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.aist.kyushu-u.ac.jp

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

Naoto Fukumoto
Graduate School of Information Science and Electrical Engineering
Kyushu University
fukumoto@soc.aist.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
With power consumption being a first-order constraint of microprocessors, they are required to achieve high performance within a strictly limited power budget. For example, the capping of peak power consumption is strongly desired for large-scale data centers and vast high-performance computing machines. In the future, many-core processors are expected to host a variety of workloads with different characteristics and requirements. Therefore, a runtime environment must be developed that can manage these applications in an energy-efficient manner.

A traditional approach for achieving energy-efficient execution under power consumption constraints is to apply dynamic voltage and frequency scaling (DVFS), which optimizes the trade-off between performance and power consumption [9]. DVFS offers efficient execution of single-threaded applications and multi-threaded applications running on chip multiprocessors [4]. However, such efficiency is not ensured for heavily multi-threaded programs being executed on many-core processors. Setting the central processing unit (CPU) frequency and supply voltage to their maximum available values according to the power budget, is an efficient method of controlling both performance and power consumption. However, another principal factor also exists that determines the performance of multi-threaded applications: scalability or parallelism. Utilizing all of the underlying core counts does not guarantee that the highest performance or the most energy-efficient execution is realized by multi-threaded programs being executed on many-core processors.

In this paper, we propose a dynamic core and frequency scaling (DCFS) technique to optimize the power-performance trade-off for multi-threaded applications. Under power consumption constraints, our technique adjusts the core counts and central processing unit frequency, dependent on the parallelism of the applications. DCFS dynamically controls the optimization parameters according to the behavior within the applications. Evaluation results show that we can, on average, achieve a 6% performance improvement across ten applications taken from the PARSEC benchmark suite, with an improvement of up to 35% for the dedup application. ([Keywords?]]

1. Introduction
Recently, multi-core processors have become the mainstream architecture for microprocessors. The performance of single-core processors has been limited by power consumption constraints, which prevent traditional performance enhancement techniques, such as increasing operating frequency or implementing complex out-of-order wide-issue superscalar processors. The number of cores equipped on a single chip has tended to increase as technology shrinks, and the many-core era is expected to arrive in the near future [5, 7, 8]. However, the key to achieving both high performance and low power in multi-core processors is efficient parallel processing, and the importance of this becomes even greater for many-core processors.

With power consumption being a first-order concern in today’s microprocessors, future processors are required to achieve high performance within a strictly limited power budget. The capping of peak power consumption is strongly desired in large-scale data centers, which are now typically managed by server consolidation, and vast high-performance computing machines in which heavily multi-threaded applications take place. In the future, many-core processors must be able to host a variety of workloads with different characteristics and requirements. Therefore, a runtime environment is needed that can manage these applications in an energy-efficient manner.

A traditional approach for achieving energy-efficient execution under power consumption constraints is to apply dynamic voltage and frequency scaling (DVFS), which optimizes the trade-off between performance and power consumption [9]. DVFS offers efficient execution of single-threaded applications and multi-threaded applications running on chip multiprocessors [4]. However, such efficiency is not ensured for heavily multi-threaded programs being executed on many-core processors. Setting the central processing unit (CPU) frequency and supply voltage to their maximum available values according to the power budget, is an efficient method of controlling both performance and power consumption. However, another principal factor also exists that determines the performance of multi-threaded applications: scalability or parallelism. Utilizing all of the underlying core counts does not guarantee that the highest performance or the most energy-efficient execution is realized by multi-threaded programs being executed on many-core processors.

In this paper, we propose a dynamic core and frequency scaling (DCFS) technique to optimize the power-performance trade-off...
core counts so as not to exceed the power consumption constraint. We assume that the maximum CPU frequency is decided by the **blackscholes** processors, and each processor has eight cores. Therefore, the system is a symmetric multi-processor machine that includes four processors. The system used in the experiment is shown in Table 1. The system is an AMD Opteron-based real system in this experiment.

Table 1. Configuration of the experimental system (AMD Opteron)

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

First, we explain the experimental setup. The system configuration used in the experiment is shown in Table 1. The system is a symmetric multi-processor machine that includes four processors, and each processor has eight cores. Therefore, the system has 32 cores in total. We select three applications from PARSEC (blackcholes, dedup and x264) and use the “native” input set.

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

This section shows that performance characteristics based on core counts and CPU frequency change dependent on the type of application or the phases executed under a power constraint. Note that we execute applications from the PARSEC benchmark suite [2] on an AMD Opteron-based real system in this experiment.

#### 2.1 Experimental setup

First, we explain the experimental setup. The system configuration used in the experiment is shown in Table 1. The system is a symmetric multi-processor machine that includes four processors, and each processor has eight cores. Therefore, the system has 32 cores in total. We select three applications from PARSEC (blackcholes, dedup and x264) and use the “native” input set.

#### 2.2 Assumption on 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 power consumption constraint, $P_{\text{constraint}}$, when all of the applications executed on the AMD Opteron platform. The $x$-axis represents the number of cores assigned to each application, and the $y$-axis represents the normalized performance, where a value of one denotes the performance of a single core execution with the lowest frequency of 0.8 GHz in our experimental setup. The five lines in each figure correspond to executions 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 has an almost 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 the CPU frequency. This notion is quite intuitive because the power consumption of a CPU is proportional to square of the positive supply voltage ($V_D^2$) and the CPU frequency. For example, a comparison of doubling the CPU frequency and doubling the core counts shows that the power consumption doubles in both cases; however, increasing the CPU frequency consumes more power because $V_D^2$ must be increased. This trend can be clearly seen in Figure 1(a), where the 0.8 GHz execution with 32 cores achieves the highest performance within the power constraint.

For applications that saturate the performance by increasing the core counts, such as x264 (Figure 1(b)) or dedup (Figure 1(c)), we can achieve a higher performance within the power envelope by restricting the number of cores and using the surplus power to boost performance by increasing the CPU frequency. As seen from Figure 1(b), the highest performance is found for the 1.5 GHz execution of x264 with 12 cores. The results for dedup are quite interesting, in that increases in CPU frequency give higher performance.

(32) cores run on the minimum available CPU frequency:

$$P_{\text{constraint}} = a \cdot N_{\text{all cores}} \cdot C \cdot f_{\text{min}} \cdot V_{\text{min}}^2.$$  

Here, $a$ is the switching activity of the circuit, $N_{\text{all cores}}$ is the total number of cores on a chip, $C$ is the load capacitance per core, $f_{\text{min}}$ is the minimum CPU frequency, and $V_{\text{min}}$ is the minimum supply voltage. We assume that the capacitance of a processor is proportional to its number of cores. Thus, we calculate the maximum available CPU frequency for each core count such that their power consumption does not exceed $P_{\text{constraint}}$.

Table 2 shows the maximum CPU frequencies and supply voltages that we assume in this study according to the number of cores.

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>CPU frequency [GHz]</th>
<th>Supply voltage [V]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–5</td>
<td>2.400</td>
<td>1.300</td>
</tr>
<tr>
<td>6–8</td>
<td>1.900</td>
<td>1.213</td>
</tr>
<tr>
<td>9–12</td>
<td>1.500</td>
<td>1.125</td>
</tr>
<tr>
<td>13–19</td>
<td>1.100</td>
<td>1.038</td>
</tr>
<tr>
<td>20–32</td>
<td>0.800</td>
<td>0.950</td>
</tr>
</tbody>
</table>

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

2 All executions create 32 threads, and threads are "packed" onto the number of cores, which is represented on the $x$-axis [3]. We use this "thread packing" technique for the remainder of our study.
The total number of dynamic instructions is stable among the performance. IPS is a good indicator of performance because committed instructions per second (IPS), which corresponds to handle different frames of the input video. The y-axis shows the flow of time by showing the results of five consecutive loops, which handle different frames of the input video. The x-axis expresses the flow of time by showing the results of five consecutive loops, which handle different frames of the input video. The y-axis shows the committed instructions per second (IPS), which corresponds to the performance. IPS is a good indicator of performance because the total number of dynamic instructions is stable among thread packing executions with different core counts [3].

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

Figures 1 and 2, and the associated discussion, motivate us to develop a dynamic optimization technique that attempts to maximize the performance within a fixed power budget by controlling the number of cores and the operating CPU frequency. The key concepts are (1) detect the performance characteristics, namely, 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 applications executed on many-core processors under a power consumption constraint. Traditionally, multi-threaded applications are executed by creating a number of threads greater than or equal to the underlying logical core counts in order to fully utilize the system, and the operating system (OS) allocates each thread to all of the cores. However, as we have seen in the previous section, such an execution does not necessarily give the maximum performance or the most energy-efficient execution.

Therefore, we propose a sophisticated technique that dynamically controls both the number of cores and the CPU frequency according to the characteristics of the application. We call this “dynamic core and frequency scaling (DCFS)”. For fully parallelized applications, such as blackscholes in Figure 1(a), we should allocate as great a number of cores as possible. However, for applications with medium to low scalability, optimization is better achieved through other means, as we have discussed earlier. To achieve efficient execution, we have to appropriately allocate the limited power consumption such that two parameters are controlled: the CPU core throttling and frequency scaling.

3.2 Determination of core counts and CPU frequency

Our DCFS technique is composed of two phases: “training” and “execution”. In the training phase, we execute the application under different configurations (different combinations of allocated core number and CPU frequency), each for a short period of time, to identify the associated characteristics. In this study, IPS is used as a measure for this purpose. After the training phase, the optimal core count and CPU frequency that maximize the performance are estimated from the recorded IPS values. The execution phase then simply applies this configuration to the application. These phases are repeated iteratively until the end of the application such that we can follow the dynamic behavior. Note that this technique is totally dynamic and does not require any static information or modification to the application binary. Figure 3 shows an overview of this technique, which we will now explain in detail.

- **Training**
  The training phase dynamically measures IPS while changing the configuration until the optimal core count–CPU frequency pairing has been determined. The key idea is to find the core count with the highest performance for every possible frequency and then compare these to find the best pairing. The algorithm works as follows. First, the application is executed with all cores and the maximum possible frequency (e.g., 32 cores and 0.8 GHz, as seen in Figure 1 (we will continue to refer to the numbers in this Figure in this explanation)) and IPS is measured for a short period of time (called the “training period”). Next, we decrease the number of cores while retaining the CPU frequency (e.g., 24 cores and 0.8 GHz) and again measure the IPS. We continue to decrease the number of cores until the IPS decreases. At that point, we are able to determine the optimal core count for that frequency (0.8 GHz). Here we assume a convex curve for the relation between the CPU frequency and performance, which is somewhat general, and is true for almost all of the benchmarks evaluated in our experiments. Next, we increase the frequency (e.g., 1.1 GHz) and, by again starting from the maximum available core count according to the power cap, decrease the core count while measuring IPS to the point where we see a performance degradation. We repeat this process for all possible frequencies (e.g., 0.8, 1.1,
1.5, 1.9, and 2.4 GHz), and the optimal configuration achieving the highest IPS is determined by comparing the maximum IPS value of each frequency.

### Execution

After the training phase, the optimal core count–frequency combination is applied to the application and executed. However, as we can see from Figure 2, the behavior of the application changes during runtime. Therefore, IPS is measured periodically (every 1 s in our work) to detect any changes, and DCFS returns to the training phase if the current IPS falls outside of a certain range in comparison with the optimal 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 enable periodic access to the performance counters in order to measure IPS. Additionally, we use sched_setaffinity(2), a standard Linux API, to control the CPU affinity of the evaluated program in order to bind the processes to a specific number of cores.

An important parameter of our proposed technique is the training period. We measured the time taken for application behavior to stabilize after changing both the affinity and CPU frequency. The maximum time was 30 ms for the worst case of changing the core count and frequency from their minimum to maximum values.

Therefore, we ignore the first 30 ms after changing the configuration, and use the next 30 ms as the training period. In the execution phase, behavioral changes in the executed application are detected if the current IPS increases or decreases by more than 10 % compared with the optimal IPS, and the technique then switches back to the training phase.

### 4. Evaluation

#### 4.1 Evaluation Platforms

We evaluate our proposed DCFS technique with two types of platforms: AMD Opteron and Intel Xeon. The configuration of the Opteron system has already been shown in Table 1, and that of Xeon is listed in Table 3. Similar to the Opteron platform, in the Xeon 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). The maximum available CPU frequency for each core count in the Xeon platform is then calculated as in Section 2.2 (Table 4). Note that we disable both the Turbo Boost (TB) and Hyper-Threading technologies in the Xeon platform to obtain stable results.

For evaluation, we choose 10 benchmarks from the PARSEC benchmark suite 2.1 [1] and use the “native” input set. We measure the scalability of the benchmarks using the Opteron platform, and classify them into three types according to their parallelisms.
Table 3. Configuration of the evaluation system (Intel Xeon)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Xeon X5670</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of processors</td>
<td>2</td>
</tr>
<tr>
<td>Number of cores per processor</td>
<td>6</td>
</tr>
<tr>
<td>Total number of available cores</td>
<td>12 (2 × 6)</td>
</tr>
<tr>
<td>L1 I/D cache</td>
<td>32 KB × 12</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 KB × 12</td>
</tr>
<tr>
<td>Shared L3 cache</td>
<td>12 MB × 2</td>
</tr>
<tr>
<td>Main memory</td>
<td>96 GB (DDR3-1333)</td>
</tr>
<tr>
<td>Bus speed</td>
<td>6.4 Gb/s</td>
</tr>
<tr>
<td>Technology Size</td>
<td>32 nm</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>CPU frequency [GHz]</th>
<th>Supply voltage [V]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>2.527</td>
<td>2.350</td>
</tr>
<tr>
<td>3</td>
<td>2.527</td>
<td>1.32</td>
</tr>
<tr>
<td>4</td>
<td>2.527</td>
<td>1.025</td>
</tr>
<tr>
<td>5</td>
<td>2.128</td>
<td>0.968</td>
</tr>
<tr>
<td>6</td>
<td>1.995</td>
<td>0.914</td>
</tr>
<tr>
<td>7</td>
<td>1.862</td>
<td>0.859</td>
</tr>
<tr>
<td>8, 9</td>
<td>1.729</td>
<td>0.805</td>
</tr>
<tr>
<td>10–12</td>
<td>1.596</td>
<td>0.750</td>
</tr>
</tbody>
</table>

(2012/3/14)

4.2 Results of the Xeon platform

Figure 4(a) shows the performance results of executions on the Xeon platform. The x-axis lists the benchmarks and the y-axis shows the performance normalized to the traditional execution with 12 cores at 1.596 GHz. On this platform, DCFS cannot achieve speedup in any of the evaluated benchmarks, except swaptions. The reason for this lack of speedup is that the performance of these benchmarks remain unsaturated even when they are executed on all 12 cores, which suggests that the traditional execution is close to the ideal case. Moreover, DCFS has the additional overhead of the training phase, which slows down the execution. swaptions has the characteristic that it achieves a higher performance when executed with core counts that are equal to a power of two. Therefore, DCFS detects the optimal number of cores, which is eight, and improves the performance by 18%. DCFS-WD achieves the highest performance among all the DCFS techniques for all benchmarks except swaptions. This result indicates that DCFS-WD successfully reduces the number of unnecessary training phases by detecting behavioral changes in the executed programs.

4.2.2 Results of the Opteron platform

Figure 4(b) shows the performance results of the executions on the Opteron platform. The structure of the plot is the same as in Figure 4(a), except that the DCFS results are much more interesting. As anticipated, for the four benchmarks with high parallelism (blackcholes, swaptions, vips, and ferret), DCFS cannot improve execution performance. The reason behind this is the same as for the Xeon platform. canneal and streamcluster also show almost no/very slight performance improvements. The reason for this is discussed in the next subsection.

However, DCFS does improve the performance of four benchmarks: freqmine, x264, bodytrack, and dedup. As can be seen from Table 5, all of these benchmarks have medium–low parallelism. In particular, the performance improvement of dedup is the highest among all of the applications and has a speedup of 35%. Further analysis of this improvement is discussed in detail in the next section, along with an additional experiment. By using DCFS-WD, the mean performance improvement over all 10 benchmarks is 6% and over the four benchmarks showing considerable improvement is 20%.

DCFS-WD achieves a higher performance compared with DCFS-3 for all benchmarks, except ferret. DCFS-3 switches to the training phase every 3 s which is too short, whereas DCFS-WD reduces the number of unnecessary training phases, and in turn reduces the overhead. However, compared with DCFS-10, DCFS-WD achieves a higher performance for only three of the benchmarks: blackcholes, swaptions, and freqmine. This result indicates that our algorithm to detect the behavioral changes within an application has yet to be perfected.

5. Discussion

5.1 Further improvement of DCFS

We now analyze the applications in which DCFS did not work well, and we discuss how we can improve our technique in future studies. The Xeon results clearly show that our DCFS technique is not effective for this platform, because almost all the applications show large or moderate scalability. Therefore, the traditional execution using all cores at minimum frequency achieves the best performance. However, DCFS shows a considerable improvement in several of the benchmarks on the Opteron platform. This was expected from Table 5, where the applications show a variety of parallelisms. Clearly, greater variability is found in the scalability of application when the number of cores is increased, and the advantage of DCFS becomes greater in such situations.

For applications such as blackcholes, swaptions, vips, and ferret, which are classified as having high parallelism in Table 5, the best configuration is execution using all of the cores in the system. Even though our DCFS technique is able to find this configuration, the time spent in training phase becomes an overhead and degrades the overall performance. Theoretically, we can avoid this overhead by switching to the training phase only if a behavioral change is detected, such as in DCFS-WD. However, as comparison between the results of DCFS-WD and DCFS-10 showed, our implementation is not perfected and we need to implement a better detection technique. We leave this for future work.

Even though canneal and streamcluster do not have high parallelism (Table 5), DCFS cannot improve execution performance. According to the work by Bienia et al., these two benchmarks are the most memory-bound of the 10 applications that we evaluated [2]. DCFS relies on the application characteristics, and we attain performance benefits by trading off the number of cores and CPU frequency. However, memory-bound applications are not sped up by increasing the CPU frequency, because this is not the main factor influencing their performance. Figure 5 shows similar
Table 5. Classification of evaluated benchmarks according to parallelism

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Benchmark</th>
<th>Speedup against 1 core (Opteron)</th>
<th>Speedup against 1 core (Xeon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>blackscholes</td>
<td>31.6  &amp;</td>
<td>11.9  &amp;</td>
</tr>
<tr>
<td></td>
<td>swaptions</td>
<td>31.6  &amp;</td>
<td>7.1  &amp;</td>
</tr>
<tr>
<td></td>
<td>vips</td>
<td>29.7  &amp;</td>
<td>11.6  &amp;</td>
</tr>
<tr>
<td></td>
<td>ferret</td>
<td>21.4  &amp;</td>
<td>9.4  &amp;</td>
</tr>
<tr>
<td>Medium</td>
<td>freqmine</td>
<td>18.4  &amp;</td>
<td>10.1  &amp;</td>
</tr>
<tr>
<td></td>
<td>x264</td>
<td>16.3  &amp;</td>
<td>10.0  &amp;</td>
</tr>
<tr>
<td></td>
<td>canneal</td>
<td>13.0  &amp;</td>
<td>10.1  &amp;</td>
</tr>
<tr>
<td></td>
<td>bodytrack</td>
<td>12.4  &amp;</td>
<td>5.4  &amp;</td>
</tr>
<tr>
<td></td>
<td>streamcluster</td>
<td>10.4  &amp;</td>
<td>7.7  &amp;</td>
</tr>
<tr>
<td>Low</td>
<td>dedup</td>
<td>3.1  &amp;</td>
<td>3.5  &amp;</td>
</tr>
</tbody>
</table>

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

plots to Figure 1, and assists this analysis by illustrating the normalized performance of canneal and streamcluster. The plots tell us that the performance improvement by increasing CPU frequency is very small compared with the applications shown in Figure 1. Hence, similar to the highly scalable applications, DCFS spends considerable time searching for the best configuration, which becomes a direct overhead. This overhead can be avoided by detecting the characteristics alongside the number of memory accesses or last-level cache misses. When we search for the optimal configuration in the training phase, we assume a convex curve for the relation between the core counts and performance. streamcluster is an exception to this, as we can see from Figure 5(b); however, we can...
still find the best configuration in this case since the execution with
32 cores gives the optimal performance.

5.2 Detailed analysis showing the benefits of DCFS
We analyze the value of the DCFS technique by examining the best
performing application, dedup, in detail. Figure 6 shows IPS (y-
axis) and the execution time (x-axis) of dedup on the Opteron
platform for three different executions: the 0.8 GHz traditional
execution with 32 cores, 0.8 GHz execution with dynamic core
scaling along with detection of dynamic behavioral changes (DCS-
WD or DCFS-WD without CPU frequency scaling), and DCFS-
WD. IPS values are measured every five seconds. Additionally, the
values beneath the plotted points represent the number of cores.
The upper values are for DCFS-WD: 0.8 GHz and the lower values
are for DCFS-WD. The total execution time of DCFS-WD is a few
seconds longer than that of the 0.8 GHz traditional execution with
32 cores because of the training overhead. The proposed DCFS-
WD considerably improves performance by dynamically allocating
different configurations while achieving high IPS throughout the
execution. This suggests that the overhead of runtime training for
DCFS-WD is well compensated for by the obtained additional
performance improvement.

6. Related Work
We now select three works from the literature that are similar to
ours. We describe each of these works and, by making comparisons
with DCFS, outline their shortcomings.

6.1 Feedback-driven Threading [10]
Suleman et al. ’s feedback-driven threading (FDT) improves per-
formance and reduces power consumption of multi-threaded ap-
lications by dynamically controlling the number of threads using runtime information. FDT predicts the optimal number of threads
for loop iteration dependent on two application characteristics: the
amount of data-synchronization and the degree of bandwidth de-
mand. The performance of a multi-threaded application does not
always increase with the number of threads; for applications with
heavy data sharing, the performance might be saturated or wors-
ened because of data synchronization; for data-parallel applica-
tions, the performance might be restricted because of conflicts over
off-chip bus bandwidth.

In FDT, the compiler divides the loop iterations into two parts.
One part estimates the optimal number of threads by measuring the
time spent executing critical sections or the amount of off-chip bus
utilization. The other part processes the application in parallel with
the estimated number of threads. FDT can both improve perfor-
man cne and reduce power consumption if the performance of the
application decreases by increasing the core counts. However, for ap-
plications whose performance saturates at some point, FDT can re-
duce power consumption by decreasing the number of threads, but
cannot improve performance. With the proposed DCFS technique,
we can achieve higher performance by reallocating the power bud-
get, which is originally allocated to non-performance contributing
cores, to performance contributing cores by raising their CPU fre-
cquency.

6.2 Intel’s TB technology [6]
TB technology achieves high performance through the ability to run
a processor at a frequency higher than its base operating frequency.
TB automatically and dynamically allows cores to run faster than
their base frequency if the power consumption of the processor is
below its thermal design power (TDP). CPU frequency is changed
dependent 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 of the cores are
processing, there is little room for improvement. Therefore, TB is
effective in cases when some of the 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
with only one core processes. However, in parallel portions, the OS
allocates threads to all cores on a chip such that TB cannot play an
active role. DCFS can dynamically adapt the number of active cores
depending on the characteristics of the application. Applications
that do not have sufficient scalability to make use of all of the cores
can be executed at a higher frequency by making some of the cores
idle. Thus, higher performance is achieved than for a traditional
execution with TB.

6.3 Pack & Cap [3]
A more recent method proposed by Cochran et al., called “Pack
& Cap”, is the work most similar to ours. They use the same two
parameters: adapting the number of cores and CPU frequency. Pack
& Cap aims to adapt these two parameters to meet user-defined
power constraints that change dynamically during runtime.

The main differences between their technique and ours are the fol-
lowing. (1) They use an offline regression classifier that esti-
mates the optimal thread packing and CPU frequency as a func-
tion of user-defined peak power constraints, and query this model
online to dynamically optimize the setting. In contrast, our tech-
nique is totally dynamic such that it requires no static informa-
tion, and can find an optimal configuration with a reasonable overhead.
(2) They use a quad-core platform for evaluation that does not en-
counter the scalability problem that we face with a larger number
of cores on the Opteron platform. This will become more important
in the future when dealing with a greater number of cores, and the

![Graph a](image1.png)

(a) canneal

![Graph b](image2.png)

(b) streamcluster

Figure 5. Relation among performance, core counts and CPU fre-
cquency on Memory-bound application.
applicability of our optimization technique will increase, as shown in the evaluation results.

7. Conclusions

Industry is shifting toward many-core processors as the size of technology shrinks. Since power consumption is a first-order constraint when building microprocessors, future processors are required to achieve high performance within a strictly limited power budget. The traditional approach for resolving this problem is to apply dynamic voltage and frequency scaling (DVFS), which optimizes the trade-off between performance and power consumption. While DVFS offers efficient execution for single-threaded applications, efficiency is not ensured for multi-threaded applications executed on many-core processors. Varying the operating frequency is an effective method for controlling both the performance and power consumption of single-threaded programs. However, another important factor to consider in multi-threaded applications is parallelism, since this does not always guarantee that using the whole system gives the optimal performance.

We have proposed a dynamic core and frequency scaling (DCFS) technique to optimize the power-performance trade-off for multi-threaded applications. Our proposed technique adjusts the core count and CPU frequency parameters under a power consumption constraint, dependent on the parallelism of the application. DCFS dynamically controls these parameters to optimize the performance of applications by having two phases: training and execution. Additionally, DCFS dynamically detects behavioral changes in an executed application. Compared with execution using all of the cores equipped on a chip at minimum frequency, DCFS achieves a performance improvement of up to 35% for the dedup application, and a 6% improvement, on average, among ten applications from PARSEC benchmark suite.

For future work, we would like to evaluate the proposed technique under different power consumption constraints and on several platforms in order to show the effectiveness of DCFS. Moreover, the algorithm to find the optimal core count–CPU frequency must be improved in order to reduce overhead. Furthermore, we plan to reduce the overhead of the training phase by implementing the runtime system on the OS kernel in order to eliminate the costs associated with system calls.

Acknowledgment

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

References


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