Abstract—Scheduling in High-Performance Computing (HPC) has been traditionally centered around computing resources (e.g., processors/cores). The ever-growing amount of data produced by modern scientific applications start to drive novel architectures and new computing frameworks to support more efficient data processing, transfer and storage for future HPC systems. This trend towards data-driven computing demands the scheduling solutions to also consider other resources (e.g., I/O, memory, cache) that can be shared amongst competing applications. In this paper, we study the problem of scheduling HPC applications while exploring the availability of multiple types of resources that could impact their performance. The goal is to minimize the overall execution time, or makespan, for a set of moldable tasks under multiple-resource constraints. Two scheduling paradigms, namely, list scheduling and pack scheduling, are compared through both theoretical analyses and experimental evaluations. Theoretically, we prove, for several algorithms falling in the two scheduling paradigms, tight approximation ratios that increase linearly with the number of resource types. As the complexity of direct solutions grows exponentially with the number of resource types, we also design a strategy to indirectly solve the problem via a transformation to a single-resource-type problem, which can significantly reduce the algorithms’ running times without compromising their approximation ratios. Experiments conducted on Intel Knights Landing with two resource types (processor cores and high-bandwidth memory) and simulations designed on more resource types confirm the benefit of the transformation strategy and show that pack-based scheduling, despite having a worse theoretical bound, offers a practically promising and easy-to-implement solution, especially when more resource types need to be managed.

I. INTRODUCTION

Scientific discovery now relies increasingly on data and its management. Large-scale simulations produce an ever-growing amount of data that has to be processed and analyzed. As an example, it was estimated that the Square Kilometer Array (SKA) could generate an exabyte of raw data a day by the time it is completed [3]. In the past, scheduling in High-Performance Computing (HPC) has been mostly compute-based (with a primary focus on the processor/core resource) [31]. In general, data management is deferred as the time it is completed [3]. Both scheduling paradigms have been studied by the literature, mostly under a single-resource constraint (see Section II for a literature review). Figure I shows an example of applying both scheduling paradigms to a same set of tasks under a single-resource constraint.

Compared to some more sophisticated scheduling algorithms that could guarantee better theoretical bounds (e.g., [22], [18]), list-based and pack-based algorithms often produce simple yet efficient schedules that can be easily implemented by practical runtime systems. Furthermore, these algorithms can be adopted to the online and/or heterogeneous scheduling environment with minimal change, thus offering more general applicability to a wide range of scheduling scenarios. Between these two paradigms, list scheduling can make better use of the available resources (at least in theory) by reducing the idle times between the tasks and thus maximizing the resource utilization. However, pack scheduling has been advocated by some recent studies [2], [27] for being practically valuable due to its ease of implementation (as batch processing) and for incurring less scheduling overhead. The
relative merits of the two scheduling paradigms are yet to be evaluated under multiple-resource constraints.

In this paper, we provide a comprehensive study of the two scheduling paradigms through both theoretical analyses and experimental evaluations in the presence of multiple resource types. In this context, we make the following contributions:

- We present several scheduling algorithms under multiple-resource constraint in both list and pack scheduling paradigms, and prove tight approximation ratios for these algorithms (2d for list and 2d + 1 for pack, where d is the number of resource types).
- We design a transformation strategy that reduces the problem of scheduling multiple resource types to single-resource-type scheduling, thereby significantly reducing the algorithms’ running times without compromising the approximation ratios.
- We conduct experimental evaluations of these scheduling algorithms on the Intel Xeon Phi Knights Landing (KNL) processor, which offers two resource management options (processor cores and high-bandwidth memory).
- We project the performance of both list- and pack-based scheduling solutions under more than two resource types using simulations on synthetic parallel workloads that extend some classical speedup profiles.

Overall, the experimental/simultion results confirm the benefit of the transformation strategy and show that pack scheduling, despite having a slightly worse theoretical bound, can indeed offer a practically promising yet easy-to-implement solution compared to its list counterpart, especially when more resource types need to be managed. The insights derived from these results will be especially relevant to the emerging architectures that will likely offer in the resource management systems more scheduling possibilities across the memory hierarchy to embrace the shift towards data-driven computing.

The rest of this paper is organized as follows. Section II reviews some related work. Section III formally defines the multiple-resource-type scheduling model. Section IV presents several scheduling solutions and proves their approximation ratios. Section V is devoted to experimental evaluation of different algorithms and heuristics. Finally, Section VI concludes the paper with hints on future directions.

II. RELATED WORK

A. Parallel Task Models

Many parallel task models exist in the scheduling literature. Feitelson [8] classified parallel tasks into three categories, namely, rigid tasks, moldable tasks and malleable tasks. A rigid task requires a fixed amount of resources (e.g., number of processors) to execute. A moldable task can be executed with a varying amount of resources, but once the task has started the resource allocation cannot be changed. The most flexible model is the one of malleable tasks, which allows the amount of resources executing the task to vary at any time during the execution. For the latter two models, two important parameters are the speedup function and the total area (or work). The speedup function relates the execution time of a task to the amount of resources allocated to it, and the total area is defined as the product of execution time and resource allocation. Many prior works [2], [5], [22] have assumed that the speedup of a task is a non-decreasing function of the amount of allocated resources (hence the execution time is a non-increasing function) and that the total area is a non-decreasing function of the allocated resources. One example is the well-known Amdahl’s law [1], which specifies the speedup of executing a parallel task with s sequential fraction using p processors as $1/(s + (1-s)/p)$. Another example used by some scheduling literature [4], [25], [14] is the speedup function $p^\alpha$, where p represents the amount of allocated resources to the task and $\alpha \leq 1$ is a constant. In particular, this function has been observed for processor resources when executing parallel matrix operations [24] as well as for characterizing the cache behaviors in terms of the miss rate of data access (which is directly related to the execution time) [15]. We refer to this function as the power law.

All works above apply to the case with a single resource type, and, to the best of our knowledge, no explicit speedup model is known to include multiple resource types. In Section V, we extend these speedup functions to construct synthetic parallel tasks with multiple-resource demands. We also perform a profiling study of the Stream benchmark [21] on a recent Intel Xeon Phi architecture that includes two resource types (processor core and high-bandwidth memory).

B. Parallel Task Scheduling

Scheduling a set of independent parallel tasks to minimize the makespan is known to be strongly NP-complete [11], thus much attention has been directed at designing approximation or heuristic algorithms. Most prior works focused on allocating a single resource type (i.e., processors) to the tasks, and the constructed schedules are either pack-based (also called shelf-based or level-based) or list-based.

Approximation Algorithms: Scheduling rigid tasks with contiguous processor allocation can be considered as a rectangle packing or 2D-strip packing problem. For this problem, Coffman et al. [6] showed that the Next-Fit Decreasing Height (NFDH) algorithm is 3-approximation and the First-Fit Decreasing-Height (FFDH) algorithm is 2.7-approximation. Both algorithms pack rectangles onto shelves, which are equivalent to creating pack-based schedules. The first result (i.e., 3-approximation) has also been extended to the case of
moldable task scheduling [2], [30]. Turek et al. [30] presented a strategy to extend any algorithm for scheduling rigid tasks into an algorithm for scheduling moldable tasks in polynomial time. The strategy preserves the approximation ratio provided that the makespan for the rigid-task problem satisfies certain conditions. The complexity of such an extension was improved in [20] with possibly a worse schedule than the one obtained in [30] but without compromising the approximation ratio. Thanks to these strategies, a 2-approximation algorithm was devised for moldable tasks based on list scheduling [30], [20], extending the same ratio previously known for rigid tasks [30], [10]. Using dual-approximation techniques, Mounie et al. [22] presented a 1.5-approximation algorithm for moldable tasks while assuming that the total area of a task is a non-decreasing function of the allocated processors. Also, for moldable tasks, Jansen and Porkolab [18] presented, for any $\epsilon > 0$, a $(1 + \epsilon)$-approximation scheme when the number of processors is a fixed constant.

While all results above are for scheduling under a single resource type, only a few papers have considered scheduling under multiple resource types. Garey and Graham [10] proved that a simple list-scheduling algorithm is $(d + 1)$-approximation for rigid tasks, where $d$ is the number of resource types. He et al. [17] proved the same asymptotic result for scheduling a set of malleable jobs, each represented as a direct acyclic graph (DAG) of unit-size tasks. For moldable tasks, Shachnai and Turek [26] presented a technique to transform a $c$-approximation algorithm on a single resource type to a $c \cdot d$-approximation algorithm on $d$ types of resources. Partially inspired by these results, this paper presents techniques and algorithms with improved approximations for pack scheduling and new algorithms for list scheduling under multiple resource types.

**Heuristics:** Some papers have proposed heuristic solutions for multiple-resource-type scheduling under various different models and objectives. Levinberga et al. [19] considered rigid tasks and proposed two heuristics that attempt to balance the usage of different resource types through backfilling strategies. He et al. [16] studied online scheduling of DAG-based jobs and proposed a multi-queue balancing heuristic to balance the sizes of the queues that contain ready tasks under different resource types. Ghodsi et al. [12] proposed Dominant Resource Fairness (DRF), a multiple-resource-type scheduling algorithm for user tasks with fixed resource demands. It aims at ensuring the resource allocation fairness among all users by identifying the dominant resource share for each user and maximizing the minimum dominant share across all users. Grundl et al. [13] considered scheduling malleable tasks under four resource types (CPU, memory, disk, and network), and designed a heuristic, called Tetris, that packs tasks to a cluster of machines. This is similar to pack-based scheduling but instead of creating packs in time it creates them in space. Tetris works by selecting a task with the highest correlation between the task’s peak resource demands and the machine’s resource availabilities, with the aim of minimizing resource fragmentation. NoroozOliae et al. [23] considered a similar cluster scheduling problem but with two resources only (CPU and memory). They showed that the simple Best Fit plus Shortest Task First scheduling outperforms other packing heuristics (e.g., First Fit, FCFS) in terms of resource utilization and task queueing delays.

### III. Model

This section presents a formal model of the scheduling problem, which we call $d$-RESOURCE-SCHEDULING. Suppose a platform has $d$ different types of resources subject to allocation (e.g., processor, memory, cache). For each resource type $i$, there is a total amount $P^{(i)}$ of resources available. Consider a set of $n$ independent tasks (or jobs), all of which are released at the same time on the platform, so the problem corresponds to scheduling a batch of applications in HPC environment. For each task $j$, its execution time $t_j(p_j)$ is a function of the resource allocation vector $p_j = (p_j^{(1)}, p_j^{(2)}, \ldots, p_j^{(d)})$, where $p_j^{(i)}$ denotes the amount of the $i$-th resource allocated to task $j$. In reality, the execution times are typically obtained by application profiling or interpolation and curve-fitting from historic data. Suppose $p_j$ and $q_j$ are two resource allocation vectors, and we define $p_j \preceq q_j$ if $p_j^{(i)} \leq q_j^{(i)}$ for all $1 \leq i \leq d$. Here, the execution time is assumed to be a non-increasing function of the resource allocation, i.e., $t_j(p_j) \preceq t_j(q_j)$. This means that increasing the allocation of any one type of resource without decreasing the others will not increase the execution time.\footnote{This assumption is not restrictive, as we can discard an allocation $q_j$ that satisfies $p_j \preceq q_j$ and $t_j(p_j) < t_j(q_j)$. This is because any valid schedule that allocates $q_j$ to task $j$ can be replaced by a valid schedule that allocates $p_j$ to the task without increasing the execution time, thus rendering $q_j$ useless.} We also assume that the resource allocations $p_{j'}^{(i)}$’s and the total amount of resources $P^{(i)}$’s are all integer values. This holds naturally true for discrete resources such as processors\footnote{We do not allow fractional processor allocation (typically realized by timesharing a processor among several tasks) as assumed by some prior work.}. For other resources (e.g., memory, cache), it can be justified as most practical resource management systems allocate the resources in discrete chunks (e.g., memory blocks, cache lines). Note that some tasks may not require all resource types in order to execute, hence its resource allocation $p_j$ for a particular resource can be zero, in which case the execution time remains validly defined. In contrast, other tasks may require a minimum amount of certain resource in order to execute. In this case, the execution time can be defined as infinity for any amount of resource below this threshold. The model is flexible to handle both scenarios.

In this paper, we focus on moldable task scheduling [8], where the amount of resources allocated to a task can be freely selected by the scheduler at launch time but they cannot be changed after the task has started the execution. Moldable task scheduling strikes a good balance between practicality and performance by incurring less overhead than malleable task scheduling yet achieving more flexible task executions than rigid task scheduling. For the $d$-RESOURCE-SCHEDULING problem, the scheduler needs to decide, for each task $j$, a resource allocation vector $p_j$ along with a starting time $s_j$. At any time $t$, a task is said to be active if it has started but not yet completed, i.e., $t \in [s_j, s_j + t_j(p_j))$. Let $J_t$ denote the set of active tasks at time $t$. For a solution to be valid, the resources used by the active tasks at any time should not exceed the total amount of available resources for each resource type, i.e., $\sum_{j \in J_t} p_j^{(i)} \leq P^{(i)}$ for all $i$ and $t$. The objective is to minimize the maximum completion time, or the makespan, of all tasks, i.e., $T = \max_{j \in J_t}(s_j + t_j(p_j))$.

Since $d$-RESOURCE-SCHEDULING is a generalization of classical makespan minimization problem with a single re-
source type, it is strongly NP-complete. Hence, we are interested in designing approximation and heuristic algorithms. In this paper, we will focus on and compare two major scheduling paradigms, namely, list scheduling and pack scheduling. A scheduling algorithm $S$ is said to be $c$-approximation if its makespan satisfies $T_{S} \leq c \cdot T_{OPT}$ for any instance, where $T_{OPT}$ denotes the makespan by an optimal moldable scheduler. Note that the optimal scheduler needs not be restricted to either list scheduling or pack scheduling.

### IV. THEORETICAL ANALYSIS

In this section, we present polynomial-time algorithms for the $d$-RESOURCE-SCHEDULING problem under both list- and pack-scheduling paradigms, and we prove tight approximation ratios for these algorithms.

#### A. Preliminaries

We start with some preliminary definitions that will be used throughout the analysis. Table I provides a summary of the main notations used.

<table>
<thead>
<tr>
<th>For platform</th>
<th>$d$</th>
<th>Number of resource types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P^{(i)}$</td>
<td>Total amount of $i$-th type of resource ($i = 1, \ldots, d$)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For any task $j$</th>
<th>$p_{j}^{(i)}$</th>
<th>Amount of $i$-th type of resource allocated to task $j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{p}_{j}$</td>
<td>Resource allocation vector of task $j$</td>
<td></td>
</tr>
<tr>
<td>$t_{j}(\vec{p}_{j})$</td>
<td>Execution time of task $j$ with vector $\vec{p}_{j}$</td>
<td></td>
</tr>
<tr>
<td>$a_{j}(\vec{p}_{j})$</td>
<td>Area of task $j$ with vector $\vec{p}_{j}$</td>
<td></td>
</tr>
<tr>
<td>$s_{j}$</td>
<td>Starting time of task $j$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For set of all tasks</th>
<th>$n$</th>
<th>Number of all tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{p}$</td>
<td>Resource allocation matrix for all tasks</td>
<td></td>
</tr>
<tr>
<td>$A(\vec{p})$</td>
<td>Total area of all tasks with matrix $\vec{p}$</td>
<td></td>
</tr>
<tr>
<td>$t_{max}(\vec{p})$</td>
<td>Maximum execution time of all tasks with matrix $\vec{p}$</td>
<td></td>
</tr>
<tr>
<td>$T(\vec{p})$</td>
<td>Makespan of all tasks with matrix $\vec{p}$</td>
<td></td>
</tr>
<tr>
<td>$L(\vec{p}, s)$</td>
<td>Maximum of $A(\vec{p})/s$ and $t_{max}(\vec{p})$ for any $s &gt; 0$</td>
<td></td>
</tr>
<tr>
<td>$L_{min}(s)$</td>
<td>Minimum $L(\vec{p}, s)$ over all matrix $\vec{p}$</td>
<td></td>
</tr>
</tbody>
</table>

### Definition 1

Define $\vec{p} = (p_{1}, p_{2}, \ldots, p_{n})^{T}$ to be a resource allocation matrix for all tasks, where $p_{i} = (p_{i}^{(1)}, p_{i}^{(2)}, \ldots, p_{i}^{(d)})$ is a resource allocation vector for task $j$.

### Definition 2

Given a resource allocation matrix $\vec{p}$ for the $d$-RESOURCE-SCHEDULING problem, we define the following:

- For task $j$, $a_{j}(\vec{p}_{j}) = \sum_{i=1}^{d} p_{j}^{(i)}$: $t_{j}(\vec{p}_{j})$ is the task's area$^3$;
- $A(\vec{p}) = \sum_{j=1}^{n} a_{j}(\vec{p}_{j})$ is the total area of all tasks;
- $t_{max}(\vec{p}) = \max_{j} t_{j}(\vec{p}_{j})$ is the maximum execution time of all tasks;
- $L(\vec{p}, s)$ = max \ $(\frac{A(\vec{p})}{s}, t_{max}(\vec{p}))$ for any $s > 0$.

The last quantity $L(\vec{p}, s)$ is related to the lower bound of the makespan. In particular, we define

$$L_{min}(s) = \min_{\vec{p}} L(\vec{p}, s),$$

and we will show later that it is a lower bound on the makespan when $s$ is set to be $d$.

In the following, we will present two general techniques for constructing the scheduling solutions. The first technique is a two-phase approach, similarly to the one considered in scheduling under a single resource type [20]:

- **Phase 1**: Determines a resource allocation matrix for all the tasks;
- **Phase 2**: Constructs a rigid schedule based on the fixed resource allocation of the first phase.

The second technique is a transformation strategy that reduces the $d$-RESOURCE-SCHEDULING problem to the $1$-RESOURCE-SCHEDULING problem, which is then solved and whose solution is transformed back to the original problem.

Section IV-B presents a resource allocation strategy (Phase 1). Section IV-C presents a transformation strategy, followed by rigid task scheduling schemes (Phase 2) in Section IV-D under both pack and list scheduling. Finally, Section IV-E puts these different components together and presents several complete scheduling solutions.

#### B. A Resource Allocation Strategy

This section describes a resource allocation strategy for the first phase of the scheduling algorithm. It determines the amounts of resources allocated to each task, which are then used to schedule the tasks in the second phase as a rigid task scheduling problem.

The goal is to find efficiently a resource allocation matrix that minimizes $L(\vec{p}, s)$, i.e.,

$$p_{s}^{*} = \arg\min_{\vec{p}} L(\vec{p}, s).$$

Let $P = \prod_{i=1}^{d} (P^{(i)} + 1)$ denote the number of all possible allocations for each task, including the ones with zero amount of resource under a particular resource type. A straightforward implementation takes $O(P^{n})$ time, which grows exponentially with the number of tasks $n$. Algorithm 1 presents a resource allocation strategy $RA_{d}(s)$ that achieves this goal in $O(nP\log P + \log n + d)$ time. Note that when the amounts of resources under different types are in the same order, i.e., $P^{(i)} = O(P^{(i+1)})$ for all $i = 1, \ldots, d$, where $P_{max} = \max_{i} P^{(i)}$, the running time grows exponentially with the number of resource types $d$, i.e., the complexity contains $O(P^{d})$. Since the number of resource types is usually a small constant (less than 4 or 5), the algorithm runs in polynomial time under most realistic scenarios.

The algorithm works as follows. First, it linearizes and sorts all $P$ resource allocation vectors for each task and eliminates any allocation that results in both a higher execution time and a larger total area (Lines 2-17), for such a vector can be replaced by another one that leads to the same or smaller $L_{min}(s)$ (Equation (1)). Each task then ends up with an array of admissible allocations in increasing order of execution time and decreasing order of total area. The complexity for this part is $O(n(P\log P + Pd))$, which is dominated by sorting each task’s allocations and computing its area. Then, the algorithm goes through all possible $t_{max}$, i.e., the maximum execution time, from the remaining admissible allocations, and for each $t_{max}$ considered, it computes the minimum total area $A$ and hence the associated $L$ value (Lines 18-40). This is achieved by maintaining the tasks in a priority queue with their longest execution times as priorities and updating the queue with a task’s new priority when its next longest execution time is considered. The algorithm terminates either when at least one task has exhausted its admissible allocations, in which case the priority queue will have fewer than $n$ tasks (Line 20), or when $L$ becomes dominated by $\frac{A}{s}$ (Line 30), since the total area only increases and the maximum execution time decreases.
Algorithm 1: Resource Allocation Strategy RA_d(s)

Input: Set of n tasks, execution time t_j(\vec{p}) ∀ j, \vec{p} and resource limit P(\vec{p}) ∀ i.

Output: Resource allocation matrix p_{min} = \arg \min_{p} L(p, s).

begin
1. P = \prod_{i=1}^{P(\vec{p}+1)};
2. A = 0;
3. for j = 1 to n do
4. Linearize all P resource allocation vectors for task j in an array \text{res_alloc} and sort it in non-decreasing order of task execution time;
5. \text{admissible_allocations}_j \leftarrow \text{list}(())
6. \text{min_area} \leftarrow \infty;
7. for h = 1 to P do
8. \vec{p} \leftarrow \text{res_alloc}(h);
9. if \vec{a}(\vec{p}) < \text{min_area} then
10. \text{min_area} \leftarrow \vec{a}(\vec{p});
11. \text{admissible_allocations}_j, \text{append}(\vec{p});
12. end
13. end
14. \vec{p}_j \leftarrow \text{admissible_allocations}_j, \text{last_element}();
15. A \leftarrow A + \vec{a}(\vec{p}_j);
16. Build a priority queue Q of n tasks with their longest execution time \vec{t}_j(\vec{p}_j)'s as priorities;
17. T_{\text{min}} \leftarrow \infty;
18. while Q.size() > 0 do
19. k \leftarrow Q.highest_priority_element();
20. \vec{p}_k \leftarrow \text{admissible_allocations}_k, \text{last_element}();
21. t_{\text{max}} = t_k(\vec{p}_k);
22. L = \max \{ A \cdot t_{\text{max}} \}
23. if L < T_{\text{min}} then
24. T_{\text{min}} \leftarrow L;
25. p_{\text{min}} \leftarrow (\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_n)^T;
26. end
27. if A ≥ t_{\text{max}} then
28. break;
29. end
30. \text{admissible_allocations}_k, \text{pop_last}();
31. Q.remove_highest_priority_element();
32. if \text{admissible_allocations}_k, \text{is_nonempty}() then
33. \vec{p} \leftarrow \text{admissible_allocations}_k, \text{last_element}();
34. A \leftarrow A + \vec{a}(\vec{p}) - \vec{a}(\vec{p}_k);
35. Q.insert_element(k) with priority t_k(\vec{p})
36. end
end
end

end

The inequality is because \text{P}(i) \cdot T(p) is the maximum volume for resource type i that can be allocated to all the tasks within a total time of T(p).

C. A Transformation Strategy

This section describes a strategy to indirectly solve the d-RESOURCE-SCHEDULING problem via a transformation to the 1-RESOURCE-SCHEDULING problem.

Algorithm 2 presents the transformation strategy TF, which contains three steps. First, any instance I of the d-RESOURCE-SCHEDULING problem is transformed to an instance I’ of the 1-RESOURCE-SCHEDULING problem. Then, I’ is solved by any moldable task scheduler under a single resource type. Lastly, the solution obtained for I’ is transformed back to obtain a solution for the original instance I.

Now, we show that the resource allocation matrix p_{min} returned by Algorithm 1 while setting the parameter s to be d can be used by a rigid task scheduling strategy in the second phase to achieve good approximations.

Theorem 1. If a rigid task scheduling algorithm R_d that uses the resource allocation matrix p_{min} obtained by the strategy RA_d(d) produces a makespan

\[ T_{R_d}(p_{min}) \leq c \cdot L_{min}(d), \]

then the two-phase algorithm RA_d(d) + R_d is c-approximation for the d-RESOURCE-SCHEDULING problem.

To prove the above theorem, let us define p_{opt} to be the resource allocation matrix of an optimal schedule, and let T_{opt'} denote the corresponding optimal makespan. We will show in the following lemma that L_{min}(d) serves as a lower bound on T_{opt'}. Then, Theorem 1 follows directly, since T_{RA_d(d)+R_d} = T_{RA}(p_{min}) \leq c \cdot L_{min}(d) \leq c \cdot T_{opt'}.

Lemma 1. T_{opt} \geq L_{min}(d).

Proof. We will prove that, given a resource allocation matrix p, the makespan produced by any rigid task scheduler using p must satisfy T(p) ≥ max t_{\text{max}}(p) and T(p) ≥ 4/3 OPT. Thus, for the optimal schedule, which uses p_{opt}, we have T_{opt} ≥ max (t_{\text{max}}(p_{opt})) \cdot 4/3 = L(p_{opt}, d) ≥ L_{min}(d). The last inequality is because L_{min}(d) is the minimum L(p, d) among all possible resource allocations including p_{opt} (Equation 1).

The first bound T(p) ≥ max t_{\text{max}}(p) is trivial since the makespan of any schedule should be at least the execution time of the longest task. For the second bound, we have, in any valid schedule with makespan T(p), that:

\[ A(p) = \sum_{i=1}^{n} \sum_{j=1}^{d} p_{i,j} \cdot t_j(\vec{p}_j) \]

\[ = \sum_{i=1}^{d} \frac{1}{P(i)} \cdot \sum_{j=1}^{n} p_{i,j} \cdot t_j(\vec{p}_j) \]

\[ \leq \sum_{i=1}^{d} \frac{1}{P(i)} \cdot P(i) \cdot T(p) \]

\[ = d \cdot T(p). \]

The inequality is because P(i) \cdot T(p) is the maximum volume for resource type i that can be allocated to all the tasks within a total time of T(p). \qed

C. A Transformation Strategy
Algorithm 2: Transformation Strategy TF

Input: Set of n tasks, execution time $t_j(p_i) \forall j, p_i$. resource limit $P^{(i)}_j \forall j$
Output: Starting time $s_j$ and resource allocation vector $\vec{p}_j \forall j$
1) Transform d-RESOURCE-SCHEDULING instance I to 1-RESOURCE-SCHEDULING instance $I'$:
   • $I'$ has the same number $n$ of tasks as I;
   • The resource limit for the only resource of $I'$ is $Q = \frac{\text{lcm}}{n} P^{(i)}$;
   • For any task $j'$ in $I'$, its execution time with any amount of resource $q \in [0, 1, 2, \ldots, Q]$ is defined as $t_j(q) = t_j\left(\left(\frac{q P^{(i)}}{Q}\right)_{i=1}^n\right)$

2) Solve the 1-RESOURCE-SCHEDULING instance $I'$;
3) Transform 1-RESOURCE-SCHEDULING solution $S'$ back to d-RESOURCE-SCHEDULING solution $S$:
   • For any task $j$ in $I$, its starting time is $s_j = s_{j'}$, where $s_{j'}$ is the starting time of task $j'$ in $I'$ and its resource allocation vector is $\vec{p}_{j'} = (\left(\frac{q P^{(i)}}{Q}\right)_{i=1}^n)$, where $q_j$ is the allocation of the only resource for task $j'$ in $I'$.

Consider any resource allocation $p = (\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n)^T$ for $I$. For each task $j$, let $k_j$ denote its dominating resource type in terms of the proportion of resource used, i.e., $k_j = \arg\min_q \frac{p_j^{(i)}}{Q}$. We construct $q(p) = (q_1, q_2, \ldots, q_n)^T$ for $I'$ by setting, for each task $j'$, an allocation $q_j = \frac{p_j^{(i)}}{Q}$. As $P(k_j)$ divides $Q$, $q_j$ is an integer and hence a valid allocation.

According to Step (1) of the transformation, the execution time of task $j'$ in $I'$ satisfies $t_j'(q_j) = t_j\left(\left(\frac{p_j^{(i)}}{Q}\right)_{i=1}^n\right) \leq t_j(\vec{p}_j)$. The inequality is because $p_j^{(i)} \leq \frac{p_j^{(i)}}{Q} \forall i$, which we get by the choice of $k_j$ and the integrality of $p_j^{(i)}$, and because we assumed that the execution time is a non-increasing function of each resource allocation (in Section III). As a result, the area of each task $j'$ in $I'$ also satisfies

$$a_j'(q_j) = q_j Q t_j'(q_j) \leq a_j(k_j) t_j(\vec{p}_j) = \sum_{i=1}^n a_j(p_j) Q = A(p).$$

Hence, the maximum execution times and the total areas for the two instances $I$ and $I'$ satisfy

$$t_{\max}(q(p)) = \max_j t_j'(q_j') \leq t_{\max}(p) = t_{\max}(\vec{p}_j).$$

and $A'(q(p)) = \sum_j a_j'(q_j') \leq \sum_j a_j(p_j) = A(p)$. This leads to $L'(q(p), d) = \max \left(\frac{A'(q(p))}{d}, t_{\max}(q(p))\right) \leq \max \left(\frac{A(p)}{d}, t_{\max}(p)\right) = L(p, d)$.  

(Proof of Theorem 2). Based on the above results, we can derive:

$$T_{TF+M_1} = T'_{M_1},$$

where $T'_{M_1}$ is the makespan of the scheduled $M_1$, which combines the transformation strategy $\text{TF}$ and the scheduler $M_1$ for a single resource type.

**Lemma 2.** Any valid solution by scheduler $M_1$ for $I'$ transforms to a valid solution by algorithm $\text{TF + M}_1$ for $I$, and $T_{TF+M_1} = T'_{M_1}$.

**Proof.** According to Steps (1) and (3) of the transformation, the execution time for each task in $I$ is the same as that of $j'$ in $I'$, i.e., $t_j(\vec{p}_j) = t_j'(q_j')$. Hence, the makespan is equal for both $I$ and $I'$, since the corresponding tasks also have the same starting time, i.e., $s_j = s_{j'}$. If the schedule for $I'$ is valid, i.e., $\sum_{j' \in I'} q_j' \leq Q \forall t$, then we have $\sum_{j' \in I'} p_j^{(i)} = \sum_{j \in I} q_j P^{(i)} Q \leq P^{(i)} \sum_{j' \in I'} q_j' \leq P^{(i)} \forall i, t$, rendering the schedule for $I$ valid as well.

The following lemma relates the makespan lower bound of the transformed instance to that of the original instance.

**Lemma 3.** $L_{\min}'(d) \leq L_{\min}(d)$.

**Proof.** We will show that given any resource allocation $p$ for $I$, there exists an allocation $q(p)$ for $I'$ such that $L'(q(p), d) \leq L(p, d)$. Thus, for $p_{\min}$ that leads to $L_{\min}(d)$, there is a corresponding $q(p_{\min})$ that satisfies $L'(q(p_{\min}), d) \leq L(p_{\min}, d) = L_{\min}(d)$, and therefore $L_{\min}(d) = \min_p L'(q, d) \leq L'(q_{\min}, d) \leq L_{\min}(d)$.

D. Rigid Task Scheduling

This section presents strategies to schedule rigid tasks under fixed resource allocations. The strategies include both list-based and pack-based scheduling. The results extend the theoretical analyses of single-resource-type scheduling [30], [2] to account for the presence of multiple resource types.

1) List Scheduling: We first present a list-based scheduling strategy for a set of rigid tasks under multiple resource types. A list-based algorithm arranges the set of tasks in a list. At any moment starting from time 0 and whenever an existing task completes execution and hence releases resources, the algorithm scans the list of remaining tasks in sequence and schedules the first one that fits, i.e., there is sufficient amount of resource to satisfy the task under each resource type. Algorithm 3 presents the list scheduling strategy $LS_d$ with $d$ resource types, and it extends the algorithm presented in [30] for scheduling under a single resource type. The complexity of the algorithm is $O(n^d)$, since scheduling each task incurs a cost of $O(nd)$ by scanning the taskList and updating the resource availability for the times in sortedTimeList.

The following lemma shows the performance of this list scheduling scheme.

**Lemma 4.** For a set of rigid tasks with a resource allocation matrix $p$, the list scheduling algorithm $LS_d$ achieves, for any
Algorithm 3: List Scheduling Strategy LS\(_{d}\)

**Input:** Resource allocation matrix \(p\) for the tasks and resource limit \(p^{(i)} \forall i\)

**Output:** Starting time \(s_j \forall j\)

1. Arrange the tasks in a list `taskList`;
2. `sortedTimeList` ← \(\{\}\);
3. while `taskList` nonempty() do
   1. \(t = \text{sortedTimeList}.pop_{\text{first}}();\)
   2. for \(j = 1\) to `taskList.size()` do
      1. if task \(j\) fits at time \(t\) then
         1. Schedule task \(j\) at time \(t\), i.e., \(s_j = t;\)
         2. Update available resources for all times before \(s_j + t\(p_j)\);
   end
end

**Algorithm 4: Packing Scheduling Strategy PS\(_d\)**

**Input:** Resource allocation matrix \(p\) for the tasks and resource limit \(p^{(i)} \forall i\)

**Output:** Set of packs \(\{B_1, B_2, \ldots\}\) containing the tasks and starting times \(s_m\) for each pack \(B_m\)

1. Sorted the tasks in non-increasing order of execution time \(t_j(p_j);\)
2. \(m \leftarrow 1;\)
3. \(B_1 \leftarrow \emptyset;\)
4. \(S_1 \leftarrow 0;\)
5. for \(j = 1\) to \(n\) do
   1. if task \(j\) fits in pack \(B_m\), then
      1. \(B_m \leftarrow B_m \cup \{j\};\)
   else
      1. \(m \leftarrow m + 1;\)
      2. \(B_m \leftarrow \{j\};\)
      3. \(S_m \leftarrow S_{m-1} + \max_{j \in B_{m-1}} t_j(p_j);\)
   end
end

Parameter \(s \geq 1\), a makespan

\[T_{LS_d}(p) \leq 2s \cdot L(p, s) . \tag{5}\]

**Proof.** We will prove in the following that \(T_{LS_d}(p) \leq 2s \cdot \max(t_{\text{max}}(p), A(p))\). For any \(s \geq 1\), it will then lead to \(T_{LS_d}(p) \leq 2s \cdot \max(t_{\text{max}}(p), A(p)) = 2s \cdot L(p, s)\).

First, suppose the makespan satisfies \(T_{LS_d}(p) \leq 2t_{\text{max}}(p)\), then the claim holds trivially. Otherwise, we will show \(T_{LS_d}(p) \leq 2A(p)\), thus proving the claim. To this end, consider any time \(t \in [0, T_{LS_d}(p)]\), and define \(t' = t + \frac{T_{LS_d}(p)}{2}\).

Since \(t_{\text{max}}(p), p_{\text{max}}(p), p_{\text{max}}(p) = 2s \cdot L(p, s)\).

Define \(U_t^{(i)} = \sum_{j \in \mathcal{J}_t} \frac{p_j^{(i)}}{p_{\text{max}}(p)}\) to be the total utilization for resource \(i\) at time \(t\). Therefore, we should have \(\exists i\), s.t. \(U_t^{(i)} + U_{t'}^{(i)} \geq 1\), for otherwise any active task at time \(t'\) could have been scheduled by the algorithm at time \(t\) or earlier. Thus, we can express the total area as:

\[
A(p) = \int_{t=0}^{T_{LS_d}(p)} \frac{d}{dt} U_t^{(i)} dt = \int_{t=0}^{T_{LS_d}(p)} \sum_{i=1}^{d} U_t^{(i)} dt \geq \frac{T_{LS_d}(p)}{2} .
\]

2) Pack Scheduling: We now present a pack-based scheduling strategy. Recall that a pack contains several concurrently executed tasks that start at the same time and the tasks in the next pack cannot start until all tasks in the previous pack have completed. Algorithm 4 presents the pack scheduling strategy PS\(_d\) with \(d\) resource types. It extends the algorithm presented in [2] for scheduling under a single resource type. Specifically, the tasks are first sorted in non-increasing order of execution times (which are fixed due to fixed resource allocations). Then, they are assigned one by one to the last pack if it fits, i.e., there is sufficient amount of resource under each resource type. Otherwise, a new pack is created and the task is assigned to the new pack. The complexity of the algorithm is \(O(n \log n + d)\), which is dominated by the sorting of the tasks and by checking the fitness of each task in the pack.

The following lemma shows the performance of this pack scheduling scheme.

**Lemma 5.** For a set of rigid tasks with a resource allocation matrix \(p\), the pack scheduling algorithm PS\(_d\) achieves, for any parameter \(s > 0\), a makespan

\[T_{PS_d}(p) \leq (2s + 1) \cdot L(p, s) . \tag{6}\]

**Proof.** We will prove in the following that \(T_{PS_d}(p) \leq 2A(p) + t_{\text{max}}(p)\), which will then lead to \(T_{PS_d}(p) \leq (2s + 1) \cdot \max(\frac{A(p)}{s}, t_{\text{max}}(p)) = (2s + 1) \cdot L(p, s)\).

Suppose the algorithm creates \(M\) packs in total. For each pack \(B_m\), where \(1 \leq m \leq M\), let \(A_m\) denote the total area of the tasks in the pack, i.e., \(A_m = \sum_{i \in B_m} a_i(p_i)\), and let \(T_m\) denote the total execution time of the pack, which is the same as the execution time of the longest task in the pack, i.e., \(T_m = \max_{i \in B_m} t_i(p_i)\).

Consider the time when the algorithm tries to assign a task \(j\) to pack \(B_m\) and fails due to insufficient amount of \(i\)-th resource, whose remaining amount is denoted by \(p^{(i)}\). As the tasks are handled by decreasing execution time, we have \(A_m \geq \left(1 - \frac{p^{(i)}}{m_{\text{max}}(p)}\right) t_j(p_j)\). Also, because task \(j\) cannot fit in pack \(B_m\), it means that its allocation \(p^{(i)}_j\) on resource \(i\) is at least \(p^{(i)}\). This task is then assigned to pack \(B_{m+1}\), so we have \(A_{m+1} \geq a_j(p_j) \geq p^{(i)} t_j(p_j)\). Since task \(j\) is the first task put in the pack and all the following tasks have smaller execution times, we have \(T_m = t_j(p_j)\). Hence,

\[A_m + A_{m+1} \geq t_j(p_j) = T_m + 1 .
\]

Summing the above inequality over \(m\), we get \(\sum_{m=0}^{M} T_m \leq 2\sum_{m=1}^{M} A_m = 2A(p)\). Finally, as the longest task among all tasks is assigned to the first pack, i.e., \(T_1 = t_{\text{max}}(p)\), we get the makespan \(T_{PS_d}(p) = \sum_{m=1}^{M} T_m \leq 2A(p) + t_{\text{max}}(p)\).

**E. Putting Them Together**

This section combines various strategies presented in the previous sections to construct several moldable task scheduling algorithms under multiple resource types. Two solutions are presented for the \(d\)-RESOURCE-SCHEDULING problem under each scheduling paradigm (list vs. pack) depending on if the problem is solved directly or indirectly via a transformation to the 1-RESOURCE-SCHEDULING problem. The following shows the combinations of the two solutions:

**Direct-based solution:** \(R_{A_0}(d) + R_d\)

**Transform-based solution:** \(TF + R_{A_1}(d) + R_d\)
Here, RA denotes the resource allocation strategy (Algorithm 1), TF denotes the transformation strategy (Algorithm 2), and R denotes the rigid task scheduling strategy LS or PS (Algorithm 3 or 4). Four algorithms can be resulted from the above combinations, and we call them D(RECT)-PACK, D(RECT)-LIST, T(RANSFORM)-PACK and T(RANSFORM)-LIST, respectively.

We point out that when the LCM of all the \( P(i) \)'s is dominated by \( P_{\text{max}} \) (e.g., when \( P(i) \)'s are powers of two), the transform-based solution will reduce the overall complexity of the direct-based solution exponentially, thus significantly improving the algorithms’ running times. The approximation ratios of the transform-based solutions, however, will not be compromised. The following theorem proves the ratios of these algorithms.

**Theorem 3.** The list-based algorithms (D-LIST and T-LIST) are \( 2d \)-approximations and pack-based algorithms (D-PACK and T-PACK) are \( (2d + 1) \)-approximations. Moreover, the bounds are asymptotically tight for the respective algorithms.

**Proof.** We prove the approximation ratios for the two pack-based scheduling algorithms. The proof for the list-based algorithms is very similar and hence omitted.

For D-PACK (RA\(_d\)(d) + PS\(_d\)): Using \( p_{\text{min}}^d \) found by RA\(_d\)(d), we have \( T_{\text{PS}_d}(p_{\text{min}}^d) \leq (2d + 1) \cdot L(p_{\text{min}}^d, d) = (2d + 1) \cdot L_{\text{min}}(d) \) from Lemma 5 (by setting \( s = d \)). Then, based on Theorem 1, we get \( T_{\text{RA}_d(d)+\text{PS}_d} \leq (2d + 1) \cdot T_{\text{OPT}}. \)

For T-PACK (TF + RA\(_1\)(d) + PS\(_1\)): Applying PS\(_1\) to the transformed instance using \( q_{\text{min}}^d \) found by RA\(_1\)(d), we have \( T'_{\text{RA}_1(d)+\text{PS}_1}(q_{\text{min}}^d) \leq (2d + 1) \cdot L'(q_{\text{min}}^d, d) = (2d + 1) \cdot L_{\text{min}}'(d) \) from Lemma 5 (again by setting \( s = d \)). Then, based on Theorem 2, we get \( T_{\text{TF}+\text{RA}_1(d)+\text{PS}_1} \leq (2d + 1) \cdot T_{\text{OPT}}. \)

The proof on the tightness of these bounds is omitted due to space constraint and can be found in the full version [29].

**V. EXPERIMENTS**

In this section, we conduct experiments and simulations whose goals are twofold: (i) to validate the theoretical underpinning of moldable task scheduling; and (ii) to compare the practical performance of various list- and pack-scheduling schemes under multiple resource types.

Experiments are performed on an Intel Xeon Phi Knights Landing (KNL) machine with 64 cores and 112GB available memory, out of which 16GB are high-bandwidth fast memory (MCDRAM) and the rest are slow memory (DDR). MCDRAM has \( \approx 5x \) the bandwidth of DDR [28], thus offering a higher data-transfer rate. The applications we run are from the Stream benchmark [21] configured to explore the two available resources of KNL (cores and fast memory) with different resource allocations. Simulations are further conducted to project the performance under more resource types using synthetic workloads that extend some classical speedup profiles.

**A. Evaluated Algorithms**

Each algorithm in Section IV-E is evaluated with two variants, depending on if it is list-based or pack-based. A list-based algorithm can arrange the tasks according to two widely applied heuristics: Shortest Processing Time First (SPT) and Longest Processing Time First (LPT), before scheduling them as shown in Algorithm 3. For pack-based scheduling, Algorithm 4 applies the Next Fit (NF) heuristic, which assigns a task to the last pack created. First Fit (FF) is another heuristic that assigns a task to the first pack in which the task fits. From the analysis, the approximation ratio can only be improved with this heuristic. Coupling these heuristics with the two solutions (direct vs. transformation) under the two scheduling paradigms (list vs. pack), we have a total of eight scheduling algorithms that we will evaluate in the experiments.

**B. Application Profiling**

In order to manage the fast memory as a resource, we first configure the KNL machine in flat mode [28], where MCDRAM and DDR together form the entire addressable memory. We will also explore MCDRAM in cache mode later (see next subsection). The applications in the Stream benchmark are then modified by allocating a specified portion of their memory footprints on MCDRAM (and the remaining on DDR), thereby impacting the overall data-transfer efficiency and hence the execution time. The profiles are obtained by varying both the number of cores (1-64) and the amount of fast memory (16GB divided into 20 chunks) allocated to each application and recording their corresponding execution times. In addition, we vary the total memory footprints of the applications to create tasks of different sizes.

Figure 2 shows the profiles of the triad application when its memory footprint occupies 100%, 80% and 60% of the fast memory, respectively. Similar profiles are also observed for the other applications (e.g., write, ddot) in the benchmark. First, we observe that the execution time is inversely proportional to the number of allocated cores, suggesting that the application has nearly perfect speedup. Also, the execution time reduces linearly as more fast memory is allocated to the point where the application’s memory footprint fits entirely in fast memory; after that the execution time remains fairly constant since allocating additional fast memory no longer benefits the execution. The profiled data for different applications under various sizes are subsequently used to compute the schedules by different algorithms.

**C. Experimental Results**

In order to validate our theoretical analysis and to compare the performance of different algorithms, we implement a basic scheduler that executes the tasks according to the schedules given by these algorithms based on the application profiles. Each task is launched at the specified time using the specified amount of resources (cores and fast memory) on the KNL machine. If any required resource of a task is not available at start time because of the delays in the preceding tasks (possibly due to performance variations), the execution of the task will be correspondingly delayed until there are enough resources available. We expect the delay (if any) to be small for each task but the effect can accumulate throughout the course of the schedule. This additional overhead is assessed in our experiments.

Figure 3(a) shows the makespans of the eight scheduling algorithms normalized by the theoretical lower bound (Equation (1) with \( s = d \); see also Lemma 1) while running 150 tasks. We have also varied the number of tasks but observed no significant difference in their relative performance. First, we can see that the scheduling overhead, which manifests as the difference between the theoretical prediction and the experimental result, is less than 10% for all algorithms. In general, list scheduling (left four) and pack scheduling (right
four) have comparable performance despite that the former admits a better theoretical bound. Moreover, transform-based solutions fare slightly better than direct-based solutions due to the balanced resource requirements of different applications in the benchmark. Lastly, in line with the intuition and conventional wisdom, LPT and FF are effective for reducing the makespan for list and pack scheduling, respectively, except for the T-LIST algorithm, which incurs a larger overhead with the LPT variant in the actual execution.

Figure 3(b) further compares the makespans of four transform-based algorithms when MCDRAM is not used in flat mode (in which case all data are allocated on the slow DDR memory), and when KNL is configured in cache mode (in which case MCDRAM serves as a last-level cache to DDR). In both cases, only one type of resource (i.e., cores) is scheduled, and as a result, the transformation produces the same solutions as the direct-based schedules. Not surprisingly, cores-only scheduling in flat mode has the worst makespan for all algorithms, since MCDRAM is completely unutilized in this case. Having MCDRAM as cache to DDR significantly improves the makespan, but due to interference from concurrently running applications, the makespan is still worse than scheduling both cores and MCDRAM in flat mode. The results confirm the benefit of explicitly managing multiple resource types for improving the application performance.

D. Simulation Results with More Resource Types

In this section, we conduct simulations to project the performance of list-based and pack-based scheduling algorithms under more resource types. We simulate up to four types of resources that can represent different types of components, such as CPUs, GPUs, fast memory, I/O bandwidth, all of which would be shared among different applications. The total amount of discrete resource for each type is set to be 64, 32, 16 and 8, respectively. Due to the lack of realistic application profiles under multiple-resource constraints, we generate synthetic workloads by extending two classical speedup functions (described in Section II-A) to model moldable applications:

- **Extended Amdahl’s law [1]:**
  - (i) $\frac{1}{s_0 + \sum_{i=1}^{d} \frac{1}{p_i}}$;
  - (ii) $\frac{1}{s_0 + \frac{1-s_0}{\prod_{i=1}^{d} p_i^{\alpha_i}}}$;
  - (iii) $\frac{1}{s_0 + \max_{i=1..d} \frac{s_i}{p_i^{\alpha_i}}}$;

- **Extended power law [24],[15]:**
  - (i) $\frac{1}{s_0 + \frac{1}{\prod_{i=1}^{d} p_i^{\alpha_i}}}$;
  - (ii) $\prod_{i=1}^{d} (p_i^{\alpha_i})$;
  - (iii) $\frac{1}{\prod_{i=1..d} \max_p \frac{s_i}{p_i^{\alpha_i}}}$.

The extension of both speedup laws comes in three different flavors (i.e., sum, product, max), characterizing how allocations of different resource types work together to contribute to the overall application speedup (i.e., sequential, collaborative, concurrent). In the simulations, we generate 150 tasks, each randomly selected from the six extended profiles. The sequential fraction $s_0$ is uniformly generated in $(0,0.2]$, and the parallel fraction $s_i$ for each resource type $i$ is uniformly generated in $(0,1]$ and normalized to make sure that $\sum_{i=1}^{d} s_i = 1 - s_0$. The parameter $\alpha_i$ is uniformly generated in $[0.3,1]$. The total work of a task is chosen uniformly from a range normalized between 0 and 1 (the relative performance of the algorithms is not affected by the normalization).

Figure 4 plots the performance distributions of the scheduling algorithms with up to four resource types. Each boxplot is obtained by running 100 instances. Only LPT (for list) and FF (for pack) are included, since they have better overall performance. First, we observe that the normalized makespans indeed increase with the number of resource types, corroborating the theoretical analyses, although they are far below those predicted by the worst-case approximation ratios. Also, list-based schedules now produce consistently smaller makespans compared to the pack-based ones, which we believe is due to the larger variability in the simulated task profiles than that of the Stream benchmark. This allows the list-based schedules to explore more effectively the gaps between successive task ex-

---

**Figure 2.** Profiles of the triad application in Stream benchmark when its memory footprint occupies 100%, 80% and 60% of the entire fast memory.

**Figure 3.** (a) Normalized makespans of eight scheduling algorithms; (b) Makespans of four transform-based scheduling algorithms in three different resource management configurations.

**Figure 4.** (a) Normalized makespans and (b) running times of four scheduling algorithms with up to four different resource types.
ecutions (as reflected by the theoretical bounds). However, the makespan difference between the two scheduling paradigms becomes smaller with more resource types, suggesting that pack-based scheduling is promising when more resource types are to be managed. Finally, transform-based solutions are superior compared to the direct-based ones in terms of both makespan and algorithm running time (for which there are 2-3 orders of magnitude difference with more than two resource types). In general, for workloads with balanced resource requirements, transform-based scheduling is expected to offer both fast and efficient scheduling solutions.

Lastly, we point out that it may seem in Figure 4(a) that the makespan increases with more resource types. In fact, it is the ratio to the optimal solution (or lower bound), not the makespan itself, that increases with \( d \). In practice, the value of that optimal solution will decrease with more resource types, since there is more room for optimization. Hence, scheduling more resource types potentially improves the overall makespan, which is what we observed experimentally when managing one and two types of resources as shown in Figure 3(b).

VI. CONCLUSION AND FUTURE WORK

List scheduling and pack scheduling are two scheduling paradigms that have been studied extensively in the past. While most prior works focused on scheduling under a single resource type, in this paper we joined a handful of researchers on scheduling under multiple-resource constraints in these two scheduling paradigms. Our analyses and evaluation results show that scheduling becomes more difficult with increasing number of resource types, but considering all available resources is essential for improving the application performance. Despite the better theoretical bound of list scheduling, pack scheduling has been shown to have comparable or, in some scenarios, even better performance, especially when more types of resources are included in the scheduling decision. The results offer useful insights to the design of emerging architectures and systems that need to incorporate multiple dimensions in the resource management.

Future work will be devoted to the design of improved list or pack scheduling algorithms (or any other practical algorithms beyond the two scheduling paradigms) with multiple-resource constraints. An immediate open question is whether the \( 2d \)-approximation for list scheduling can be improved, e.g., to \((d+1)\)-approximation as proven by Garey and Graham for rigid tasks [10], which is equivalent to Phase 2 of our list-based algorithm but with fixed resource allocation. Given our list algorithm’s lower bound of \( 2d \) for moldable tasks (Theorem 3), any improvement will likely be achieved through the design of more sophisticated resource allocation strategies (Phase 1) or through a more coupled design/analysis of the two phases. Prior work [22] on scheduling under a single-resource type has shed light on some possible directions (e.g., by using dual approximation).

ACKNOWLEDGEMENTS

We would like to thank Nicolas Denoyelle for sharing the code to run the modified Stream benchmark on KNL. This research is supported in part by the National Science Foundation under the award CCF 1719674. Finally, we thank the anonymous reviewers for useful comments.

REFERENCES