Market-inspired Dynamic Resource Allocation in Many-core High Performance Computing Systems

Amit Kumar Singh, Piotr Dziurzanski and Leandro Soares Indrusiak
Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK.
Email: {amit.singh, piotr.dziurzanski, leandro.indrusiak}@york.ac.uk

***Abstract*—Many-core systems are envisioned to fulfill the increased performance demands in several computing domains such as embedded and high performance computing (HPC). The HPC systems are often overloaded to execute a number of dynamically arriving jobs. In overload situations, market-inspired resource allocation heuristics have been found to provide better results in terms of overall profit (value) earned by completing the execution of a number of jobs when compared to various other heuristics. However, the conventional market-inspired heuristics lack the concept of holding low value executing jobs to free the occupied resources to be used by high value arrived jobs in order to maximize the overall profit. In this paper, we propose a market-inspired heuristic that accomplish the aforementioned concept and utilizes design-time profiling results of jobs to facilitate efficient allocation. Additionally, the remaining executions of the held jobs are performed on freed resources at later stages to make some profit out of them. The holding process identifies the appropriate jobs to be put on hold to free the resources and ensures that the loss incurred due to holding is lower than the profit achieved by high value arrived jobs by using the free resources. Experiments show that the proposed approach achieves 8% higher savings when compared to existing approaches, which can be a significant amount when dealing in the order of millions of dollars.**

***Keywords*—Many-core, High Performance Computing, Resource allocation, Profit, Value curves.**

I. Introduction

Many-core architectures are widely adopted to fulfill the need in various computing fronts such as general purpose and high performance computing (HPC) [1], [2]. These architectures enable parallel processing of various processes on different cores and thus achieve high performance. In HPC, the many-core resources can be arranged in several possible configurations. The bottom part of Figure 1 shows one possible configuration of a *many-core HPC platform*. The platform contains several nodes (Node 1,...,Node N), where each node contains a set of processing elements (PEs) connected by an on-chip interconnection network [2], [3]. The PEs are also referred to as processing cores. The nodes of the platform can be connected via different communication infrastructures such as conventional bus, PCI express, network switch, etc., to facilitate communication between the cores of different nodes. Depending upon the performance requirement and hardware cost, an appropriate infrastructure can be employed.

It has been well proven that resource allocation is one of the most complex problems in large many-core systems, and in general it is considered NP-hard [4]. Therefore, a well-tuned search algorithm needs to evaluate hundreds of thousands of distinct allocations before it finds one solution that meets the systems performance requirements. Since such evaluation is expected to take a long time, maybe hours to days, it cannot be applied to find the solution quickly, which is desired in the contexts of dynamic resource allocation for the jobs arriving at different unknown moment of times. A job may contain a number of tasks or processes. Further, the search algorithms normally consider static workload, e.g., a fixed number of jobs known a priori, and thus cannot handle dynamic workload, where unknown number of jobs may arrive at different moment of times. The top part of Figure 1 shows jobs (e.g., JOB 1 and JOB m) having different arrival time. These dynamically arriving jobs can be allocated to the platform resources by employing light-weight run-time heuristics that can find an allocation quickly.

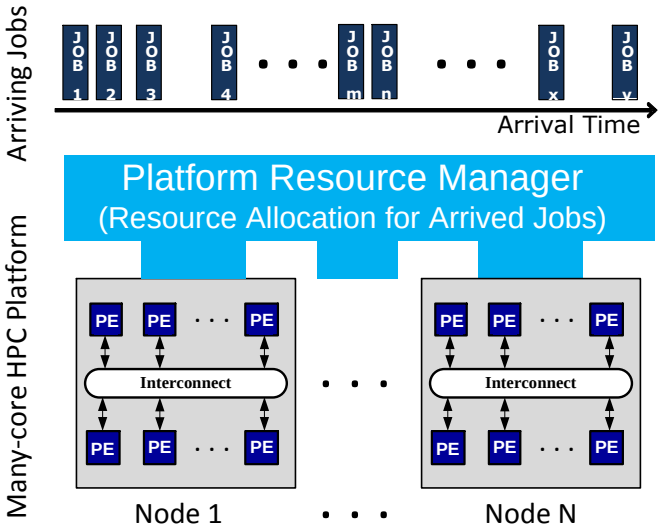

Fig. 1. Jobs arrival and resource allocation for them on a many-core HPC platform.

Resource allocation process takes the list of unallocated arrived jobs as input at some particular time and tries to find an efficient allocation for each job based on current platform status, i.e. availability of resources (cores). A *platform resource manager*, as shown in Figure 1, keeps the updated platform status and performs the resource allocation process by employing efficient light-weight run-time heuristics. Extensive literature exists for such run-time heuristics [1]. However, *in overload situation* where demand for available resources is higher than the supply, these heuristics can lead to starvation, missed deadlines and reduced throughput [5], [6]. In such situation, it becomes difficult to decide what jobs to discard (or hold for later allocation) and what to enter into the system when resources become available due to completion of a job.

To handle the overload situation, notion of values (economic or otherwise) of the jobs have been introduced by previous researchers to define the importance level of jobs [5], [6]. For example, in a time-critical system, jobs are supposed to accomplish certain services upon execution, and thus each job has a particular importance to the overall system functionality. This helps in deciding to hold the low value jobs for late allocation and allocating limited resources to the high value jobs. The value of a job can change over time to reflect the impact of the computation over the business processes. For example, finishing a computation earlier may result in increasing earnings (achieved profit) for a specific product, whereas a late finish may result in low earnings. Such a variation in the value of a job adds complexity to the allocation process.

The notion of values of the jobs has led to the investigation of several *market-inspired* or *value-based* heuristics for various computing platforms such as clusters, distributed databases, grids, etc. [7], [8]. These heuristics use available platform capacity measured in terms of *bids* within an auction-like allocation process and try to find allocations resulting in high system performance in terms of profit or value. In Figure 1, to initiate auction-like process, the bids from different computing nodes (Node 1, ..., Node N) in terms of available processing power (cores) can be placed to the platform resource manager and the node having the highest bid can be chosen to allocate the current highest value job to maximize the value returned by system. Existing market-inspired heuristics use several similar concepts, but they do not employ the concept of holding low value executing jobs to free the occupied resources to be used by high value arrived jobs in order to maximize the overall profit. Additionally, they do not employ any design-time profiling of the jobs, which can facilitate efficient resource allocation to the jobs.

Contribution: This paper addresses shortcomings of existing market-inspired heuristics by proposing *1)* a design-time profiling concept for jobs obtained from the historical data to facilitate efficient dynamic allocation, and *2)* the concept of holding low value executing jobs for later allocation in order to allocate freed resources by holding to the high value arrived jobs. Since the jobs put on hold, i.e. preempted jobs, are tried to be allocated later on, our approach can also be referred to as preemptive or suspend/resume approach. The holding (preemption) process in initiated when the bids received by platform resource manager from different nodes (Node 1, ..., Node N in Figure 1) are zero in terms of number of free cores and none of the arrived jobs can be allocated to the platform cores. The holding process selects the appropriate low value executing jobs to stop their execution such that holding will lead to improved overall profit. The remained executions of the held jobs are resumed (performed) on freed resources at later times if the jobs still hold some values. The profiling and holding/resuming concept can be easily augmented into the existing market-inspired heuristics in order to achieve improved overall profit. We evaluate the proposed approach for HPC workloads containing dynamically arriving jobs and observe improvement in the overall profit when compared to existing approaches. The HPC workloads are obtained from High Performance Computing Center Stuttgart (HLRS) of the University of Stuttgart as the historical data over the last year, and many-core HPC system and heuristics are modelled to fulfill the needs of a real HPC system to be deployed in later stages.

The remainder of the paper is organized as follows. In Section II, related works regarding the dynamic resource allocation involving market concepts are discussed. The models of job, value of a job and HPC platform along with the problem definition are introduced in Section III. The proposed market-inspired approach is discussed in Section IV. Section V presents the experimental results and Section VI concludes the paper.

II. Related Work

Resource allocation on many-core systems is a well studied topic. The need for dynamic resource allocation arose to handle dynamic workloads, which is encountered in several computing systems such as embedded and HPC. The dynamic allocation process normally employ a heuristic following some fundamental optimization procedure to identify an efficient allocation at run-time. Several heuristics have been proposed to accomplish this aim [1], [9]. Some fundamental procedures employed in the heuristics are iterative hierarchical allocation to reduce energy consumption while satisfying the required Quality of Service (QoS), incremental dynamic allocation, hybrid mapping to perform intensive computations at design time and using the design-time analysed results at run time, etc. [10]. In overload situation, as described earlier, these heuristics can lead to starvation, missed deadlines, and reduced throughput. Further, these heuristics do not take into account any notion of values of jobs to users.

Market-inspired resource allocation heuristics employing notion of values representing importance of jobs have been studied to perform well in overload situations [7], [11], [12]. Some researchers have considered fixed value of a job [13], whereas others consider values that can change with time, described with so-called value curve of the job [5], [6]. In such curve, the value of a job normally decreases with computation time and reflects the impact of computation over the business process. It has been shown that using value curves instead of fixed values of jobs gives greater market efficiency in the long run [14]. During the course of allocation, the resource manager receives the list of unallocated jobs and the bids obtained from different nodes (Figure 1) to identify appropriate allocation for each job [15].

The employed heuristic can allocate the jobs in several ways. For example, the highest value job to the node having highest bid, which can also be referred to as *bidding based on highest value* [13]. The problem with this approach is that a high value job might require large amount of resources, and thus leaving less resources for rest of the jobs. A remedy to this problem could be to allocate resources first to several small size jobs requiring less resources, but a higher profit cannot be guaranteed as the values of jobs requiring more resources and having higher values might become very low or zero by the time resources are available. In order to overcome such problems, *bidding based on highest value density* was introduced [16], where tasks value divided by the amount of required computational homogeneous resources is considered as the value density. Variants of value density based approaches have also been proposed [17]–[19]. Another heuristic termed as *minimum value remaining* has been proposed to ensure that the job that is going to lose its value soon, i.e., has minimum remaining value, should be allocated first [20]. The remaining value is calculated as the area under the value

curve from the current time to the time when its value is zero. These approaches are similar to *Backfilling* approaches in cluster schedulers, where small jobs are moved forward in the prioritized job queue to utilize the idle computers or cores [21], [22]. These approaches might not guarantee higher overall profit than the bidding based on the highest value as higher value jobs might be postponed for later allocation due to their low value density or high minimum remaining value. Further, they do not use design-time profiling results and lack the concept of holding low value executing jobs to allocate the freed resources to high value jobs.

In contrast to the above heuristics, our approach uses profiling results and employ the concept of holding the low value executing jobs for allocating high value arrived jobs and resuming the held job back to operation. Further, some of existing heuristics consider a uni-process system (e.g., [19]) and a fixed value of each job (e.g., [13]). Our approach is applicable to many-core systems and jobs having varying values over time, which is desired for the modern HPC systems.

III. System Model and Problem Definition

We model our workload and many-core HPC platform based on the typical industrial HPC scenario. This section provides a brief overview of the workload and platform model along with the problem definition.

A. Job Model

An HPC workload consists of a number of jobs, where each job j is modelled as a directed graph $TG = (T; E)$, where T is the set of tasks of the job and E is the set of directed edges representing dependencies amongst the tasks. Figure 2 (a) shows an example job that contains 7 tasks (t_1,..,t_7) connected by a set of edges. Each task $t \in T$ has attributes execution time (ExecTime) and memory requirement, when mapped on a core. The ExecTime for each task is considered as its worst-case execution-time (WCET) and remains fixed. Each edge $e \in E$ represents data that is communicated between the dependent tasks. A job j is also associated with its arrival time AT_j. The jobs of the HPC workload are obtained from our project partner High Performance Computing Center Stuttgart (HLRS) and are based on the historical data.

B. Value Curve of a Job

The value curve represents the value forecast related to the completion of a given computation over time. Therefore, for each job j, the value curve VC_j is a function of the value of the job to the user depending on the completion time of the job. The value curve is usually a monotonically-decreasing function and trends towards zero with the increasing completion time, as shown in Figure 2 (b), where appropriate benefits (profits or values, on vertical axis) for completing the job at different times (on horizontal axis) are shown. A similar value curve model has been used in several works reported in the literature, e.g., [5], [6], [23], [24]. We assume that value curve of each job of the HPC system is given, which is generally perceived from the business unit by following an economic model. The description of the economic model is orthogonal to our approach and out of scope of this paper.

The value curves facilitate bidding based on the available processing capacity on different platform nodes computed as bids towards maximizing the profit for each node. For example, a node with a high available processing capacity can bid for the maximum value (500 in Figure 2 (b)) to maximize its profit. This also helps to finish the job as soon as possible, e.g. in 15 time-units if bid is for 500, enabling faster release of the occupied resources that can be used for future incoming jobs.

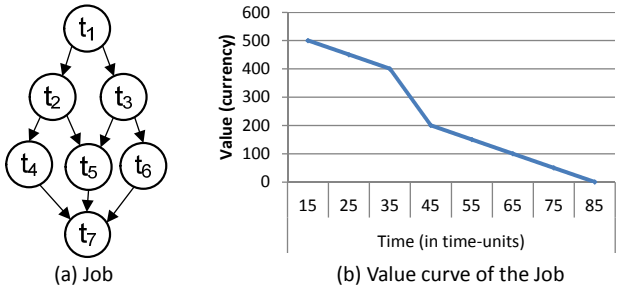

Fig. 2. An example job model and its value curve.

C. Many-core HPC Platform Model

The HPC platform HP contains a set of nodes $(PG_1, ..., PG_N)$, where each node contains a set of homogeneous cores (PEs). A node n is represented as a directed graph $PG_n = (C_n; V_n)$, where C_n is the set of cores of the node and V_n represents the connections amongst the cores. The bottom part of Figure 1 shows an example HPC platform. The communication amongst the cores of a node is established by employing dedicated connections.

A *platform resource manager* (as shown in Figure 1) is used to manage the platform resources and perform resource allocation for the arrived jobs. During system operation, the manager keeps up to date status of the platform resources, i.e., which resources are busy and which are idle, such that accurate and efficient allocations can be made. In our case, the platform status is maintained as the number of available (idle) cores at different nodes and resource allocation has also been referred to as core allocation.

D. Problem Definition

In HPC system, jobs $(j_1, ..., j_M)$ arriving at different moment of times need to be efficiently allocated on the resources (cores) of the platform nodes $(PG_1, ..., PG_N)$ in order to maximize the overall system profit P earned by servicing these jobs. It is assumed that the tasks of a job are allocated to only one node in order to avoid huge communication delay between different nodes. To summarize, the problem targeted in this paper considers the following set of input, constraints and objective.

- **Input:** Workload, i.e., Job set $(j_1, ..., j_M)$, Value curve of each job VC_j, Arrival time of each job AT_j $(j \in 1, \ldots, M)$, Cores of the HPC platform nodes $(PG_1, ..., PG_N)$.
- **Constraints:** Limited resources (cores) on each node of HP.
- **Objective:** Maximize overall profit P.

IV. Proposed Market-inspired Approach

The proposed market-inspired approach is presented in Figure 3. It consists of two main steps: *1) design-time (off-line) profiling* of the jobs obtained from the historical data, and *2) run-time (on-line) resource allocation* for the jobs considering their arrival time and profiled results. The *platform resource manager* is invoked to perform the resource allocation process for the arrived jobs.

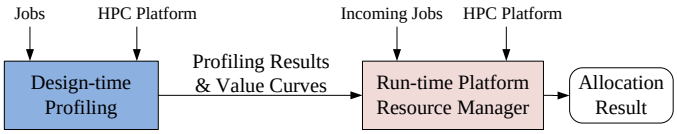

Fig. 3. Proposed approach.

ALGORITHM 1: Design-time Profiling

Input: Job j, HPC Platform HP.
Output: Minimum response times and corresponding allocations at different number of used cores.
Find $maxUsedCores$ by j;
for $usedCore = 1$ *to* $maxUsedCores$ **do**
  Find *allocation* using $usedCore$ cores that provides minimum *response time* by GA [25];
end
return response times and allocations;

A. Design-time Profiling

For each job in the workload, the profiling step estimates the performance (expressed as response time) when utilizing different amount of computing power in terms of number of cores. The response time of a job can also be referred to as the completion time after the job has been allocated resources for execution, i.e., different between the end time and start time of the job. Algorithm 1 describes the profiling procedure for a job. First, maximum number of used cores by the job ($maxUsedCores$) is found, which is equal to the number of tasks in the job. Since each task can occupy only one core, the $maxUsedCores$ cores can exploit all the parallelism present in the job and thus there is no point of allocating more cores than the number of tasks in the job. To estimate the response time at different number of used cores, we follow a genetic algorithm (GA) based evaluation, similarly as in [25]. The same GA approach is run repeatedly by providing the number of cores as input in order to find an efficient allocation leading to minimal response time. Choosing the minimal response time value helps us to complete the job as soon as possible.

Figure 4 shows some outcomes of the profiling step for the example considered job in Figure 2. The different number of used cores and the corresponding minimal response time values are plotted on the right vertical (# Cores) and horizontal (time) axis, respectively. The response time values are computed by assuming worst-case execution times of the tasks in the job, so that the most pessimistic run-time system behaviour can be taken into account. The profiling output (in red color) is plotted along with the given value curve (in blue color), which provide enriched information for the job to perform efficient run-time resource allocation. Similar profiling is performed for all the jobs in the workload. For each job, this step associates information about the required computing power (# Cores) to achieve a certain value by executing the job over a fixed amount of time. These information along with the allocation decisions at different number of used cores are stored (*Profiling Results & Value Curves* as shown in Figure 3) to be used for performing efficient run-time resource allocation.

B. Run-time Resource Allocation

In order to allocate platform resources to the incoming jobs at run-time, the platform resource manager is invoked to find

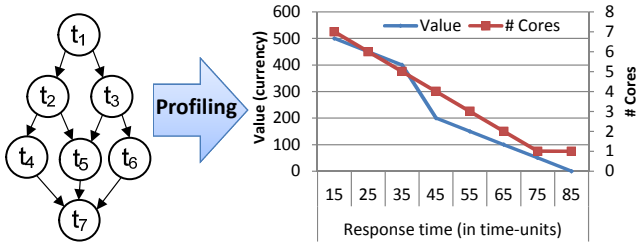

Fig. 4. Profiling output.

an allocation. The manager takes the profiling results of the jobs from the storage along with their value curves and arrival times as input, and identifies profit maximizing allocation for each job based on the number of available cores at different nodes in the platform. This helps to achieve high overall profit by servicing (completing) different jobs. For each job, it is assumed that all of its tasks will be allocated to one node in the platform, i.e., the tasks of a job cannot be allocated to more than one nodes in order to avoid huge communication delay between two nodes. In case of a newly arrived job for which profiling result is not available, the profiling step in employed followed by the run-time resource allocation step based on the available number of cores. In such a case, the profiling step needs to identify an allocation for the available number of cores, which can be done in the order of millisecond or seconds. Since this step is quite fast as compared to tasks execution times that are in the order of minutes or hours, it brings only a small timing overhead that can be neglected.

The proposed resource allocation heuristic followed by the manager is summarized in Algorithm 2. The profiling results used as input are the minimum response time values achieved at different computing power (number of used cores) and the corresponding allocation decisions. At each time step, the heuristic checks for three events as follows: 1) *any already allocated job(s) finish execution* to update the platform resources, *2) any job(s) arrive into the platform* to put into a job queue, and *3) job queue contains job(s) having non-zero values at current time step* to perform resource allocation for such jobs.

To perform resource allocation for all valuable queued jobs (i.e., jobs having positive values), all of them (*counter* = 0 to *JobQueue.size()*) are tried to be allocated on the platform resources as along as any resource is available or profit can be made by holding some executing jobs. First, bids (in terms of number of available cores) from different platform nodes are collected, then the maximum bid ($maxBid$) and the corresponding node is selected. Choosing such a node helps to achieve better load balancing amongst nodes and thus better resource utilization. In case more than one nodes have the same amount of bid, any of them is chosen. If the estimate of $maxBid$ is greater than zero ($maxBid > 0$), i.e., at least one resource is available in the platform, the profits of jobs utilizing $maxBid$ resources are computed and the job leading to maximum profit is selected ($maxProfitableJob$) to be allocated to resources of the node having $maxBid$ resources provided the maximum profit is a positive value ($profit > 0$). The profit computation for each job considers the exact number of cores to be used by the job and its value at the current time step. If $maxBid$ is greater than the number of cores to be used to achieve maximum profit (maximum parallelism exploitation), the latter one is chosen as the exact number of cores to be used; otherwise the former one is chosen. The resource allocation

ALGORITHM 2: Run-time Resource Allocation

Input: Incoming Jobs with arrival times, Value curves of Jobs with profiling results, HPC Platform HP.
Output: Resource Allocation for Incoming Jobs.
for *each* $time\_step$ **do**
  if *allocated_job(s) finish execution* **then**
    Update platform resources;
  end
  if *job(s) arrive* **then**
    Put the *job(s)* in *JobQueue*;
  end
  if *JobQueue contains job(s) having positive values* **then**
    $counter$ = 0;
    repeat
      Collect $bids$ from different nodes;
      Select $maxBid$ from $bids$;
      if $maxBid > 0$ **then**
        Compute $profits$ of jobs by utilizing $maxBid$ resources;
        Select $maxProfitableJob$ and its $profit$;
        if $profit > 0$ **then**
          Allocate resources of $maxBid$ node to $maxProfitableJob$;
          Update platform resources;
        end
      else
        //hold low value jobs and allocate later;
        Find executing $jobs\_to\_hold$ in the $best\_suitable\_node$ for recently arrived *maxProfitJob* (from *JobQueue*) and *max_hold_profit* by **Algorithm** 3;
        if *holding profitable (max_hold_profit > 0)* **then**
          Hold jobs $jobs\_to\_hold$ and put in *JobQueue* for later allocation;
          Release used resources by held jobs;
          Allocate resources of $best\_suitable\_node$ to *maxProfitJob*;
          Update platform resources;
        end
      end
      $counter$++;
    until *counter != JobQueue.size()*;
  end
end

ALGORITHM 3: Jobs Holding Heuristic

// *max_hold_profit* = 0;
for *each recently arrived job* $j \in JobQueue$ **do**
  Find *executing_jobs* in each platform node;
  Sort *executing_jobs* in each node in ascending order based on their *start_times*;
  for *each node* n *of platform* **do**
    for *each executing_job of* n **do**
      Find *net_profit* (Equation 1) by holding *executing_job*;
      if *net_profit > max_hold_profit* **then**
        max_hold_profit = net_profit;
        maxProfitJob = j;
        $best\_suitable\_node$ = n;
        Add *executing_job* to list $jobs\_to\_hold$;
      end
    end
  end
end

on the exact number of cores of the node containing $maxBid$ cores is done based on the allocation achieved on the same number of cores during design-time profiling. The allocation process allocates tasks within a job to the cores (PEs) of a node. The platform resources are updated after each allocation process to have up to date resources' availability information for the next allocation instance. Such information helps to achieve an accurate and efficient allocation.

In case no resource is available in the platform, i.e. $maxBid$ = 0, it is checked if any profit can be made by holding low value executing jobs that are supposed to lead to small amount of profit. For a profitable holding, *max_hold_profit* is greater than zero (*max_hold_profit* > 0). The jobs holding logic in presented in Algorithm 3, which provides the executing $jobs\_to\_hold$ in the $best\_suitable\_node$, and the maximum profitable queued job (*maxProfitJob*) along with the achieved profit (*max_hold_profit*) by utilizing the freed cores of the held jobs. The holding process is carried out for the recently arrived jobs (i.e., at current time step) to avoid the same process for all the queued jobs at each time step.

The holding process works as follows. For each queued job that has arrived at current time step, first, executing jobs of each platform node are found and sorted based on their start time of the execution. Then, the *net_profit* made by holding the executing jobs in each node is computed by Equation 1, where $Profit$ is calculated by allocating the current queued job on the freed cores and $Loss$ is the earlier profit achieved by the executing $jobs\_to\_hold$.

$$net\_profit = Profit - Loss \tag{1}$$

The allocation uses either all the freed cores or some of them. If the number of freed cores is higher than the number of cores required by the job to make the maximum profit, the later one is chosen as the number of cores to be used; otherwise the former one is chosen for the same. Sorting the executing jobs based on their start times helps us to choose, first, the job having the latest start time, then the latest start time job along with the second latest start time, then latest and second latest start time jobs along with the third latest start time, and so on. Such consideration helps to identify and hold the jobs that have started recently and avoids holding of jobs that have been executed for a long time. This process tries to identify the most profitable instance in terms of jobs to hold. For example, holding the latest start time job might not be profitable, but it might be profitable to hold the latest and second latest start time jobs together. In such cases, the profitable instance would be only when both the latest and second latest start time jobs are put on hold. Further, this also avoids considering all the possible job combinations that might be quite huge for large number of executing jobs in a node.

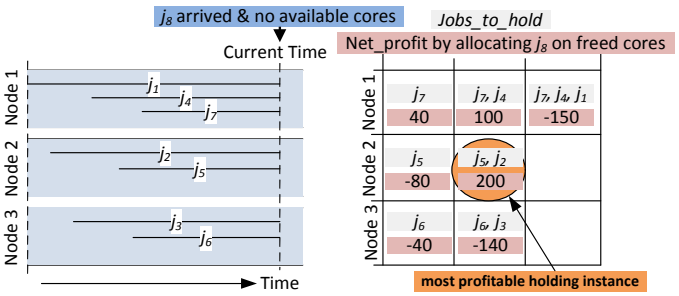

Fig. 5. Holding Demonstration.

Holding Demonstration: Figure 5 demonstrates the holding process, where three platform nodes are executing different set of jobs at *the current time*, e.g., Node 1 is executing jobs j_1, j_4 and j_7. The executing jobs started at different moments of time. At the current time, job j_8 has arrived and no resource is available in the platform, therefore, the holding process tries to identify the set of jobs to be put on hold. The table on the right hand side shows the jobs to hold in various nodes and the corresponding net_profit by allocating the freed resources to job j_8. It can be realized that sometimes the net_profit (in appropriate currency) by holding two jobs may be higher than that of one job, e.g., 100 by holding j_7 and j_4, whereas only 40 by holding j_7. The net_profit can also be negative (e.g., -80 by holding j_5), representing a loss if jobs are put on hold, i.e., the achieved profit is less than the loss. The *most profitable holding instance* is to hold the jobs j_5 and j_2 of Node 2, which results in a net_profit of 200. The holding process will choose this instance.

If holding is profitable ($max\_hold\_profit > 0$), the jobs $jobs\_to\_hold$ of node $best\_suitable\_node$ are put on hold for later allocation and used cores are released. Then, the incoming job is allocated to the freed cores based on the profiling allocation decisions and resources are updated. The holding process helps to achieve a higher profit for some jobs, whereas jobs on hold achieve lower profits due to allocation at later time steps with decreased values of the jobs. In case holding is not profitable, the recently arrived jobs remain in the job queue and resource allocation for them is performed later when resources become available by completing the executing job(s).

The allocation process also ensures that a queued job having zero value at the current time step is dropped from the queue as no profit can be made out of it. Further, the allocation for a queued job that was put on hold starts from the hold point (i.e., it is resumed) to ensure that only the fraction of the job left after holding is executed, but not the whole job from the beginning. The allocation process continues until all the arrived jobs are allocated or dropped due to having zero value while waiting in the job queue.

V. Experimental Results

The proposed market-inspired heuristic has been implemented in a C++ prototype and integrated with a SystemC functional simulator. As a workload to evaluate the quality of the heuristic, historical data of an industrial HPC system at High Performance Computing Center Stuttgart (HLRS) has been considered. The workload contains a set of jobs having varying arrival time. Each job contains a set of tasks that have known worst-case execution times (WCETs), which can be obtained via prior executions of tasks from the historical data. The number of tasks in the jobs vary from 5 to 10.

TABLE I. Approaches considered for comparison

Approaches	Abbreviation	References
Simple Job Queuing	SJQ	Baseline
Maximum Value Queued Job	maxV	[13]
Maximum Value Density Queued Job	maxVD	[16]
Minimum Value Remaining Queued Job	minVR	[20]
Simple Job Queuing with Holding	SJQH	Proposed
Maximum Value Queued Job with Holding	maxVH	Proposed
Maximum Value Density Queued Job with Holding	maxVDH	Proposed
Minimum Value Remaining Queued Job with Holding	minVRH	Proposed

The considered HPC platform model contains a set of 3 nodes, where each node consists of 9 cores. However, any number of nodes and cores within them can be considered based on the physical limitation of hardware integration. The considered platform is modelled based on a working HPC platform deployed in the HPC centre at University of Stuttgart.

The platform manager employs a heuristic to find an allocation for each job of the workload by considering its arrival time, given value curve and profiled information representing the computing power (used number of cores) and the corresponding allocation decision to achieve a certain value by executing over a fixed amount of time. The profiling information is achieved by employing the design-time profiling step described in Section IV-A. The platform status (resource availability) is also taken into account during the allocation process.

We present results obtained from our proposed approach to perform efficient resource allocation for the jobs in the workload and compare them with various relevant existing approaches reported in the literature, which are abbreviated in Table I. In SJQ, the jobs are queued when no resource (core) is available in the platform and they are processed in the queuing order to allocate the freed cores at later time steps if they still hold values. This approach is the simplest one and is considered as the baseline. In maxV, maxVD and minVR as well, the jobs are put in queue when no platform core is available, but the queued jobs are processed based on their value, value density and the remaining value, respectively, when utilizing available (or required) computing power (number of cores) of highest bid node ($maxBid$ node). The maxV approach chooses the maximum value queued job first, whereas maxVD and minVR approaches choose maximum value density queued job and minimum value remaining queued job, respectively. The value density of a job is computed by dividing the achieved value by the number of used cores. This indicates that a job providing high value and requiring less cores is allocated first, which also leaves cores for later arriving jobs. The remaining value of a job is calculated as the remaining area under the value-time curve from the current time to the time when value becomes zero. This signifies that the job that is going to loose its value soon, i.e., has minimum remaining value, is chosen first in the assumption that most of the jobs will be serviced and some values will be achieved out of them. However, in doing so, a high value job might have a very low value by the time resources are available to perform the allocation process for it. Therefore, it might result in low overall profit.

The proposed approach employs the holding logic to hold the low value executing jobs to allocate the freed cores to high valued arrived jobs. The variants of the proposed approach when employing holding (H) are referred to as SJQH, maxVH, maxVDH and minVRH, as tabulated in Table I. The SJQH,

TABLE II. PROFILING RESULTS.

Job	Response time at different number of used cores								
	1	2	3	4	5	6	7	8	9
j_1	86	43	29	23	19	17	15	15	15
j_2	58	29	20	17	13	11	11	×	×
j_3	83	42	28	22	22	22	×	×	×
j_4	74	37	26	25	25	×	×	×	×

maxVH, maxVDH and minVRH employ holding logic in approaches SJQ, maxV, maxVD and minV, respectively. All the approaches try to use the maximum bid node first, i.e. the node having highest processing power, and thus inherently employ load balancing. Further, it should be noted that all the heuristics in Table I use the profiling results to make a fair comparison between them.

A. Profiling Results

Table II shows the profiling results in terms of minimum response time values at different number of used cores for four jobs j_1, j_2, j_3 and j_4 of the considered workload. Similar results are obtained for the remaining jobs in the workload. It can be observed that the response time decreases with the number of used cores growth due to higher parallelism exploitation. However, it becomes saturated after using a certain number of cores, which implies that the maximum parallelism has already been exploited and the response time is governed by sequential execution of a task belonging to the critical path. The $\times$ symbol indicates that the number of tasks in the job is lower than the number of cores and an allocation using that number of cores is not possible. Therefore, the number of used cores corresponding to the last valid entry for a job represents the number of tasks in the job, e.g., jobs j_1 and j_2 consist of 9 and 7 tasks, respectively. The profiling results also include the allocation information corresponding to each response time value and are stored along with the value curves to perform efficient dynamic resource allocation.

B. Overall profit by executing different number of jobs

Figure 6 shows the overall profit obtained by employing the proposed approaches for varying job sets, which are derived by choosing different number of jobs from the workload. A small number of jobs in a set reflects the execution of jobs in the HPC centre for a fixed small amount of time, e.g. few minutes or hours, but not for the time as with the whole historical data. This helps to achieve results without performing time consuming simulations. A couple of observations can be made from the Figure 6. *1)* The profit obtained by the approaches employing the holding process (e.g., maxVDH and maxVH) is always higher than that of the corresponding approaches without employing holding. This improvement is achieved by holding low value executing jobs and allocating the freed resources to high value incoming jobs. Since holding is performed only when it is profitable, the approaches employing holding achieve higher overall profit. *2)* The overall profit increases with the number of jobs in a workload as profit is made out of higher number of jobs. *3)* The maxVH approach achieves maximum overall profit for the considered job sets. This is due to the fact the choosing maximum value queued job leads to more favourable situations to maximize the profit by completing different jobs. On an average, maxVH achieves 8% higher profit than that of maxV, which can be a significant value when serving (completing) a large amount of jobs.

C. Analysing holding effect

Figure 7 shows the profit obtained by different jobs when approaches SJQ and SJQH are employed for the job set

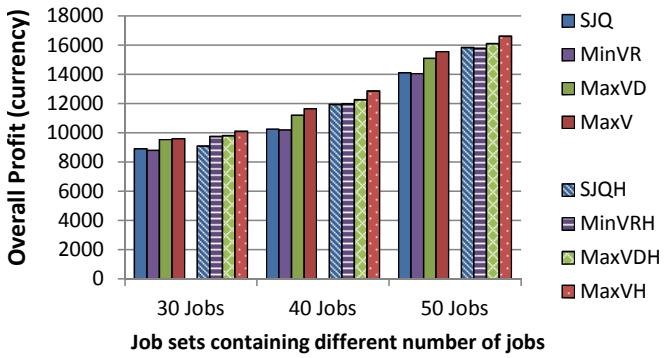

Fig. 6. Overall profit for executing different job sets.

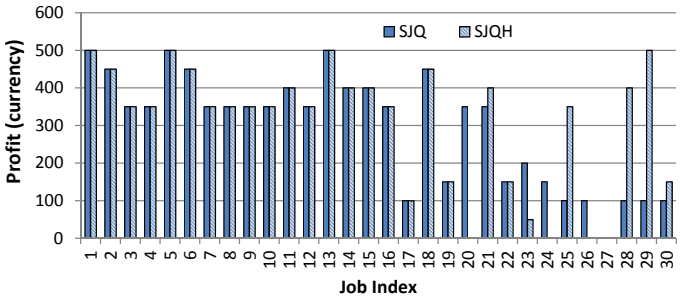

Fig. 7. Profit by different jobs.

containing 30 jobs. The interesting observations that can be made from the figure are as follows. *1)* SJQ makes profit for most of the jobs and jobs achieving zero profit are those that were queued and whose value decreased to zero when resources become available. An example of such zero profit making job is job 27. *2)* SJQH makes profit for lower number of jobs than that of SJQ as SJQH holds some jobs and makes zero profit for some of them. Examples of such jobs are job 20 and job 24. It should be noted that some zero profit making jobs by SJQH could be due to the same reason as that of SJQ, i.e. due to some queued jobs for whom profitable holding was not possible at their arrival and value for them becomes zero when resources become available. Similar results are obtained for other job sets.

D. Overall profit with varying holding penalty

We also have evaluated the overall achieved profit in case there is some penalty to hold the low value executing jobs. This will be more favourable situation for the customers as they know that most likely their submitted jobs will be serviced with the initial promised quality; otherwise the cluster managing agency has to pay them back in terms of some penalty. We have assumed that if a job is put on hold then there will be a penalty of some percentage of the maximum value that could be achieved for the job. However, for the queued jobs that can lead to zero profit making, we have not considered any penalty since the job is not put on hold but went to out of profit making point due to resources unavailability. The holding penalty percentage has been varied to evaluate its impact on the overall profit achieved by the most promising approach maxVH.

Figure 8 shows the overall profit obtained by employing maxVH when holding penalty percentage is varied from 0% to 70% for the job set containing 30 tasks. The overall profits obtained by employing maxVH with no holding penalty

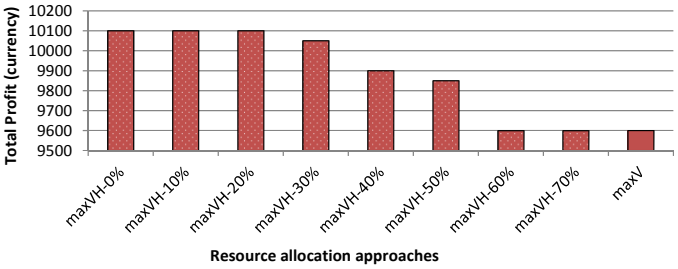

Fig. 8. Profit by maxVH with varying holding penalty and by maxV.

(maxVH-0%) and maxV have also been plotted for the comparison purposes. It can be observed that the overall profit by maxVH decreases as the holding penalty percentage increases and becomes saturated after a particular holding penalty percentage. The profit at lower penalties remains the same as the penalty is not sufficient enough to affect the allocation decisions for the jobs. The decreasing trend is obtained as lesser holdings are performed with increased penalty and thus making low overall profit. The later constant profit region indicates that no further holdings are performed due to high cost (penalty). It should be noted that the holding penalty also determines the holding decision and thus different jobs are put on hold with the changed penalty. Further, it can be observed that the overall profit with higher holding penalty is the same as that of maxV as the approach maxVH performs resource allocation exactly in the same manner as that of maxV, i.e., no jobs are put on hold due to high penalty and incoming job is put into the job queue for later allocation when resources become free.

VI. Conclusions and Future Work

We proposed a market-inspired dynamic resource allocation approach for many-core HPC systems. We show that the approach exploits design-time profiling results and employs the concept of holding low value executing jobs to free resources for high value arrived jobs towards maximizing the overall profit of the system. It has been shown that the exploitation of profiling results and holding concept leads to higher overall profit. In future, we plan to consider multi-criteria optimization, e.g. jointly optimizing for overall profit and energy consumption, which is desired for the high energy consuming HPC centres.

Acknowledgment

This work is funded by EU FP7 DreamCloud project under grant agreement no. 611411. We would like to thank our project partner at University of Stuttgart to provide us the industrial historic HPC workload in order to evaluate our proposed and existing heuristics.

References

[1] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends," in *Proceedings of ACM Design Automation Conference (DAC)*, 2013, pp. 1:1–1:10.

[2] W. Jiang and G. Agrawal, "MATE-CG: A Map Reduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters," in *Proceedings of IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2012, pp. 644–655.

[3] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the ACM/IEEE Conference Supercomputing*, 2004, pp. 47–47.

[4] S. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, 1981.

[5] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini, "The Meaning and Role of Value in Scheduling Flexible Real-time Systems," *J. Syst. Archit.*, vol. 46, no. 4, pp. 305–325, 2000.

[6] B. Khemka, R. Friese, L. Briceno, H. Siegel, A. Maciejewski, G. Koenig, C. Groer, G. Okonski, M. Hilton, R. Rambharos, and S. Poole, "Utility Functions and Resource Management in an Oversubscribed Heterogeneous Computing Environment," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2014.

[7] C. S. Yeo and R. Buyya, "A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing," *Softw. Pract. Exper.*, vol. 36, no. 13, pp. 1381–1419, 2006.

[8] O. O. Sonmez and A. Gursoy, "A Novel Economic-Based Scheduling Heuristic for Computational Grids," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 1, pp. 21–29, Feb. 2007.

[9] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, "Mapping of applications to MPSoCs," in *Proceedings of IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2011, pp. 109–118.

[10] A. K. Singh, A. Kumar, and T. Srikanthan, "A Hybrid Strategy for Mapping Multiple Throughput-constrained Applications on MPSoCs," in *Proceedings of ACM Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 175–184.

[11] S. K. Garg, C. S. Yeo, A. Anandasivam, and R. Buyya, "Environment-conscious Scheduling of HPC Applications on Distributed Cloud-oriented Data Centers," *J. Parallel Distrib. Comput.*, vol. 71, no. 6, pp. 732–749, 2011.

[12] J. Sun, X. Wang, K. Li, C. Wu, M. Huang, and X. Wang, "An Auction and League Championship Algorithm Based Resource Allocation Mechanism for Distributed Cloud," in *Advanced Parallel Processing Technologies*. Springer, 2013, pp. 334–346.

[13] T. Theocharides, M. K. Michael, M. Polycarpou, and A. Dingankar, "Hardware-enabled Dynamic Resource Allocation for Manycore Systems Using Bidding-based System Feedback," *EURASIP J. Embedded Syst.*, vol. 2010, pp. 3:1–3:21, 2010.

[14] K. Lai, "Markets Are Dead, Long Live Markets," *SIGecom Exch.*, vol. 5, no. 4, pp. 1–10, 2005.

[15] D. P. Bertsekas, "Auction algorithms for network flow problems: A tutorial introduction," *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, 1992.

[16] C. D. Locke, "Best-effort Decision-making for Real-time Scheduling," Ph.D. dissertation, Pittsburgh, PA, USA, 1986, aAI8702895.

[17] P. Li and B. Ravindran, "Fast, Best-Effort Real-Time Scheduling Algorithms," *IEEE Trans. Comput.*, vol. 53, no. 9, pp. 1159–1175, 2004.

[18] N. Bansal and K. R. Pruhs, "Server Scheduling to Balance Priorities, Fairness, and Average Quality of Service," *SIAM J. Comput.*, vol. 39, no. 7, pp. 3311–3335, 2010.

[19] S. Aldarmi and A. Burns, "Dynamic value-density for scheduling real-time systems," in *Proceedings of the Euromicro Conference on Real-Time Systems*, 1999, pp. 270–277.

[20] A. M. Burkimsher, "Fair, responsive scheduling of engineering workflows on computing grids," Ph.D. dissertation, UK, 2014.

[21] D. A. Lifka, "The ANL/IBM SP Scheduling System," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pp. 295–303.

[22] A. K. L. Wong and A. M. Goscinski, "Evaluating the EASY-backfill Job Scheduling of Static Workloads on Clusters," in *Proceedings of IEEE International Conference on Cluster Computing*, 2007, pp. 64–73.

[23] D. E. Irwin, L. E. Grit, and J. S. Chase, "Balancing Risk and Reward in a Market-Based Task Service," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2004, pp. 160–169.

[24] K. Chen and P. Muhlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function," *Real-Time Syst.*, vol. 10, no. 3, pp. 293–312, 1996.

[25] M. Sayuti and L. Indrusiak, "Real-time low-power task mapping in Networks-on-Chip," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2013, pp. 14–19.