A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs

Yang Ding, Mahmut Kandemir, Padma Raghavan, Mary Jane Irwin
Department of Computer Science & Engineering
Pennsylvania State University, University Park, PA 16802, USA
{yding, kandemir, raghavan, mji}@cse.psu.edu

Abstract

In parallel to the changes in both the architecture domain – the move toward chip multiprocessors (CMPs) – and the application domain – the move toward increasingly data-intensive workloads – issues such as performance, energy efficiency and CPU availability are becoming increasingly critical. The CPU availability can change dynamically due to several reasons such as thermal overload, increase in transient errors, or operating system scheduling. An important question in this context is how to adapt, in a CMP, the execution of a given application to CPU availability change at runtime. Our paper studies this problem, targeting the energy-delay product (EDP) as the main metric to optimize. We first discuss that, in adapting the application execution to the varying CPU availability, one needs to consider the number of CPUs to use, the number of application threads to accommodate and the voltage/frequency levels to employ (if the CMP has this capability). We then propose to use helper threads to adapt the application execution to CPU availability change in general with the goal of minimizing the EDP. The helper thread runs parallel to the application execution threads and tries to determine the ideal number of CPUs, threads and voltage/frequency levels to employ at any given point in execution. We illustrate this idea using two applications (Fast Fourier Transform and MultiGrid) under different execution scenarios. The results collected through our experiments are very promising and indicate that significant EDP reductions are possible using helper threads. For example, we achieved up to 66.3% *and* 83.3% *savings in EDP when adjusting all the parameters properly in applications FFT and MG, respectively.*

1 Introduction

Chip multiprocessors (CMPs) are becoming increasingly popular as performance improvements brought by increasing clock frequency alone are approaching their limits. Other factors, such as ease of verification/validation of individual cores (as compared to complex unicore architectures) and the ability to exploit both thread level (coarse grain) and instruction level (fine grain) parallelism, also boost trends towards chip multiprocessing. Unfortunately, while several CMPs [1, 2, 5, 23, 42] have already made their way into the commercial market, software support for CMPs is still in its infancy, and is expected to be the main roadblock to the effective use of CMPs [38].

In addition to the changes in architecture domain, we are also witnessing changes in application characteristics and target optimization metrics. More specifically, applications are getting increasingly complex and data intensive (particularly large codes from scientific computing, database and embedded image/video processing domains), and optimization metrics other than performance are becoming increasingly important. Two of these metrics are availability and energy consumption. In many execution scenarios where CMPs are involved, satisfying both these metrics (i.e., achieving high availability and low energy consumption) can be critical. One of the interesting problems in this context is to adapt application execution to varying hardware resources in a performance and energy efficient manner, that is, use the available resources carefully to achieve both good performance and low power consumption.

Motivated by these observations, this paper studies the problem of how an execution can cope with CPU availability change. Our goal is to decide – at runtime – the best strategy to employ when the number of CPUs available to an application is changed, considering the energy-delay product (EDP).[1] In other words, we want to adapt the execution to CPU availability change with the goal of minimizing the EDP. The approach proposed in this paper employs a *helper thread* for this purpose. More specifically, we use a helper thread whose primary job is to collect – using performance counters information (or data) provided by the CMP architecture and a power model – energy-delay product statistics during the course of execution and decide the most appropriate number of CPUs, number of threads, and voltage/frequency levels to use when a variation on resource availability occurs. In making these decisions, the

[1] We want to emphasize that energy-delay product (EDP) [22] is an important metric as it captures our desire of both achieving high performance and reducing energy consumption, both of which are critical in data intensive computing.

helper thread uses curve fitting and data interpolation methods [44]. Assuming long application execution time, using such a helper thread, our approach can collect a sufficient number of data points (CPU count, thread count, voltage/frequency level, and EDP value) at runtime. Based on these data points, it can accurately predict the behavior of the application and can then select the CPU count, thread count and voltage level to use when the number of CPUs available to it dynamically changes at runtime.

We implemented our approach using a full system simulator [6] and conducted experiments using two data-intensive applications: Fast Fourier Transform (FFT) and MultiGrid (MG). In our experimental evaluation of the proposed approach, we tested its three variants: (1) using a fixed number of threads and a fixed voltage/frequency level (i.e., adapting only the number of CPUs); (2) adapting both CPU count and thread count (under a fixed voltage/frequency level); and (3) adapting the CPU count, thread count and voltage level together in a coordinated fashion. The experimental results we collected indicate that our proposed helper thread based dynamic adaptation scheme is very successful in practice even if all the overheads brought by the helper thread are accounted for. Specifically, we are able to reduce the EDP value in scenarios (1), (2) and (3) mentioned above by as much as 21.7%, 54.6%, and 66.3% in FFT, and as much as 35.5%, 77.2%, and 83.3% in MG. The corresponding average EDP improvements are 7.4%, 26.3% and 46.1% in FFT under scenarios (1), (2) and (3), and 15.3%, 59.7% and 70.5% in MG. In addition, we also compared our approach to an *optimal* – but unimplementable – adaptation strategy. The collected statistics with our helper thread based approach and the optimal adaptation scheme reveal that, on average, we are within 5.9% of the optimal. Overall, our results indicate that, in order to minimize the EDP in a CMP based architecture, we need to select the number of CPUs, number of threads, and voltage/frequency levels very carefully, and a helper thread can be very useful for this purpose. Note that, while we present our analysis using two applications, our approach is quite general and can be used for other application domains as well.

The rest of this paper is structured as follows. The next section discusses the related work. Section 3 introduces the CMP architecture targeted by our work and explains the execution scenario considered. Our helper thread based approach to the CPU availability problem is discussed in Section 4. Experimental setup and the results from our experimental evaluation are presented in Sections 5 and 6, respectively. Section 7 gives a summary of our major conclusions and a brief outline of the planned future work.

2 Related Work

This section presents a discussion of the prior work on CMPs, CPU adaptation, voltage scaling and other related topics, and compares these efforts to our work.

Chip multiprocessors have been studied in the past from the perspective of hardware [11, 30, 41] as well as software [36, 37, 45]. In comparison to these efforts, the work described in this paper focuses on application adaptation under varying CPU availability. There have been several prior publications on deciding the number of CPUs to employ in different program phases. Earlier work in this area by Hall and Martonosi [25] points out that compiler-parallelized applications may waste various computational resources in different program phases. To remedy this, they propose a mechanism to dynamically adjust the number of CPUs which in turn improves workload performance. There are two main differences between this study and our work. First, our main goal is to minimize the energy-delay product, whereas they focus mainly on performance. Second, in their work, the number of CPUs is adjusted to react program behavior, while our approach aims to adapt the application execution to variations in CPU availability.

Curtis-Maury et al [15] build a user-level library framework for online adaptation of multi-threaded codes targeting low-power and high-performance, which changes the processors/threads configuration as the program executes. An online performance prediction model is used to predict performance, power and combined metrics based on phases, and the model is evaluated on a server composed of four processors. In comparison, our work focuses more on adapting application execution to the dynamic CPU availability. Also, we use a full system CMP simulator and use a power model based on program characteristics, which is not the case in their work. In addition, we employ voltage/frequency scaling. This paper builds upon our previous work [18] which looks at optimal savings that are possible when adapting program at runtime.

Dynamic Voltage-Frequency Scaling (DVFS) has been discussed in the past to reduce power consumption of CMPs. Wu et al [45] study the effectiveness of a dynamic compiler-driven voltage scaling scheme. In comparison, our work considers multiple techniques in adapting application execution. Li et al [33] explore a multi-dimensional design space for CMPs, which includes the CPU count and operating voltage/frequencies. However, they do not consider dynamic adaptation at runtime. Li and Martinez [32] discuss the viability of changing the number of concurrent processors/threads at run-time to accommodate changes in the execution environment. Their work studies several heuristics to prune the search space for determining the optimum number of CPUs to employ and the voltage/frequency levels to use. They assume at most one application thread is executed on each CPU, which restricts potential search space. As a result, they do not consider modifying the thread structure of the application. Also, their main target metric is power consumption, whereas we consider energy-delay product, which we believe is a more suitable metric for many data-intensive computing environments.

Thread migration in the context of CMP has been studied in the past [13, 21, 26, 27, 35, 39]. Constantinou et al [13] investigate the performance implications of single thread migration on CMPs. The experimental results they provide show that the performance loss due to activity mi-

gration can be kept at minimum when several techniques are used. Thread migration has also been applied in several studies [21, 26] to manage power density. The main difference between most of these prior efforts and ours is that we focus on adapting application behavior to CPU availability change and target the energy-delay product.

Recently, there have been several efforts on dynamic resource partitioning in the context of CMPs. Guo et al [24] focus on microarchitecture and software support in order to provide a guarantee of a certain level of performance. They propose optimization techniques to improve throughput when applications with diverse requirements exercise the same CMP. Chu et al [12] propose a profile-guided method for partitioning memory accesses across distributed data caches. Such a data partitioning reduces stall cycles for memory accesses and achieves overall performance speedup. In comparison, our work focuses on how to utilize the resources with both good performance and low power consumption.

Our work is also different from the classical fault tolerance oriented research [7]. Specifically, our focus is not on the techniques for failure recovery or load balancing. Rather, we try to minimize the EDP at runtime through careful selection of CPU count, thread count and voltage/frequency levels in a CMP based architecture.

3 CMP Architecture and Target Scenario

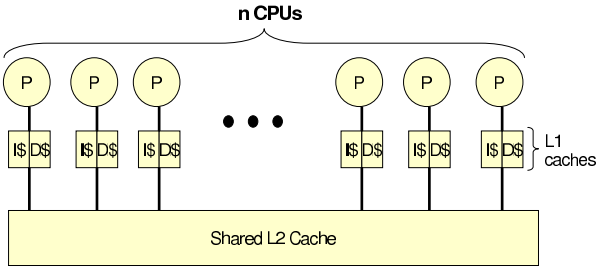

Figure 1. CMP architecture considered in our work. L1 instruction and data caches are private, whereas the unified L2 cache is shared across all CPUs.

In this section, we introduce the CMP architecture assumed by our work and explain the CPU availability problem. Our solution to this problem is detailed in the next section. The CMP architecture considered in this work is of the type shown in Figure 1. There are n CPUs in total. Each CPU has its own private instruction and data L1 caches, and all CPUs share a unified on-chip L2 cache. Note that several commercial and academic CMP architectures discussed in literature fit into this template [1, 2, 42], although future CMPs are likely to have a distributed and banked L2 cache accessed through an interconnection network. We assume that, if the application is not using a CPU, that CPU and its associated L1 caches can be turned off to save power. As leakage power is becoming an increasingly important component of the overall power budget [8], turning off a CPU and its L1 caches can reduce the chip-wide power consumption dramatically. Several architectural techniques [20, 46] can be used for turning off L1 caches. We further assume that the CPUs in our CMP can be voltage/frequency scaled. Scaling down the frequency of a CPU increases the length of its clock period (which in turn increases application execution latency), and scaling down its voltage reduces power consumption. Therefore, interesting power-performance tradeoffs can be studied by playing with the voltages/frequencies of CPUs in a CMP environment. In this CMP architecture, multiple applications can execute at the same time, but we assume that different applications do not share any CPU. That is, a given CPU can only run threads from a single application at any time in execution.

Figure 2(a) illustrates the execution scenario we focus in this paper. In this scenario, an application executing on a CMP (say, for simplicity, n threads running on n CPUs) is informed during its execution that one or more of the CPUs it currently uses are about to be taken away from it. This can be due to several reasons such as a pending thermal emergency, increase in transient errors, or as a result of an operating system (OS) decision. The approach described in this paper is applicable to all these scenarios; in fact, the actual reason behind such unavailability is orthogonal to the main focus of this paper. Note that the CPU availability can change in the other direction as well (i.e., the number of available CPUs to an application can increase at runtime). Our approach handles the increased or reduced CPU availability with a general scheme. Therefore, without loss of generality, we assume that the OS decides (based on its global resource management policies) to take away some of the CPUs on which the application is running. In this case the execution should somehow adapt to this new condition. As our focus is not on techniques for failure recovery, issues of thread imaging and restart after CPU failure is beyond the scope of this paper. In the rest of this section, we discuss the different adaptation options that can be considered:

• The first, and the most intuitive option, is to let the OS re-map (migrate) the threads that were originally running on the CPUs to be taken away to available CPUs. For example, if m of the original n CPUs become unavailable, the threads on these m CPUs can be migrated to the remaining $n - m$ available CPUs. While many types of migration (re-mapping) schemes can be employed for this purpose, the different re-mappings can lead to dramatically different results. Figure 2(a) depicts a specific migration scheme with $n = 16$ and $m = 2$. In this migration scenario, m of the $n - m$ available CPUs take the m orphan threads, and this leads to an increase in their workload.

• While trying to use all available $(n - m)$ CPUs may be reasonable from the performance perspective alone, it may *not* be the best option when we consider the EDP. More specifically, *is it possible to use fewer CPUs than $n - m$ and achieve a lower energy-delay product?* This may be possible as the unused CPUs (and their L1 caches)

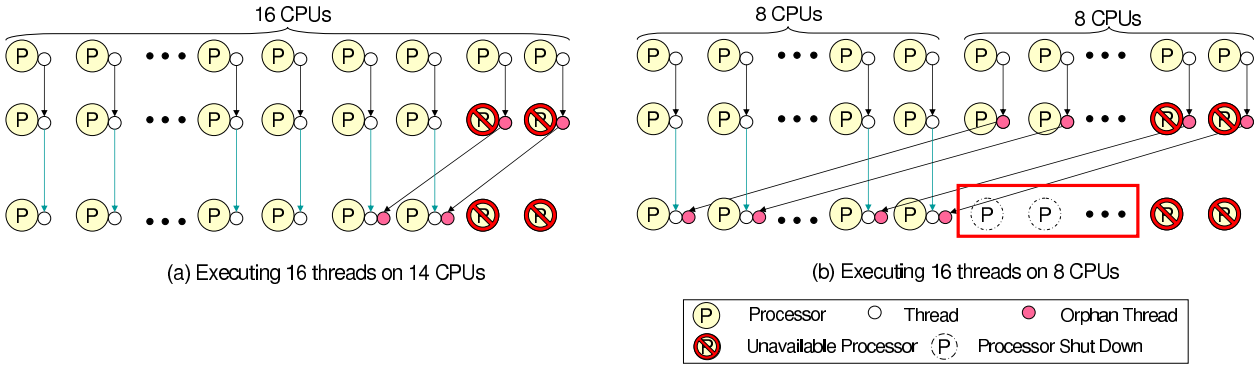

Figure 2. Illustration of reduced CPU availability. (a) When the two (rightmost) CPU become unavailable, their threads are migrated (re-mapped) to available CPUs. (b) When using fewer CPUs, unused ones can be turned off, potentially leading to a lower EDP value.

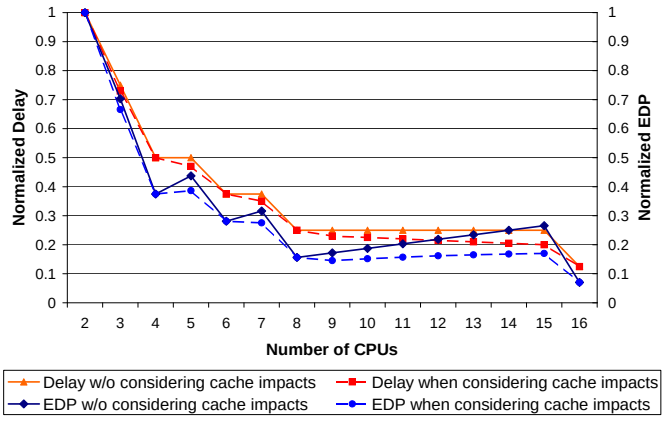

Figure 3. Example delay and EDP curves.

can be turned off, as explained earlier, to save leakage energy. Meanwhile, such a reduction in the number of CPUs and L1 caches used does not always significantly affect the performance. As a potential scenario, based on the thread migration scheme given in Figure 2(a) and omitting the potential changes in cache behavior due to CPU/L1 turnoff, one can expect the execution delay and EDP curves to be as shown in Figure 3 (assuming n is 16). The main observation from this plot is that the large drops in latency (due to parallelization) occur only in certain CPU counts. This is a typical characteristic of many multi-threaded applications [19] because load balance changes along with the number of available CPUs. In fact, the execution delay curve drawn in Figure 3 follows $\lceil \frac{16}{number\ of\ CPUs} \rceil$. Therefore, as an example, if $n = 16$ and two CPUs become unavailable (i.e., $m = 2$), we may work with $q = 8$ CPUs (instead of $n - m = 14$ CPUs) as the latencies with these two configurations are expected to be similar (see Figure 2(b)), save for the cache behavior. In addition, since 14 CPUs will have higher power consumption than 8 CPUs, EDP with 8 CPUs will be lower than that with 14 CPUs, as also illustrated in Figure 3 (in fact, as we increase the CPU count from 8 to 14, the EDP value continuously increases without considering cache impacts). Note however that 14 CPUs have more aggregate on-chip L1 cache capacity, and as a result, in reality using 14 CPUs may result in lower execution latency than using 8 CPUs. On the other hand, the operating system support for systems with different number of CPUs can bring different performance overheads. Therefore, in practice, the minimum EDP value could be caught with a CPU count of q, where $8 \leq q \leq 14$, as also depicted in Figure 3.

• So far, we implicitly assumed that, when m CPUs become unavailable, we still try to continue execution with n threads. Now, supposing that we are also able to obtain a version of the application that can use a different number of threads, we may want to use an r-thread version such that the total EDP value is further minimized. The important point here is the question of how such a version (re-threading) can be obtained. One option is dynamic code modification which can generate any desired version at runtime. However, this may be costly in some cases as compilation time required for re-threading contributes to the overall execution latency. We are investigating the possible overheads and benefits of this option using the Microsoft Phoenix framework [3]. Note that, for applications using OpenMP [16], the cost of re-threading can be much less since the number of threads can be set at runtime using an environment variable [31]. Another option would be to prepare, for each function of the application, a different version (that uses a different number of threads) beforehand at compile time and use (link) the appropriate (multi-threaded) version at runtime when the CPU unavailability takes place. This option, also called static versioning, is possible in general since we can expect the same set of functions to execute for many times in a large and long-running application. In practice, we may not need all possible versions of each function with all possible thread counts. Instead, we can do a reasonably good job with only a small set of versions (e.g., 2, 4, 8, and 16 thread versions) and switch to the appropriate (multi-threaded) version at runtime. An important issue at this point is the place in the code at which the new version will be invoked. Obviously, such an invocation cannot take place at any arbitrary point since this would re-

quire us to prepare versions considering all possible points where CPU unavailability can occur. Instead, one can adopt the following strategy: when the unavailability occurs, we select a suitable number of CPUs and continue executing the application without changing the number of threads until a function boundary is reached. When this boundary is reached, we change the number of threads as well. This approach is illustrated in Figure 4. In this example, if the CPU unavailability occurs in function $f2$ at iteration i, (i.e., during the ith visit of the function), we can continue to use $n = 16$ threads on q CPUs (where q is the number of CPUs we select considering the EDP) until the return of $f2$. When this happens, we start to use r threads ($r \leq 16$) and execute them on p CPUs (again r and p are selected based on the EDP). In this paper, we use the notation of (a, b) to represent an execution of the application using a threads running on b CPUs.

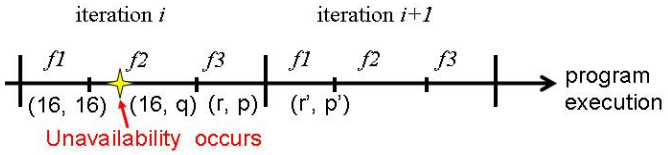

Figure 4. Illustration of how we change the number of CPUs and threads to cope with reduced CPU availability.

• The results may be even better, from an EDP perspective, if voltage/frequency scaling is employed. Suppose that, based on the analysis outlined above, we decided to use p CPUs and r threads. It may be possible to reduce the EDP further by using p' ($p' > p$) CPUs and r' ($r' > r$) threads with the voltages and frequencies scaled down. In this work, we assume that all the CPUs use the same voltage/frequency level. That is, when voltage/frequency scaling is applied, their voltages/frequencies are scaled to the same value.

Clearly, one can expect the best (minimum) EDP value when the number of threads, number of CPUs and voltage/frequency level to use are selected carefully, considering the interactions among them. In the rest of the paper, we explain and experimentally evaluate a helper thread based approach to this dynamic adaptation problem.

4 Helper Thread Based Approach to CPU Availability Problem

In this section, we explain our helper thread based approach to the CPU availability problem. The motivation for helper threads comes from the observation that future CMPs will include a large number of CPUs and will execute applications that involve a large number of threads [9]. In such an execution environment, it is entirely feasible to allocate some of the CPUs and threads to various tasks that enable better application execution. For example, one can use helper threads to collect valuable statistics about application execution and execution environment, using performance counters and sensors. As another example, several recent efforts [29, 34] employed helper threads for prefetching data in cache memories. In a scenario with a large number of CPUs and threads, one can expect the additional costs brought by such helper threads to be more than compensated by the EDP benefits they bring.

In our context, we use a single helper thread which collects performance counter information and implements *curve fitting* to predict the ideal number of CPUs, threads, and voltage/frequency level to use when a variation in CPU availability occurs. In the following paragraphs, we explain the use of the helper thread under three different scenarios, each of them being more complex than the previous ones.

In the first scenario, we assume that the only control parameter we have is the number of CPUs. That is, in reacting to a variation in CPU availability at runtime, we can only change the number of CPUs; the number of threads and voltage/frequency levels are not changed (we use the current number of threads and the maximum voltage/frequency level). As explained earlier in Section 3, when m of the original n CPUs become unavailable to an application, it may not always be the best choice to use all $n - m$ available CPUs when considering the EDP as the primary metric of optimization. In this scenario, the functionality of the helper thread can be explained as follows. As the execution progresses, the helper thread collects statistics, with the help of performance counters and a power model, that keep track of the EDP values obtained under various number of CPUs experienced so far. When a variation on CPU availability occurs, the helper thread uses the data collected so far (which is essentially a set of [Number of CPUs, EDP Value] pairs observed thus far in execution), and makes use of curve fitting to predict the ideal number of CPUs to use to achieve the lowest EDP value. While our current implementation uses a particular curve fitting scheme (piecewise cubic spline interpolation [44]), the selection of the curve fitting scheme to employ is really orthogonal to the main focus of our approach. Note that, when an adaptation takes place, the helper thread continues to record the EDP value observed and updates its database to achieve better predictions in the future. It is important to point out that some of the [Number of CPUs, EDP Value] pairs can also be obtained using profiling or from prior executions of the same application. These pairs can then be reused in the current execution for predictions. To summarize, our helper thread implements two functionalities. First, it maintains the applications EDP values observed under different CPU counts. Second, it decides the next configuration to use through curve fitting.

Figure 5 gives an example of the curve fitting in such scenario, assuming two CPUs out of the original 16 CPUs become unavailable to our application at some point during the course of execution. If we have initial points of EDP at CPU counts of 2, 8 and 16, we can predict that the optimal point to operate is with 14 CPUs as illustrated by the dotted curve in Figure 5. After the execution of current function

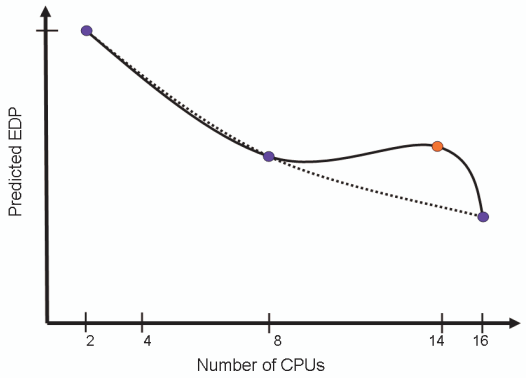

Figure 5. One-dimensional (curve) fitting. Both curves capture the predicted EDP values. As more data points are collected at runtime, the curve fitting becomes more accurate.

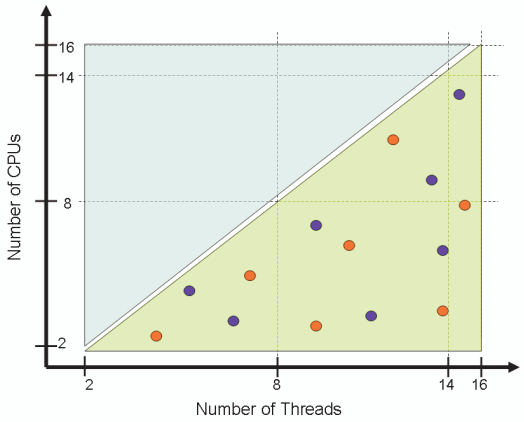

Figure 6. Two-dimensional (surface) fitting. Data points in the lower triangle can be used to predict the next configuration to minimize the EDP.

call, the energy-delay product can be calculated (using the performance counters supported by the underlying CMP architecture) and used as another data point (at CPU count of 14) for future curve fittings. This is captured by the solid curve in Figure 5, and as a result, 9 CPUs are chosen to continue the execution. That is, as more data points are collected, our curve fitting becomes more accurate. We explain the further details of this adaptation scheme in our experimental evaluation.

In the second scenario, the control parameters we have are the number of CPUs and the number of threads. That is, in reacting to a variation in CPU availability, the helper thread selects a new CPU count and a new thread count. However, as explained earlier in Section 3, thread count changes can only be performed at function boundaries (our current implementation postpones the thread count change to the next function boundary). As in the case of first scenario, the helper thread makes its prediction using curve fitting. Clearly, the curve fitting in this case is a two dimensional one (i.e., it is actually surface fitting) as it involves predicting both thread and CPU counts, and our current implementation uses triangle-based linear interpolation for this purpose [44].

Figure 6 illustrates the two-dimensional space for the number of CPUs and the number of threads, assuming both of them are no more than 16. It is clear that using more CPUs than the number of threads can bring no additional performance gains but only wastes extra power. Therefore, the exploration space in Figure 6 is actually only the lower triangle portion. Ideally, we want to find the optimal point in the triangle (i.e., the best (a, b) configuration), when rethreading is possible. Note that, if the number of threads is fixed as in the first scenario, the exploration translates into finding the optimal point in a certain vertical line in Figure 6. Therefore, the first scenario is just a special (and simpler) case of the second one.

In the third scenario, the control parameters we use include CPU count, thread count and voltage/frequency levels. That is, we also predict the new voltage/frequency level to use over the second scenario explained above. As stated earlier, in this study, all the CPUs use the same voltage/frequency level, but it is possible to extend our approach to cover cases where we have the flexibility of changing the voltage/frequency level of each CPU independent of the other CPUs. As in the pervious two scenarios, we use curve fitting (in this case it is three dimensional) to predict the best operating point. The particular curve fitting scheme used in this work is triangle-based linear interpolation [44].

5 Experimental Setup

5.1 Applications

In this section, we briefly describe the two applications used in this paper: Fast Fourier Transform (FFT) and Multi-Grid (MG). Both of these programs are from the NAS Parallel Benchmark Suite [4]. We used their OpenMP implementations in version 3.2 with the class W inputs [28]. Both these codes represent computations that are used in a large variety of modeling and simulation applications based on finite element, finite difference and spectral methods.

Fourier Transforms are important in many domains such as digital signal processing and solving partial differential equations. Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. A Discrete Fourier Transform (DFT) of length N can be represented as a sum of two DFTs of length $N/2$, and this process can be repeated recursively to obtain a divide and conquer algorithm. This partitioning significantly reduces the computational cost of the DFT, hence the term Fast Fourier Transform (FFT). The FFT imple-

mentation we use has six major iterations, which altogether take 90% of the serial execution time. Each iteration performs the kernel of a three-dimensional FFT in function *fft*. This three-dimensional FFT kernel implements three one-dimensional FFT's, corresponding to three function calls within each iteration (*cffts1*, *cffts2* and *cffts3*). The behavior of these three functions are not the same but very similar.

The MultiGrid (MG) method is typically used for solving elliptic partial differential equations on a discretized physical domain. The main idea behind MG is to represent the physical domain with a hierarchy of discretization from fine grain to coarse grain. The solution for the physical domain is obtained with extrapolation between coarser and finer grids. The MG implementation we use has four major iterations, which consume 70% of the total serial execution time. It computes the solution of the three-dimensional scalar Poisson Equation. Each loop iteration consists of the multigrid operation (function *mg3P*) and the residual calculation (function *resid*). The multigrid operation is performed with a sequence of different computations, implemented in functions such as *rprj3*, *psinv*, *interp* and *resid*. Since there are a large number of function invocations within one iteration, we divide them into 7 function groups separately by each function call of *resid*. This partition makes our experiments easier.

5.2 Simulation Setup

We used the Simics toolset [6] to perform our experiments. Simics is a multiprocessor simulator that can be used to perform full system simulation. The abstraction of the CMP architecture used in this study is given earlier in Figure 1. In our simulator, each CPU runs unmodified Solaris 9 operating system with Sun Studio compilers and tools to support OpenMP [14]. Table 1 gives the major simulation parameters used in this study with their default values. In this work, $(16, 16)$ corresponds to our default configuration, before the CPU unavailability occurs (i.e., 16 threads running on 16 CPUs). If no re-threading is used, after the unavailability, we use the configuration $(16, x)$, where $x < 16$. If re-threading is also an option, other configurations are also possible. We assume that, in both FFT and MG, the CPU unavailability takes place after the kernel computation starts (i.e., when entering the major loops). If m CPUs become unavailable, configuration $(16, 16 - m)$ is used as default to continue the execution. Our *baseline* (against which we compare our approach) is the case when the thread count and the CPU count do not change, and the remaining computations are executed using the default configuration of $(16, 16 - m)$.

Accurate estimation of the EDP values is critical in evaluating our work as we target at reducing the EDP. We rely on the various performance counters in Simics for this purpose, along with existing support from proper power models. To calculate the EDP value, we need to calculate the energy consumption and the application execution latency. The execution latency can be easily calculated using the number of cycles spent in execution (which can be obtained from performance counters) and the clock frequency used. The energy consumption we calculate include both leakage energy and dynamic energy components for CPUs, L1 caches, and shared L2 cache. We obtain the leakage power and dynamic power for the cache reads and writes from CACTI 5.0 tool [43] for both L1 and L2 caches. Together with the number of cache accesses (hits/misses) from performance counters in the full system simulator, we can estimate the energy consumption for caches. As for energy consumption for CPUs, we employ a power model similar to the one used in Wattch [10] and scale it appropriately to get dynamic and leakage power numbers separately.

Table 1. Our major simulation parameters and their default values.

Parameter	Value
Number of CPUs (n)	16
Number of Threads	16
Highest CPU Frequency	2GHz
Highest Voltage Level	1.1V
Number of Voltage/Frequency Levels	5
CPU Issue Width	1
L1 Data Cache	64K, 2-way, 2 banks
L1 Instruction Cache	64K, 2-way, 2 banks
Unified L2 Cache	4MB, 16-way, 2 banks
Process Technology	70nm

Figure 7 presents the EDP values for the FFT and MG benchmarks over different execution steps under the $(16, x)$ configurations. Each curve shows the normalized EDP values for one application step (a function or a function group) when executed with different number of CPUs. All the results are normalized with respect to the maximum EDP value observed. An important point to note from these results is that, the curves of EDP results over different steps are similar in FFT, but they differ more in the case of MG. This is because the functions in FFT behave similarly (one-dimensional FFT). In contrast, as mentioned earlier, we divide the functions in MG into different groups. As a result, the curves that belong to the steps in the same group are similar, but they exhibit a different trend from those falling into the different function groups.

This difference between the FFT and MG applications allows us to test the effectiveness of our helper thread based approach with two general types of applications. First, for applications that operate over similar tasks iteratively, a single model can be used to apply our curve fitting method and predict the next configuration to choose. Second, for applications with complex iterations (i.e., each iteration calls different functions and executes for long periods of time), we can divide the complex iteration into several phases and build different models for different phases. There exists several studies [17, 40] that can be used to identify application phases based on dynamic runtime characteristics. For simplicity, we partition a complex iteration in MG based on the static code structure in our study. Overall, the trends exhibited by the curves in Figures 7 indicate that curve fitting can be successful in predicting the next (most appropriate) configuration to go.

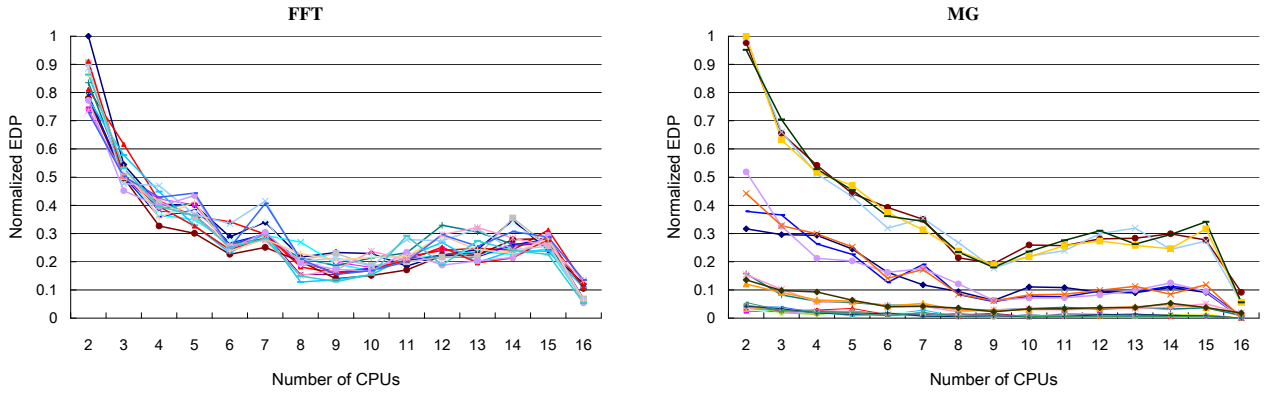

Figure 7. EDP values for different steps in the FFT and MG benchmarks. Each curve captures the normalized EDP values for one application step when executed with different number of CPUs. Steps in FFT exhibit similar behavior, whereas the steps in MG fall into several groups.

6 Experimental Results

In our experiments with FFT and MG, we start curve fitting with initial data points collected before the current execution. For example, in the FFT application, we can measure the EDP of the first function call under different number of CPUs and different number of threads. Note that, we do not need a complete set of data points to start curve fitting. As we will show in the next section, a few data points spread across the exploration space are sufficient in most cases. Once the CPU availability changes, we predict the next configuration based on the initial data points to minimize the EDP. After each function call, the EDP value for the function execution is calculated, which is subsequently used to add or update the reference data points for future curve fittings. The application continues to execute with the best predictable setting. This section discusses the results we obtain under several execution scenarios.

6.1 Results without re-threading

We first examine the results with one-dimensional curve fitting (i.e., to adapt to a variation in CPU availability, we change only the CPU count). Figure 8 shows the EDP values for different approaches, when using 16 threads throughout the program execution of FFT and MG applications. All the results are *normalized* with respect to the EDP value when no change is made to the number of CPUs (i.e., when all the available CPUs are used); this baseline result is captured by the first bar in each group of bars. In both of these plots, we present the results of curve fitting (denoted 1-D fitting) under the different number of initial data points to start exploration. We also show the minimum EDP value (the last bar in each group of bars) that can be achieved if the EDP of each step with any setting is known beforehand (so that we can select the best alternative). This minimum value is also the optimal result that we can possibly achieve using the one-dimensional curve fitting case. The specific selection of initial data points used in these experiments is as follows. When the number of initial data points is 3, 4, 5, 8 and 15, the corresponding sets of CPU counts with available EDP values are $\{2, 8, 16\}$, $\{2, 4, 8, 16\}$, $\{2, 4, 8, 12, 16\}$, $\{2, 4, 6, 8, 10, 12, 14, 16\}$, and $\{2...16\}$ respectively; i.e., we start execution with these EDP values ready as our initial data points.

As we can see from these plots, when the CPU availability becomes low (e.g., less than 8), there is not much room for improvement. In contrast, the EDP savings are significant when the CPU availability remains relatively high (e.g., larger than 10). We also see that the EDP benefits achieved using curve-fitting guided configurations are not as good as the optimal but much better than the baseline case. Note that, the results with our curve fitting based approach can sometimes be worse than the baseline case. The reason for this is that the EDP curves are in general not smooth. Figure 9 provides a snapshot of the one-dimensional curve fitting for EDP values in FFT. We observe that the curve fitting results closely follow the trend in actual EDP values. However, due to several factors (such as on-chip cache behavior), the EDP value at some point may not be predicted precisely by the curve fitting method (see the data point at 15 CPUs in Figure 9 for an example).

In summary, although the specific EDP savings vary with the number of available CPUs and the number of initial data points we have, our helper thread based method clearly brings benefits in most cases. For example, as shown in Figure 8, for the FFT application, the EDP savings are 7.4% on average and can go up to 21.7% in the best curve fitting case with 15 initial data points. Even in the worst curve fitting case with 8 initial data points, the savings we achieve are about 4.8% on average and can be as much as 17.1%.

6.2 Results with re-threading

Figure 10 provides a snapshot of the two-dimensional curve fitting for FFT. The circles in the curve fitting surface are the data points used in the triangle-based linear interpolation. Note that the curve fitting results are shifted by

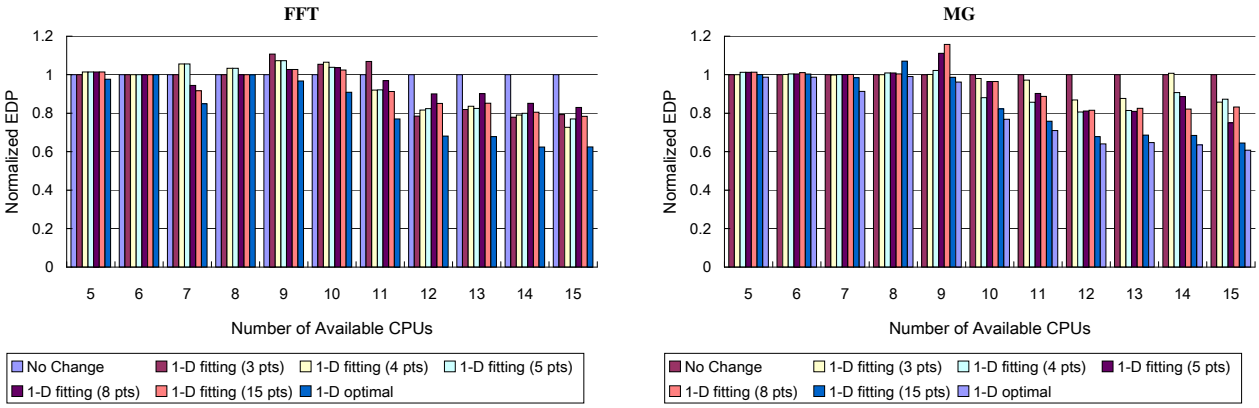

Figure 8. One-dimensional fitting results for FFT and MG. A bar corresponding to a value of p on the x-axis, gives the normalized EDP value when the number of CPUs drops from 16 to p. For our curve fitting approach (denoted 1-D fitting), the value within the parentheses indicate the number of initial data points.

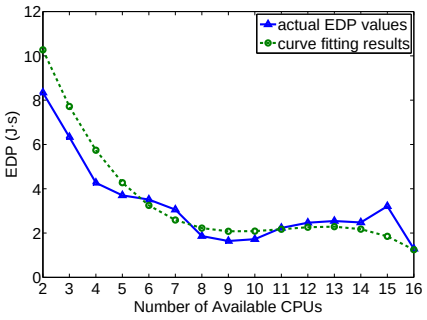

Figure 9. One-dimensional fitting snapshot.

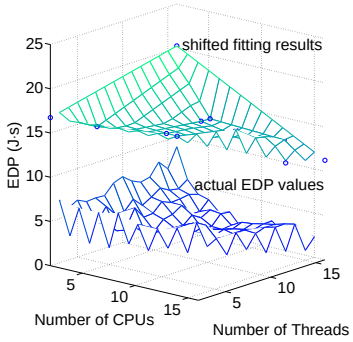

Figure 10. Two-dimensional fitting snapshot.

10 to make both surfaces visible and comparable. Taking this into account, we see that the curve fitting results are close to the actual EDP values. Even in this example with few data points, the fitting method still captures the trends in most parts of the surface. The EDP results for FFT and MG applications using two-dimensional curve fitting methods are shown in Figure 11. All the results are *normalized* with respect to the optimal value that can be obtained under the one-dimensional curve fitting method, namely, the minimum EDP value if the number of threads does not change. As expected, when we consider re-threading, the search space becomes much larger. The two-dimensional curve fitting in these figures starts with 20 data points. These points are chosen randomly from the 120 data points in the lower triangle portion of the space shown in Figure 6. Due to the randomness, the plots present the minimum (Min), maximum (Max) and average (Avg) EDP results among $1,000$ experiments with random data points in the set. For comparison, the minimum EDP value is also shown (marked as 2-D optimal in the figures), which represents the optimal result that can be achieved if the EDP value for any configuration at any step is known beforehand.

Our first observation from the results in Figure 11 is that changing the number of threads along with the number of CPUs can reduce the EDP much further. In fact, the minimum EDP values of two-dimensional fitting (2-D optimal) are 31.1% and 62.5% less for FFT and MG, respectively, than the optimal results if we can only change the number of CPUs. This is due to the much larger search space considered. For instance, when CPU availability reduces, the number of CPUs that can be chosen without re-threading is usually less than the current number of threads. This may sometimes exacerbate the load imbalance and waste computing resources. In such cases, changing the number of threads to be the same as the number of CPUs can possibly give better results. Another observation to note is that two-dimensional fitting also benefits from scenarios where the number of threads is greater than the number of CPUs. For example, our experimental results indicate that using 4 threads on 2 CPUs generates lower EDP than using 2 threads on 2 CPUs for some functions of both FFT and MG applications.

The difference between the minimum EDP (2-D optimal) and the best results with our two-dimensional curve fitting approach is minimal in most cases and on average within 5.9%. Across our experiments with 20 random initial

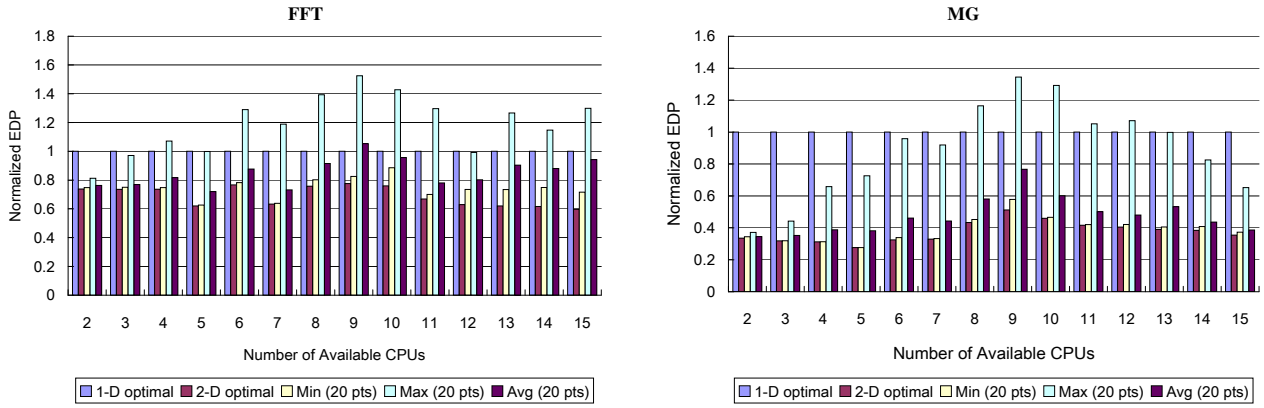

Figure 11. Two-dimensional fitting results for FFT and MG with 20 initial data points. All bars are normalized to the first bar in each group of bars. The initial data points are randomly selected and results shown here are based on $1,000$ experiments.

data points, the two-dimensional curve fitting on average reduces the EDP values to be much lower than the optimal of one-dimensional curve fitting (14.9% for FFT and 52.5% for MG). These savings translate into 26.3% and 59.7% less than the baseline case. Note that the difference between the best case and the worst case for two-dimensional curve fitting is significant. We found that the higher EDP value in some cases results from the poor distribution of initial data points which has few data points around the diagonal in Figure 6. The data points on the diagonal represent the scenarios where equal number of CPUs and threads are running, these points are often close to the optimal settings for most function calls. Therefore, in practice, a careful selection of initial data points can avoid the worse cases and achieve at least the average benefits.

6.3 Results with DVFS

In this set of experiments, we assumed five levels for dynamic voltage/frequency scaling, with the supply voltage (V_{dd}) ranging from $0.7V$ to $1.1V$ using steps of $0.1V$. Under these V_{dd} values, we obtain the scaling factors for frequency, dynamic power and leakage power from HSPICE simulations. We use these scaling factors in calculating the EDP values when the CPUs are running at different voltage/frequency levels. Note that the curve fitting in this case becomes three-dimensional as we control the number of CPUs, number of threads, and voltage levels. The experimental results we collected indicate that additional EDP savings are possible over the two-dimensional curve fitting case. Specifically, in comparison to the baseline case, the EDP in FFT can be reduced by 46.1% on average and by as much as 66.3% in some cases. Similarly in MG, the EDP saving is 70.5% on average and can go up to 83.3%. Detailed voltage scaling results are omitted due to space concerns.

6.4 Sensitivity Analysis

As technology scales, it is expected that more processor cores will become available in a CMP [9]. Therefore, it is important to study the effectiveness of our approach when the number of CPUs and the number of threads in the system are larger. We performed experiments with the one-dimensional curve fitting approach on a system with 24 CPUs. We observed that our approach is very effective in most cases, especially when the CPU availability is not too low. Another set of experiments investigated the EDP trends under 32 threads running on a CMP system of 16 CPUs. Recall from Figure 8, that little room is there to reduce EDP when the CPU availability is very low (e.g., less than half of the thread count). However, we observed that, with 32 threads, even if the CPU availability is low, it is still possible to achieve EDP improvements using our curve fitting based approach. This tells us that, as the number of threads an application uses increases, there is more room to make better tradeoffs between performance and power consumption.

The number of data points used in curve fitting is important in our approach. It affects the accuracy of the estimations and relates to the goodness of final results. We performed a sensitivity study that employs different number of initial data points. Figure 12 shows the results for the one-dimensional curve fitting case with the number of initial data points varying from 3 to 15, when the number of available CPUs drops from 16 to 14, and the number of threads is fixed at 16. Figure 13 shows the results for two-dimensional curve fitting with the number of initial data points ranging from 5 to 120, also assuming that the number of available CPUs drops from 16 to 14. One would expect better results when more initial data points are used to start curve fitting. The results with MG in the one-dimensional and two-dimensional cases indicate such a trend clearly. However, the curves become flat beyond

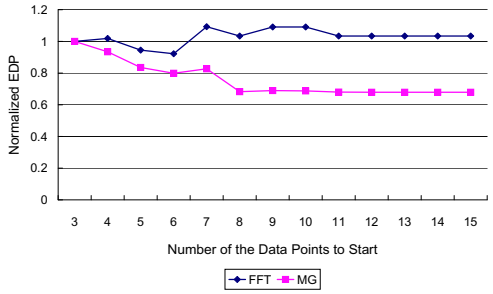

Figure 12. One-dimensional fitting with the different number of initial data points. We assume that the number of available CPUs drops from 16 to 14 and the thread count remains at 16.

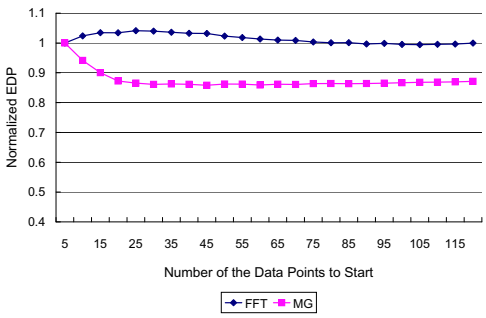

Figure 13. Two-dimensional fitting with the different number of initial data points. We assume that the number of available CPUs drops from 16 to 14.

a certain number of initial data points. This is because by then the accuracy has already become good enough. The important point is that, for both the one-dimensional curve fitting and two-dimensional curve fitting cases, the number of initial data points needed to reach close-to-best results is small compared to the whole search space. On the other hand, the sensitivity study with FFT shows less clear trends. We also observe that in some cases using a larger number of initial data points may even cut the EDP savings. There are two reasons for this behavior. First, as long as some data points close to the optimal settings are chosen, the overall results can be generally good. Second, there might be some unpredictable points in the curve fitting due to unpredicted changes in EDP caused by on-chip cache behavior or other runtime factors.

6.5 Limited Number of Configurations

So far in our study, we assume that a function can be executed with any arbitrary number of threads after re-threading. This may not always be the case in practice. We now assume that only a few multi-threaded versions exist for the functions in our applications. For example, suppose that only (pre-compiled) 2, 4, 8 and 16 threaded versions of a function (instead of all potential versions for all threads from 1 to 16) are available to choose from at runtime. In this case, if the number of the available CPUs drops from 16 to 14, the version with 8 threads is chosen by our scheme. As a result, the EDP saving we achieve without applying DVFS is 36.8% over the baseline case. This saving is not as much as the corresponding average EDP saving (42.8%) if any number of threads is allowed, but is still larger than the optimal EDP saving that can be achieved without re-threading (33.2%).

6.6 More General Variations

In all the experiments discussed above, we have assumed that the number of available CPUs drops from 16 to a lower number, and this CPU availability change occurs only once. As stated earlier, our helper thread based application adaptation scheme can adapt to the availability variations in the other direction as well, i.e., when the number of CPUs available to an application increases. In fact, as our scheme is a dynamic one, the number of availability changes that may occur has no effect on its applicability. That is, the number of CPUs available to an application may change in any direction in any number of times during the course of execution. We now use an example scenario to illustrate the benefits in a more general case of CPU availability variation. We assume that the FFT application starts execution with 16 CPUs, and the number of available CPUs changes in the sequence of 10, 13, 9, 14, 11, 15 at boundaries of the six main iterations of the application. Using our helper thread based adaptation scheme, the EDP can be reduced by about 22% without re-threading and about 30% with re-threading. In addition, the EDP savings can jump to 48.4% when voltage/frequency scaling is also employed. For this availability variation pattern, Figure 14 gives the different configurations selected at runtime by three different schemes when the thread count is fixed. The first one is to use all the CPUs available; the second one is our one-dimensional curve fitting with 4 initial data points (at CPU counts of $2, 4, 8, 16$); and the last one is the optimal one that can be achieved if the EDP values for all potential configurations are known a priori. It can easily be observed that the configurations selected by the one-dimensional curve fitting strategy are close to the optimal selections in most cases. Exceptions may occur when certain unpredictable data points are witnessed as explained in Section 6.1. Compared to the optimal adaptation, changes among different configurations are also less when using curve fitting due to the nature of curve fitting method used.

Figure 15 gives the results for two-dimensional curve fitting with 20 initial data points. At each step of the FFT application, the configuration ([Number of Threads, Number of CPUs]) chosen by our two-dimensional fitting scheme and the optimal configuration are shown as the points on the corresponding curves. Again, configuration changes

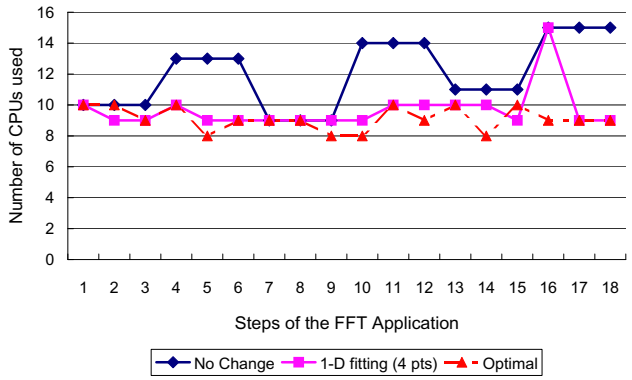

Figure 14. One-dimensional fitting for a general CPU availability variation pattern.

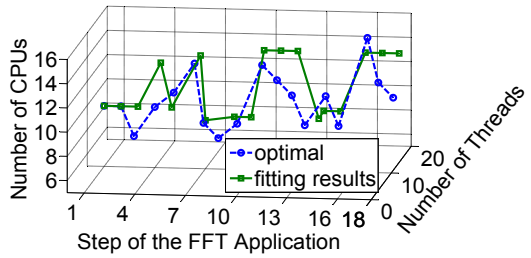

Figure 15. Two-dimensional fitting results for a general CPU availability variation pattern.

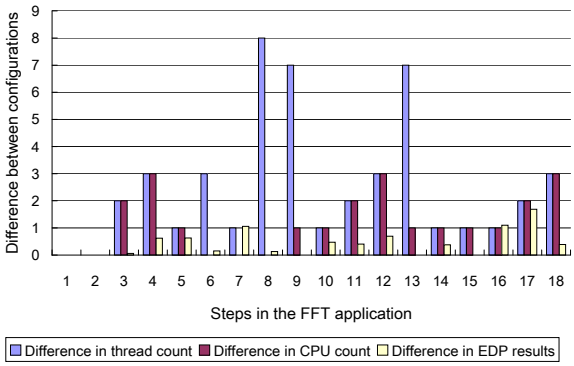

Figure 16. Difference in the configurations chosen by our two-dimensional fitting scheme and the optimal scheme. Each group of bars indicate the difference in thread count, CPU count, and EDP value achieved. The EDP value differences are normalized to the optimal values, whereas the differences in thread/CPU count are in absolute terms.

happen less frequently when use curve fitting in contrast to the optimal configuration selections. Figure 16 presents the difference between the configurations chosen by our two-dimensional fitting scheme and the optimal scheme. We observe that the difference in CPU count is usually low, but the difference in thread count can be large between the two scenarios. However, even in the cases where the thread count chosen by our approach is largely different from the optimal configuration, the resulting EDP values are very similar in most cases. The EDP differences in some cases are relatively large due to the continuity of fitting and the lack of data points around the unpredictable optimal configurations. For example, as shown in Figure 15, in moving from step 16 to step 18, the configurations chosen by two-dimensional fitting remain at $(14, 14)$, while the optimal configurations in this case would be $(15, 15)$, $(12, 12)$ and $(11, 11)$.

6.7 Quantifying of the Helper Thread Overheads

Although our helper thread based application adaptation scheme brings significant improvements in EDP as demonstrated by our experimental evaluation presented so far, it is also important to capture and quantify its overheads. Recall that our helper thread collects performance counter statistics, calculates EDP, implements different curve fitting schemes, and changes the number of CPUs, number of threads and voltage/frequency levels at runtime. We quantified the performance and power overheads of our helper thread and found that its energy overhead (even in the worst case scenario) is about 1% of the total application energy consumption. Since its performance overhead is almost completely hidden in parallel execution, we conclude that the contribution of helper thread overheads to the overall EDP is negligible in practice (note that thread count changes take place in function boundaries in our implementation and this limits the potential overheads significantly). We also observed during our analysis of the helper thread behavior that most of the overheads it brings are due to implementing the curve fitting strategy. In addition, one can expect the overhead contribution of helper threads to be even lower in the future CMPs where the number of CPUs and the number of threads will be much higher than the current values we assume in our experiments. To summarize, we can conclude that our helper thread based adaptation approach brings significant EDP benefits even if all the overheads brought by it are accounted for.

7 Conclusions and Future Work

The main contribution of this paper is the discussion of an adaptive approach targeting varying CPU availability in a CMP based environment. The proposed approach makes use of the helper thread concept and reacts to CPU

availability change by selecting the number of CPUs, number of threads, and voltage/frequency levels to use through curve fitting. We implemented three different variants of our helper thread based approach and performed experiments using two application codes. The experimental results we collected clearly show the success of the proposed approach. For example, when we adapt application execution by changing all control parameters (i.e., thread count, CPU count, voltage/frequency level), we achieved on average 46.1% and 70.5% EDP savings in applications FFT and MG, respectively. We also conducted a set of sensitivity experiments where we changed the default values of some of our simulation parameters, and observed that our EDP improvements are consistent across a wide range of experiments. As our future work, we plan to explore other potential uses of helper threads in the context of CMP architectures. Work is also underway in retargeting our approach to other optimization metrics such as chip temperature and reliability. Also, we are in the process of implementing curve fitting based models that account for program phases.

Acknowledgements

This work is supported in part by NSF grants CCF 0444345, CNS 0720645, CCF 0702519, and a grant from Microsoft Corporation. The authors also acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. We thank Xiaoxia Wu for the help on the HSPICE simulation and Suzanne Shontz for the helpful discussion on curve fitting methods.

References

[1] AMD Quad-Core Opteron. `http://www.amd.com/quadcore`.

[2] Intel Core Duo Processors. `http://www.intel.com/products/processor/coreduo/`.

[3] Microsoft Phoenix Academic Program, `http://research.microsoft.com/phoenix/`.

[4] NAS NPB benchmarks 3.2. `http://www.nas.nasa.gov/Software/NPB/`.

[5] UltraSPARC T1 processor. `http://www.sun.com/processors/UltraSPARC-T1/`.

[6] Virtutech Simics 3.0. `http://www.virtutech.com/`.

[7] M. Banatre and P. A. Lee. *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives (Lecture Notes in Computer Science)*. Springer-Verlag, 1994.

[8] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[9] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual conference on Design Automation*, 2007.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture*, 2000.

[11] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[12] M. Chu, R. Ravindran, and S. Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the 40th Annual IEEE/ACM Symposium on Microarchitecture*, 2007.

[13] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Computer Architecture News*, 33(4), 2005.

[14] N. Copty. OpenMP support in Sun Studio compilers and tools, `http://developers.sun.com/solaris/articles/studio_openmp.html`.

[15] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual International Conference on Supercomputing*, 2006.

[16] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1), 1998.

[17] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[18] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. Adapting application execution to reduced CPU availability. In *Proceedings of the 11th workshop on Interaction between Compilers and Computer Architectures*, 2007.

[19] X. Feng, R. Ge, and K. W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[20] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the International Symposium on Computer Architecture*, Anchorage Alaska, May 2002.

[21] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[22] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. In *Proceedings of the International Symposium on Low Power Electron-*

ics, October 1995.

[23] M. Gschwind. Chip multiprocessing and the Cell broadband engine. In *Proceedings of the ACM Computing Frontiers 2006*, 2006.

[24] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM Symposium on Microarchitecture*, 2007.

[25] M. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *SUIF Compiler Workshop*, Stanford University, CA, Aug. 1997.

[26] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2003.

[27] H. Jiang and V. Chaudhary. Compile/run-time support for thread migration. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[28] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report *NAS-99-011*, 1999.

[29] C. Jung, D. Lim, J. Lee, and Y. Solihin. Helper thread prefetching for loosely-coupled multiprocessor systems. 2006.

[30] J. Kim, D. Park, C. Nicopoulos, N. Vijaykrishnan, and C. R. Das. Design and analysis of an NoC architecture from performance, reliability and energy perspective. In *Proceedings of the Symposium on Architecture for Networking and Communications Systems*, 2005.

[31] M. Klemm, M. Bezold, S. Gabriel, R. Veldema, and M. Philippsen. Reparallelization and migration of OpenMP programs. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[32] J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2006.

[33] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2006.

[34] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the Sun UltraSPARC CMP processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[35] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2004.

[36] S. H. K. Narayanan, M. Kandemir, and O. Ozturk. Compiler-directed power density reduction in NoC-based multi-core designs. In *Proceedings of International Symposium on Quality Electronic Design*, 2006.

[37] O. Ozturk, M. T. Kandemir, and S. Tosun. An ILP based approach to address code generation for digital signal processors. In *Proceedings of ACM Great Lakes Symposium on VLSI*, 2006.

[38] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. In *Proceedings of the EuroSys*, 2007.

[39] K. Shaw and W. Dally. Migration in single chip multiprocessors. *Computer Architecture Letters, IEEE*, (1), 2002.

[40] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the International Symposium on Computer Architecture*, 2003.

[41] D. Shin and J. Kim. Power-aware communication optimization for networks-on-chips with voltage scalable links. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[42] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5), 2005.

[43] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi. CACTI 5.0. Hewlett-Packard technical report HPL-2007-167, 2007. `http://quid.hpl.hp.com:9082/cacti/`.

[44] D. F. Watson. *Contouring: A guide to the analysis and display of spacial data*. Pergamon, 1994.

[45] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the International Symposium on Microarchitecture*, 2005.

[46] S.-H. Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture*, February 2002.