Power-Capped DVFS and Thread Allocation with ANN Models on Modern NUMA Systems

Satoshi Imamura* Hiroshi Sasaki† Koji Inoue* Dimitrios S. Nikolopoulos‡
*Graduate School and Faculty of Information Science and Electrical Engineering, Kyushu University, Fukuoka, Japan
†Department of Computer Science, Columbia University, New York, NY, USA
‡School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast, UK
s-imamura@soc.ait.kyushu-u.ac.jp, sasaki@cs.columbia.edu, inoue@ait.kyushu-u.ac.jp, d.nikolopoulos@qub.ac.uk

***Abstract*—Power capping is an essential function for efficient power budgeting and cost management on modern server systems. Contemporary server processors operate under power caps by using dynamic voltage and frequency scaling (DVFS). However, these processors are often deployed in non-uniform memory access (NUMA) architectures, where thread allocation between cores may significantly affect performance and power consumption. This paper proposes a method which maximizes performance under power caps on NUMA systems by dynamically optimizing two knobs: DVFS and thread allocation. The method selects the optimal combination of the two knobs with models based on artificial neural network (ANN) that captures the non-linear effect of thread allocation on performance. We implement the proposed method as a runtime system and evaluate it with twelve multithreaded benchmarks on a real AMD Opteron based NUMA system. The evaluation results show that our method outperforms a naive technique optimizing only DVFS by up to 67.1%, under a power cap.**

I. Introduction

The number of cores available on server systems has increased as technology shrinks. The era of manycore servers containing tens or hundreds of cores on a rack unit has come. However, power consumption also keeps increasing and becomes the primary concern for such systems. Power capping, which enables us to set an arbitrary power constraint, consequently becomes an essential function for efficient power budgeting and cost management [4], [10], [17]. Maximizing performance under power caps is a critical problem to address.

The NUMA architecture is most common in servers, due to its better scaling properties than architectures based on uniform memory access [6]. On the NUMA architecture, multiple nodes, each with processor(s) and DRAM, are connected via a system-level interconnection network. Each processor on a node typically includes multiple cores and a last level cache (LLC) that is shared between those cores. The physical memory address space is globally shared and therefore all cores on a system can access DRAM in all nodes. Since memory accesses to a remote node must additionally traverse the interconnect and a memory controller on the remote node, remote memory accesses take longer than local memory accesses.

On a manycore system based on the NUMA architecture, multithreaded programs are often executed to make full use of plenty hardware resources. In this case, several factors such as data locality, LLC contention, cache coherence overhead due to data sharing and contention on memory controllers and buses affect performance and power consumption [2], [16], [19], [20]. The impacts of these factors largely depend on how threads are allocated across multiple nodes (i.e., thread allocation).

Processors such as Intel's Sandy Bridge family and AMD's family 15h enable power capping by using DVFS. For example, Intel and AMD support RAPL [11] and TDP Limiting [1], respectively. However, when we execute a multithreaded program under a power cap on a NUMA system, performance is often not maximized even with DVFS. This is because threads are allocated by the OS scheduler without consideration of an underlying NUMA architecture and the impacts of thread allocation on performance and power consumption are ignored. In order to address this challenge, we need to find alternative methods to simultaneously optimize the two knobs and explore their power and performance tradeoff.

In this work, we propose and implement a method which maximizes performance of a multithreaded program executed on a NUMA system under power caps by dynamically optimizing DVFS and thread allocation. This paper makes the following contributions. (i) Based on performance and power consumption measurements using a set of multithreaded benchmarks on our experimental platform, we show that the simultaneous optimization of DVFS and thread allocation improves performance by up to 45.3% under a power cap, compared to DVFS-only optimization. (ii) We propose ANN-based models (one for performance prediction and another for power prediction) to estimate performance and power consumption for various combinations of the two knobs, based on data sampled from hardware performance counters. ANN can capture the complex and non-linear performance implications of thread allocation. It is also simpler to implement than statistical regression models that are commonly used for developing predictive models because ANN requires less rigorous statistical training [22]. (iii) We propose a method to select the optimal combination of the two knobs under power caps, based on estimated performance and power consumption. (iv) We implement the method as a runtime system and evaluate its performance and power consumption with several multithreaded benchmarks on a real NUMA system. The evaluation results show that it outperforms a naive technique optimizing only DVFS by up to 67.1% under a power cap.

II. Experimental Setup

A. Platform

Our experimental platform is a quad socket IBM System x3755 M3 server containing four 16-core AMD Opteron 6282 SE processors based on the Bulldozer architecture. Each processor includes two dies. The server is an 8-node NUMA system where each node contains a processor die and 12GB of DRAM. The nodes are interconnected with AMD's HyperTransport. The total number of cores is 64 and the total

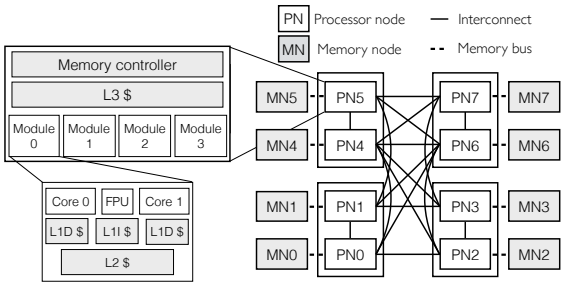


Fig. 1. Configuration of our 8-node NUMA platform. A processor node integrates four AMD Bulldozer modules, an 8MB shared L3 cache and a memory controller. Each module contains two cores, two 16KB L1 data caches and three components shared by the two cores: a floating point unit (FPU), a 64KB L1 instruction cache and a 2MB L2 cache.

size of main memory is 96GB. Fig. 1 shows the configuration of this platform. Five levels of DVFS are available: 1.4, 1.7, 2.0, 2.3 and 2.6 GHz. To remove the effect of automatic CPU frequency control by hardware we disable AMD's Turbo Core in the BIOS.

Our system runs Linux (kernel version is 3.9.4). DVFS level is controlled with the `cpufreq` utility in Linux and all the cores operate at the same level. We use Likwid [21] for sampling hardware performance counters, controlling thread allocation and binding threads to cores (see Section II-D for details). Power consumption of the entire system is measured with a WattsUp? Pro power meter [23] at the maximum available sampling rate (one sample per second).

B. Benchmarks

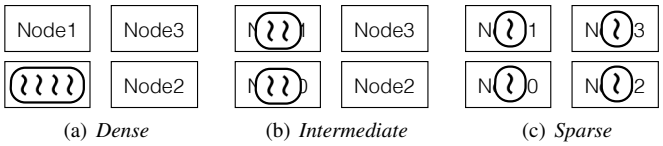
We choose twelve benchmarks that are sensitive to thread allocation in terms of performance from the PARSEC 3.0, Splash-2X [3], and NAS Parallel Benchmark (NPB) suites, all implemented with OpenMP [12]. We use the *native* input size for PARSEC 3.0 and Splash-2X benchmarks and the Class B problem size for NPB. We measure and report performance and power only during the parallel region (ROI: region of interest) of each benchmark.

C. Memory Page Allocation

Linux provides two page allocation policies: *Local* and *Interleaving*. *Local* is the default policy where pages are allocated on-demand on the node a thread requesting them runs. This policy aims to obtain good DRAM access locality by minimizing remote accesses. *Interleaving* evenly allocates pages across nodes. For a program where the main thread allocates almost all pages sequentially, *Local* will incur high contention on memory resources because the pages will be allocated on a single node. In this case, *Interleaving* significantly improves performance by reducing contention [6]. One benchmark in our suite, `canneal`, has this property and requires *Interleaving* policy to maximize performance. In all other benchmarks we use the *Local* policy. We select the page allocation policy with the `libnuma` library.

D. Thread Binding

Multithreaded programs executing on NUMA platforms often exhibit performance variation across execution due to variations in page allocation and non-deterministic thread migrations imposed by the OS. To curb this problem, we bind each thread to a specific core with Likwid that provides a function to bind threads to cores at the time of thread creation. Threads are mapped to cores in the order of their creation, starting from the core with the lowest ID. Our experiments show that thread binding keeps the coefficient of variation (CV) of the execution time among five runs below 0.03 for all benchmarks. However, thread binding significantly degrades performance only in one benchmark, `x264`. This benchmark performs dynamic thread allocation, which introduces load imbalance if threads are bound to fixed cores. Therefore, we execute this benchmark without thread binding to prevent performance degradation. The CV of `x264` is below 0.01 without thread binding.



(a) *Dense* (b) *Intermediate* (c) *Sparse*

Fig. 2. Examples of three types of thread allocations with four threads.

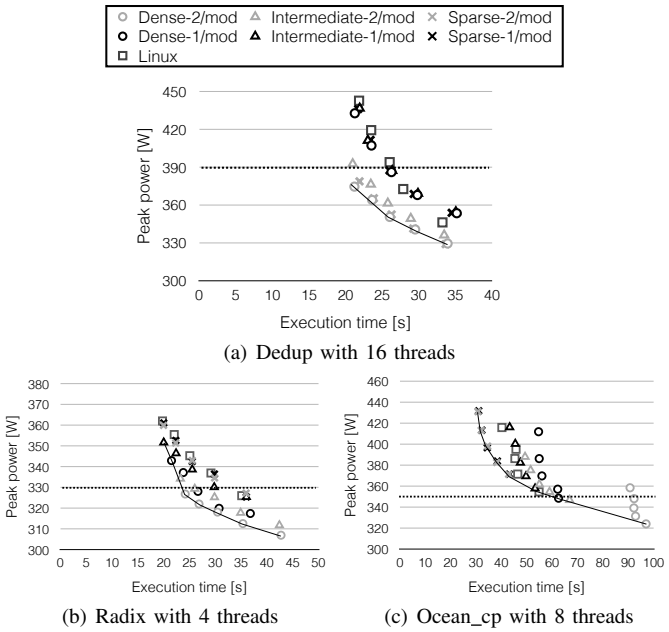
III. Motivation

A. Thread Allocation

As thread allocation has great impact on both performance and power consumption, we need to carefully explore its optimization space. Since our target platform consists of multiple nodes and each node includes several modules, we can consider two granularities of thread allocations. We define three types of coarse-grain allocation policies: *Dense*, *Sparse*, and *Intermediate* as depicted in Fig. 2. For fine-grain (or module level) allocations that attempts to assign a module to each thread, on the other hand, two policies are introduced: *1-thread per module* and *2-threads per module*. By leveraging the two different granularities in thread allocation, six policies are available in total. The detail of coarse-grain allocation policies is as follows.

- **Dense** (Fig. 2(a)) allocates threads on as few nodes as possible. This allocation can obtain high data access locality and reduce data sharing overhead, but may also cause high LLC contention. Moreover, if the page allocation policy is *Local*, it may cause high contention of memory bus and memory controller.
- **Sparse** (Fig. 2(c)) evenly distributes threads between nodes and therefore it has the opposite characteristics to *Dense*. In terms of power consumption, *Dense* tends to consume lower power than *Sparse* because it allocates threads on fewer nodes [2].
- **Intermediate** (Fig. 2(b)) attempts to assign the average number of nodes to be used on *Dense* and *Sparse*. The digit after a decimal point of the number of nodes is rounded up or down for load balancing. For example in Fig. 2, the average number of nodes is 2.5 (*Dense* uses one node and *Sparse* uses four nodes) and we have four threads. Since using two nodes makes load balancing better than a three-node assignment, the allocation policy rounded down the average to 2.0.

For the fine-grain (or module-level) thread allocation policies, the negative effects of resource contention must be considered. As Fig. 1 shows, an L1 instruction cache, an L2 cache and an FPU are shared by two neighbor cores in a module. If two threads are allocated in different modules, i.e.,



(a) Dedup with 16 threads

(b) Radix with 4 threads

(c) Ocean_cp with 8 threads

Fig. 3. Power consumption and performance of combinations of DVFS and thread allocation. Different dot symbols indicate different thread allocation policies. Gray and black curves imply allocations with 2-threads and 1-thread per module, respectively. We represent five DVFS levels with five dots, for any given thread allocation.

1-thread per module policy, the threads will not compete for shared resources but power consumption may increase due to the increased number of activated modules. On the other hand, if the two threads are packed into a single module, i.e., *2-threads per module* policy, we can expect power reduction but contention for shared resources may degrade performance.

We can choose 1-thread or 2-threads per module for each of *Dense*, *Intermediate* and *Sparse* policies. For instance, when eight threads are running, the combination of *Dense* and *1-thread per module* uses two nodes with four modules each, whereas *Dense* with *2-threads per module* allocation uses four modules of a single node. We use abbreviations such as *Dense-1/mod* (meaning *Dense* thread allocation with *1thread per module*) to represent thread allocations.

B. Power-Performance Tradeoff of DVFS and Thread Allocation

In order to show that simultaneous optimization of DVFS and thread allocation is necessary, we measure execution time and peak power consumption throughout the execution of each benchmark, with various combinations of the two knobs. We explore scenarios where programs with limited scalability are executed with the optimal number of threads [18]. Thus, we run each benchmark with 4, 8, 16, or 32 threads to comply with benchmarks that require power-of-two threads. Fig. 3 plots the results. We focus on three benchmarks with representative characteristics. The black line at the lower left on each graph means Pareto frontier and dots along the line show the optimal combinations, namely those that achieve the best performance under different power caps. *Linux* represents executions where thread allocation is left to the Linux scheduler. This is our baseline for comparisons to our thread allocation schemes.

Taking `dedup` with 16 threads as an example in Fig. 3(a), we find that the optimal thread allocation is always *Dense-2/mod*. Therefore, we can maximize performance by selecting that thread allocation and optimizing only DVFS given a power cap. To meet a 390W cap, CPU frequency can be set to 2.6 GHz with *Dense-2/mod* while it must be set to 1.7 GHz with *Linux*. In this case, *Dense-2/mod* can improve performance by 31.5%, compared to *Linux*.

In `radix` with four threads (Fig. 3(b)), the optimal thread allocation varies with different power caps. While *Intermediate-1/mod* achieves the highest performance without a power cap, *Dense-2/mod* does so under a 330W power cap. Without a power cap, performance is identical between the optimal thread allocation and *Linux*. However, under a 330W cap, *Dense-2/mod* at 2.6 GHz outperforms *Linux*, which must scale down frequency to 1.4 GHz to meet the power cap, by as much as 45.3%.

Furthermore, in `ocean_cp` with eight threads (Fig. 3(c)), *Sparse-2/mod* achieves the highest performance under power caps over 350W. However, the same thread allocation and *Linux* can not meet power caps below 350W even at the minimum DVFS level. In this case, we need to select a sub-optimal thread allocation to meet power caps. Overall, we find that optimizing performance under power capping for different applications requires varying and non-obvious combinations of DVFS and thread allocation settings.

IV. Optimization of DVFS and Thread Allocation with ANN-Based Models

A. Overview

Our proposed method dynamically optimizes two knobs (DVFS and thread allocation) with ANN-based models. Its objective is to maximize performance under power capping. Our method repeats the following three steps during program execution. First, it periodically samples data to collect inputs of ANN-based models. Second, it estimates performance and power consumption for all combinations of the two knobs with the models. Third, it selects the combination that is expected to maximize performance while keeping power consumption under a given power cap.

In order to search for the optimal knob settings at runtime, one may consider an exhaustive approach. The runtime system can find optimal combinations of settings by comparing the performance and power consumption of each combination. The runtime system can either use iterative execution of the whole program, or change knob settings and measure performance and power in the same run, if the program is itself iterative and its control and data flow remain unchanged across iterations. Unfortunately, an exhaustive approach can be impractical. In our experimental setup, we would need to dynamically compare 30 combinations (5 DVFS levels $\times$ 6 thread allocations) for every exhaustive search, in each program. Although the overhead to change CPU frequency is slight (in the order of microseconds), changing thread allocation is not trivial because it requires thread migrations, for which the overhead can be tens or hundreds of milliseconds. We would need at least six expensive changes of thread allocation. If the search must be repeated to handle dynamic behavior in the program (e.g., different phases of execution with different performance and power characteristics), the search overhead may offset any performance improvement obtained by selecting the optimal combination of knobs. To combat the overhead of searching the optimization space for maximizing performance under a power cap, we adopt a model-based approach.

TABLE I. 22 INPUTS OF ANN-BASED MODELS

	Parameter	Source of value
Execution status	DVFS level	(Given)
	# of threads	(Given)
	# of used modules	# of threads and thread allocation
	# of used nodes	
Sampling data	Voluntary context switches	*/proc/[pid]/status*
	Involuntary context switches	
	Per-core CPU utilization	*/proc/stat*
	Retired instructions	PMC
	Instruction cache misses	PMC
	L2 cache misses	PMC
	Retired floating point ops	PMC
	L3 cache misses	NBPMC
	Local/Remote memory accesses	NBPMC
	Memory access latency to node 0-3/4-7	NBPMC
	Memory requests to node 0-3/4-7	NBPMC
	DRAM page conflicts and misses	NBPMC
	Memory controller read requests	NBPMC
	Memory controller write requests	NBPMC
	Power consumption	Wattsup? PRO

B. ANN-Based Models

1) Overview of ANN: ANN is a statistical modeling tool that follows the principles of operation of the human brain [22]. The reason why we choose ANN is because it can capture the complex non-linear impact of thread allocation on performance and it is simple to implement, as it requires less rigorous statistical training compared to other non-linear models such as statistical regression models. Moreover, ANN can detect all possible interactions between independent variables. An ANN-based model consists of input layer, hidden layer(s) and output layer. The input and output layers correspond to independent and dependent variables, respectively. The hidden layer(s) allows the network to model complex non-linear relationships between inputs and outputs [22]. The following four parameters need to be decided when implementing an ANN-based model: the number of hidden layers, and number of neurons in each of input, hidden and output layers. As explained later, the input and output layers contain 22 and 30 neurons, respectively. We use only one hidden layer which contains the mean number of neurons between the numbers of neurons in the input and output layers, based on a common ANN implementation approach [9]. To estimate performance and power consumption, we use two distinct models.

2) Inputs: We use 22 independent variables as inputs which correspond to the execution status (e.g., the number of threads, the number of used nodes, etc.) and data sampled from hardware performance counters to capture power and performance characteristics of a running program. TABLE I summarizes the inputs. The second column shows the parameter measured for each input and the third column shows the source from where the value of each parameter is obtained. DVFS level and the number of threads are given by the our optimization algorithm. The number of used modules and used nodes are obtained based on the selected number of threads and thread allocation. The number of context switches and per-core CPU utilization (the average of used cores) can be read from the */proc* file system on Linux. On the AMD Bulldozer architecture, various performance counter events are available, which are classified into core performance counter events (PMC) and North bridge performance counter events (NBPMC) [1]. Moreover, we assume availability of a register to store the power consumption but actually obtain the power value from the WattsUp? Pro power meter.

TABLE II. AVERAGE PREDICTION ERRORS OF OUR MODELS

Model	4 threads	8 threads	16 threads	32 threads
Performance	9.1%	9.6%	14.1%	12.0%
Power	2.1%	3.1%	4.0%	7.8%

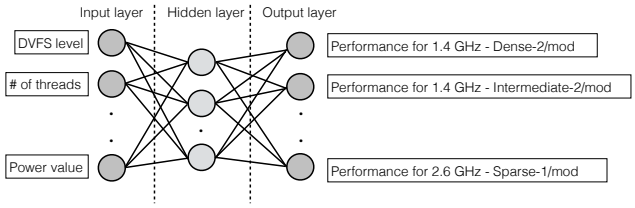


Fig. 4. Structure of performance model. The power model has the same structure, except from the output values.

3) Outputs: Our performance and power models output relative performance and absolute power consumption, respectively. The output layer of each model contains 30 neurons, which correspond to all combinations of DVFS level and thread allocation. The relative performance is calculated as the performance (inverse of execution time) of a program with each combination normalized by performance with the best (fastest) combination. Fig. 4 shows the structure of our performance model. The structure of the power model is the same, except from its output values.

4) Training and Prediction Accuracy Evaluation: We create different models for performance and power consumption, for any given number threads used in the program. We thus create four performance models and four power models in our setup, representing executions with 4, 8, 16, and 32 threads. For training we measured execution time and power consumption, and sampled the metrics shown in TABLE I using 12 benchmarks with changing combinations of DVFS level and thread allocation. PMC and NBPMC data is obtained with Likwid. ANN-based models tend to overfit on training data, thus leading to low prediction accuracy when presented with unseen input data that has not been used for training. To address this problem, we apply a *leave-one-out cross validation* method. In particular, we use measured data from 11 out of 12 benchmarks for training and the data from the remaining benchmark to evaluate prediction accuracy. We use the *neuralnet* package [8] in R 3.1.0 [15] for training. TABLE II summarizes prediction errors of performance and power models for different thread counts. Each value indicates the average error of 12 models trained with leave-one-out cross validation. The error is defined as:
$(MeasuredValue - EstimatedValue)/MeasuredValue$

C. Algorithm

Fig. 5 shows the flow chart of our power-capped performance optimization algorithm. At the beginning of program execution, DVFS level is set to the lowest in order to avoid violating the power cap (P_{cap}). Thread allocation is initialized to *Sparse-2/mod* for allocating memory pages across as many nodes as possible with the *Local* page allocation policy and avoiding high contention on memory bus and controller. To handle possible prediction errors of the power models, we use P_{target} which is a power threshold that the proposed method should actually stay under. P_{target} is initialized to $P_{cap} \times (1 - AvgError)$ where $AvgError$ is the average error of the power model shown in TABLE II. This initial value

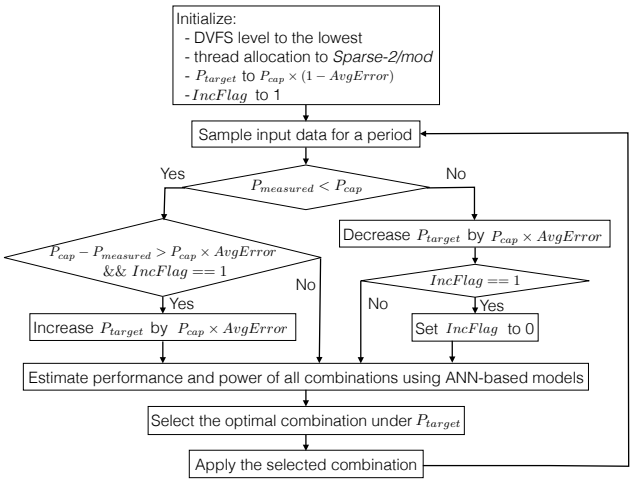


Fig. 5. Algorithm flow chart of our model-based technique.

```
1: Store the measured power to P_measured
2: if level_increased == 1 then
3:     powerdiff[dvfs_level − 1] ← P_measured − P_prev
4:     level_increased ← 0
5: end if
6: if P_measured < P_cap then
7:     if dvfs_level ≠ MAX_LEVEL then
8:         if powerdiff[dvfs_level] == NULL then
9:             d ← powerdiff[dvfs_level − 1]
10:        else
11:            d ← powerdiff[dvfs_level]
12:        end if
13:        if P_cap − P_measured > d then
14:            P_prev ← P_measured
15:            level_increased ← 1
16:            dvfs_level + +          ▷ Increment DVFS level
17:        end if
18:    end if
19: else
20:    dvfs_level ← 0          ▷ Decrease DVFS level to the minimum
21: end if
```

Fig. 6. Algorithm of *Naive-DVFS* to control DVFS level under P_{cap}.

means that a combination of DVFS level and thread allocation should conservatively take into account potential underestimation of power consumption. The proposed method may also select a combination which accounts for overestimation if P_{target} is $P_{cap} \times (1 + AvgError)$. P_{target} is dynamically adjusted by $P_{cap} \times AvgError$. Note that P_{target} is not used as an input to the models. Furthermore, $IncFlag$, which is a flag to decide whether P_{target} should be increased during execution or not, is initialized to 1.

Following initialization, the input data for the models is sampled for a period. If the measured power ($P_{measured}$) is less than P_{cap}, the algorithm checks whether the difference between P_{cap} and $P_{measured}$ is larger than $P_{cap} \times AvgError$ and whether $IncFlag$ is 1 or not. If these two conditions are true, P_{target} is increased by $P_{cap} \times AvgError$. On the other hand, if $P_{measured}$ exceeds P_{cap}, P_{target} is decreased by $P_{cap} \times AvgError$. $IncFlag$ is then reset to 0, which implies that P_{target} should not be increased further once $P_{measured}$ exceeds P_{cap}.

After tuning P_{target}, the relative performance and power consumption for all combinations are estimated using ANN-based models with the input data shown in TABLE I. Based on the estimated performance and power consumption, the combination of DVFS level and thread allocation that is expected to achieve the highest performance under P_{target} is selected and applied. The steps between data sampling and applying the optimal combination are repeated during program execution.

We measure the execution time of the algorithm itself (from data sampling to selection of the optimal combination) on our experimental platform. The longest execution time of the algorithm that we measured was 6 ms. For long-running applications, this overhead is negligible. The major overhead of our proposed method is thus the overhead of thread migrations, which nevertheless occur only when the models predict a change in the optimal thread allocation.

D. Implementation

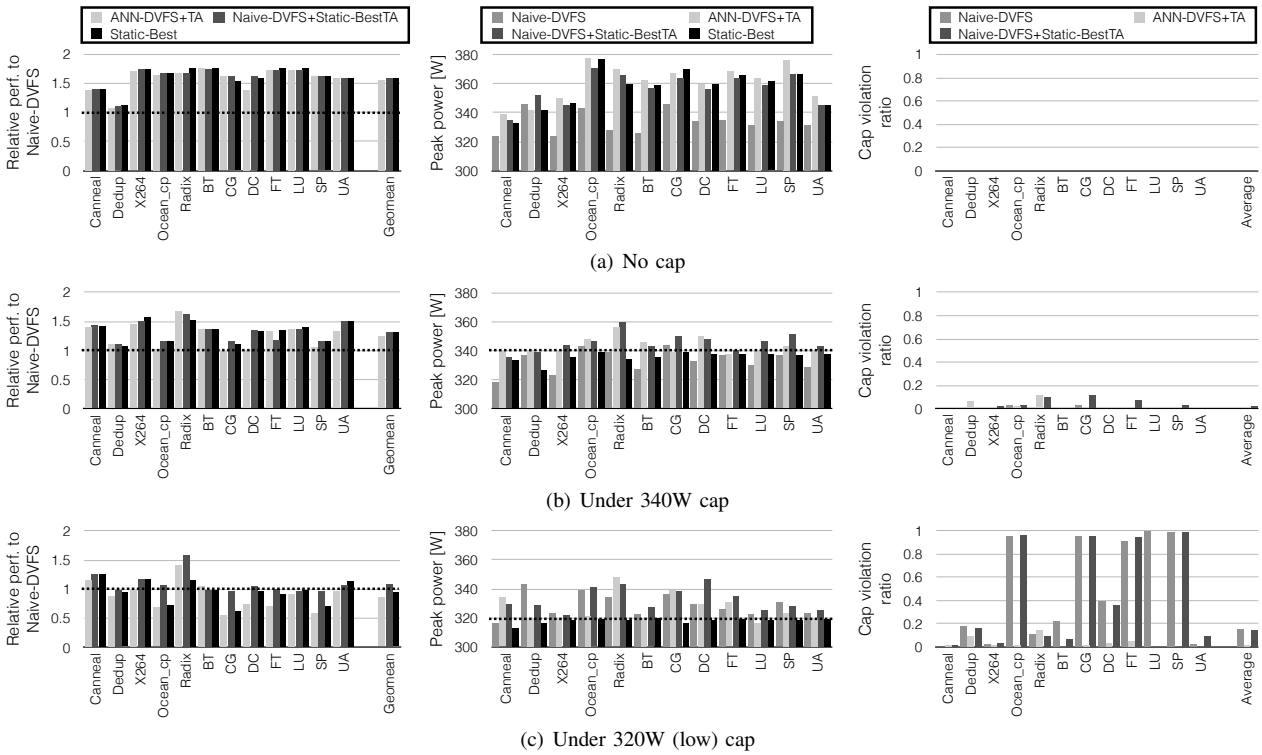
We implement the proposed method as a runtime system based on Likwid, where we sample input data for the models from hardware performance counters. Our platform has only four counters for NBPMC events, which is a limitation, since the runtime system needs to sample ten events. Due to this limitation, we cannot sample all of the events during once execution. In order to address this limitation, we modify Likwid so that counters for NBPMC events can be time-shared for more events. Concretely, we divide ten NBPMC events into three groups and sample events in each group for one-third of the sampling period. As discussed in Section IV-B2, we obtain power consumption from a power meter. Since one second is the shortest possible sampling interval of the meter, we set our overall sampling period to one second. Given this coarse sampling period, our method can hide the overhead of changing thread allocation and sustain high performance (see Section V for details). If an infrastructure to collect power measurements at finer granularity was available, we would need to trade the performance improvement obtained from fine-grain optimization with the overhead imposed by more frequent changes of thread allocation. We intend to explore this problem in future work. We control DVFS level with the `cpufreq` utility of Linux and thread allocation with the `sched_setaffinity(2)` system call.

V. Evaluation of ANN Model-Based Technique

A. Baselines

We consider three baselines to our approach: *Naive-DVFS*, *Naive-DVFS+Static-BestTA* and *Static-Best*. *Naive-DVFS* dynamically controls DVFS level to satisfy a power cap and leaves thread allocation to the Linux scheduler. We develop a simple DVFS algorithm for *Naive-DVFS* as explained in Fig. 6 that is invoked every one second[1]. The algorithm compares the measured power $P_{measured}$ with P_{cap}. If there is enough power headroom (i.e., P_{cap} - $P_{measured}$ is enough large), DVFS level $dvfs\_level$ is incremented. We predict the power to be increased by incrementing $dvfs\_level$ with $powerdiff$, which is an array recording the power differences when DVFS level is previously incremented. For instance, the first entry of $powerdiff$ stores the power difference when $dvfs\_level$ is incremented from zero to one. Since five DVFS levels are available on our platform, $powerdiff$ contains four entries. Note that the following three steps are executed before the

[1]*Naive-DVFS* can also be implemented with an interface to automatically control DVFS level under a power cap, such as Intel's RAPL. We leave the comparison to a hardware implementation for future work.



(a) No cap

(b) Under 340W cap

(c) Under 320W (low) cap

Fig. 7. Evaluation results with 4 threads.

algorithm in Fig. 6 is repeated. First, $dvfs\_level$ is set to zero and each entry of $powerdiff$ is initialized to $NULL$. Second, the measured power is stored to P_{prev}. Third, $dvfs\_level$ is incremented to one and $level\_increased$, which is a flag to know whether DVFS level is increased in the previous iteration, is set to one.

Naive-DVFS+Static-BestTA and *Static-Best* are oracular and thus impractical approaches, since they need to compare performance and power consumption of knob settings offline. We investigate how our proposed method appropriately selects the optimal combination by comparing it with these two baselines. *Naive-DVFS+Static-BestTA* dynamically controls DVFS level to satisfy a power cap using the same algorithm as *Naive-DVFS*. It statically selects the best thread allocation that maximizes performance without a power cap. *Static-Best* statically selects the best combination of DVFS level and thread allocation that maximizes performance under a power cap. The program is executed with the best settings without changing them at runtime.

B. Experimental Results and Discussions

We evaluate performance and power consumption of our model-based method and three baselines with 4, 8, 16 and 32 threads, while changing the power cap. Fig. 7 shows the evaluation results with 4 threads. Our proposed method is labeled as *ANN-DVFS+TA* in the figure. The graphs on the right present the cap violation ratio, which is the ratio of power values violating power caps to all power values measured during program execution. We assume that power cap violations for a very small fraction of program execution (i.e., low cap violation ratio) does not affect power cost and system reliability. Since *Static-Best* never violates a power cap, it exhibits a zero violation ratio and is not plotted in the graph. Each data point of all techniques except *Static-Best* in these graphs represents the best of five runs in terms of performance.

Fig. 7(a) shows the results without a power cap. Since all of four alternatives can execute the program at the highest DVFS level, performance depends only on thread allocation. We observe in the graph on the left that *ANN-DVFS+TA* outperforms *Naive-DVFS* by a significant margin. The geometric mean (Geomean) of performance improvement across the 12 benchmarks is 56.1%. Moreover, our method achieves comparable performance to *Naive-DVFS+Static-BestTA* and *Static-Best*. The Geomean of performance degradation compared to these oracular techniques is only 2.0% and 2.6%, respectively. These results show that our method appropriately selects the optimal thread allocation on the highest DVFS level.

We show results under a 340W cap in Fig. 7(b). This cap is set as the middle value between the highest power that is consumed by the most power-consuming of 12 benchmarks (`ocean_cp` consumes 376W with the optimal knob setting) and the idle power of our platform, which is around 280W. While all techniques except *Static-Best* violate the cap for some benchmarks (middle graph), cap violation ratios are low (right graph). Under the 340W cap, *ANN-DVFS+TA* outperforms *Naive-DVFS* (Geomean of performance improvement is 24.6%) and achieves comparable performance to *Naive-DVFS+Static-BestTA* and *Static-Best* (Geomean of performance degradation is 5.8%).

Interestingly, in `radix`, *ANN-DVFS+TA* outperforms all other techniques. The performance improvement is 67.1% compared to *Naive-DVFS*, 2.5% compared to *Naive-DVFS+Static-BestTA*, and 10.4% compared to *Static-Best*. Fig. 8 provides insight in the reason behind this result. In the top graph, we observe that power consumption suddenly increases near the end of execution. Although *Naive-DVFS*

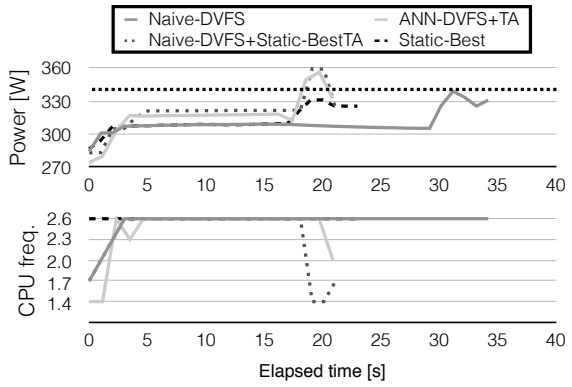


Fig. 8. Runtime results of `radix` with four threads under a 340W cap.

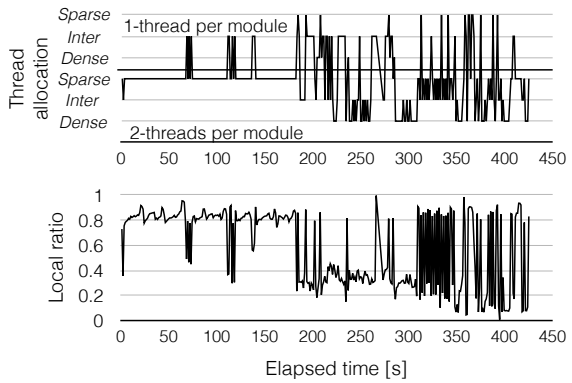


Fig. 9. Thread allocation and local to total memory accesses ratio of *ANN-DVFS+TA* when `DC` is executed with four threads under a 340W cap.

keeps the highest DVFS level, it performs much worse than other techniques because it ignores thread allocation. Fig. 3(b) shows that *Intermediate-1/mod*, *Sparse-1/mod* or *Sparse-2/mod* are the best thread allocations without a power cap for this benchmark. *Static-Best* selects a sub-optimal (but low power) allocation to keep peak power consumption under the cap. In contrast, *ANN-DVFS+TA* and *Naive-DVFS+Static-BestTA* select one of the best thread allocations at the highest DVFS level, while power consumption stays below the cap. Although these methods violate the power cap at the end of execution, cap violation ratios are relatively low at 11.1% and 9.1%.

We show results under a 320W power cap in Fig. 7(c). This cap is relatively low for multithreaded executions, even with four threads. In the left graph, we observe that performance of *ANN-DVFS+TA* and *Static-Best* is significantly lower than *Naive-DVFS* and *Naive-DVFS+Static-BestTA* for several benchmarks. However, *Naive-DVFS* and *Naive-DVFS+Static-BestTA* violate the cap during large portions of execution time (right graph). This is because they cannot select a low-power thread allocation, even though they do set DVFS level to the minimum. In contrast, *ANN-DVFS+TA* can save power by setting DVFS level to the minimum and selecting low-power thread allocations, which translate to drastic reduction of the cap violation ratio. Considering the Geomean of 12 benchmarks, *ANN-DVFS+TA* achieves 86.0%, 79.4% and 90.8% of the performance of *Naive-DVFS*, *Naive-DVFS+Static-BestTA* and *Static-Best* respectively, but keeps the cap violation ratio at a mere 2%.

Fig. 7 (left) suggests that the performance of *ANN-DVFS+TA* is much lower compared to *Naive-DVFS+Static-BestTA* and *Static-Best* for some benchmarks. In particular, the degradation is 33.6% and 33.0% for `DC` under a 340W power cap. The overhead of changing thread allocation is one of the reasons for the observed low performance. However, our proposed method changes thread allocation at most once per second because the data sampling period is one second. In fact, the number of thread allocation changes per second is at most 0.74 across all experiments explained in this section. Thus, the overhead may not fully explain the performance degradation. We explore another explanation using the experimental results of `DC` with four threads under a 340W cap, shown in Fig. 9. The bottom graph presents the ratio of the number of local memory accesses to the number of all memory accesses. This benchmark has inherent data locality because the aforementioned ratio is 0.93, when the benchmark is executed by *Static-Best* with the *Local* page allocation policy. For a large part of the execution that spans 180 seconds, *ANN-DVFS+TA* selects the *Sparse-2/mod* thread allocation. The local to total memory access ratio remains high (0.8) because each thread keeps running on the node where it allocates its own pages. However, thread allocation is frequently changed after 180 seconds by our policy and this increases dramatically remote memory accesses. Performance thus drops due to contention in memory resources. Similar behavior is observed in other benchmarks where the performance degradation from our algorithm is significant. We expect to address this problem by exploiting dynamic page migration [7], [13].

TABLE III. EVALUATION RESULTS OF 8, 16, AND 32 THREADS

Cap	Relative perf. to *Naive-DVFS*			Cap violation ratio		
	ANN -DVFS +TA	*Naive-DVFS +Static -BestTA*	*Static -Best*	*Naive -DVFS*	*ANN -DVFS +TA*	*Naive-DVFS +Static -BestTA*
8 threads						
No cap	1.58	1.58	1.61	0	0	0
380W	1.19	1.27	1.28	0.02	0.02	0.03
340W	0.81	1.06	0.85	0.21	0.02	0.22
16 threads						
No cap	1.37	1.42	1.45	0	0	0
450W	1.20	1.27	1.26	0.03	0.03	0.04
390W	0.96	1.11	0.98	0.20	0.03	0.17
32 threads						
No cap	1.13	1.13	1.18	0	0	0
550W	1.13	1.14	1.17	0.05	0.04	0.05
450W	0.98	1.07	1.01	0.21	0.04	0.13

Due to space limitations, we summarize the evaluation results of 8, 16, and 32 threads in TABLE III. We show the relative performance compared to *Naive-DVFS* and the power cap violation ratio. The reported values of relative performance and power cap violation ratio are geometric and arithmetic means across our 12 benchmarks, respectively. We observe that the performance improvement of our technique drops with an increasing number of threads. This is because the benefit of optimizing thread allocation decreases as the number of threads increases. For example, when a program that needs more LLC space is executed with eight threads, *Sparse-1/mod* can achieve much higher performance by utilizing eight LLCs on eight nodes than *Dense-2/mod*, which uses one LLC on one node. However, when the same program is executed with 32 threads, *Dense-2/mod* uses four LLCs while *Sparse-1/mod* uses eight LLCs. Therefore, the performance difference between them diminishes, compared to the execution with eight threads.

VI. Related Work

Performance Optimization under Power Caps: Ma et al. propose a technique to control DVFS level under a given power budget for a mix of single-threaded and multithreaded programs executed on manycore systems [14]. Cochran et al. [4], Imamura et al. [10] and Sasaki et al. [17] aim to dynamically optimize the number of cores and their DVFS level. While Cochran et al. and Imamura et al. target multithreaded programs, Sasaki et al. target a workload composed of multiple multithreaded programs. Our technique tries to maximize performance under power caps by optimizing DVFS and thread allocation for standalone multithreaded programs executed on NUMA systems.

Optimization of Thread Allocation on NUMA Systems: Tang et al. investigate the tradeoff between improving data locality and reducing LLC contention for several web-service workloads of Google on NUMA systems [20]. Rao et al. consider two types of thread allocations similar to the *Dense* and *Sparse* allocations used in our work, and reveal the impact of data locality, LLC contention, and data sharing overhead on performance [16]. They also propose a NUMA-aware algorithm to optimize thread allocation. Bae et al. consider similar thread allocations and show that thread allocation affects performance and power consumption in a multithreaded program and propose a system that dynamically chooses between the two allocations according to program characteristics [2]. In contrast, we stress the necessity of simultaneous optimization of DVFS and thread allocation to maximize performance under power caps.

ANN Model-Based Approaches: Yoo et al. use an ANN-based model to capture the non-linear relation between workload configurations and performance [24]. Curtis-Maury et al. propose an ANN model-based technique to reduce energy consumption by dynamically optimizing the number of threads to execute a multithreaded program [5]. Furthermore, Su et al. try to optimize performance, energy delay product, or performance per watt by dynamically controlling the number of threads and thread allocation on a NUMA architecture. They estimate the optimal number of threads based on their ANN-based model and then optimize thread allocation to reduce remote memory accesses and contention on memory controllers based on critical path analysis [19]. We use ANN-based models to estimate performance and power consumption for combinations of DVFS level and thread allocation under power capping.

VII. Conclusions

In this work, we show that simultaneous optimization of DVFS and thread allocation is effective to maximize performance under power caps on modern NUMA systems. We propose an ANN model-based technique that dynamically optimizes the two knobs and implement it as a runtime system. The evaluation of the proposed technique using various multithreaded benchmarks on a real NUMA system show that our technique outperforms a naive technique optimizing only DVFS by up to 67.1%.

Acknowledgment

This research was supported in part by JSPS Grant-in-Aid for JSPS Fellows (Grant Number 13J02888), JSPS Postdoctoral Fellowships for Research Abroad, the Japan Science and Technology Agency (JST) CREST program, the UK Engineering and Physical Sciences Research Council under Grants EP/L000055/1 (ALEA), EP/L004232/1 (ENPOWER) and EP/K017594/1 (GEMSCLAIM) and the European Commission under Grant Agreements FP7-323872 (SCoRPiO) and FP7-610509 (NanoStreams).

References

[1] AMD, "Bios and kernel developer's guide (bkdg) bios and kernel developer's guide for amd family 15h models 00h-0fh processors," January 2013.

[2] C. S. Bae, L. Xia, P. Dinda, and J. Lange, "Dynamic adaptive virtual core mapping to improve power, energy, and performance in multi-socket multicores," in *HPDC '12*, 2012.

[3] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[4] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive dvfs and thread packing under power caps," in *MICRO 44*, 2011.

[5] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. De Supinski, and M. Schulz, "Identifying energy-efficient concurrency levels using machine learning," in *CLUSTER '07*, 2007.

[6] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: a holistic approach to memory placement on numa systems," in *ASPLOS '13*, 2013.

[7] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on linux," in *IPDPS '09*, 2009.

[8] F. Gunther and S. Fritsch, "neuralnet: Training of neural networks," *The R Journal*, vol. 2, no. 1, pp. 30–38, 2010.

[9] J. Heaton, *Introduction to Neural Networks for Java, 2nd Edition*. Heaton Research, Inc., 2008.

[10] S. Imamura, H. Sasaki, N. Fukumoto, K. Inoue, and K. Murakami, "Optimizing power-performance trade-off for parallel applications through dynamic core and frequency scaling," in *RESoLVE '12*, 2012.

[11] Intel, "Programming guide volume 3b-2 of intel 64 and ia-32 architectures software developer's manual," December 2011.

[12] H. Jin, M. Frumkin, and J. Yan, "The openmp implementation of nas parallel benchmarks and its performance," Technical Report NAS-99-011, NASA Ames Research Center, Tech. Rep., 1999.

[13] H. Löf and S. Holmgren, "Affinity-on-next-touch: Increasing the performance of an industrial pde solver on a cc-numa system," in *ICS '05*, 2005.

[14] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *ISCA '11*, 2011.

[15] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2014.

[16] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in numa multicore systems," in *HPCA '13*, 2013.

[17] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated power-performance optimization in manycores," in *PACT '13*, 2013.

[18] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura, "Scalability-based manycore partitioning," in *PACT '12*, 2012.

[19] C. Su, D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. d. Supinski, and E. A. León, "Model-based, memory-centric performance and power optimization on numa multiprocessors," in *IISWC '12*, 2012.

[20] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing google's warehouse scale computers: The numa experience," in *HPCA '13*, 2013.

[21] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *ICPPW '10*, 2010.

[22] J. V. Tu, "Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes," *Journal of clinical epidemiology*, vol. 49, no. 11, pp. 1225–1231, 1996.

[23] watts up?, "Watts up? pro," https://www.wattsupmeters.com/secure/products.php?pn=0&wai=0&spec=4, Last checked: May 2014.

[24] R. M. Yoo, H. Lee, K. Chow, and H.-H. Lee, "Constructing a non-linear model with neural networks for workload characterization," in *IISWC '06*, 2006.