

Adapting Application Execution to Reduced CPU Availability

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

As CMPs become increasingly popular, issues such as performance, CPU availability and energy efficiency require careful considerations. An important question in this context is how to adapt the execution of a given application to reduced CPU availability at runtime. This paper studies this problem, targeting the energy-delay product as the main metric to optimize. Specifically, we want to adapt the application execution using several techniques to reduced CPU availability with the goal of minimizing the energy-delay product. In doing so, our approach considers four factors: (1) specific application level behavior; (2) thread migration; (3) dynamic code modification or static code versioning; and (4) voltage/frequency scaling. We implemented our approach using a simulation based environment and conducted experiments with two scientific applications: Fast Fourier Transform (FFT) and Multi-Grid (MG). Our results show that, in order to adapt the application execution to reduced CPU availability, considering all four factors mentioned above is critical. We also performed a sensitivity analysis by changing the values of some simulation parameters, and observed significant reductions in the energy-delay product.

1. INTRODUCTION

Chip multiprocessors (CMPs) are increasingly popular as performance improvements based on increasing clock frequency alone are approaching their limits. Other factors, such as ease of verification/validation and ability of exploiting both thread level (coarse grain) and instruction level (fine grain) parallelism, also boost trends towards chip multiprocessing. [16, 27, 17, 18, 26, 6]

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 for large scientific codes), and optimization metrics other than performance are becoming more and more widespread. Two of these metrics are availability and energy. In many execution scenarios where CMPs are involved, satisfying both these metrics can be critical. One of the interesting problems in this context is to adapt application execution to varying hardware resources in an energy efficient manner.

Motivated by this observation, this paper studies the problem of how an execution can cope with reduced CPU availability. Our goal is to decide the best approach to employ when the number of processors is reduced, considering the energy-delay product. In other words, we want to adapt the execution to reduced CPU availability with the goal of minimizing the energy-delay product. Our approach considers four factors: (1) specific application level behavior; (2) thread migration; (3) dynamic code modification or static code versioning; and (4) voltage/frequency scaling. We want to emphasize that energy-delay product [9] is an important metric as

it captures our desire of both achieving high performance and reducing energy consumption, both of which are critical in scientific computing.

We implemented our approach using a simulation based environment and conducted experiments with two scientific applications: MultiGrid (MG) and Fast Fourier Transform (FFT). Our results show that in order to adapt the application execution to reduced CPU availability, considering all four factors mentioned above is very important. Our results indicate, for example, that, in the FFT application, thread migration provides 20.1% reduction in the energy-delay product, code versioning brings an additional 41.1% improvement on top of this, and voltage/frequency scaling increases the savings to 44.6%. Overall, our results indicate that, in order to minimize the energy-delay product, we need to select the number of CPUs, number of threads, and voltage/frequency levels very carefully. Note that, while we present our analysis using only two applications, these applications represent an important class of scientific applications and our conclusions may be extended to other scientific applications as well.

The rest of this paper is organized as follows. The next section discusses the related work. Section 3 introduces the applications. Section 4 describes our problem in more detail and discusses different approaches to it. Experimental setup and the results from our experimental evaluation are presented in Section 5. Section 6 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 [15, 24, 5] as well as software [20, 22, 27, 8]. In comparison, our work targets the execution of single scientific application on a CMP and looks at the problem from the energy-delay product perspective.

Dynamic Voltage-Frequency Scaling (DVFS) has been used in recent years to reduce power consumption of CMPs. Previous proposals mostly focused on multiprogrammed workloads [7, 26, 21]. Li and Martinez [16] discuss the viability of changing the number of concurrent processors/threads at run-time to accommodate changes in the execution environment. Their work proposes heuristics to prune the search space for determining the optimum number of CPUs to employ and the DVFS levels to use. Some of the major differences between this work and our paper can be summarized as follows. First, 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. Second, they approximate the leakage as a fraction of the dynamic power consumption instead of using an accurate leakage model. In addition, their main target metric is power consumption, whereas we consider energy-delay product, which we believe is more proper for scientific computing environments. Third, they assume a known target execution time and maximum allowable power dissipation. Such information may not be readily available in general. Wu et al [27] study the effectiveness of dynamic compiler driven voltage scaling. In comparison, our work considers multiple techniques in adapting application execution. Li et al [18] 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.

There have been several publications on deciding the number of CPUs to employ in different program phases. Earlier work [10] in this area points out that compiler-parallelized applications may waste computational resources in different program phases. To remedy this, they propose a mechanism to dynamically adjust the number of processors 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 [10] focuses mainly on performance. Second, in their work, the number of CPUs is adjusted to react program behavior, while our approach aims to adapt to variations in CPU availability.

Thread migration has been studied in the past, and the representative studies include [6, 12, 19, 11]. The main difference between most of these prior efforts and ours is that we focus on adapting application behavior to reduced CPU availability and target the energy-delay product.

Our work dealing with reduced CPU availability is also different from standard fault tolerance research. Our focus is not on techniques for failure recovery or load balancing. Rather, we try to minimize energy-delay product.

3. OUR APPLICATIONS

In this section, we briefly describe the two applications used in this paper: Fast Fourier Transform (FFT) and MultiGrid (MG). Both are from the NAS Parallel Benchmark Suite [1]. We used their OpenMP implementations with the class W inputs. 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 [13].

3.1 Fast Fourier Transform (FFT)

Fourier Transforms are important in many domains. 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. The terms of the summation in the DFT can be divided into two parts. The first part can be formed using the even numbered terms and the second part can be formed using the odd numbered terms. This partitioning significantly reduces the computational cost of the DFT from $O(N^2)$ to $O(N \log_2 N)$, hence the term Fast Fourier Transform (FFT).

This algorithm is essentially recursive and the data accesses have the property of spatial locality. At each step of the recursion, the terms in each summation are split into halves. Thus, the underlying algorithm naturally works best on list sizes that are powers of two. Likewise, its parallelization most benefits from mapping the computation to p processors, where p is also a power of 2.

The FFT implementation we use has six major iterations, which . Each iteration performs a similar computation, and altogether they take 90% of the serial execution time.

3.2 MultiGrid Method (MG)

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 Ω with a hierarchy of discretization from fine grain to coarse grain, i.e. $\Omega_0, \Omega_1, \dots, \Omega_m$. The method starts with restricting the initial guess of the solution on the finest grid Ω_0 to the coarsest grid Ω_m to obtain an approximate solution associated with Ω_m . Then, the solution for Ω_m is interpolated to a finer grid Ω_{m-1} , and refined through a few iterations of a stationary method. This process is repeated until the solution for Ω_0 is obtained.

The computation is performed on a stack of different arrays. A typical MG implementation doubles the dimension of the grid for each finer level, increasing the cost of refinement at each level accordingly. This recursive doubling once again favors computing in parallel with p processors, when p is a power of 2. The MG implementation has four major iterations, which consume 70% of the total serial execution time.

4. CMP ARCHITECTURE, CPU UNAVAILABILITY & DIFFERENT APPROACHES

Our goal in this paper is to study how varying the number of CPUs, number of threads, and voltage/frequency levels affects the energy-delay product if we want to tune application execution based on reduced CPU availability. It is not an objective of this paper to determine the optimal number of threads, CPUs or voltage/frequency levels to use automatically; rather, this paper shows how these parameters affect the energy-delay product of a parallel scientific application. Of course, the observations made in this paper can also be used for developing a fully automated strategy, which is in our future research agenda.

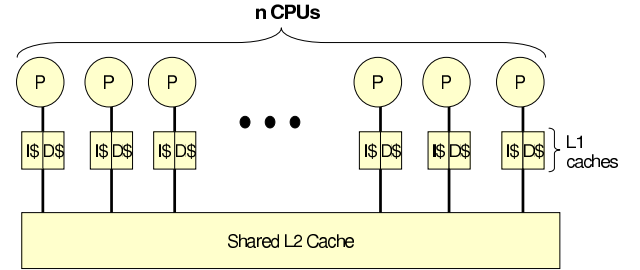


Figure 1: CMP architecture considered in our work. L1 instruction and data caches are private, whereas the unified L2 cache is shared.

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. 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, turning off a CPU and its L1 caches can reduce the leakage power consumption dramatically. Several architectural techniques such as [28, 7] can be used for turning off L1 caches. We further assume that each CPU can be voltage/frequency scaled, independently from the other CPUs. Scaling down the frequency of a CPU increases the length of its clock period, (which in turn increases execution latency), and scaling down its voltage reduces its energy consumption. Therefore, interesting energy-performance tradeoffs can be studied by playing with the voltages/frequencies of individual CPUs.

cur. Therefore, we adopt the following strategy: when the unavailability occurs, we select a suitable number of CPUs and continue executing the application with $n = 16$ threads on that number of CPUs until a suitable point in the program code is reached where we switch to the version with a different number of threads (so that the energy-delay product is further minimized). While such a point in the program code may not be easily identified in general programs, for the class of scientific applications of interest, this is possible. It is because as mentioned earlier these applications are of iterative nature, i.e., there is an outermost loop that iterates over the entire program code in a fixed number of times or until a condition is satisfied. So, if the CPU unavailability occurs at iteration i , we can continue to use $n = 16$ threads on q CPUs (where q is the number of CPUs we select considering the energy-delay product) until the end of iteration i is reached. When this happens, we start to use r threads ($r < 16$) and execute them on r CPUs (i.e., the iterations $i + 1$ and higher are executed with r threads and r CPUs, again r is selected based on the energy-delay product). This approach is illustrated in Figure 4.¹ We use the notation of (a, b) to represent an execution of the application using a threads running on b CPUs.

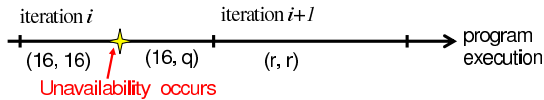


Figure 4: Illustration of how we change the number of CPUs and threads to cope with reduced CPU availability.

- The results may be even better if voltage/frequency scaling is employed. Suppose that, based on the analysis outlined above, we decided to use r CPUs and r threads, and the total execution latency with this combination is L cycles. It may be possible to reduce the energy-delay product further by using s CPUs and s threads where $s > r$ and still achieve the same execution latency (L) by scaling down the voltages and frequencies of these s CPUs carefully.

Clearly, one can expect the best (minimum) energy-delay product when the number of threads, number of CPUs and the voltage/frequency levels to use are selected carefully, considering the interactions among these decisions. In the rest of the paper, after presenting our experimental setup, we discuss experimental data collected using our simulation based platform and two benchmarks (FFT and MG). Our goal is to check whether the approaches described above behave the way we expect them to do.

5. EXPERIMENTAL SETUP AND RESULTS

5.1 Setup

We used the Simics toolset [2] 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. Table 1 gives the major simulation parameters used in this study with their default values. Later in our experiments we modify the default values of some parameters to conduct a sensitivity study. The leakage and dynamic power numbers for caches are collected using the Cacti tool [25]. The default CPU energy figures are obtained using Wattch [4]. As our default, we assume that in both FFT and MG, the CPU unavailability takes place after the first iteration (recall that FFT has six

¹As will be discussed later in the paper, our experiments show that, with a given CPU count r , the best latency values are obtained with an r -thread version of the application. See Figure 13 as an example.

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

Parameter	Value
Number of CPUs (n)	16
Number of Threads	16
Number of Unavailable CPUs (m)	2
Highest CPU Frequency	2GHz
Highest Voltage Level	1.5V
Number of Voltage/Frequency Levels	4
CPU Issue Width	2
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

iterations and MG has four iterations). $(16, 16)$ corresponds to our default configuration, before the CPU unavailability occurs. If no re-threading is used, after the unavailability, we use the configuration $(16, x)$. Otherwise, other configurations can also be used.

5.2 Results with 16 Threads

We start by presenting the execution latency for the two benchmarks under the $(16, x)$ configurations. The results for the FFT and MG benchmarks are given in Figure 5 and 6 respectively. An important point to observe from these results is that, the cycle results do not change too much between 9 and 15 CPUs. However, the difference between 15 and 16 CPUs is dramatic for both benchmarks. These results can be explained by considering the structure of these applications as well as the thread mapping strategy employed. As mentioned earlier, due to the thread mapping shown in Figures 2, the cycle counts between 8 and 15 are not expected to be significantly different. This is more or less what we observe from the results in Figures 5 and 6, except that in both benchmarks, some additional reductions in cycle count are achieved when the number of CPUs is increased beyond 8, as a result of increased L1 cache capacity with higher CPU counts (Recall Figure 3 which shows our expectations when cache effects are accounted for).

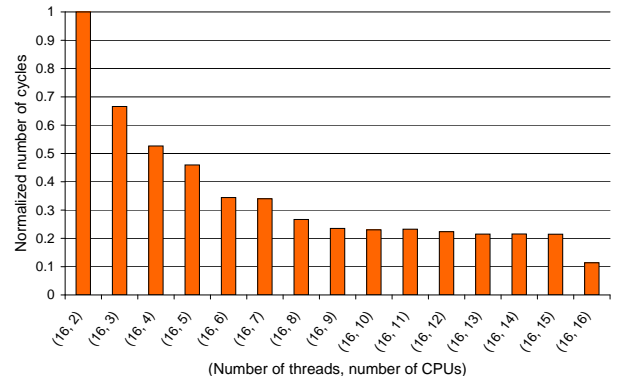


Figure 5: Number of execution cycles for the FFT benchmark. All bars are normalized with respect to the first one.

Figures 7 and 8 present the energy consumption results for FFT and MG, respectively. These results exhibit a completely different picture than the performance results presented in Figures 5 and 6. Each bar in these figures is divided into three components: CPU energy, cache leakage energy (including both L1 and L2) and cache dynamic energy (including both L1 and L2). We see a significant increase in the energy consumption in FFT when we go beyond 9

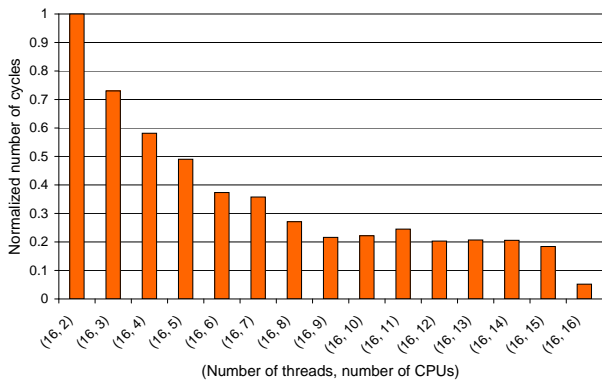


Figure 6: Number of execution cycles for the MG benchmark. All bars are normalized with respect to the first one.

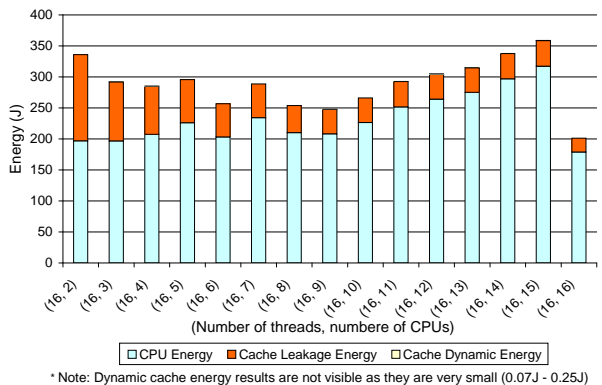
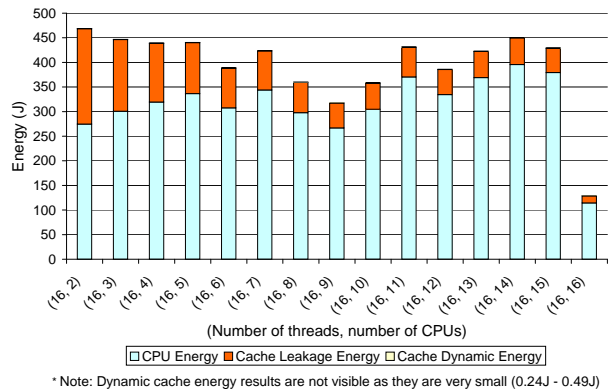


Figure 7: Energy consumption results for the FFT benchmark.

CPUs, until we reach 16 CPUs where energy consumption drops sharply, as a result of the reduction in leakage due to the reduction in execution cycles when 16 CPUs are used.² In Figures 7 and 8, as far as caches are concerned, leakage energy dominates dynamic energy very badly, as it constitutes a much higher percentage of energy consumption in the process technology we work with (70nm).

The energy-delay product results with FFT and MG are given in Figures 9 and 10. All the results are normalized with respect to the first bar. Let us now consider the FFT results in more detail, assuming that two of the original sixteen CPUs are to be taken away from this application after its first iteration (i.e., $n = 16$ and $m = 2$). The question, at this point, is *what is the optimum number of CPUs to use to execute the remaining iterations if we are to minimize the energy-delay product?* In this case that this number is nine. 9 CPUs are preferred over 14 CPUs ($n - m$; maximum possible), because using 14 CPUs requires five more CPUs and their associated L1 caches to be active, consuming significant additional leakage, which in turn increases the energy-delay product (recall that leakage energy dominates dynamic energy in the process technology we work with). On the other hand, using 9 CPUs is also a better choice than using 8 CPUs based on the energy-delay product results, mainly because of the increased on-chip L1 capacity. Therefore, for the FFT application, when two CPUs are taken away, the best strategy is to execute the remaining iterations using

²Note that we assumed two CPUs become unavailable during execution; consequently, continuing the execution with 15 or 16 CPUs is not possible. The reason that we give the results with these CPU counts as well is for comparison purposes only.



* Note: Dynamic cache energy results are not visible as they are very small (0.24J - 0.49J)

Figure 8: Energy consumption results for the MG benchmark.

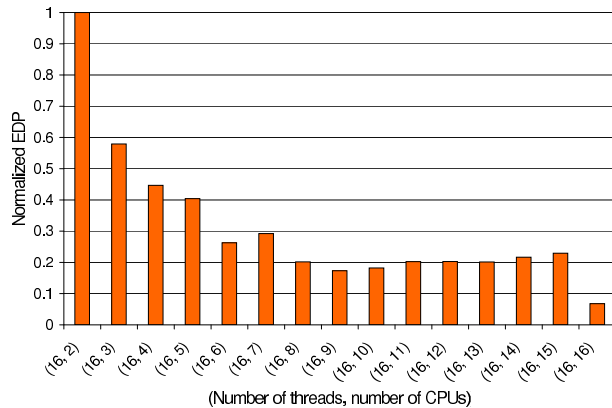


Figure 9: EDP values for the FFT benchmark.

nine CPUs (i.e., (16, 9) configuration), to minimize the value of the energy-delay product.

Considering Figure 9, let us now consider another possible scenario where the original number of CPUs is 8 (i.e., $n = 8$) and one of these CPUs becomes unavailable after the first iteration of the benchmark (i.e., $m = 1$). In this case, we see that the best option is to use 6 CPUs instead of 4 or 7, based on a similar argument made above for the case where $n = 16$ and $m = 2$. Similar observations can be made for the MG application as well, based on Figure 10.

5.3 Results with Varying Number of Threads

Our discussion so far has focused on selecting the right number of CPUs to execute the remaining part of an application when some CPUs become unavailable. However, we assumed that always 16 threads will be used. Consequently, in the FFT example discussed above, the best combination was to execute 16 threads on 9 CPUs.

Let us assume now, in addition to being able to change the number of CPUs on which the application runs, we are also able to change the number of its threads. As mentioned earlier, this can be achieved by either dynamic code modification (re-threading) or static versioning, which is pre-compiling different versions with different thread counts at compile time and selecting the appropriate one at runtime. In this work, we do not study the potential overheads associated with these schemes (and in fact dynamic code modification can be costly in practice); rather, we study the potential benefits from the energy-delay perspective of using the different number of threads.

Figures 11 and 12 give the energy-delay products for FFT and MG, respectively, when the number of threads is equal to the num-

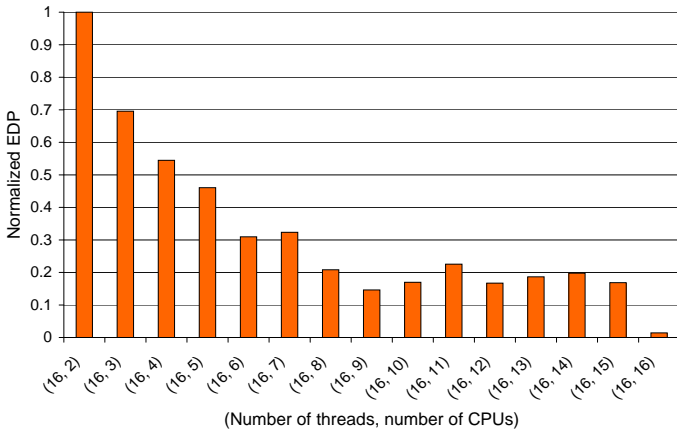


Figure 10: EDP values for the MG benchmark.

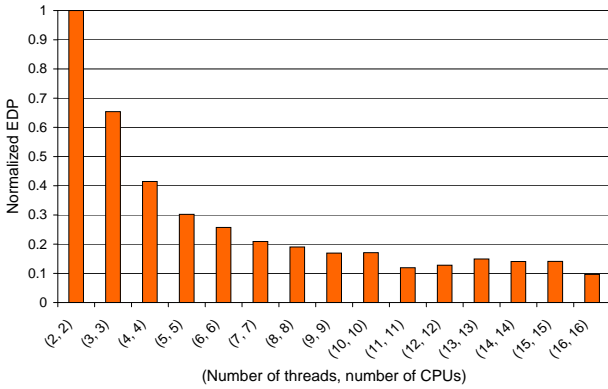


Figure 11: EDP values for the FFT benchmark.

ber of CPUs. We see from these results that (11, 11) generates the best value for the FFT benchmark. In other words, if we have the option of changing the number of threads as well, using 11 threads on 11 CPUs is a better option than using 16 threads on 9 CPUs. To explain why this occurs, let us now look at Figure 13 which shows, for this benchmark, the cycle counts (execution latency) with different thread/CPU combinations. The first of these curves reproduces the results from Figure 5 and the other one shows the results when the number of CPUs is the same as the number of threads. We see that this latter curve continuously exhibits better behavior than the former one when the number of CPUs is fixed at any value, which partly explains the trends witnessed in Figure 11. For the results with the MG benchmark (Figure 12), the minimum energy-delay product is achieved (considering all available 14 CPUs) with the combination (14, 14). Therefore, in this case, if we have the flexibility of re-threading the applications code, we would prefer to use 14 threads and execute them on 14 CPUs.

5.4 Voltage Scaling Results

We now consider CPU voltage/frequency scaling, and study how much additional savings it can bring in the energy-delay product. Let us first focus on the FFT application. Recall that this application generated the best energy-delay product so far under the configuration (11, 11). We performed another set of experiments where we ran x threads on x CPUs where $11 < x \leq 14$, such that voltage/frequency scaling is used (on these (x, x) configurations) to make sure that the resulting execution latency is similar to the latency of the (11, 11). In other words, our goal is to explore

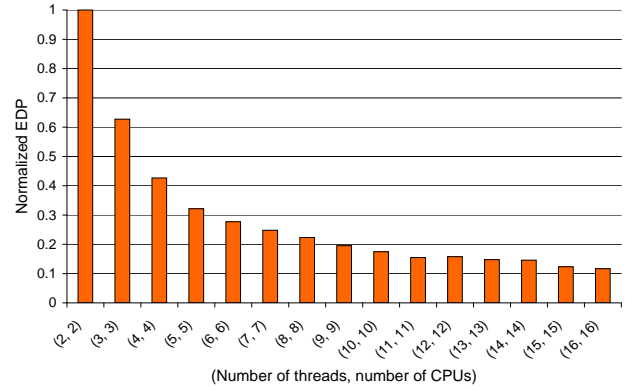


Figure 12: EDP values for the MG benchmark.

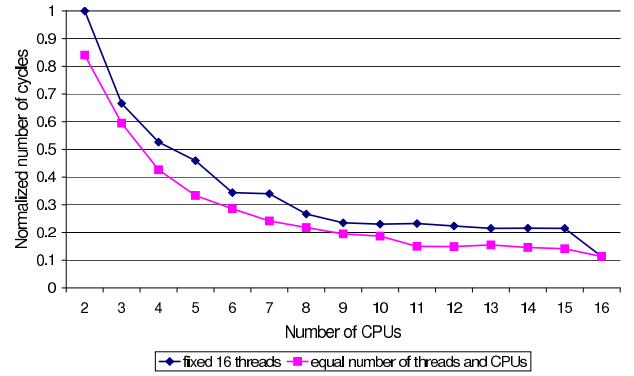


Figure 13: Normalized number of cycles for the FFT benchmark. The results are normalized with respect to that of configuration (16, 2).

the possibility of reducing the energy-delay product even further by using voltage/frequency scaling while not increasing the execution cycles beyond that of (11, 11), the best configuration so far. Note that while reducing the voltage helps cut both dynamic and leakage energy, increasing number of CPUs also increases leakage consumption. Therefore, it is not clear whether using voltage scaling along with more CPUs will reduce the energy-delay product further. The results with the FFT benchmark are given in Figure 14. Assuming that the difference between the original number of cycles of (11, 11) and (14, 14) allows us to reduce the voltage from 1.5V to 1.3V, we see that the energy-delay product can be reduced by an additional 3.5%. When we consider the MG benchmark, however, we see that voltage scaling does not bring any additional benefits in terms of energy-delay product. This is due to the fact that the (x, x) configuration generates the minimum execution cycles for this benchmark when $x = 14$ (among available CPU counts). That is, there is no room for voltage scaling in this benchmark.

5.5 Sensitivity Analysis

In this section, we vary the default values of some of our simulation parameters, and study their impacts on energy-delay product. Recall that the CPU power used in our experiment so far was 11.3W. We see from the results in Figures 7 and 8 that the CPU energy constitutes a large fraction of overall energy consumption. Therefore, it is important to study the impact of different CPU power values on our results. Figure 15 shows the energy-delay products, for the FFT benchmark, with 5W and 15W CPU power values for the (x, x) configurations. It also reproduces our previous

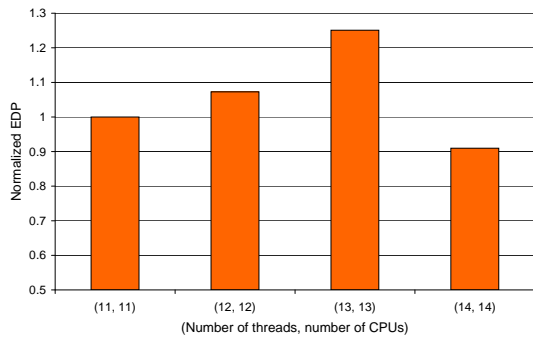


Figure 14: EDP for FFT after voltage scaling.

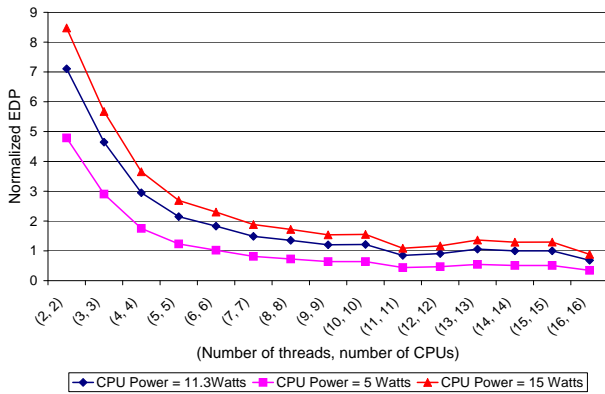


Figure 15: EDP values for the FFT benchmark with the different CPU power numbers.

results. In this graph, all values are normalized with respect to the EDP of configuration (14, 14) with default CPU power. While the actual values may differ (based on the CPU power assumed), the trends exhibited by these curves are similar to each other.

In our second sensitivity experiment, we study the influence of leakage energy on our results. Recall from Figures 7 and 8 that cache leakage energy dominates cache dynamic energy by a large margin. However, there are several circuit level and architectural level techniques [3, 23, 29, 7, 14] that can be employed for reducing cache leakage energy. To see how much such techniques can help, we performed two new sets of experiments. In the first of these, the default cache leakage values are reduced by 50%, and in the second one by 90%. The energy-delay products for the FFT benchmark are given in Figure 16. The normalization is the same as in Figure 15. Again, we see from these results that, while the actual values may change depending on the leakage contribution assumed, the trends do not change in any significant way, and as before, (11, 11) generates the best energy-delay product.

While we do not present detailed results here, we also performed sensitivity experiments with the (16, x) configurations. The results are similar to those discussed above for the (x , x) configurations. In particular, in this case, (16, 9) generated the minimum energy-delay product values for the FFT benchmark. Overall, our sensitivity experiments show that our conclusions do not change too much with varying leakage and CPU power values.

5.6 Summary of Results

Figure 17 summarizes the behavior of the different approaches discussed in this paper in the context of the FFT application. Recall that our main scenario was that two of the sixteen CPUs be-

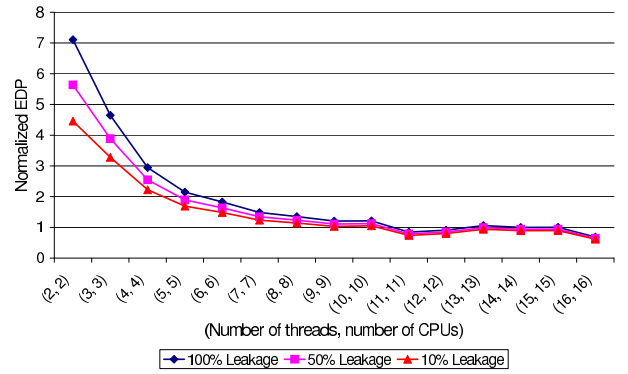


Figure 16: EDP values for the FFT benchmark with the different leakage power contributions.

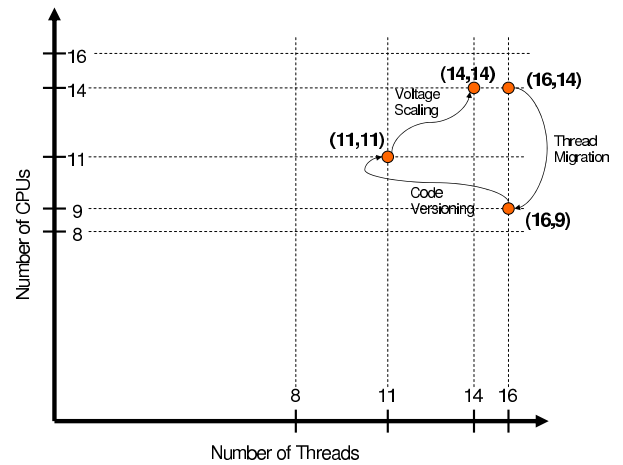


Figure 17: Summary of the results for the FFT benchmark.

come unavailable at some point during the course of execution. Therefore, the simplest move would be using (16, 14), i.e., using all of the available CPUs. However, as noted earlier, we may want to work with fewer CPUs if our goal is to minimize the energy-delay product. In fact, our experiments revealed that (16, 9) is the best configuration for this benchmark as long as we stick to 16 threads and do not consider re-threading. After that, we studied static versioning which reduced the value of the energy-delay product further and required configuration (11, 11) for achieving this. Finally, when voltage/frequency scaling is considered, configuration (14, 14) generated the minimum energy-delay product. Clearly, a different application would follow a different trajectory in this two-dimensional (thread, CPU) space, and the example trajectory shown in Figure 17 represents only a particular application. One of our future research directions will be further exploring this space using different target metrics and different applications.

6. CONCLUSIONS AND FUTURE WORK

Current trends indicate that CMPs will be used as building blocks for constructing future parallel machines. It is critical to study power, performance and availability characteristics of these systems. This paper studies the execution adaptation problem when a subset of the CPUs in the system is taken away from an application during its execution. Our results clearly show that, in adapting application execution to this reduced CPU availability, one needs to consider the number of CPUs to use, the number of threads to ac-

commodate and the voltage/frequency levels to employ (if the CMP has this capability) carefully. Targeting the energy-delay product (which we believe is becoming increasingly important in scientific computing), we show that all these factors may play an important role in adapting application execution to variances in CPU availability. Our future work includes developing a fully automated approach that determines the best combination of the number of threads, number of CPUs, and voltage/frequency levels to use automatically. Work is also underway in experimenting with other types of scientific applications.

7. ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for useful comments. This work is supported in part by NSF Grant No. 0093082, No. 0444345, and a grant from GSRC.

8. REFERENCES

- [1] NAS NPB benchmarks 3.2. <http://www.nas.nasa.gov/Software/NPB/>.
- [2] Virtutech Simics 3.0. <http://www.virtutech.com/>.
- [3] A. Agarwal and K. Roy. A noise tolerant cache design to reduce gate and sub-threshold leakage in the nanometer regime. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2003.
- [4] 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.
- [5] 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.
- [6] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Sez nec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Computer Architecture News*, 33(4), 2005.
- [7] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *International Symposium on Computer Architecture*, Anchorage Alaska, May 2002.
- [8] M. Goma, 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.
- [9] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. In *Proceedings of the International Symposium on Low Power Electronics*, October 1995.
- [10] M. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *SUIF Compiler Workshop*, Stanford University, CA, Aug. 1997.
- [11] 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.
- [12] H. Jiang and V. Chaudhary. Compile/run-time support for thread migration. In *Proceedings of International Parallel & Distributed Processing Symposium*, 2002.
- [13] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, October 1999.
- [14] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings of the International Symposium on Computer Architecture*, 2001.
- [15] 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.
- [16] 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.
- [17] Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, energy, and thermal considerations for SMT and CMP architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] M. Nikitovic and M. Brorsson. A multiprogrammed workload model for energy and performance estimation of adaptive chip-multiprocessors. In *International Parallel and Distributed Processing Symposium*, 2004.
- [22] 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.
- [23] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.
- [24] 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.
- [25] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.2. <http://quid.hpl.hp.com:9081/cacti/>.
- [26] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.
- [27] 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.
- [28] 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.
- [29] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction cache leakage optimization. In *Proceedings of the International Symposium on Microarchitecture*, 2002.