A Low Cost, High Performance Dynamic-Programming-Based Adaptive Power Allocation Scheme for Many-Core Architectures in the Dark Silicon Era

Xiaohang Wang*, Zhiming Li*, Mei Yang†, Yingtao Jiang†, Masoud Daneshtalab‡, Terrence Mak§
*Guangzhou Institute of Advanced Technology, CAS, China
Email: {xh.wang, zm.li}@giat.ac.cn
†University of Nevada, Las Vegas, USA
Email: mei.yang@unlv.edu, yingtao@egr.unlv.edu
‡University of Turku, Finland
Email: masdan@utu.fi
§The Chinese University of Hong Kong, China
Email: stmak@cse.cuhk.edu.hk

***Abstract*—Power consumption of many-core chips increases in such a rapid pace that it will soon exceed the chip's affordable power budget. As a result, design of a many-core chip has to address a significant performance challenge under a tight power budget constraint. This problem becomes more prevalent as more of the frequencies and/or voltages of on-chip resources in a many-core chip can be tuned, where heuristics based power allocation approaches often lead to poor performance. Another important problem is that the input power budget of a many-core chip might actually undergo a rapid change at run time. In this paper, the performance optimization problem is formally formulated, and an Optimal Power Allocation method using Dynamic programming (OPAD) is proposed to solve this problem. OPAD has a linear time complexity, and it is quite scalable to the problem size. Extensive experimental results have confirmed lower application execution time of OPAD than that of other competing power allocation methods, *i.e.*, 20%$\sim$30% reduction in applications' execution time over three competing methods. The runtime and hardware overhead of OPAD are also shown to be very small, making it suitable for adaptive power allocation in future many-core systems.**

***Keywords*—many-core, power budgeting, dynamic programming**

I. Introduction

Driven by continuous advances of CMOS technology and request of emerging applications, multi-core and many-core chips are widely used in cloud computing, mobile computing, high-performance computing, as well as many other important areas [1]. Explosive growth of computing performance, however, comes with a rapid increase of power consumption. According to ITRS, by the year 2020, the chip power consumption will increase by as much as a factor of 10 over the year 2012 [1]. This high power number, unfortunately, will be far higher than the affordable power budget (maximum power supply or thermal-compliant power budget). As a result, the real performance lagged far behind the idealized performance [2]. For instance, although transistor count will increase by a factor of 32 in 2020, the performance speedup can only be $7 \sim 8$ folds, according to a study indicated in [2]. This gap is caused by the "dark silicon" phenomenon [2] that only half of the chip's transistors can actually be powered on at a given time due to its power budget limit.

A subtle solution to this dark silicon problem is to maximize chip performance under a limited power budget. Optimization of such power budgeting [3] problem can become very complicated due to two challenging facts. First, in a state-of-the-art many-core chip, operating frequencies of many of its cores (or IPs) are tunable. For example, consider a 16-tile many-core system, where the frequency of each core can take one of the four allowed frequencies. The total frequency combination is as large as 4^{16}, which is about 4×10^9, and the optimal solution is obtained by exhaustively searching through all these solutions. Heuristic-based approaches [4]–[6] can only find sub-optimal solutions, which might severely impact the chip performance.

Second, the power budget might change very fast and suddenly due to IR drop (resistive voltage drop), power supply noise, or change in external power supply. As so, a good power budgeting algorithm must be running fast enough (low run time overhead) to track these power budget changes. However, most existing power management algorithms [4]–[6] take unacceptably long time (often in the range of millions of clock cycles), leading to unknown chip behavior or poor performance.

To address two aforementioned challenges, a novel power allocation approach, Optimal Power Allocation using Dynamic programming (OPAD), is proposed in this paper. Essentially, the performance optimization under power budget problem is first formulated and it is solved by OPAD featuring a novel dynamic programming network with a linear time complexity. We summarize our main contributions as follows.

1) OPAD can generate globally optimal power allocation solution under a given power budget.
2) The run time of OPAD is much lower (e.g., a few to dozens of cycles) than that of any other known power allocation

This work was supported in part by National Science Foundation (NSF) under grant no. CNS-1126688 and NSF China under grant no. 61228501.

algorithms. One big advantage of OPAD is that it can achieve much better performance when the power budget undergoes a rapid change.
3) OPAD can be used for many-core systems to control the power consumption of various on-chip resources at a finer grain by hierarchically forming frequency domains, which is scalable in terms of network size.
4) Extensive experimental results confirm the superiority of OPAD over other power allocation schemes in terms of performance.

The paper is organized as follows. Section II reviews the related work. Section III introduces the performance-power model followed by the problem definition in Section IV. The description of the DPN based power allocation scheme is detailed in Section V. Section VI provides the experimental results and analysis of the proposed approach. Finally, Section VII concludes the paper.

II. Related work

As power is becoming one of the major barriers for many-core systems, optimizing performance under given power budget has become an increasingly important problem, which brings the so-called power budgeting problem [3] up to the front. In the literature, online power budgeting can be achieved by applying various circuit and/or architectural techniques, including frequency/voltage scaling [3], [7], or power gating to control the power consumed by the resources [6].

Frequency/voltage scaling [6], [8] can be performed either at the chip or at the core level. Chip-wide frequency scaling [8] treats the frequencies of the on-chip cores as one variable, and these frequencies are scaled up/down altogether with the same scaling factor. This rigid scheme, although simple, tends to result in quite poor chip performance.

The problem of performance optimization under power budget is solved either using some sort of heuristics [3], [4], [6], [7], or formulating it as a linear programming program [7]. Heuristics-based approaches do not generate optimal solutions and their performance can degrade severely with the increase of the number of resources to be controlled. The linear programming-based approach might take long time to find the solution.

III. Models and notations

This section defines the performance and power models which help in formulating the problem.

A. Architectural model

We are targeting an NoC-based many-core system, where each tile consists of a processor, a router and an L2 cache bank. A tile can just have an L2 cache and a router, or a processor and a router (without L2 cache). In the case of heterogeneous systems, a tile can be any resources connecting to the NoC. Tiles can be grouped virtually into regions where each tile inside one region runs at the same frequency.

B. Performance model

Frequency scaling is used to balance between power consumption and performance. Suppose there are a total of N regions in the chip whose respective frequencies are $f_1, ..., f_N$. Each of these regions can be either a single tile or a cluster of multiple tiles. Performance given in execution cycles is modeled in terms of the frequencies of the regions/tiles, as follows,

$$Cycle = g_{cycle}\,(f_1,\, \ldots, f_N) \tag{1}$$

The NoC of a many-core system is hierarchically partitioned into N regions to trade power control granularity for a more manageable solution space size. The tile(s) inside one region has (have) the same frequency. The total number of regions now is the same as the number of frequency variables in Eqn. 1.

Regression models [9] are used to find the g_{Cycle} in Eqn. 1 through curve fitting. Our experiment has indicated that the following model in Eqn. 2 can result in less errors, and this model will be validated in the experimental part in Section VI.B.

$$\ln Cycle = a_0 + \sum_{i=1}^{N} a_i \cdot \sqrt{f_i} \tag{2}$$

where a_i is the regression coefficient w.r.t. f_i. At the k-th time interval, the frequencies of N regions are set randomly with a vector $\overrightarrow{f_k} =< f_1, ..., f_N >$ and the cycles $Cycle_k$ is measured. With K time intervals, the training data $\{< \overrightarrow{f_k}, Cycle_k >, k = 1, ..., K\}$ are collected. A linear regression model with the maximum likelihood estimator [9] will find the coefficients (a_i's) with the training data set.

C. Power model

Assuming all the cores are operating in the same voltage level (dynamic frequency scaling only), the total dynamic power of a many-core chip of interest can be determined as follows:

$$\sum_{i=1}^{N} \alpha_i \cdot C_i \cdot f_i \cdot V^2 = \sum_{i=1}^{N} b_i \cdot f_i \tag{3}$$

where α_i is the switching activity, C_i is the effective capacitance, V is the voltage, $b_i \equiv \alpha_i \cdot C_i \cdot V^2$.

If, besides the core frequencies, the core voltages can also be adjusted (dynamic voltage and frequency scaling), the dynamic power can be calculated as

$$\sum_{i=1}^{N} \alpha_i \cdot C_i \cdot {f_i}^3 / K^2 = \sum_{i=1}^{N} d_i {f_i}^3 \tag{4}$$

where K is a constant. Similarly, $d_i \equiv \alpha_i \cdot C_i / K^2$.

IV. Problem Definition

Following the performance and the power models described above, the power budgeting problem aims to minimize the overall execution time under the input power budget. It is formulated as

$$\min Cycle = g_{cycle}\,(f_1,\,\ldots,f_N) \tag{5}$$

subject to

$$\sum_{i=1}^{N} b_i \cdot f_i \le P \tag{6}$$

for each

$$f_i \in \{F_1,\,\ldots,F_M\} \tag{7}$$

The power model in Eqn. 6 is determined by applying either Eqn. 3 or 4; P is the input power budget, which could be a function of time t. Eqn. 7 specifies a discrete set of frequency values that each frequency variable f_i can take.

The problem can be viewed as, given N tile regions whose frequencies can take $\{F_1,\,\ldots,F_M\}$ and the power budget P, find the assignment of the frequencies of the regions, so as to minimize the execution cycles modeled in Eqn. 2.

Dropping the ln notation in Eqn. 2 and converting the min operator to max in the objective equation, the above problem definition has the same form as a bounded knapsack problem [10], which is NP-hard. On the other hand, the power budget P might also change fast, which requires the problem be solved with low run time and hardware overhead. Heuristic based approaches, like [3], [4] might fail to find a good solution with low execution cycles, due to the fact that the number of possible combinations of assigning N variables grows exponentially. Linear programming [7] or exhaustive search take unacceptable long time to find the optimal solution.

The above challenge pushes us to develop a new algorithm to find optimal solutions with high parallelism in nature and low run time and hardware overhead, as in next section.

V. Dynamic Programming Based Optimal Online Power Allocation

A. *Overview of the OPAD algorithm*

To solve the problem defined in Section IV optimally, the OPAD algorithm transforms the problem into a dynamic programming network based solver as shown in Fig. 1. A regression algorithm is applied either online or offline to obtain the performance model (section III.B), which relates the frequencies of relevant regions to performance (measured as cycles) in Eqn. 1. The regression model according to Eqn. 2 can be found in a curve fitting manner.

Inspired by the dynamic programming approach to solve the knapsack problem, the above problem can also be solved in polynomial time as follows.

Since the logarithmic function in the objective (Eqn. 2) is monotonic, minimizing Eqn. 2 is equivalent to minimizing $\sum_{i=1}^{N} a_i \cdot \sqrt{f_i}$. Let $C_{i,p}$ denote the minimum cycles of assigning

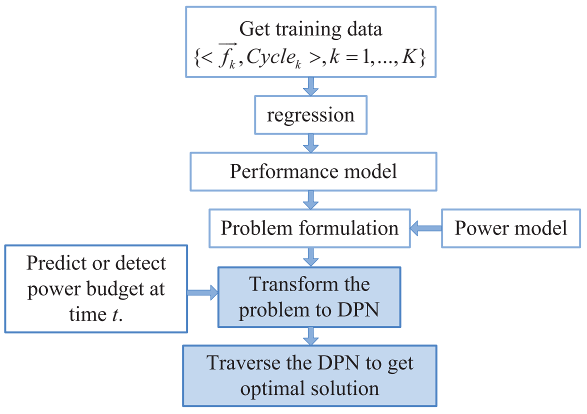

Get training data
$\{<\overrightarrow{f_k}, Cycle_k>, k=1,\ldots,K\}$
regression
Performance model
Problem formulation
Power model
Predict or detect power budget at time t.
Transform the problem to DPN
Traverse the DPN to get optimal solution

Fig. 1. The OPAD system block diagram.

$\{f_1, \ldots, f_i\}$ with $\sum_{j=1}^{i} b_j \cdot f_j \le p$ and $0 \le p \le P$. Thus, for each f_i,

- if $\sum_{j=1}^{i} b_j \cdot f_j > p$, $C_{i,p} = C_{i-1,p}$.
- Otherwise, $C_{i,p} = \min\{C_{i-1,p}, C_{i-1,p-b_i f_i} + a_i\sqrt{f_i}\}|_{f_i=F_j}$.

In this way, $C_{N,P}$ is the minimum cycles by assigning the N variables. The dynamic programming algorithm can be summarized in Algorithm 1.

Algorithm 1: Dynamic Programming Based Frequency Assignment

```
Input: $a_i$, $b_i$: the coefficients in Eqns. 2 and 3
Output: $C_{i,p}$: the minimum cycles given the power
        budget $\le p$.
Function: Find the minimum cycles.
begin
    Initialize all the $C_{i,p}$ to be 0;
    for each $f_i$ do                          /* $i = 1, ..., N$ */
        for each $F_j$, do                     /* $\{F_1, \ldots, F_M\}$ */
            for each $p \le P$ do
                if $\sum_{j=1}^{i} b_j \cdot f_j > p$ then
                    $C_{i,p} = C_{i-1,p}$;
                else
                    $C_{i,p} = \min\{C_{i-1,p}, C_{i-1,p-b_i f_i} +$
                    $a_i\sqrt{f_i}\}|_{f_i=F_j}$;
            end
        end
    end
end
```

To accelerate the computation, a multi-stage dynamic programming network (DPN) is designed to solve the problem with linear time complexity. It is first constructed by mapping the terms in the constraints (Eqn. 3) and objective (Eqn. 2) equations to the weights of vertices or edges. Then the DPN is traversed to find a minimum weight path corresponding to the optimal solution.

In DPN construction, each vertex represents a different power budget. An edge exists between two vertices in adjacent stages if the power consumption of assigning the frequency equals to the difference in the power budgets of the two vertices.

In DPN traversal, each edge is assigned a weight by the term in Eqn. 2, *i.e.*, minimum cycles by assigning frequency to a variable. Thus, finding the minimum weight path is equivalent to an optimal frequency assignment. The traversal can be done in parallel. Each vertex at current stage selects the edge such that, the sum of the edge weight and the minimum cycles achieved by the later stage is minimum. This sum is transmitted back to vertices in the previous stage. In this manner, after reaching the source, the minimum weight path is found in linear time.

In the following, the transformation of the problem to a DPN (which runs Algorithm 1 in parallel) is introduced first, followed by the two operations in DPN which correspond to finding the optimal solution.

B. The definition of the dynamic programming network

The dynamic programming algorithm (Algorithm 1) can be mapped to a DPN which finds solutions in parallel. Fig. 2 shows the transformation of the problem into a dynamic programming network, *i.e.*, construction of the DPN. Once the DPN is constructed, it only needs minor update to remove some vertices and edges according to current power budget.

Definition 1. *Dynamic programming network. A dynamic programming network is denoted as a graph $DPN(V,E)$, with V and E represent the sets of vertices and edges, respectively.*

- Each vertex is assigned with two properties, $Cycle$ and P, where $Cycle\left(v_{i,j}\right)$ denotes the minimum cycles given the power budget $P\left(v_{i,j}\right)$ equal to j.
- The weight of an edge $e_{i,j,k} = (v_{i,j}, v_{i+1,k})$, represented as $w_{i,j,k}$. $w_{i,j,k} = a_i\sqrt{f_i}|_{f_i=(j-k)/b_i}$ equals to the corresponding term in Eqn. 2.

Let N denote the number of frequency variables, M be the number of values that can be taken by each frequency variable, and P be the power budget given at time t. Of the total of $P\times(N+1)$ vertices in a DPN, the network is organized as $N+1$ stages (as there are N frequency variables), and each stage has P vertices (corresponding to the available power budget levels). Two dummy vertices, S and D, are added before stage 1 and after stage $N+1$, respectively, as in Fig. 2.

Two steps are involved for the transformation to DPN, *i.e.*, DPN construction and traversal.

C. DPN construction

The DPN construction has three steps as follows.

- Let
$$P(v_{i,j}) = j \tag{8}$$
- An edge $e_{i,j,k}$ is added between vertices $\left(v_{i,j}, v_{i+1,k}\right)$, if
$$k + b_i \cdot f_i|_{f_i=F_l} = j, \text{for } l \in \{1,...,M\} \tag{9}$$
where b_i is defined in Eqn. 3.

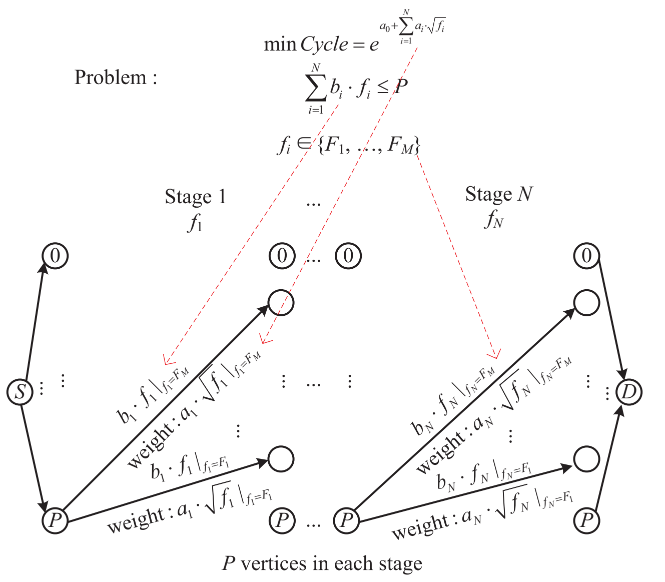

Problem :
$\min Cycle = e^{a_0+\sum_{i=1}^{N} a_i\cdot\sqrt{f_i}}$
$\sum_{i=1}^{N} b_i \cdot f_i \le P$
$f_i \in \{F_1, ..., F_M\}$
Stage 1 f_1
...
Stage N f_N
0
S
P
D
$b_1 \cdot f_1|_{f_1=F_M}$
weight : $a_1 \cdot \sqrt{f_1}|_{f_1=F_M}$
$b_1 \cdot f_1|_{f_1=F_1}$
weight : $a_1 \cdot \sqrt{f_1}|_{f_1=F_1}$
$b_N \cdot f_N|_{f_N=F_M}$
weight : $a_N \cdot \sqrt{f_N}|_{f_N=F_M}$
$b_N \cdot f_N|_{f_N=F_1}$
weight : $a_N \cdot \sqrt{f_N}|_{f_N=F_1}$
P vertices in each stage

Fig. 2. Transformation of the optimization problem to a dynamic programming network. In total, there are $P\times(N+1)$ vertices. In the first step, each vertex represents the power budget after assigning a frequency variable in its previous stage. In the second step, a minimal weight path corresponds to the optimal solution is found by dynamic programming.

- Repeat the above steps to connect N stages of the vertices.

Note that f_m with $1 \le m < i$ (of the stags proceeding stage i) are assigned a value at stage i, while f_n with $i < n \le N$ (stages after stage i) are yet to be assigned. Eqn. 9 states that an edge $e_{i,j,k}$ is added between two vertices $v_{i,j}$ and $v_{i+1,k}$ located at two adjacent stages if $k + b_i \cdot f_i|_{f_i=F_l} = j$, for some $l \in \{1,...,M\}$, which means, there is a path between two adjacent vertices if the difference in their power budgets equals to the power consumption by assigning a frequency F_l to the variable with $l \in \{1,...,M\}$.

D. DPN traversal

The DPN traversal has two steps.

- The edge $e_{i,j,k}$ is assigned with weight
$$w_{i,j,k} = a_i\sqrt{f_i}|_{f_i=(j-k)/b_i} \tag{10}$$
where a_i and b_i are defined in Eqns. 2 and 3.
- Find a minimum weight path from S to D, which corresponds to the optimal solution of the problem.

The weight of each edge equals to the corresponding term in Eqn. 2 as shown in Fig. 2. When the power budget P changes, some vertices $v_{i,j}$ and the corresponding edges are removed if the power budget of $v_{i,j}$ (*i.e.*, j) exceeds P. To find the minimum weight path under the power constraint in linear time, the following dynamic programming equations can be calculated within N iterations [11], [12] following a backward moving procedure (from D back to S). At each stage, each vertex calculates the following value, a modified version of the Bellman equations [11], [12].

$$Cycle_{MIN}(v_{i,j}) = \min_{\forall v_{i+1,k}, \text{an edge } e_{i,j,k} \text{ exists between}(v_{i,j}, v_{i+1,k})} \{w_{i,j,k} + Cycle_{MIN}(v_{i+1,k})\} \tag{11}$$

where $Cycle_{MIN}(D) = 0$, and $w_{S,1,k} = w_{N,D,k} = 0$, with $k \in [1, P]$, *i.e.*, the edges connecting S to the vertices in the first stage and the vertices in stage N to D have a weight of 0.

Thus, the calculation of Eqn. 11 will become to find the minimum weight path (optimal solution) under the power constraint in N steps as follows,

$$\begin{aligned} Cycle^*_{MIN}(PATH_{S,D}) &= \min_{e_{i,j,k} \in PATH_{S,D}} \{ \sum_{i=1,j=1,k=1}^{i=N,j=P,k=P} w_{i,j,k} \} \\ &= \min_{f_i \in \{F_1,\ldots,F_M\}} \sum_{i=1}^{N} a_i \sqrt{f_i} \end{aligned} \tag{12}$$

where $PATH_{S,D}$ is the set of all the paths from S to D. The optimal path from $v_{i,j}$ to $v_{i+1,\mu}$ at each vertex $v_{i,j}$ (corresponding to the assignment of each frequency variable f_i by $(j-\mu)/b_i$) along the minimum weight path can be obtained as follows:

$$v_{i+1,\mu} = \arg\min_{\forall v_{i+1,k}, \text{an edge } e_{i,j,k} \text{ exists between}(v_{i,j}, v_{i+1,k})} \{w_{i,j,k} + Cycle^*_{MIN}(v_{i+1,k})\} \tag{13}$$

where the optimal assignment of f_i is $F_l = (j-\mu)/b_i$. In this way, the optimal assignment of each frequency variable is found and the system can be tuned to run under this frequency configuration.

The pseudo-code in Algorithm 2 shows the traversal of the DPN to find the minimum weight path from S to D. Both DPN construction and traversal take N iterations. Each iteration only involves *add* and *compare* operations which could be done in one cycle. Thus, the run time is $2N$ cycles.

```
Algorithm 2: FindMinWeightPath
Input: $e_{i,j,k}$: the weight of each edge, for
       $i, j \in [1, N+1]$ and $k \in [1, P]$.
Output: $Cycle(v_{i,j})$:the minimum cycles of each
        vertex after assigning $f_i$.
Function: Find the minimum weight path &
corresponding to the optimal solution.
begin
    Initialize all the $Cycle(v_{i,j})$ to be $\infty$, except
    $Cycle(D) = 0$;
    for stages $i$ from $N-1$ to $0$ do
        for each vertex $v_{i,j}$ do
            for adjacent vertex $v_{i+1,k}$ do /* an edge
            $e_{i,j,k}$ connecting $v_{i,j}$ and $v_{i+1,k}$ */
                if
                $Cycle(v_{i+1,j}) + w_{i,j,k} < Cycle(v_{i,j})$
                then
                    $Cycle(v_{i,j}) =$
                    $Cycle(v_{i+1,j}) + w_{i,j,k}$;
                    $f_i = (j-k)/b_i$;
            end
        end
    end
end
```

TABLE I. PARAMETERS USED IN THE SIMULATION

Number of processors	64 (MIPS ISA 32 compatible)
Fetch/Decode/Commit size	4 / 4 / 4
ROB size	64
L1 D cache (private)	16KB, 2-way, 32B line, 2 cycles, 2 ports, dual tags
L1 I cache (private)	32KB, 2-way, 64B line, 2 cycle
L2 cache (shared) MESI protocol	64KB slice/node, 64B line, 6 cycles, 2 ports
Main memory size	2GB
On-chip network parameters	
NoC flit size	72-bit
Data packet size	5 flits
Meta packet size	1 flit
NoC latency	router 2 cycles, link 1 cycle
NoC VC number	4
NoC buffer	5×12 flits

TABLE II. BENCHMARKS

PARSEC	streamcluster, swaptions, ferret, fluidanimate, blackscholes, freqmine, dedup, canneal, vips
SPLASH-2	barnes, raytrace

VI. EXPERIMENTAL RESULTS

A. Experimental setup

Experiments are performed using an in-house developed event-driven many-core simulator [13], [14]. Table I lists the configuration of the many-core simulator. The Orion 2.0 power library [15] is admitted into our simulator. Evaluation is performed over a suite of benchmarks: 9 benchmarks in PARSEC [16] and all the benchmarks in SPLASH-2 [17]. These benchmarks are cross-compiled into MIPS-compatible binaries. For the sake of space, only two representatives from SPLASH-2, barnes and raytrace, are listed in Table II.

In all the experiments, we select a 8×8 2D mesh as the underline NoC topology. Four regions are formed in the NoC, where all the tiles within one region run at the same frequency. We compare the performance of the proposed OPAD against three other best known schemes: (1) PGCapping [6], where both the number and the frequency of tiles can be adjusted, (2) PEPON [3], where the frequency of each processor and last-level cache bank can be adjusted, and (3) DPPC [7], a linear programming based approach.

In what follows, we will present the verification of regression model of the performance vs. frequency first. Next the performance of the proposed OPAD is compared against that of related approaches. In the end, the hardware cost and overhead of the DP-based optimization are analyzed.

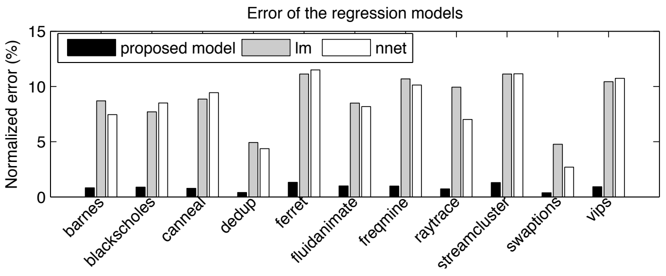


Fig. 3. Comparison of the three models, linear regression (lm), neural network (NNet) and the proposed model as in Eqn. 2. The NoC is partitioned into 4 regions with each region has a block size of 2×8. All the errors (NRMSE) are normalized to the mean value of the cycles.

B. Precision of the performance model

Suitable performance model plays an important role in the problem formulation. To justify the accuracy of the performance model in Eqn. 2, we compare it against two other approaches, the linear regression model (lm), and neural network model (nnet). The metric used here is the normalized root square mean squared error (NRMSE), which is defined in Eqn. 14.

$$NRMSE = \sqrt{\frac{\sum_{t=1}^{K} \left(\widehat{Cycle_t} - Cycle_t\right)^2}{K}} / (Cycle_{\max} - Cycle_{\min}) \tag{14}$$

where $Cycle_t$ are execution cycles, $\widehat{Cycle_t}$ are the cycles by regression, K is the number of total training data, $Cycle_{max}$ and $Cycle_{min}$ are the respective maximum and minimum execution times when the training data are applied.

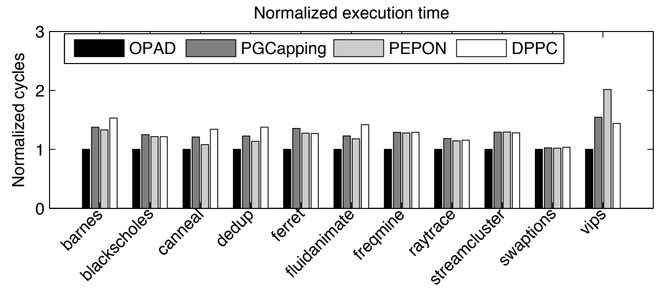
Fig. 3 shows the errors of the three regression models, where the proposed regression model has shown significantly smaller error than the other two.

The parameters might need to be updated as time goes by. Thus, in the experiments, the model parameters in Eqn. 2 are updated every 10M cycles similarly as in Section III.B. Note that the update of the model parameters is independent of the DPN calculation. Once the model is updated, the edge weights will also be updated.

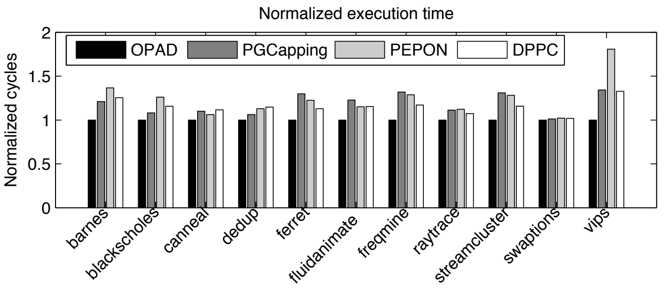
C. Comparison of the power allocation methods

In this set of experiments, a single application is run each time. OPAD is compared against PGCapping, PEPON, and DPPC, and Fig. 4 shows the performance of the four methods when the power budget is high, 150W as in (a), and low, 90W as in (b). From Fig. 4 one can see that OPAD has the lowest execution time under both high and low power budgets. When the power budget is high, OPAD records an average of 26 %, 28 % and 30 % less time that of PGCapping, PEPON, and DPPC , respectively. When the power budget becomes low, on average, OPAD needs 19 %, 25 % and 16% less execution time than that of PGCapping, PEPON, and DPPC, respectively.

To verify the real time power adaptiveness of the four methods, Fig. 5 compares the four methods with a varying power budget. The input power budget in Fig. 5 has two



(a)



(b)

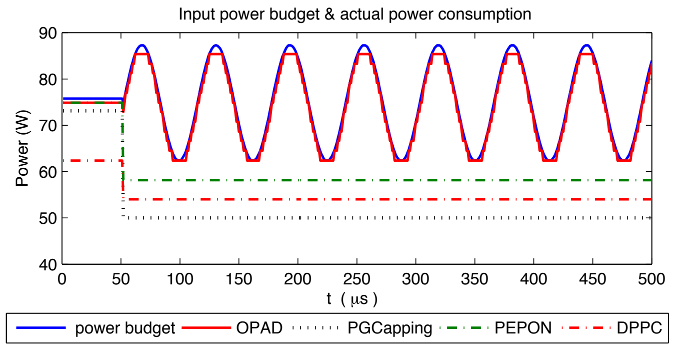
Fig. 4. Execution time of the four approaches under the power budget of (a) 150W and (b) 90 W.

phases: a fast changing phase that the power budget varies rapidly, in the range of microseconds, and a slow changing phase that the power budget varies slowly, in the range of milliseconds. The closer the actual power consumption is to the input power budget, the less the energy loss, and thus the better result. Using this criteria, from Fig. 5 (a), one can see that OPAD outperforms all the other three methods, especially when the power changes rapidly. This is due to low runtime overhead of OPAD, in the level of a few dozen cycles, while the other three methods need a few to ten or even hundred million cycles. With such long run time overhead, the three methods fail to track the varying power budget, which leads to high energy loss. Fig. 5 (b) shows the power consumption when the input power budget varies at a slow pace. Compared to Fig. 5 (a), the matching of the power consumption to the input power budget of PGCapping, PEPON, DDPC is improved. Yet, the matching of these three methods is still poor compared to OPAD. For both cases in Fig. 5 (a) and (b), OPAD has the best match of power consumption with the input budget, resulting in the least energy loss.

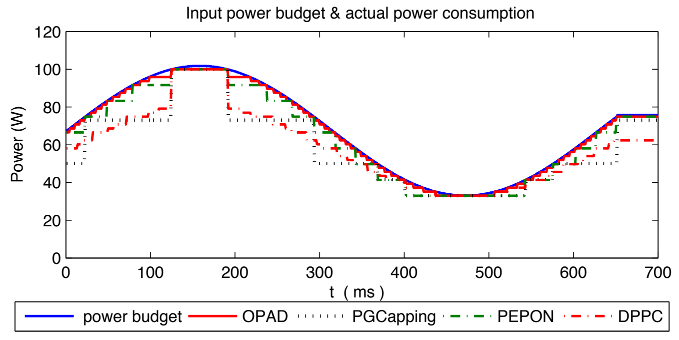
From the above experiments, OPAD can have lower execution time given the same input power budget compared to three state-of-the-art methods, PGCapping, PEPON, and DPPC. On the other hand, as OPAD has much lower overhead in power allocation, it is a beneficial method for many-core systems in which the power budget varies rapidly. Power consumption of OPAD matches the input budget much better than the other three methods, resulting in less energy loss. Thus, OPAD is suitable for online adaptive power allocation.

D. Hardware cost and runtime overhead

The hardware cost of the proposed OPAD is mainly due to the nodes in the dynamic programming networks. Each node operation includes a 16-bit comparator and an adder. Each



(a)



(b)

Fig. 5. Input power budget vs. actual power consumption under different power budget variation rates. The power budget varies at the scale of (a) microsecond and (b) millisecond.

node has an area of 121 μm^2 and consumes 20 μW of power (assuming switching activity of 0.5) using Synopsys Design Compiler under 65nm TSMC library. As there are a total of $P \times (N + 1)$ vertices in DPN, for a system with 16 regions and P in Eqn. 3 normalized to 10, the whole DPN area can be 20570 μm^2 and consumes 3.4 mW of power. For reference, a single 5×5 router with 2 virtual channels, flit size of 75 bits, and 4 flit-depth FIFO has an area of 145890 μm^2 and consumes 8 mW of power. That is, the total area and power consumption of the DPN is 14 % and 42 %, respectively, of a single router.

The total run time of the OPAD to allocate power online is $2N$ cycles, where N is the region number (number of frequency variables). For an 8-region partition, only 16 cycles is needed. As a comparison, PGCapping, PEPON, and DPPC each typically takes about $10 \sim 100$ M cycles, which is 6 to 7 orders of magnitude higher than that of OPAD. In OPAD, the regression model takes 10^5 cycles to compute. However, this is infrequently invoked; it only occurs at the initialization or at update interval for model error correction (typically, every 10 million cycles).

VII. Conclusion

In this paper, an optimal adaptive power allocation method was proposed to optimize performance under a given power budget. The proposed method has two major steps. First, a performance-power model is created either online or offline that relates the performance/power to the frequency variables of various on-chip resources. Second, a dynamic programming network is used to solve the power allocation problem in an optimal way. The proposed method was compared against three state-of-the-art power management methods, *i.e.*, PGCapping, PEPON, and DPPC via extensive experiments. On average, the proposed method can reduce as much as 30 % execution time over the other three methods. One big advantage of the proposed method is its extremely low run-time overhead and hardware cost, making it particularly suitable for power adaptation in many-core systems with rapidly changing power budget.

References

[1] S. Borkar, “Thousand core chips: a technology perspective,” in *Proc. Design Automation Conf*, pp. 746–749, ACM, 2007.

[2] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proc. Int’l Symp. Computer Architecture*, pp. 365–376, IEEE, 2011.

[3] A. Sharifi, A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das, “PEPON: performance-aware hierarchical power budgeting for NoC based multicores,” in *Proc. Int’l Conf. Parallel Architectures and Compilation Techniques*, pp. 65–74, ACM, 2012.

[4] 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,” *Proceedings of the RESoLVE*, 2012.

[5] S. Reda, R. Cochran, and A. Coskun, “Adaptive power capping for servers with multi-threaded workloads,” *IEEE Micro*, vol. 32, no. 5, pp. 64–75, 2012.

[6] K. Ma and X. Wang, “PGCapping: exploiting power gating for power capping and core lifetime balancing in CMPs,” in *Proc. Int’l Conf. Parallel Architectures and Compilation Techniques*, pp. 13–22, ACM, 2012.

[7] K. Ma, X. Wang, and Y. Wang, “DPPC: dynamic power partitioning and control for improved chip multiprocessor performance,” *IEEE Trans. Computers, in press*, 2013.

[8] J. Li and J. F. Martinez, “Dynamic power-performance adaptation of parallel computation on chip multiprocessors,” in *Proc. Int’l Symp. High-Performance Computer Architecture*, pp. 77–87, IEEE, 2006.

[9] C. M. Bishop, *Pattern recognition and machine learning*. Springer New York, 2006.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.

[11] T. Mak, P. Cheung, K. Lam, and W. Luk, “Adaptive routing in network-on-chips using a dynamic-programming network,” *IEEE Trans. Industrial Electronics*, vol. 58, no. 8, pp. 3701–3716, 2011.

[12] D. P. Bertsekas, *Dynamic programming and optimal control*, vol. 1. Athena Scientific Belmont, 1995.

[13] X. Wang, T. Mak, M. Yang, Y. Jiang, M. Daneshtalab, and M. Palesi, “On self-tuning networks-on-chip for dynamic network-flow dominance adaptation,” in *ACM/IEEE Int’l Symp. Networks-on-Chip, in press*, 2013.

[14] J. Xue, A. Garg, B. Ciftcioglu, J. Hu, S. Wang, I. Savidis, M. Jain, R. Berman, P. Liu, M. Huang, H. Wu, E. G. Friedman, G. Wicks, and D. Moore, “An intra-chip free-space optical interconnect,” in *Proc. Int’l Symp. Computer architecture*, pp. 94–105, ACM, 2010.

[15] K. Samadi, A. Kahng, B. Li, and L. S. Peh, “Orion 2.0: a fast and accurate NoC power and area model for early-stage design space exploration,” in *Proc. Design, Automation and Test in Europe Conf. and Exhibition*, pp. 423 – 428, IEEE, 2009.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *Proc. Int’l Conf. Parallel Architectures and Compilation Techniques*, pp. 72–81, ACM, 2008.

[17] J. P. Singh, W. D. Weber, and A. Gupta, “SPLASH: stanford parallel applications for shared-memory,” *ACM SIGARCH Computer Architecture News*, pp. 5–44, 1992.