discussed in detail in the next section with an additional experiment. The geometric mean of the performance improvements with DCFS-WD are 6% for all the ten benchmarks and 20% for the four benchmarks which show performance improvements.

DCFS-WD achieves higher performance compared to DCFS-3 for all benchmarks except `ferret`. This is because DCFS-3 switches to the Training phase every three seconds which is too short, while DCFS-WD reduces the number of unnecessary Training phase to reduce the overhead. However, compared to DCFS-10, DCFS-WD achieves higher performance for only three benchmarks: `blackscholes`, `swaptions`, and `freqmine`. This result indicates that our algorithm to detect the behavior changes within a program is not perfect.

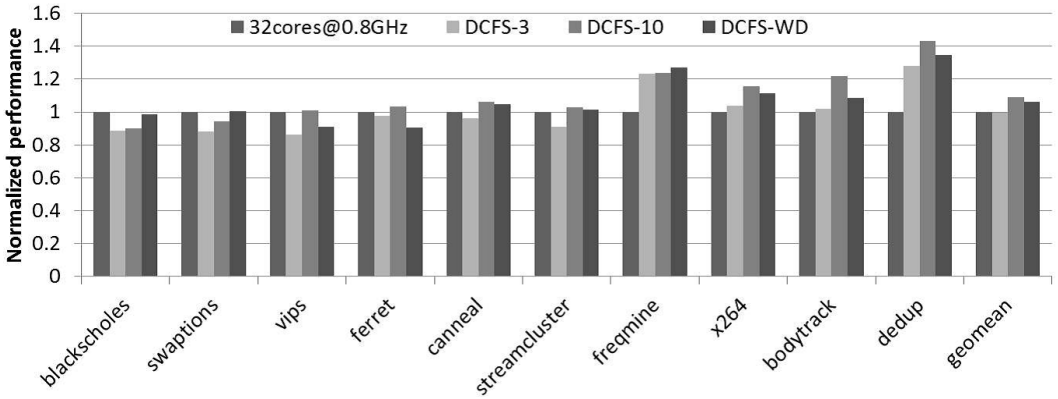
5. Discussions

5.1 Discussions for further improvement

We make some analyses on the applications that DCFS did not work well and discuss how we can improve our technique for future studies. From the result of Xeon, it is clear that our DCFS technique is not effective for this platform because almost all the



(a) Xeon



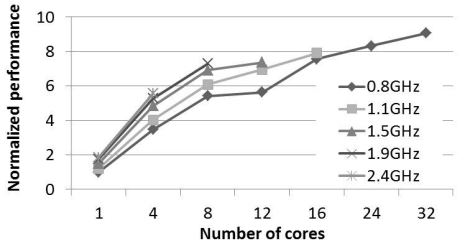
(b) Opteron

Figure 4. Performance normalized to the minimum frequency execution with all cores

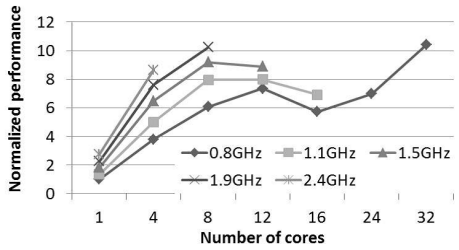
applications show great or moderate scalability. Therefore, the traditional execution with all cores on minimum frequency achieves the best performance. However, DCFS shows great improvement on several benchmarks on the Opteron platform. This was expected from Table 5 where the programs show a variety of parallelisms. It is clear that there will be more and more variability in the scalability of programs when the number of cores becomes larger, and the advantage of DCFS becomes greater in such situations.

For programs such as `blackscholes`, `swaptions`, `vips`, and `ferret` which are classified as high parallelism in Table 5, the best configuration is to execute with all the cores in the system. Even though our DCFS technique is able to find this configuration, all the time spent in Training period becomes an overhead which degrades the overall performance. Theoretically, we can avoid this overhead by switching to the Training phase only if a behavior change is detected such as in DCFS-WD. However, as the result against DCFS-10 showed, our implementation is not perfect and we need to implement a better detection technique which is left for our future work.

Even though `canneal` and `streamcluster` does not have high parallelism as shown in Table 5, DCFS cannot improve performance. According to the work by Bienia et al., these two benchmarks are the most memory-bounded applications of the ten benchmarks we have evaluated [2]. DCFS relies on the characteristics of a program where we can have benefit in performance by trading off the number of cores into additional CPU frequency. However, increasing the CPU frequency cannot speed up the memory-bounded applications because CPU frequency is not the main factor to decide their performance. Figure 5 is a similar graph as Figure 1 which assists this analysis by showing the normalized performance for `canneal` and `streamcluster`. The figures tell us that the performance improvement by increasing CPU frequency is very small compared to applications which we have shown in Figure 1. Similar to the highly scalable applications, DCFS spend time on searching for the best configuration which directly becomes an overhead. This overhead can also be avoided by detecting the characteristics with the number of memory accesses or last-level cache misses (LLC). When we search the best configuration in Training phase, we assume a convex curve for the core counts versus performance. `streamcluster` is an only exception as we can see from Figure 5(b), however, we can find the best configuration in this case as the execution with 32 cores gives us the best performance.



(a) canneal



(b) streamcluster

Figure 5. Relation among performance, core counts and CPU frequency on Memory-bound application

5.2 Detailed analysis to see the benefits of DCFS

We analyze the novelty of DCFS technique with the best performing application which is `dedup` in detail. Figure 6 shows both the IPS and execution time of `dedup` on Opteron platform with three different executions: 0.8 GHz execution with 32 cores, 0.8 GHz with dynamic core scaling along with detection of dynamic behavior changes (DCS-WD or DCFS-WD without CPU frequency scaling), and DCFS-WD. The x-axis shows the execution time and the y-axis shows the IPS. The IPS values are measured every five seconds. Additionally, values under graphs represent the number of cores. The upper values are for DCS-WD@0.8 GHz and the lower values are for DCFS-WD. Total execution time of DCS-WD is few seconds longer than that of 0.8 GHz execution with 32 cores because of the training overhead. The proposed DCFS-WD achieves a much greater performance by dynamically allocating different configurations while achieving high IPS during the whole execution. This suggests that the overhead of runtime training is well compensated for the additional performance improvement which can be obtained by DCFS-WD.

6. Related Work

We pick up three work from the literature that are close to ours and make some comparisons and discussions against them.

6.1 Feedback-Driven Threading [10]

Feedback-Driven Threading (FDT) by Suleman et al. improves the performance and reduce power consumption of multi-threaded applications by dynamically controlling the number of threads using runtime information. FDT predicts the optimal number of threads for loop iteration depending on the characteristics of the program which are the amount of data-synchronization and the degree of demand for bandwidth. Performance of multi-threaded application does not always increase with the number of threads: performance might be saturated or worsened because of data synchronization for heavily data sharing application; performance of data-parallel application might be restricted because of contentions for off-chip bus bandwidth.

In FDT, compiler divides the loop iterations into two parts. One part is for estimating the optimal number of threads by measuring the time spent in the execution of critical sections or the amount of off-chip bus utilization. In the other part, the program is processed in parallel with the estimated number of threads. FDT can both improve performance and reduce power consumption if the performance of the program decreases by increasing the core counts. However, for programs whose performance saturates at some point, FDT can reduce power consumption by decreasing the number of threads but cannot improve performance. With our proposed DCFS technique, we can achieve higher performance by reallocating the power budget which is originally allocated to non-performance contributing cores to performance contributing cores by raising their CPU frequency.

6.2 Intel's Turbo Boost technology [6]

Turbo Boost technology (TB) achieves high performance through the ability of running with a higher frequency over its base operating frequency. It automatically and dynamically allows cores to run faster than the base frequency if the power consumption of processor is below its TDP (Thermal Design Power). CPU frequency is changed depending on the condition of the processor by monitoring the power consumption.

TB heavily depends on the number of active cores. For example, when only one core is active, the power consumption is typically much lower than TDP. In this case, TB can increase the CPU frequency drastically. On the contrary, when all the cores are processing, there is not much room for improvement. Therefore, TB is effective in cases when some number of cores are totally idle.

When we execute a multi-threaded application on a many-core processor, TB can speed up the execution of the sequential portions where only one core processes. However, in parallel portions, OS allocates threads to all cores on a chip making almost no room for TB to play an active role. DCFS can dynamically adapt the number of active cores depending on the characteristics of the application. For applications which does not have enough scalability to make use of all the cores, we can let them execute on a higher frequency by making some cores idle to achieve higher performance than traditional execution with TB.

6.3 Pack & Cap [3]

A more recent work called Pack & Cap proposed by Cochran et al. is the work most close to ours. They use the same two knobs: adapting the number of cores and CPU frequency. Pack & Cap aims to adapt these two knobs to meet the user-defined power constraints which changes dynamically during runtime.

The main differences of their technique against ours is that (1) they use an offline regression classifier that estimates the optimal thread packing and CPU frequency as a function of user-defined peak power constraints, and query this model online to dynamically optimize the setting, while our technique is totally dynamic which requires no static information and can find an optimal configuration with a reasonable overhead; (2) they use a quad core platform for an evaluation that does not meet the scalability problem that we face at a larger number of cores shown for Opteron platform. This will be much more important in the future when dealing with more number of cores and the room of our optimization technique becomes larger as shown in the evaluation results.

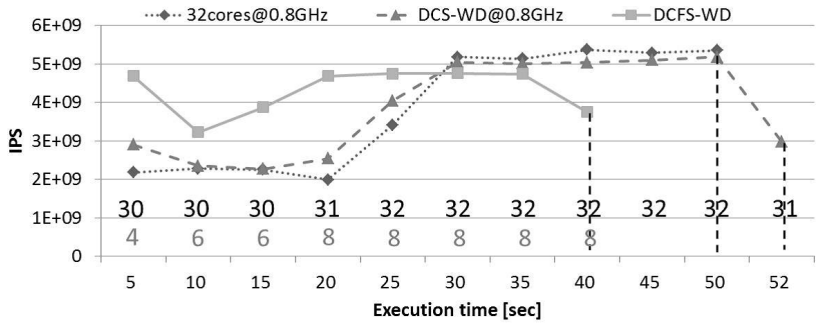


Figure 6. Comparison between execution with all 32 cores and proposal technique

7. Conclusions

Industry is shifting towards many-core processors as the technology size shrinks. Because power consumption being the first order constraint to build microprocessors, future processors are required to achieve high performance within the strictly limited power budget. Traditional approach for this problem is to apply dynamic voltage and frequency scaling (DVFS), which optimizes the trade-off between performance and power consumption. While DVFS offers an efficient execution for single-threaded applications, this is not always the case for multi-threaded applications executed on many-core processors. Operating frequency is an efficient knob to control both the performance and power consumption for single-threaded programs, however, another important factor to consider in multi-threaded applications is its parallelism which does not always guarantee us that using the whole system gives us the best performance.

We propose dynamic core and frequency scaling (DCFS) technique to optimize the power-performance trade-off for multi-threaded applications. Our proposed technique adjusts core counts and CPU frequency depending on the parallelism of applications under the power consumption constraint. DCFS dynamically controls the settings to optimize against the phases within programs by having two phases: Training and Execution. Additionally, DCFS detects the behavior changes of the executed program dynamically. DCFS achieves performance improvement of up to 35% for `dedup` and 6% on average among ten applications from PARSEC benchmarks compared against the execution with all cores equipped on a chip at minimum frequency.

For future work, we would like to evaluate our proposed technique under different power consumption constraints and on several platforms to show the effectiveness of DCFS. Moreover, the algorithm to find the best combination of core counts and CPU frequency must be improved to reduce the overhead. Furthermore, we plan to reduce the overhead of Training phase by implementing the runtime system on the OS kernel to eliminate the overhead associated with calling the system calls.

Acknowledgment

This work was supported in part by New Energy and Industrial Technology Development Organization (NEDO), Japan, Semiconductor Technology Academic Research Center (STARC), and the Grant-in- Aid for Young Scientists (A), 21680005.

References

[1] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56. Ieee, 2008.

[2] C. Bienia, S. Kumar, J. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[3] R. Cochran, C. Hankendi, A. Coskun, and S. Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *Proceedings of the 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '44, pages 175–185, Washington, DC, USA, 2011. IEEE Computer Society.

[4] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 38–43. IEEE, 2007.

[5] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[6] Intel®Corporation. Intel® turbo boost technology in intel® core microarchitecture (nehalem)based processors. Whitepaper, Intel® Corporation, November 2008.

[7] C. Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *Hot Chips 23*, 2011.

[8] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08*, Aug. 2008.

[9] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40. IEEE, 2002.

[10] M. Suleman, M. Qureshi, and Y. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. *ACM SIGPLAN Notices*, 43(3):277–286, 2008.