Coordinated Power-Performance Optimization in Manycores

Hiroshi Sasaki Satoshi Imamura Koji Inoue
Kyushu University
Fukuoka, Japan
{sasaki,s-imamura}@soc.ait.kyushu-u.ac.jp, inoue@ait.kyushu-u.ac.jp

***Abstract*—Optimizing the performance in multiprogrammed environments, especially for workloads composed of multi-threaded programs is a desired feature of runtime management system in future manycore processors. At the same time, power capping capability is required in order to improve the reliability of microprocessor chips while reducing the costs of power supply and thermal budgeting. This paper presents a sophisticated runtime coordinated power-performance management system called C-3PO, which optimizes the performance of manycore processors under a power constraint by controlling two software knobs: thread packing, and dynamic voltage and frequency scaling (DVFS). The proposed solution distributes the power budget to each program by controlling the workload threads to be executed with appropriate number of cores and operating frequency. The power budget is distributed carefully in different forms (number of allocated cores or operating frequency) depending on the power-performance characteristics of the workload so that each program can effectively convert the power into performance. The proposed system is based on a heuristic algorithm which relies on runtime prediction of power and performance via hardware performance monitoring units. Empirical results on a 64-core platform show that C-3PO well outperforms traditional counterparts across various PARSEC workload mixes.**

***Index Terms*—DVFS, manycore processor, power budget allocation, power-performance optimization, runtime system, scalability, thread packing.**

I. Introduction

Managing power and energy consumption has become a first-order concern in modern computer systems. With future manycore processors being predicted to be power limited [5], the ability to cap the peak power consumption is critical as well as optimizing performance. In order to achieve this goal, the power budget should be effectively transformed into performance while fully making use of it.

Recent work has shown that performance can be optimized under a power consumption constraint by applying thread packing, and dynamic voltage and frequency scaling (DVFS) in tandem [4]. Fig. 1(a) shows an example of such technique applied to a 64-core processor where a *single* multithreaded program is optimized by allocating 36 cores with maximum frequency while keeping the rest of the cores idle with minimum frequency.

When multithreaded programs are *simultaneously* executed, the situation becomes much more complicated. One straight-forward approach is to equally partition the resources (cores *and* power budget) to each program and apply an optimization shown in Fig. 1(a) within each partition. Fig. 1(b) shows such an example with execution of four programs, `App1`, `App2`, `App3` and `App4`, optimized by allocating 16 cores with minimum frequency (or 16-Min), 12-Mid, 16-Max and 8-Max, respectively. Although the performance is *locally* optimized for each partition, this is not always guaranteed to be the *globally* optimal assignment. For example, consider a situation where `App3` and `App4` have more power budget than are sufficient; `App3` is fully utilizing the partition with maximum frequency but there is still surplus power (i.e., not power hungry) and `App4`'s performance cannot be improved by adding extra cores (i.e., poorly scalable). Where on the other hand, `App1` and `App2` have a potential to increase their performance if more power is allowed to be consumed.

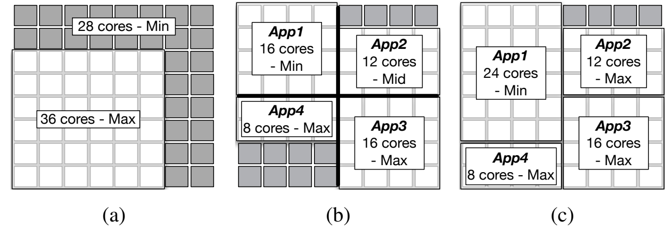


Fig. 1. Optimizing performance under a power cap. (a) A multithreaded program being executed. (b) and (c) Four multithreaded programs simultaneously being executed. Cores running the same program is grouped together.

Fig. 1(c) illustrates an example where the performance is further improved than that of Fig. 1(b) by allowing the surplus power of `App3` and `App4` to be consumed by `App1` and `App2` by increasing the number of cores assigned (`App1`) and by operating at higher frequency (`App2`). The exploration space for such global optimization is multi-dimensional where the allocation of the number of cores and the operating frequency of each program can be varied under two global conditions: sum of the allocated number of cores must be less than or equal to the total number of cores, and the power consumption must satisfy the constraint.

This paper presents the design and implementation of **C-3PO** (coordinated performance-per-power optimization), a runtime management system for manycore processors which attempts to maximize the performance while capping the peak power for multiprogrammed workloads composed of multithreaded programs. There are couple of major challenges associated with this problem. Power-performance characteristics of each program should be dynamically discovered; power consumption must be precisely controlled in order to optimize against power caps; and efficient power capping needs to be performed in a global manner by taking the discovered power-performance characteristics of all the programs into account.

To the best of our knowledge, none of the previous work has tackled this problem.

In order to overcome the challenges, C-3PO introduces a heuristic-based power-performance optimization algorithm which iteratively converges to a better assignment among the two-dimensional trade-off space composed of (1) the number of cores to pack the entire threads and (2) the voltage and frequency levels of active processors of each program. The key idea of the proposed algorithm is to dynamically "salvage" the power budget that is not contributing to performance at runtime, and to "redistribute" it to programs which are expected to effectively convert it into performance in the "right way" (among two knobs), by globally considering the power cap. Additional bonus of C-3PO is that although it is not designed to save power consumption, it is possible to reduce the total energy consumption of the system in many cases by preventing the processor from wasting the power which does not contribute to performance.

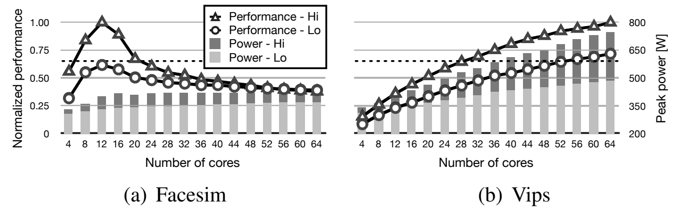
Overall, this paper makes the following major contributions:

- We present a runtime power estimation technique based on processor utilization, number of active cores, and operating frequency. Multivariate linear regression modeling is used to identify the coefficients associated with the model which show coefficient of determination of over 0.93.
- We introduce a utilization-based scalability prediction technique which identifies whether a program is effectively translating the allocated computing resources into performance or not.
- We propose C-3PO, a runtime system for manycore processors executing multiprogrammed multithreaded workloads. C-3PO dynamically estimates the power consumption and then salvages (if possible) and allocates the power budget to each program based on its predicted scalability so as to globally optimize the performance.
- We implement C-3PO on Linux where it is evaluated on an AMD 64-core platform. Experimental results show that C-3PO improves average performance by 21.0% compared to Linux with DVFS for a series of multiprogrammed workloads.

II. Motivation

A. Power-Performance Analysis of Parallel Programs

Multithreaded programs show diverse power-performance characteristics with respect to the number of allocated cores and the operating frequency. Fig. 2 shows performance curves along with its peak power consumption of the ROI (region of interest) execution for two* PARSEC benchmarks [3], `facesim` (Fig. 2(a)) and `vips` (Fig. 2(b)), on a 64-core SMP platform† operated under two processor frequencies (1.4 GHz or "Lo" and 2.6 GHz or "Hi"). The x-axis shows the number of cores allocated to the programs where they create equal or larger number of threads than the available core count (we pass "64" as an argument to programs), and are scheduled to the indicated number of cores (i.e., thread packing [4]). The y-axis on the left hand side shows the normalized performance to the maximum performance (typically, but not necessarily, 64-cores with 2.6 GHz) and the y-axis on the right hand side indicates the peak power consumption during the execution, shown as lines and bars in the figures, respectively. Note that the bottom value for the right hand side of the y-axis is 200 W.



(a) Facesim (b) Vips

Fig. 2. Power-performance characteristics of `facesim` and `vips`.

Prior work [4] has shown that performance can be optimized under a power consumption constraint by selecting the appropriate core count and the processor frequency (e.g., Fig. 1(a)) which can also be seen from Fig. 2. For example, when we assume a power budget of 600 W (shown as a dotted line in the figures), performance can be optimized by allocating 12 cores and executing with 2.6 GHz (or simply 12-Hi) for `facesim` while 36-Hi maximizes performance for `vips`.

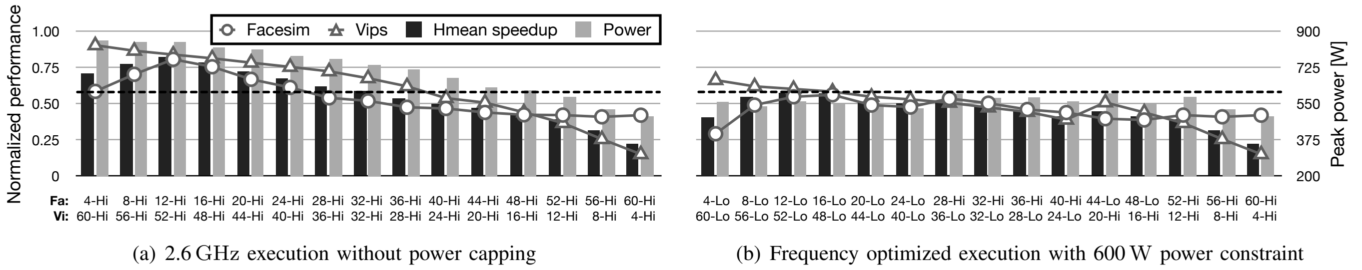
B. Power Budget Distribution in Multiprogrammed Workloads

When executing only a single program, the power budget can of course be totally consumed by that specific program. Therefore, the best performing configuration (number of allocated cores and the processor frequency, or simply assignment) can be obtained by considering its power-performance characteristics such as the ones shown in Fig. 2. However, finding the globally optimal assignment for multiprogrammed workloads is difficult because the power budget can be flexibly distributed to each program, as described in Section I, where it is further explained with examples in the following.

1) Controlling the number of cores and processor frequency: Fig. 3 shows how the performance varies with different number of cores allocated to the two programs shown in Fig. 2 when they are co-scheduled. We assume a 600 W power budget throughout this study which is about 70% of the peak power consumption of the platform. Fig. 3(a) shows the result when both programs are executed with Hi frequency without considering the power constraint and Fig. 3(b) shows the result when the operating frequency of each program is chosen so as to maximize the performance while keeping a power constraint (shown as a dotted line in the figures). The x-axis shows different configurations where the number of cores allocated to `facesim` and `vips` are varied. The number of cores allocated to `facesim` ranges from 4 to 60 in steps of four from left to right (remaining cores are allocated to `vips` while keeping all the 64 cores active). The notations on the x-axis denote the number of cores allocated along with the operating frequency of `facesim` on the top and

*Only two distinct benchmarks are shown due to space limitations.

†Evaluation system consists of four CMPs with each CMP having 16 core multi-chip module processor consisting of two eight core dies. Further details of the experimental environment are described in Section IV.



(a) 2.6 GHz execution without power capping

(b) Frequency optimized execution with 600 W power constraint

Fig. 3. Performance for simultaneous execution of `facesim` and `vips` when number of cores allocated to each program is varied (64 cores active).

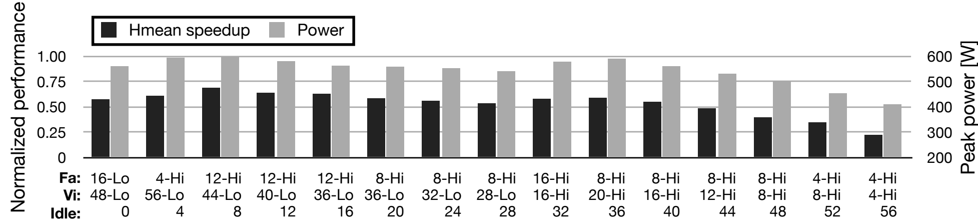


Fig. 4. Performance for simultaneous execution of `facesim` and `vips` when number of active cores is varied.

`vips` at the bottom. The y-axis on both sides are the same as Fig. 2, while additional black bars in the figure show the harmonic mean (hmean) of the per-application speedups with respect to the solo runs [14] of two programs. We evaluate the performance with hmean speedup metric because it is known to capture a notion of both performance and fairness [6]. Note that hmean speedup ranges from 0 to 1 and a higher value indicates higher performance.

When executed with only Hi frequency, most configurations except three configurations at the right end violate the power budget of 600 W as shown in Fig. 3(a). Among these three, the best hmean speedup of 0.39 is achieved with 52-Hi for `facesim` and 12-Hi for `vips`. This is apparently not optimal because allocating more than 12 cores to `facesim` degrades performance as shown in Fig. 2(a). Performance can be improved while keeping the power constraint by utilizing different frequency pairs. As shown in Fig. 3(b), 16-Lo for `facesim` and 48-Lo for `vips` achieves the best hmean speedup of 0.57 with peak power consumption of 552 W. This configuration gives us 46.1% improvement over the best pair which only uses 2.6 GHz (52-Hi for `facesim` and 12-Hi for `vips`). It is also interesting to see that different combinations of processor frequencies achieve the best performance when the number of allocated cores differs. This is because the frequencies for each application have to be chosen in tandem so that the total power consumption satisfies the power constraint.

2) Increasing the processor frequency by introducing idle cores: So far, we have considered to utilize all the available cores, however, there is still room for improvement by reducing the number of active cores while keeping some of them idle. Because idle cores consume minimum power, the processor frequency of other active cores can be increased which leads to improved performance. Fig. 4 shows the hmean speedup and the peak power consumption for the same benchmark pair as Figures 2 and 3, by varying the number of active and idle cores. The x-axis shows different configurations whose notations are similar to Fig. 3, where each pair of bars corresponds to the best configuration among a specific number of active cores. For example, the left most pair shows the result obtained from Fig. 3(b), where all the cores are active (no idle cores), and 16-Lo for `facesim` and 48-Lo for `vips` achieves the best performance. We evaluated all the possible pairs of core count and processor frequency, and the pairs with the highest hmean speedup which do not exceed 600 W are shown in Fig. 4. Compared to the results obtained from Fig. 3(b), performance can be improved when 4, 8, 12, 16, 32 and 36 cores are idle. The best hmean speedup of 0.68 is achieved with 12-Hi for `facesim` and 44-Lo for `vips` while keeping 8 cores idle, which performs 19.3% better performance than 16-Lo for `facesim` and 48-Lo for `vips`. Performance boost comes from increasing the processor frequency of `facesim` execution which shows significant performance improvement when executing with 2.6 GHz at 12 cores as can be seen in Fig. 2(a). This example shows that the performance of multiprogrammed workloads can be globally optimized by trading off active cores for increased processor frequency.

C. Potential of Coordinated Optimization

To summarize this section, we compare the hmean speedup and the peak power consumption of the best configuration, or *Globally-optimal* (12-Hi for `facesim`, 44-Lo for `vips` and 8 cores idle, which corresponds to Fig. 1(c)) to other execution policies in Fig. 5. *Linux-Hi* and *Linux-Lo* in the figure leave the scheduling to the default Linux scheduler where they denote executions with 2.6 GHz and 1.4 GHz, respectively. Additionally, we show *Locally-optimal*, the best statically obtained configuration when the resources are equally partitioned (12-Hi for `facesim`, 32-Lo for `vips`, and 20 cores idle, which corresponds to Fig. 1(b)).

Linux-Hi achieves hmean speedup of 0.68 which is equivalent to *Globally-optimal*, however, its peak power consumption

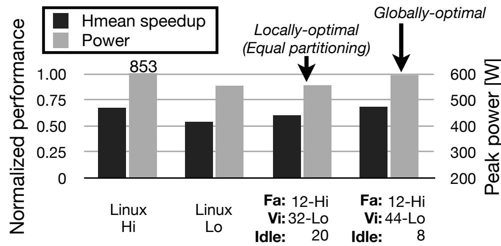


Fig. 5. Performance for simultaneous execution of `facesim` and `vips` with different execution policies.

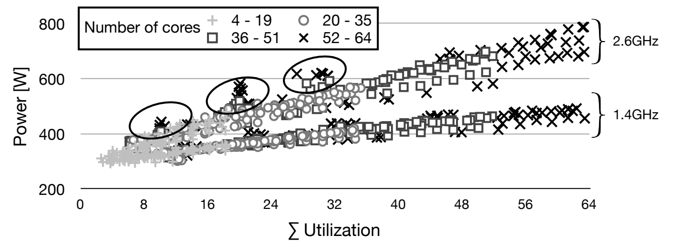


Fig. 6. Power consumption for different CPU utilizations. Results of two levels of CPU frequencies (1.4GHz and 2.6GHz) are plotted in the figure.

of 853 W greatly exceeds the power budget. *Linux-Lo* stays below the budget with peak power consumption of 555 W while achieving hmean speed up of 0.54. *Locally-optimal* achieves hmean speedup of 0.60 with peak power consumption of 558 W which still leaves room for improvement, showing the limit of local optimization.

Examples shown in this section demonstrate the potential of the global optimization policy which allocates the appropriate number of cores and the processor frequency to each program. Because the optimal assignment depends on the workload, their execution phases and the power budget, which might change at runtime, the configuration cannot be statically determined. Therefore, a dynamic optimization technique needs to be established. As we have seen in the example of Fig. 3(a), the key to achieving high performance is to not allocate extra cores to programs which cannot benefit from it (programs with poor scalability). Further performance optimization is possible by considering the power budget and by carefully controlling the two knobs as shown in Figures 3(b) and 4.

III. C-3PO Runtime System

A. Overview

In this section, we introduce the proposed C-3PO (coordinated performance-per-power optimization) runtime system. The goal of the system is to optimize the performance (maximize the hmean speedup) of manycore processors executing multiprogrammed workloads under a power constraint. The system invokes an allocation function every fixed time interval, or epoch, and determines the configuration iteratively over the program execution. The key idea of C-3PO is to "salvage" the power budget that is not contributing to performance and "redistribute" to programs that would benefit from it. In order to accurately salvage and redistribute the power budget, performance and power of the programs need to be estimated at runtime.

B. Online Power Estimation

In order to keep the power consumption within a power budget, C-3PO needs to know the total power consumption of the processor. Although some of the recent processors equip *power counters* for that purpose, we estimate the power consumption from the statistics collected via hardware performance monitoring units (PMUs) as a more general solution. Prior studies have shown that power consumption can be estimated using processor utilization [7, 11] where Ma et al. extend the prior approaches so that DVFS can be considered in the model [15]. They model the total power consumption as $P = \sum_{i=1}^{N} (U_i \times f_i \times W) + C$ where P is the total power consumption, N is the total number of cores, U_i is the utilization, which ranges from 0 to 1, of the i th core ($1 \leq i \leq N$), f_i is the frequency level of core i, W is the coefficient of proportionality of the model and C is the idle power of the system. Note that the static power is captured by C. We refer to this model as the *original model*. We have found that this model is not sufficient to characterize the power consumption as it assumes that W and C are constant regardless of the processor frequency. Therefore, we propose a new model to overcome this issue.

W: Dynamic power consumption of a processor is proportional to the product of the processor frequency (f_i) and the square of the supply voltage (which is captured by W). As the voltage needs to be increased when operating with higher frequency, it is quite intuitive to assume that W also increases for higher frequency. We consider this in the model by having different W_f for each processor frequency.

C: Fig. 6 shows power plots for 1.4 GHz and 2.6 GHz PARSEC executions with various number of allocated cores. The x-axis shows the sum of processor utilization of all processors (maximum is 64 because the evaluation platform has 64 cores) and the y-axis shows the power consumption. We can see a trend that power and utilization have an almost linear relationship as expected from the original model. However, some exceptions can also be seen in the figure where plots inside the three circles drawn are remarkable. We can see a trend here that these plots exist in low utilization area while having a large core count. This cannot be captured by the constant C of the original model. The proposed model introduces a term which indicates the static power of the *active* cores. This makes the predicted power consumption possible to differentiate between single active core with utilization 1.0 from ten cores with utilization 0.1.

The equation of the *proposed model* is shown as follows: $P = \sum_{i=1}^{M} (U_i \times f_i \times W_{f_i} + S_{f_i}) + C$, where M is the total number of active cores ($1 \leq M \leq N$), W_{f_i} is the weight of the processor frequency f_i, and S_{f_i} is the static power of an active core with f_i, and C is the constant power consumed regardless of the activity of the system. P, i, U_i and f_i are the same as the original model. We calculate the coefficients W_{f_i}, and S_{f_i} using multivariate linear regression. In order to utilize this model at runtime, we perform the calculation

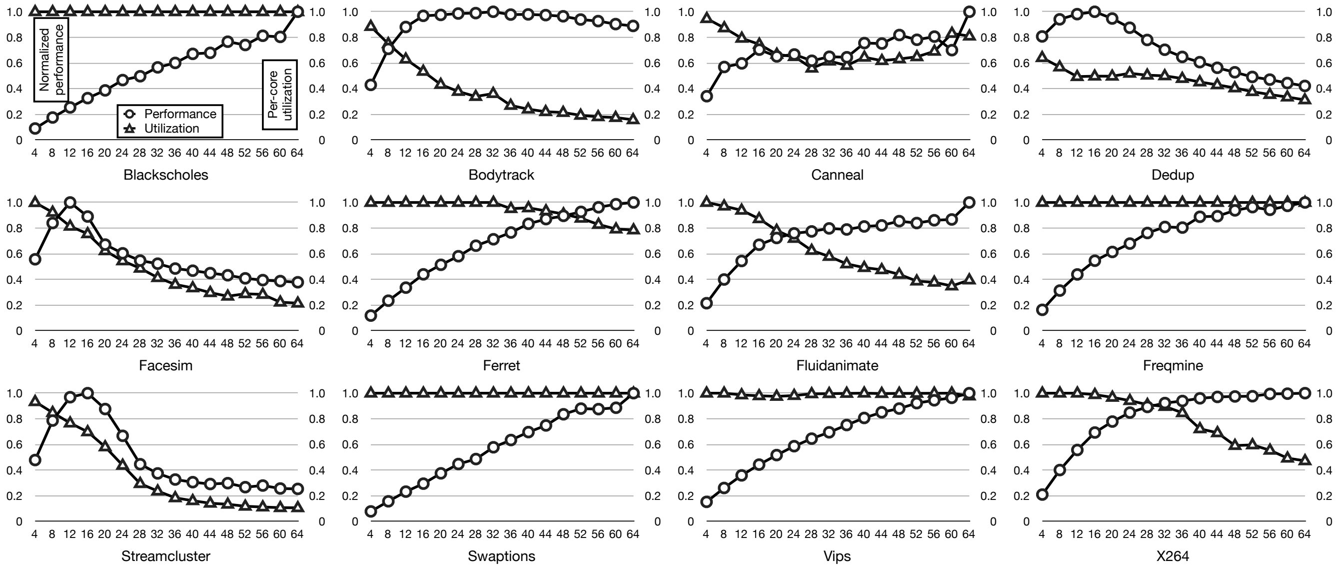


Fig. 7. Normalized performance vs. per-core utilization for PARSEC benchmarks as the number of allocated cores is varied.

statically in advance. The data sets for regression is obtained from a power meter and PMUs by running a set of programs with each operating frequency under different number of active cores (e.g., Fig. 6). C-3PO estimates the power consumption at runtime using this proposed model.

C. Online Scalability Prediction

C-3PO needs to distinguish whether the program is scalable (i.e., benefits from having additional cores) or not at runtime to salvage the power consumption not contributing to performance. It is known that each thread of highly scalable programs which favor additional cores is independent, having less communications between each other. In other words, highly scalable programs can fully utilize the CPU time allocated by the OS. On the other hand, programs which do not scale well have opposite characteristics, meaning that threads are dependent with each other having communications and synchronizations. This in turn makes the cores underutilized by wasting the CPU time allocated to the program. Therefore, we focus on the processor utilization in order to determine whether the program is scalable or not. Fig. 7 supports our idea, which shows the normalized performance to the maximum performance and the per-core processor utilization for the ROI execution of PARSEC benchmarks when different number of cores are allocated. Note that the processor frequency is set to 2.6 GHz and the per-core utilization shown in the figure is the average across the execution.

As we have imagined, highly scalable programs such as `blackscholes`, `ferret`, `freqmine`, `swaptions` and `vips` show almost constant per-core utilization of one (i.e., cores fully utilized) regardless of the number of cores. Other programs show similar trends with each other where per-core utilization remains high when the number of allocated cores is small while the utilization tends to decline at a certain point, dependent on the program. We can see that these programs show good scalability at lower number of cores but the performance saturates or even decreases as the per-core utilization declines. It can be seen from the figure that programs which have modest per-core utilization at greater number of cores (`canneal`, `fluidanimate` and `x264`) show better scalability than other programs whose per-core utilization decreases rapidly as the core count grows (`bodytrack`, `dedup`, `facesim` and `streamcluster`). Although accurate performance prediction is not possible by focusing on per-core utilization, it is clear that there is no benefit to allocate extra cores to a program whose per-core utilization is lower than a certain threshold (e.g., 0.7). Therefore, C-3PO utilizes this statistic as a guideline; it determines the number of cores to allocate to each program so that the per-core utilization does not fall below the threshold.

Focusing on the per-core utilization has two advantages over directly measuring performance related statistics such as IPC (instructions per cycle) or total instruction throughput in the following ways: (1) as utilization itself is a relative value which ranges from 0 to 1, we can set a universal threshold that each program should not fall short, which is very difficult for absolute values like IPC or total instruction throughput, and (2) as we have seen in Fig. 7, per-core utilization shows a monotonic decrease as the number of cores increases (only exception is for `canneal` when core count is greater than 56), which makes C-3PO easy to control the value above a certain threshold by simply decreasing the number of cores until it exceeds the threshold.

D. Allocation Algorithm

We describe the algorithm of C-3PO in this subsection which is based on the online power and performance estimation components described in Sections III-B and III-C, respectively. Algorithm 1 presents an outline of C-3PO. The allocation function is invoked every epoch where the algorithm is designed to iteratively converge to a better allocation. The algorithm uses the following inputs which are obtained from

Algorithm 1: C-3PO algorithm

```
Input: util[i], n_cores[i], freq[i]
Data: surplus_cores, surplus_power, Threshold, Budget
Result: allocation of next epoch
1  // Step-1: Salvage underutilized cores
2  foreach application i do
3      if util[i] < Threshold and n_cores[i] > x then
4          surplus_cores ← surplus_cores + x
5          n_cores[i] ← n_cores[i] − x
6      end
7  end
8  // Step-2: Calculate the surplus power
9  surplus_power ← UpdateSurplusPower()
10 // Step-3: Control the power
11 if surplus_power > 0 then
12     DistributePower()
13 else
14     ReducePower()
15 end
```

Algorithm 2: DistributePower()

```
Input: app_util_descending, app_#cores_ascending
1  // Step-1: Increase the # of allocated cores
2  while app_util_descending ≠ NULL and surplus_power > 0 and
   surplus_cores > x do
3      if app_util_descending > Threshold then
4          surplus_cores ← surplus_cores − x
5          n_cores[i] ← n_cores[i] + x
6          surplus_power ← UpdateSurplusPower()
7          Pop(app_util_descending)
8      end
9  end
10 // Step-2: Increase the operating frequency
11 sort_ascending(app_#cores_ascending)
12 while surplus_power > 0 do
13     app_tmp ← app_#cores_ascending
14     while app_tmp ≠ NULL and surplus_power > 0 do
15         freq[i] ← freq[i] + y
16         surplus_power ← UpdateSurplusPower()
17         Pop(app_tmp)
18     end
19 end
```

Algorithm 3: ReducePower()

```
Input: app_#cores_descending, app_util_ascending
1  // Step-1: Decrease the operating frequency
2  while surplus_power < 0 do
3      app_tmp ← app_#cores_descending
4      while app_tmp ≠ NULL and surplus_power < 0 do
5          freq[i] ← freq[i] − y
6          surplus_power ← UpdateSurplusPower()
7          Pop(app_tmp)
8      end
9  end
10 // Step-2: Decrease the # of allocated cores
11 while app_util_ascending ≠ NULL and surplus_power < 0 do
12     if n_cores[i] > x then
13         surplus_cores ← surplus_cores + x
14         n_cores[i] ← n_cores[i] − x
15     end
16     surplus_power ← UpdateSurplusPower()
17     Pop(app_util_ascending)
18 end
```

PMUs of each scheduled program i: per-core processor utilization ($util[i]$), number of the allocated cores ($n_cores[i]$) and the operating frequency ($freq[i]$). The algorithm effectively works with the following initial state: (1) processor frequency of all the cores is set to the minimum, and (2) processor cores are equally assigned to each program (if the total number of cores cannot be divided by the total number of programs, surplus cores are randomly assigned). Because an optimal assignment for a certain workload might totally be different for another workload, $freq[i]$ and $n_cores[i]$ are initialized when a program terminates or a new program is being launched.

Step-1: First, for each program, C-3PO checks whether the per-core utilization of the previous epoch is below a threshold ($Threshold$) or not. If it is below $Threshold$, the number of cores allocated to the program is reduced by x.

Step-2: Next, by assuming that the characteristics of the programs do not change for the next epoch,* C-3PO calculates the surplus power by subtracting the estimated power consumption from the power budget ($UpdateSurplusPower()$).

Step-3: Finally, if the calculated surplus power is positive, C-3PO distributes the power consumption to programs by allocating additional cores or increasing the operating frequency ($DistributePower()$ shown in Algorithm 2). On the other hand, if the surplus power is negative (i.e., power consumption exceeds the power budget, for example, this might happen if the per-core utilization increases compared to the previous epoch), the scheduler drops the frequency level or decreases the number of allocated cores so that the power consumption stays within the power cap ($ReducePower()$ shown in Algorithm 3). The algorithms of $DistributePower()$ and $ReducePower()$ are described in the following.

Before going into the details of $DistributedPower()$ and $ReducePower()$, we first describe the time overhead associated with increasing/decreasing DVFS levels and allocating/deallocating cores. Although controlling the DVFS levels and the number of allocated cores both affect the power and performance of program executions, the amount of overhead is quite different. Performing DVFS control can be accomplished in tens to hundreds of micro seconds [12] because it is purely a hardware mechanism, where changing the thread to core allocation may incur tens of milliseconds in the worst case because the program threads need to be migrated to different cores when the allocation changes (thread migration is a heavy task for an OS). Therefore, C-3PO conservatively changes the core allocation (number of cores allocated to each program is increased or decreased at a minimum amount at once) while it aggressively changes the processor frequency (DVFS levels can be increased or decreased at any level).

$DistributePower()$ introduces the following new data as inputs: a sorted list of programs in descending order of the per-core utilization ($app_{util_descending}$) and a sorted list of programs in ascending order of the number of allocated cores ($app_{\#core_ascending}$). These lists are being sorted every time

*This assumption does not always hold in practice because (1) per-core utilization of programs whose number of allocated cores is reduced tends to increase and/or (2) execution phase of programs might change at runtime.

the function is invoked. This function is comprised of two steps which first increases the number of allocated cores and next increases the operating frequency. Programs with high per-core utilization are prioritized in receiving additional cores while programs with lower number of cores are prioritized in increasing the DVFS level. This policy is reasonable because (1) we have shown in Fig. 7 that programs with high per-core utilization tend to improve performance with additional cores and (2) programs with fewer number of cores have a chance to improve performance by increasing the processor frequency as can be seen in Fig. 2.

First, it picks program i which is in the front of $app_{util_descending}$ (i.e., program with the highest per-core utilization) and increases the allocated number of cores by x. It updates the surplus power and removes the element from the front of $app_{util_descending}$. This procedure is repeated until the following conditions are valid: (I) $app_{util_descending}$ is not empty, (II) updated $surplus_power$ is positive and (III) updated $surplus_cores$ is positive. If $surplus_power$ is still positive after this first step, it enters the second step.

First, it copies $app_{\#core_ascending}$ to app_{tmp} because it might reuse $app_{\#core_ascending}$ when we need to change the processor frequency aggressively (more than y levels for a single program). The second step is similar to the first step but it favors the program with fewer number of allocated cores and increases its processor frequency level by y. It sorts $app_{\#cores_ascending}$ to earn an up-to-date list because the number of allocated cores might have changed in Step-1. After increasing the frequency level, it updates the surplus power and removes the element from the front of app_{tmp}, and the procedure is repeated while (I) app_{tmp} is not empty and (II) $surplus_power$ is positive. If the $surplus_power$ is still positive when app_{tmp} becomes empty, it repeats the while loop (line 12) again from copying the sorted list to app_{tmp}.

$ReducePower()$ increases the surplus power by operating totally the opposite of $DistributePower()$. Therefore, it introduces $app_{\#core_descending}$ and $app_{util_ascending}$ as new inputs. It first controls the processor frequency because the overhead of DVFS control is much smaller than changing the core allocation. As its flow is similar to $DistributePower()$, detailed description is omitted.

IV. Experimental Setup

A. Hardware Platform

We perform experiments on a quad socket IBM System x3755 M3 server which consists of four 16-core AMD Opteron 6282 SE microprocessors forming a 64-core platform with 96 GB of main memory. Each socket integrates two eight-core dies with a shared 16 MB L3 cache. The processor supports per-core frequency scaling with five different levels (1.4, 1.7, 2.0, 2.3 and 2.6 GHz). Note that Turbo Core technology [1] is disabled to avoid undesirable power and performance variance. The entire system power including the processors, memory and disks is measured using the Yokogawa Electric Corporation's WT1600 digital power meter [19]. The power consumption is measured every 500 ms.

TABLE I
Parameters of the power estimation model

f [GHz]	W_f	S_f	C	Coefficient of determination (R^2)
1.4	2.09	0.10	300	0.94
1.7	2.09	0.32	300	0.93
2.0	2.19	0.47	300	0.94
2.3	2.17	1.04	300	0.93
2.6	2.39	1.03	300	0.94

TABLE II
Benchmark classifications

Type	Benchmarks
Hi	Blackscholes (bl), Ferret (fe), Freqmine (fr), Swaptions (sw), Vips (vi)
Mid	Canneal (ca), Fluidanimate (fl), X264 (x2)
Lo	Bodytrack (bo), Dedup (de), Facesim (fa), Streamcluster (st)

B. C-3PO Implementation

C-3PO is implemented as a user-level runtime software. The evaluation system runs on Linux with kernel 2.6.37 where the `perf-tools` toolset is used to allow periodical access to the PMUs. The processor clock frequency is controlled through the `cpufreq` interface located under `/sys` directory. Standard Linux API (`sched_setaffinity(2)`) is used to control the CPU affinity of processes to bind the programs to specific cores. Detailed parameters of C-3PO are set as follows: epoch length is 1 s considering the overhead of thread migration vs. actual speedup (pros and cons of capturing and responding to the execution phases), and $Threshold$, x (number of cores to change) and y (DVFS levels to change) used in the algorithms are 0.7, 4 and 1, respectively.

TABLE I shows the calculated coefficients and the coefficient of determination (R^2) for the power models of each operating frequency. R^2 values for all the models are greater than 0.93, which indicates that the models fit well with the training values (Fig. 6 shows that of 1.4 GHz and 2.6 GHz). Although the R^2 value of the proposed model is high, there exists some amount of prediction errors in practice. Therefore, we set the target power constraint as $P_{real} = P_{target} + \Delta$, where P_{real} is the 600 W power cap, P_{target} is the target power constraint which is used internally in C-3PO, and Δ is the safety margin [11]. We set Δ as 10% of P_{real} which has shown to ensure power capping in most executions.

C. Workloads

We use PARSEC benchmark suite 2.1 [3] to evaluate C-3PO and its counterparts. The benchmarks are compiled with GCC-4.4.5. We use *native* input sets for all benchmarks except `dedup`. Because the execution time of `dedup` was too short compared to other programs, we use a larger input (the Fedora 16 x86_64 DVD ISO file) so that the execution time becomes comparable to other benchmarks. We use 12 out of 13 benchmarks which are shown in Fig. 7 (`raytrace` is excluded because we could not compile the parallel version of the program in our evaluation platform).

Two types of workloads are used where one is composed of two benchmarks and the other is composed of four

TABLE III
BENCHMARKS FOR WORKLOAD-2 AND WORKLOAD-4

	Benchmarks
Workload-2:	Hi (fe, fr, vi), Mid (fl, x2, ca), Lo (bo, de, st)
Workload-4:	Hi (bl, sw), Mid (fl, x2), Lo (fa, de)

benchmarks. In order to construct workloads with variety of characteristics, we classify the benchmarks into three groups as shown in TABLE II from the utilization curve shown in Fig. 7 and its associated discussions presented in Section III-C. We choose three benchmarks from each type to construct 36 ($= {}_9C_2$) workloads which are composed with two benchmarks (Workload-2). Similarly, two benchmarks from each type are picked to construct 15 ($= {}_6C_4$) workloads composed of four benchmarks (Workload-4). The benchmarks are chosen such that all the 12 benchmarks are at least selected once for either of the workload types. The benchmarks selected are summarized in TABLE III. Note that benchmark names are abbreviated with their first two letters shown within parentheses in TABLE II.

Although C-3PO is designed to effectively handle simultaneous executions of various multithreaded programs*, some programs have long serial execution regions which do not stress the system enough. Therefore, we use Berkeley Lab Checkpoint/Restart (BLCR) tools [9] along with the *hooks* library that comes with PARSEC to set up an environment which enables *checkpoint and restart* of programs to evaluate only the ROI region, by creating a checkpoint at the beginning of ROI for each program which is modified to terminate the execution at the end of ROI. The evaluation methodology is similar to the one originally proposed for SMT job scheduling [17] where a number of studies have been evaluated in a similar fashion [8, 20]. To account for the variety of execution times among programs (from tens to hundreds of seconds), we instantaneously restart a program from the checkpoint (beginning of ROI) when it finishes execution until all of them are executed at least three times to completion.

D. Counterparts

1) Linux with chip-wide DVFS: This policy (*Linux-DVFS*) leaves the thread allocation and scheduling to the Linux default scheduler. It only controls the chip-wide frequency level so that the power consumption does not exceed the power budget. It monitors the processor utilization of the previous epoch and estimates the power consumption from the proposed power estimation model with the safety margin of 10%. By assuming that the utilization value is the same for the next epoch, it selects the highest possible processor frequency among the five DVFS levels. The frequency is controlled every 100 ms.

2) Statically obtained locally- and globally-optimal allocations: Two statically determined allocations, Locally-optimal (or *L-opt*) and Globally-optimal (or *G-opt*) are also evaluated for comparisons. These are introduced in Fig. 5 and correspond

*Note that this includes programs with serial execution phases because the per-core utilization of such phase will be no greater than $1/n_cores$ (n_cores: number of allocated cores) whose core count steadily decreases while giving other programs more resources.

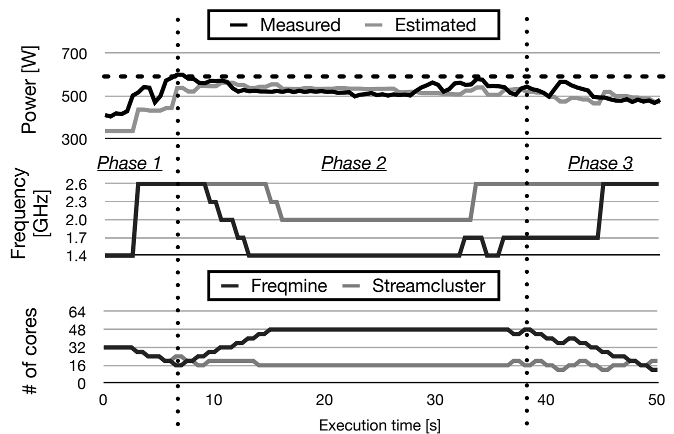


Fig. 8. DVFS and core count control for `freqmine` and `streamcluster` execution with C-3PO.

to the examples shown in Figures 1(b) and 1(c), respectively. In order to obtain these best allocations, we need to execute all the possible configurations and select the best ones, which is unfortunately too time consuming to perform. Therefore, we try to estimate them by calculating the hmean speedups and the power consumption via the proposed model from the individual performance and utilization characteristics of each program (Fig. 7 for *all* the possible frequencies, which can be easily obtained), respectively, and choose the best allocations (*L-opt* and *G-opt*) accordingly for each workload. Note that we assume that the power and performance of each program are not affected by the co-scheduled programs.

V. EVALUATION RESULTS

A. Demonstration of C-3PO Execution Traces

We first demonstrate an illustrative example of the changes in DVFS levels and allocated core count of simultaneous execution for `freqmine` and `streamcluster` in response to the dynamic phase transition occurred for `freqmine`. Fig. 8 illustrates the changes of two knobs at the bottom along with the measured and estimated power consumptions on the top. `Streamcluster` shows constant behavior with poor scalability throughout its execution while `freqmine` dynamically changes its behavior at runtime. More specifically, `freqmine` consists of three phases: it begins with a poorly scalable phase, changes to a highly scalable phase, and again changes to a poorly scalable phase and finishes the execution.

As simultaneous execution begins, C-3PO initializes the DVFS levels and the core count of programs. The numbers of cores allocated to both programs are gradually reduced while the DVFS levels instantaneously reach the maximum. `Freqmine` changes to the second phase around 8 s, where the core count gradually increases while the DVFS level decreases in concert. After `freqmine` reaches a steady state, the DVFS level of `streamcluster` is reduced. The configurations stay constant until the last phase transition occurs to `freqmine` around 38 s. Once the per-core utilization of `freqmine` starts to decline, the core count is reduced while the DVFS level of `streamcluster` is increased to the maximum.

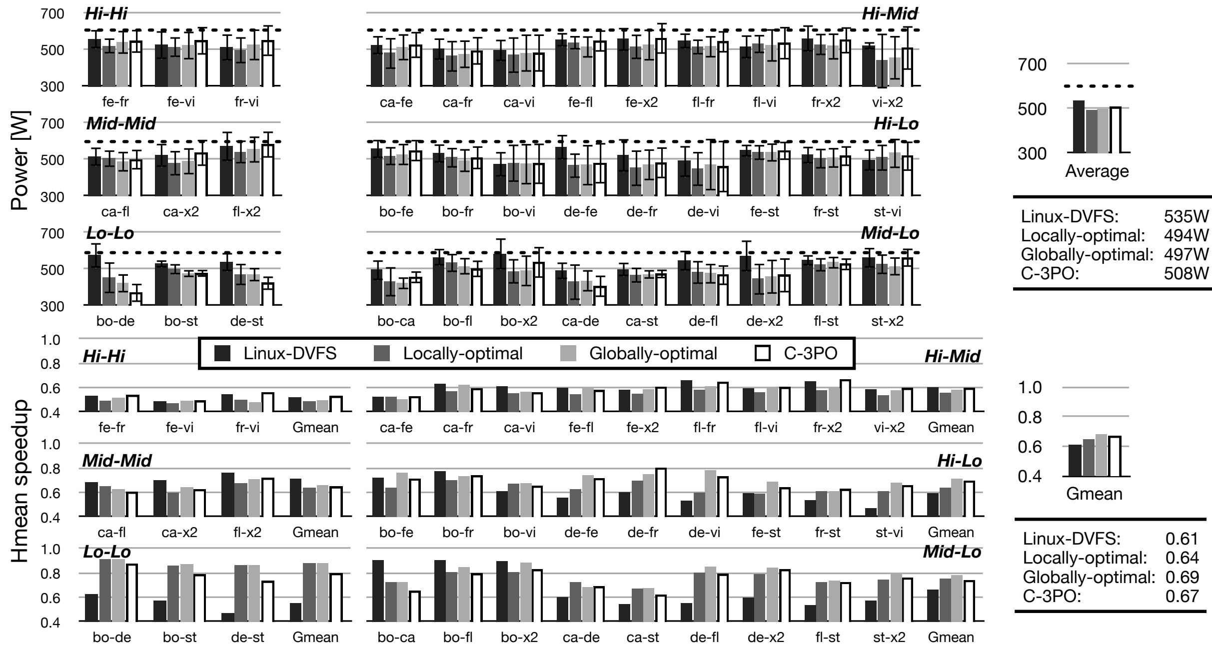


Fig. 9. Average and standard deviations of the measured power consumption (top) and hmean speedup (bottom) for all workloads of Workload-2 with Linux-DVFS, Locally-optimal, Globally-optimal and C-3PO.

We can also see from the power trace that the estimated power consumption is controlled to keep within the target power consumption $P_{target} = 540\,\text{W}$ (10% safety margin is used). The measured power consumption accurately follows the estimated power which shows the high accuracy of the proposed online power estimation technique. Another interesting thing to be noticed is that the power consumption is controlled around 500 W during the second phase of `freqmine` which is lower than P_{target}. This is because there is enough power budget salvaged from `streamcluster`, while the power consumption is redistributed to `freqmine` by increasing the core count to 48 which almost reaches its peak performance. This means that C-3PO can improve performance and at the same it has a chance to decrease the power and energy consumed for the execution. This comes from C-3PO's policy which aims to increase the performance but does not freely distribute the power budget it has salvaged, controlled by monitoring the processors' utilization.

B. Power Management Accuracy

The top figure of Fig. 9 shows the average and standard deviations (as error bars) of the measured power consumption for all the 36 workloads of Workload-2. The bars are grouped with the same types of workloads (e.g., Hi-Hi, Hi-Mid, etc), and $P_{real} = 600\,\text{W}$ is shown as dotted lines. The high power estimation accuracy shown in Fig. 8 suggests reasonable online power capping capability, which can also be seen from the bars together with the error bars for each policy in the figure.

C. System Performance

1) Workload-2: Performance of Workload-2 are shown in the bottom of Fig. 9 where the workloads are grouped as the same as the power results along with geometric mean (gmean) of the hmean speedup for each group. Before going into the detailed results of each group, we first see the overall gmean results which are shown in the right end of the figure. All three policies which partitions the processor achieve better performance than *Linux-DVFS* (gmean of 0.61). C-3PO performs 9% better than *Linux-DVFS*, 4% better than *L-opt* and 3% worse than *G-opt*.

C-3PO mainly has the following three advantages over its counterparts: (I) the ability to cope with poorly scalable programs by restricting the number of cores to allocate, (II) the flexibility of distributing the power budget and the number of cores to each program and (III) the ability to dynamically adapt to execution phase changes within programs.

(I) The first advantage allows C-3PO to achieve better performance than *Linux-DVFS* for workloads with Lo programs because *Linux-DVFS* allocates all the cores to programs regardless of their optimal core count. Lo-Lo workloads are remarkable where C-3PO achieves 45% better performance than *Linux-DVFS* by allocating appropriate number of cores which achieves the peak performance for programs such as `dedup` or `streamcluster` (both 16 cores), while the salvaged power can be redistributed to other programs. These results along with the power results show that C-3PO executes workloads with Lo types in a high-performance *and* energy-efficient manner. It can complete the workloads in shorter time with lower average power, which greatly reduces the energy consumption. C-3PO shows greater performance than *Linux-DVFS*, however, *L-opt* and *G-opt* achieve 9% better performance than C-3PO for this type of workloads. This is because the performance of Lo-Lo workloads can be optimized with small amount of power consumption *and* core count,

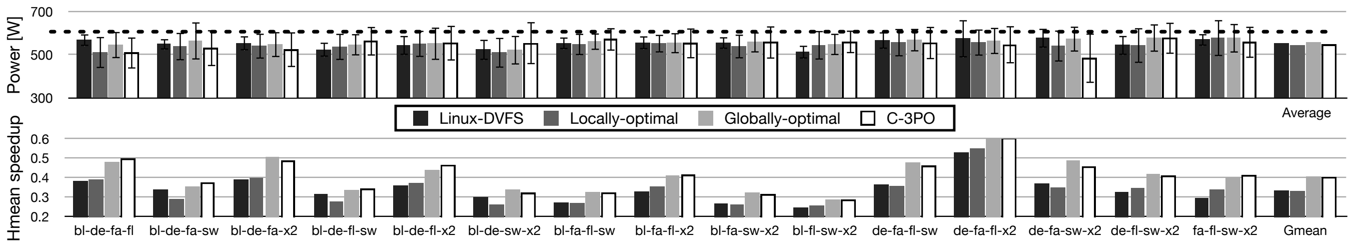


Fig. 10. Average and standard deviations of the measured power consumption (top) and hmean speedup (bottom) for all workloads of Workload-4 with Linux-DVFS, Locally-optimal, Globally-optimal and C-3PO.

which means that both *L-opt* and *G-opt* result in the same allocation.

(II) As discussed in Section I, the second advantage is the key of C-3PO which enables further optimization against *L-opt*. The results of Hi-Mid and Hi-Lo show this where C-3PO performs 7% and 9% better than *L-opt*. *L-opt* is not able to fully utilize the power budget for Mid and Lo programs while Hi programs favor additional power budget, which can be optimized in a global manner with C-3PO.

(III) Dynamically adjusting to execution phases is another essential feature of C-3PO. As we have seen in Fig. 8, `freqmine` is a program whose scalability characteristics change time over time. `X264` shows similar behavior as well [16]. We can see that C-3PO outperforms not only *L-opt* but also *G-opt* for 10 workloads which include these two programs (`fe-fr`, `fr-vi`, `fe-x2`, `fl-fr`, `fr-x2`, `vi-x2`, `bo-fr`, `de-fr`, `fr-st` and `fl-x2`).

Mid-Mid workloads show different trends compared to previous types where *Linux-DVFS* achieves the best performance. This comes from the characteristics of `fluidanimate` and `x264` which achieve their peak performance with all the cores allocated but not fully utilizing the processor (per-core utilization of $\approx$ 0.4). Therefore, better performance results are achieved by allowing the programs to use all the processors without applying processor partitioning. However, the performance gap is not significant where the gmean of this type of workloads for *Linux-DVFS* and C-3PO are 0.71 and 0.65, respectively.

2) Workload-4: The power and hmean speedup for Workload-4 are summarized in Fig. 10. We can see similar trends for average power consumption with Fig. 9, indicating that power capping is possible regardless of the number of programs. On the other hand, even though the workloads consist of a mix of programs which are diversely chosen from types Hi, Mid and Lo, in a similar manner with Workload-2, the performance trends are not diverse as is the case in Workload-2 evaluation. We can see that C-3PO achieves better performance than *Linux-DVFS* and *L-opt* for *all* the 15 workloads. Similar with the results of Workload-2 which include type Lo programs, C-3PO greatly reduces the energy consumption for some workloads. The most remarkable result is for `de-fa-sw-x2` with hmean of 0.46 which improves the performance over *Linux-DVFS* and *L-opt* for 24.3% and 31.4%, respectively; while consuming average power of 484 W which reduces the average power consumption against the counterparts for 16.2% and 10.4%, respectively. These results suggest that C-3PO can effectively find a better allocation when there exists larger optimization space. Some workloads in Workload-2 had no optimization space left (e.g., Hi-Hi workloads) where workloads in Workload-4 provide flexible optimization space which can be efficiently utilized by C-3PO. This suggests that when more and more programs with various characteristics are co-scheduled on future manycore processors, the advantage of C-3PO which globally distributes the power budget becomes apparent, resulting in high system performance. Overall, when compared with *G-opt*, we can see that C-3PO achieves equivalent gmean of 0.40, where it greatly improves performance over both *Linux-DVFS* and *L-opt* for 21.2%.

D. Analysis of C-3PO

TABLE IV shows how close the allocations chosen by C-3PO are against *G-opt* for Workload-2. Workloads are listed with their group where the results of `freqmine` and `x264` are excluded (21 out of 36 workloads are shown in the table) because they show different execution phases at runtime which makes the static allocation of *G-opt* suboptimal as discussed before. We calculate the hmean speedup for every possible allocations from the individual performance and utilization characteristics of each program, where each row shows the percentage of time spent with allocations which is meant to achieve within X% (X = 5, 10 and 20) of the best statically calculated hmean speedup (i.e, that of *G-opt*). We can see from the table that, on average, 65.0% of the time is spent with allocations which are expected to achieve hmean within 10% of the best configuration, where 90.0% of the time are spent within 20% of it. This supports the performance results shown in Fig. 9 where C-3PO performs comparable to *G-opt*.

VI. Related Work

Controlling several knobs together in order to dynamically optimize power-performance of parallel programs is studied in the literature. For single multithreaded programs, Li and Martinez proposed a heuristic-based mechanism to search for the best DVFS level and core count combination [13]. Pack & Cap [4] introduces thread packing where it orchestrates with DVFS to maximize performance within a power cap. We study multiprogrammed workloads that are beyond the scope of prior work, where we need to take the power-performance characteristics of the co-scheduled programs into account which makes the problem more challenging.

TABLE IV
PERCENTAGE OF TIME C-3PO EXECUTED WORKLOAD-2 IN ALLOCATIONS WHICH PERFORM COMPARABLE TO GLOBALLY-OPTIMAL

Error	Hi-Hi	Hi-Mid				Hi-Lo						Mid-Mid	Mid-Lo						Lo-Lo			gmean
	fe-vi	ca-fe	ca-vi	fe-fl	fl-vi	bo-fe	bo-vi	de-fe	de-vi	fe-st	st-vi	ca-fl	bo-ca	bo-fl	ca-de	ca-st	de-fl	fl-st	bo-de	bo-st	de-st	
< 5	91.7	67.7	38.2	54.9	77.2	67.0	35.1	59.1	62.7	33.0	31.4	18.8	23.9	31.2	15.0	5.1	6.4	35.0	45.7	3.3	23.9	29.6
< 10	92.6	74.3	73.2	69.3	91.7	78.2	46.4	63.4	77.1	77.1	69.2	42.0	90.9	77.8	72.0	35.8	69.0	72.7	74.9	24.8	54.9	65.0
< 20	100	96.8	100	93.0	100	97.5	78.4	78.5	79.5	98.5	99.3	89.0	94.4	95.8	79.4	72.3	89.5	98.2	90.9	78.0	89.2	90.0

Others assume a mix of single-threaded programs as target workloads [10, 18]. Winter et al. present *Steepest Drop*, a heuristic-based optimization technique which iteratively selects the application and processor pair that would provide the biggest ratio of power reduction to performance loss when the power is over the budget [18]. C-3PO follows a similar approach when trying to reduce the power consumption. Isci et al. propose *MaxBIPS* policy for global power management with per-core DVFS which takes power from low performing program and gives it to other programs to maximize the instruction throughput [10]. Our work incorporates thread packing where core count and DVFS level affect both performance and power, thus cooperatively controlling these two knobs is required.

Optimizing workloads composed of variable multithreaded programs are also studied [2, 15, 16]. Sasaki et al. apply thread packing to optimize performance where they utilize a simple performance model and sampling [16], however, they do not take DVFS control into account. Bhadauria and McKee proposed heuristic-based scheduling techniques which also apply thread packing in order to optimize the ED metric [2] but they do not take DVFS into account as well. Ma et al. studied a scalable power management technique which controls the power consumption to stay below the budget while optimizing the performance [15]. Contrary to the studies by Sasaki et al., and Bhadauria and McKee, they only consider optimizing the DVFS level for each program because the thread count per application is fixed a priori and is chosen such that the sum is less than the total core count in their study. We believe that the problem we have tackled is more general and important considering the use of future manycore processors. Our work has discovered a truly cooperative optimization of thread packing and DVFS for such an important problem that none of the previous studies have focused on.

VII. CONCLUSIONS

Future computer systems with manycore processors are required to effectively handle diverse multithreaded programs co-scheduled while keeping the peak power consumption within a power budget. This work investigates the design of an efficient runtime system which accomplishes such requirements by cooperating thread packing and DVFS in tandem. We propose a system called coordinated performance-per-power optimization or C-3PO, which corporates sophisticated online power and performance estimation capabilities. The system is designed to iteratively converge to a better allocation using a heuristic mechanism along with the prediction techniques. The key idea of C-3PO is to dynamically salvage the power budget that is not contributing to performance and allocate to programs which are expected to effectively convert the distributed power into performance, in a global manner. Empirical results show that C-3PO enables precise power control, as well as 21.2% better performance than state-of-the-art solutions, and comparable performance to statically obtained globally optimal allocations.

ACKNOWLEDGMENTS

This research was supported in part by New Energy and Industrial Technology Development Organization (NEDO) and Japan Science and Technology Agency (JST), CREST.

REFERENCES

[1] Advanced Micro Devices, “AMD ‘Bulldozer’ core technology,” white paper, Advanced Micro Devices, 2011.
[2] M. Bhadauria and S. McKee, “An approach to resource-aware co-scheduling for CMPs,” in *ICS ’10*, 2010, pp. 189–199.
[3] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, 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, pp. 175–185.
[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *ISCA ’11*, 2011, pp. 365–376.
[6] S. Eyerman and L. Eeckhout, “System-level performance metrics for multiprogram workloads,” *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.
[7] X. Fan, W.-D. Weber, and L. A. Barroso, “Power provisioning for a warehouse-sized computer,” in *ISCA ’07*, 2007, pp. 13–23.
[8] A. Fedorova, M. Seltzer, and M. Smith, “Improving performance isolation on chip multiprocessors via an operating system scheduler,” in *PACT ’07*, 2007, pp. 25–38.
[9] P. Hargrove and J. Duell, “Berkeley lab checkpoint/restart (BLCR) for linux clusters,” *Journal of Physics: Conference Series*, vol. 46, pp. 494–499, 2006.
[10] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, “An analysis of efficient multi-core global power management policies: maximizing performance for a given power budget,” in *MICRO 39*, 2006, pp. 347–358.
[11] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, “Virtual machine power metering and provisioning,” in *SoCC ’10*, 2010, pp. 39–50.
[12] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, “System level analysis of fast, per-core DVFS using on-chip switching regulators,” in *HPCA ’08*, 2008, pp. 123–134.
[13] J. Li and J. F. Martinez, “Dynamic power-performance adaptation of parallel computation on chip multiprocessors,” in *HPCA ’06*, 2006, pp. 77–87.
[14] K. Lun, J. Gummaraju, and M. Franklin, “Balancing thoughput and fairness in SMT processors,” in *ISPASS ’01*, 2001, pp. 164–171.
[15] K. Ma, X. Li, M. Chen, and X. Wang, “Scalable power control for many-core architectures running multi-threaded applications,” in *ISCA ’11*, 2011, pp. 449–460.
[16] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura, “Scalability-based manycore partitioning,” in *PACT ’12*, 2012, pp. 107–116.
[17] A. Snavely and D. Tullsen, “Symbiotic jobscheduling for a simultaneous multithreaded processor,” in *ASPLOS-IX*, 2000, pp. 234–244.
[18] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, “Scalable thread scheduling and global power management for heterogeneous many-core architectures,” in *PACT ’10*, 2010, pp. 29–40.
[19] Yokogawa Electric Corporation, “WT1600 digital power meter.”
[20] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “Addressing shared resource contention in multicore processors via scheduling,” in *ASPLOS ’10*, 2010, pp. 129–142.