Optimal Greedy Algorithm for Many-Core Scheduling

Anuj Pathania, Vanchinathan Venkatramani, Muhammad Shafique, Tulika Mitra, and Jörg Henkel

***Abstract*—In this paper, we propose an optimal greedy algorithm for the problem of run-time many-core scheduling. The previously best known centralized optimal algorithm proposed for the problem is based on dynamic programming. A dynamic programming-based scheduler has high overheads which grow fast with increase in both the number of cores in the many-cores as well as number of tasks independently executing on them. We show in this paper that the inherent concavity of extractable instructions per cycle in tasks with increase in number of allocated cores allows for an alternative greedy algorithm. The proposed algorithm significantly reduces the run-time scheduling overheads, while maintaining theoretical optimality. In practice, it reduces the problem solving time 10 000× to provide near-optimal solutions.**

***Index Terms*—Greedy algorithm, many-cores, scheduling, throughput maximization.**

I. Introduction

Many-cores are hundred core (or even thousand core) processors designed to execute several independent multithreaded tasks in parallel [9]. One crucial scheduling decision to be made on a many-core is the distribution of processing cores amongst the executing tasks in a manner such that the processor always operates at its peak performance. In this paper, we use *adaptive* many-cores [15] which allows acceleration of not just multithreaded tasks by thread level parallelism (TLP) exploitation but also acceleration of single-threaded tasks by exploitation of instruction level parallelism (ILP).

To reduce context-switching overheads, many-cores prefer to operate with one thread per-core model [5]; keeping the problem of many-core scheduling mathematically discrete. A task goes through different phases during its execution. These phases result in a high variance within the *speedup* a task can derive from a many-core as shown in Fig. 1. n-core speedup of a task is defined as the ratio of its instructions per cycle (IPC) when assigned N cores with its IPC when assigned only to a single core. Most tasks in general show sublinear speedups but sometime they also show super-linear speedups. This happens when assigning more cores open up secondary bottlenecks such as available memory.

Malleable tasks [6] by design allow their core allocations to increase or decrease in size at any time during their execution with negligible performance penalties. When operating with malleable tasks, many-core schedulers can exploit the variance in task speedups by performing core redistributions at run-time for throughput (total IPC) maximization [14].

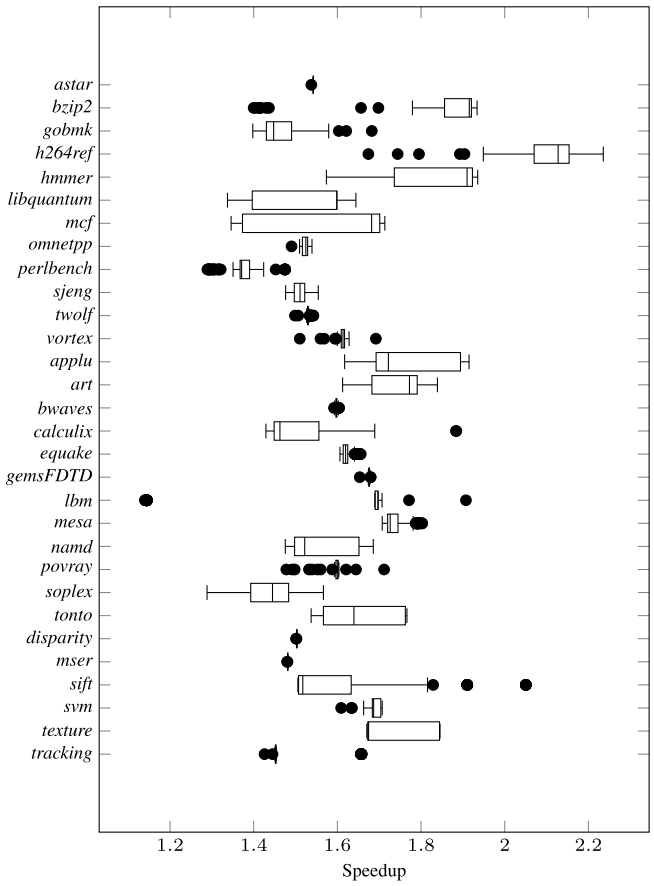

Fig. 1. Observed variance in 2-core speedup for different benchmarks sampled at every ten million instructions within their entire execution.

The problem of run-time scheduling on many-core (throughput maximization) can be solved optimally in polynomial time using dynamic programming [8]. A centralized scheduler based on dynamic programming has high overheads which grow fast as both the number of cores in a many-core and the number of tasks they can execute in parallel increase. Dynamic programming given its overheads is not suitable for scheduling at run-time due to frequent invocations. Therefore, a search for alternative low overhead algorithms for run-time many-core scheduling is mandated. In past, researchers have resorted to proposing heuristics with no guarantees on the obtained results [16]. We instead propose a low overhead solution in this paper, which preserves the theoretical optimality of results.

This search is aided by the fact that extractable IPC in tasks executing on a many-core is in general concave with increase in number of allocated cores as shown in Fig. 2. The observed concavity is due to the saturation of exploitable ILP/TLP in the tasks.

Manuscript received July 9, 2016; accepted September 29, 2016. Date of publication October 19, 2016; date of current version May 18, 2017. This work was supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" under Grant SFB/TR 89, and in part by the Singapore Ministry of Education Academic Research Fund Tier 2 under Grant MOE2014-T2-2-129. This paper was recommended by Associate Editor A. Macii. *(Corresponding author: Anuj Pathania.)*

A. Pathania and J. Henkel are with the Chair of Embedded System, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany (e-mail: anuj.pathania@kit.edu).

V. Venkatramani and T. Mitra are with the School of Computing, National University of Singapore, Singapore 117417.

M. Shafique is with Embedding Computing System Group, TU Wien, Vienna 1040, Austria.

Digital Object Identifier 10.1109/TCAD.2016.2618880

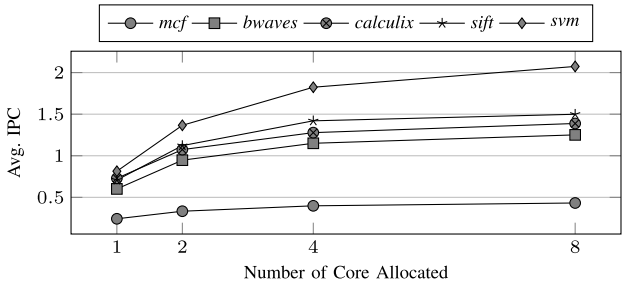

Fig. 2. Observed average IPC of different benchmarks when allocated different number of cores.

A. Our Novel Contributions

In this paper, we present a greedy algorithm for the problem of run-time many-core scheduling based on the inherent concavity in extractable IPC in independently executing tasks. We show that a scheduler based on the proposed greedy algorithm significantly reduces the processing- and space overhead in comparison to a scheduler based on dynamic programming without sacrificing on the theoretical optimality. In practice where concavity property does not always hold, the algorithm results in significantly near-optimal solutions.

II. Related Work

Multi/many-core scheduling is a well-studied problem in research [16]. Though, major focus in past for performance-oriented schedulers remained on design-time makespan minimization instead of run-time throughput maximization.

Makespan minimization involves determining a fixed schedule that finishes a given workload on a multi/many-core processor in the shortest time [7]. Makespan minimizing schedulers often assume all information about the workload such as execution time and arrival time of constituent tasks is known beforehand. They also often assume executing tasks lack any phases and thereby always progress at a fixed rate. When operating with malleable tasks with concave speedups/IPCs, Błażewicz *et al.* [4] proposed a scheduler that can optimally minimize makespan with $O(T \max\{C, T \log^2 C\})$ processing overhead, where T is the number of tasks and C is the number of processing cores.

Throughput maximization on the other hand involves determining a schedule at discrete intervals for a workload, where often neither arrival time nor the departure time of its constituent tasks are known to the scheduler [7]. A run-time scheduler when running unprofiled tasks can rely only on information available to it from online observations and performance predictions that can be made from them.

Gulati *et al.* [8] proposed a dynamic programming-based algorithm with $O(CT^2)$ processing overhead for optimal throughput maximization. Given the high-overhead for obtaining optimal solutions through dynamic programming, alternative approaches to reduce overhead of run-time many-core scheduling was proposed in form of distributed algorithms. Anagnostopoulos *et al.* [1] introduced a best-effort heuristic multiagents systems (MASs) for run-time many-core scheduling with malleable tasks. We ourselves designed an MAS in [14] that solved the problem optimally using a distributed algorithm based on cooperative game theory. Though promising in reducing the per-core processing overhead by disbursement of scheduling calculations across multiple/all cores, the reduction is accompanied by concurrent increase in the communication overhead.

We instead in this paper, improve upon best-known centralized algorithm for throughput maximization [8] by introducing another optimal centralized algorithm that decreases both the processing as well as space overhead without changing the communication overhead. The proposed algorithm when operating with malleable tasks brings the efficiency of throughput maximizing schedulers to the same level as that of makespan minimizing schedulers.

III. Greedy Algorithm for Run-Time Many-Core Scheduling

A. System Model

We begin by presenting the notations used in this paper to model a many-core system. Let T and C be the number of tasks and cores in the system, respectively. C_i be the number of cores currently allocated to task i. Let I_{C_i} denote the IPC of task i when allocated C_i cores.

Using the above notations, the problem of throughput maximization on many-core can be mathematically stated as

$$\text{Maximize } \sum_{i=1}^{T} I_{C_i}, \text{ under constraint } \sum_{i=1}^{T} |C_i| \;\leq\; C.$$

In the context of run-time many-core scheduling, the above equation needs to be solved by the scheduler nearly in every *scheduling epoch*. Scheduling epoch is the frequency at which a scheduler is invoked by an operating system (OS). In this paper, its value is assumed to be 10 ms; same as the default scheduling epoch used by Linux kernel [12].

Furthermore to operate, a scheduler also needs to know the value of IPC $I_{C'_i}$ of task i from a possible core allocation C'_i from its current observable IPC I_{C_i} when $C'_i \neq C_i$. For this, we employ regression-based performance-prediction models for adaptive many-cores developed in [17].

The concavity in IPC extraction as shown in Fig. 2 is mathematically captured by following equation:

$$\forall_{n\geq 0}\ I_{C'_i+n} - I_{C'_i} \geq I_{C_i+n} - I_{C_i} \text{ if } C'_i \leq C_i. \tag{1}$$

B. Greedy Algorithm

We now propose a greedy algorithm for the problem of run-time many-core scheduling. The proposed algorithm is composed of following sequential steps performed by the greedy scheduler before every scheduling epoch.

1) Assume $C_i = 0\ \forall i \in T$.
2) Sort all tasks in T in ascending order by using comparator $[I_{C_i+1} - I_{C_i}]$ and store in a queue.
3) Virtually assign a core to task j in front of the queue and update the corresponding I_{C_j} using performance-prediction models from [17].
4) Reposition task j according to the updated I_{C_j} in the sorted queue using a binary search insertion.
5) Repeat steps 3 and 4 till all cores are allocated.
6) Readjust real core allocations from last scheduling epoch to reflect the new optimal core allocations.
7) Execute tasks with the optimal core allocations.

The proposed greedy scheduler, like all greedy algorithms is minimalistic in its approach and is quite easy to implement. Nevertheless, its real strength come from its ability to provide optimal results. We now proceed to prove the theoretical optimality of our algorithm.

Theorem 1: The greedy core allocations are optimal.

Proof: Let $\langle C_1, C_2, \ldots, C_T \rangle$ be the core allocations chosen by our proposed greedy algorithm. We prove our theorem using proof by induction.

C. Base Case

Suppose $\langle C_1, C_2, C_x - 1, \ldots, C_y + 1, \ldots, C_T \rangle$ instead be the optimal core allocations in which task x has one less core allocated to it, which instead is allocated to task y.

Now for the optimal core allocations to be better than greedy core allocations following equation must hold:

$$I_{C_y+1} - I_{C_y} > I_{C_x} - I_{C_x-1}. \tag{2}$$

The above equation says that the benefit (in terms of increased IPC) of assigning additional core to task y must outweigh the loss of taking away that core (in terms of decreased IPC) from task x under optimal core allocations.

Our greedy algorithm does not reconsider core allocation decisions. So, the suboptimal decision of allocating an additional core to task x with $C_x - 1$ cores already allocated happens when either C_y cores or $C'_y < C_y$ cores were allocated to task y. We consider both cases below.

If suboptimal core allocation happens when task y had C_y cores, then by greedy design it implies following relation:

$$I_{C_x} - I_{C_x-1} \geq I_{C_y+1} - I_{C_y}.$$

The above equation is in contradiction to (2).

On the other hand, if the suboptimal core allocation happens when task y had $C'_y < C_y$ cores, then

$$I_{C_x} - I_{C_x-1} \geq I_{C'_y+1} - I_{C'_y}.$$

But, we know from concavity (1)

$$\begin{aligned} & I_{C'_y+1} - I_{C'_y} \geq I_{C_y+1} - I_{C_y} \\ \implies & I_{C_x} - I_{C_x-1} \geq I_{C_y+1} - I_{C_y}. \end{aligned} \tag{3}$$

Again a contradiction to (2). Hence, we prove our base case is optimal.

D. Step Case

Suppose $\langle C_1, C_2, C_x - 2, \ldots, C_y + 1, C_z + 1, \ldots, C_T \rangle$ instead be the optimal core allocations in which task x has two less cores allocated to it, which instead are allocated to tasks y and z; one each. Without loss of generality the proof will also hold if both cores from task x were allocated to tasks y or z exclusively instead.

Now for above optimal core allocations to be better than greedy core allocations following equation must hold:

$$I_{C_y+1} - I_{C_y} + I_{C_z+1} - I_{C_z} > I_{C_x} - I_{C_x-2}. \tag{4}$$

As argued in base case, suboptimal decision of allocating the first additional core to task x when allocated $C_x - 2$ happens when either $C'_y \leq C_y$ or $C'_z \leq C_z$ or both. Therefore, by design of the greedy algorithm following relations hold:

$$\begin{aligned} I_{C_x-1} - I_{C_x-2} &\geq I_{C'_y+1} - I_{C'_y} \\ &\geq I_{C'_z+1} - I_{C'_z}. \end{aligned} \tag{5}$$

Similarly, suboptimal decision of allocating the second additional core to task x with $C_x - 1$ cores already allocated happens when either $C'_y \leq C_y$ or $C'_z \leq C_z$ or both. Therefore, by design following relations also hold:

$$\begin{aligned} I_{C_x} - I_{C_x-1} &\geq I_{C'_y+1} - I_{C'_y} \\ &\geq I_{C'_z+1} - I_{C'_z}. \end{aligned} \tag{6}$$

Adding (5) and (6) we get

$$I_{C_x} - I_{C_x-2} \geq I_{C'_y+1} - I_{C'_y} + I_{C'_z+1} - I_{C'_z}.$$

But, we know from concavity (1)

$$\begin{aligned} I_{C'_y+1} - I_{C'_y} &\geq I_{C_y+1} - I_{C_y} \\ I_{C'_z+1} - I_{C'_z} &\geq I_{C_z+1} - I_{C_z}. \end{aligned}$$

Therefore

$$I_{C_x} - I_{C_x-2} \geq I_{C_y+1} - I_{C_y} + I_{C_z+1} - I_{C_z}.$$

Above equation is in contradiction to (4). Hence, we prove our step case to be optimal.

E. Assumption Case

We assume greedy allocations are optimal till n cores are removed from task x and distributed among remaining tasks in any combination. Mathematically, the following relationship is assumed to be true:

$$I_{C_x} - I_{C_x-n} \geq I_{C_y+\alpha_y} - I_{C_y} + \cdots + I_{C_T+\alpha_T} - I_{C_T} \tag{7}$$

where $\alpha_y + \cdots + \alpha_T = n$.

F. Induction Case

Now, we assume in optimal allocation $(n+1)$th core is removed from task x and without loss of generality given to task y, while previously removed n cores are distributed in same combination as in assumption case.

Now for the optimal core allocations to be better than greedy core allocations following equation must hold:

$$I_{C_y+\alpha_y+1} - I_{C_y} + \cdots + I_{C_T+\alpha_T} - I_{C_T} \geq I_{C_x} - I_{C_x-n-1}. \tag{8}$$

Since greedy algorithm chooses to allocate core to task x with $C_x - n - 1$ cores allocated instead of task y with $C_y + \alpha_y$ cores allocated, by the design of the greedy algorithm following relationship holds:

$$I_{C_x-n} - I_{C_x-n-1} \geq I_{C_y+\alpha_y+1} - I_{C_y+\alpha_y}.$$

Adding (7) to above equation we get

$$I_{C_x} - I_{C_x-n-1} \geq I_{C_y+\alpha_y+1} - I_{C_y} + \cdots + I_{C_T+\alpha_T} - I_{C_T}.$$

Above equation is in contradiction to (8), proving our induction step is optimal.

Hence, our algorithm is proven optimal by induction. ■

G. Complexity

Given that dynamic programming already provides the optimal solution for the problem of run-time many-core scheduling, the primary reason for developing a greedy algorithm is to reduce the scheduling overheads. These reductions will directly translate into improvement in performance in real-world systems.

In the proposed algorithm, one time sorting in step 2 has a processing overhead of $O(T \lg T)$. Additionally, binary search in step 4 has a processing overhead of $O(\lg T)$. Since step 4 is repeated C times, the total processing overhead of our algorithm is $O(T \lg T + C \lg T)$ or $O(\max\{C, T\} \lg T)$. This overhead is significantly less than $O(CT^2)$ processing overhead of dynamic programming.

Furthermore, the proposed algorithm requires maintenance of only one queue data structure with space overhead of $O(T)$. On the other hand, dynamic programming has a significantly higher space overhead of $O(CT)$.

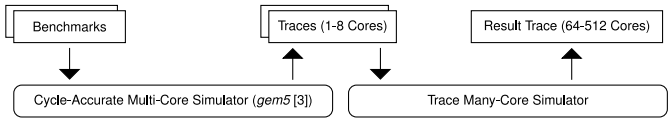

Fig. 3. Experimental setup.

TABLE I
BENCHMARKS USED FOR EXPERIMENTS

Suite	*Benchmark Name*
SPEC [10], [11]	*art astar, bwaves, bzip2, calculix, equake, gemsfdtd, gobmk, h264ref, hmmer, lbm, mcf, namd, omnetpp, perlbench, povray, sjeng, tonto, twolf, vortex*
SD-VBS [19]	*disparity, mser, sift, svm, texture, tracking*
PARSEC [2]	*blackscholes, fluidanimate, swaptions, streamcluster*
SPLASH [20]	*cholesky, fmm, lu, radix, radiosity, water-sp*

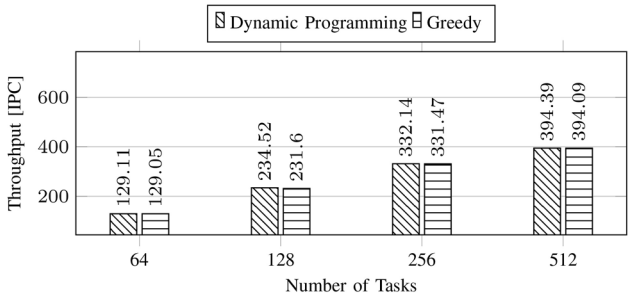

Fig. 4. Performance comparison between schedulers employing dynamic programming and greedy algorithm on closed 256-core many-core system under different size workloads.

H. Fairness

Note that optimal performance does not translate into a starvation-free optimal fairness. In fact, performance and fairness are often contradictory goals [20]. Optimal fair scheduling for many-cores remains an unsolved problem and is at present an active subject of research [13].

IV. EXPERIMENTAL EVALUATIONS

A. Setup

We use a two-stage adaptive many-core simulator as shown in Fig. 3 for our proof-of-concept empirical evaluations. First stage is based on *gem5* [3] cycle-accurate simulator with *Bahurupi* [15] adaptive cores implementing *ARM v7 ISA* and running at 1 GHz. Cores have 2-way out-of-order pipeline and have separated 4-way associative L1 instruction- and data cache of size 64 KB each. An 8-way associative 2 MB unified L2 cache is shared by all cores.

Unfortunately, cycle-accurate many-core simulations are not time-wise feasible and hence our first stage is limited to maximum of eight cores. To bypass this limitation, we take isolated execution traces of different tasks from the cycle-accurate simulator with up to eight cores allocated and extrapolate them using an in-house trace-simulator for many-core simulations. Our extrapolated trace simulations lack the modeling depth of cycle-accurate simulations but we believe them to be sufficient as an initial testbed for comparing many-core algorithms.

On software side, we took 36 benchmarks from *SPEC* [10], [11], *SD-VBS* [18], *PARSEC* [2], and *SPLASH* [19] benchmark suites as listed in Table I. All benchmarks were executed in syscall emulation mode. *SPEC*, *SD-VBS*, and *PARSEC/SPLASH* benchmarks were executed with "ref," "full-hd," and "sim-small" inputs, respectively.

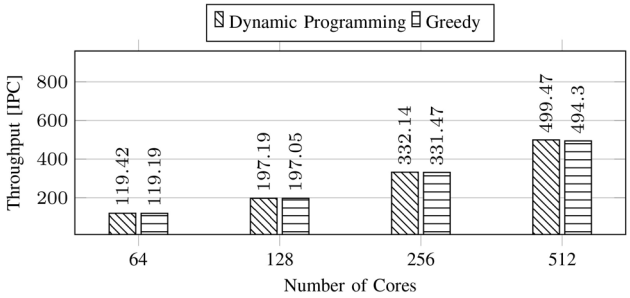

Fig. 5. Performance comparison between schedulers employing dynamic programming and greedy algorithm for a fixed 256-task workload on varisized closed many-core systems.

Diversity in these benchmarks allows for a very generic empirical evaluations.

B. Performance Evaluation

We chose to evaluate our algorithm on a closed many-core system. In a closed system, a workload is fixed at the start and constituent tasks of the workload restart execution immediately after completion. Our workload comprises of a random mix of all the available benchmarks with uniform distribution. Throughput measured in terms of average total IPC of the system over two billion cycles is selected as the metric for measuring performance.

Fig. 4 shows the performance of a closed many-core system with 256 cores under different size workloads when operated with a dynamic programming- and a greedy algorithm-based schedulers. In our experiments, we observed that the performance under the greedy scheduler can differ from dynamic programming-based scheduler by up to 1.24% even though both are proven theoretically optimal.

This discrepancy is attributed to the fact that the greedy algorithm requires IPC concavity to be observed by all benchmarks at all times, while no such requirement is mandated by dynamic programming. In practice, IPC concavity is exhibited by all benchmarks most of the time, but there are few rare exceptions. For example, benchmarks like *h264* and *sift* sometime exhibit super-linearity as shown in Fig. 1. Hence, a slight degradation in performance is observed when the concavity assumption is violated.

Fig. 5 shows the performance of closed many-core systems of various sizes under different schedulers when a given workload is executed on them. We observed that the greedy scheduler results in near-optimal solutions in under all cases; similar to observations made in Fig. 4.

C. Scalability

To get a measure of real-world benefits from a greedy scheduler against a dynamic programming-based scheduler, we ran both schedulers cycle-accurately on *gem5* within a single core in our simulated many-core with representative varisized inputs. We then recorded the simulated system-time it took for both schedulers to reach a solution in Table II on average for a scheduling epoch.

Empirical evaluations show that dynamic programming-based scheduler requires 7.985 ms to solve the problem of run-time many-core scheduling for a 64-core many-core running 32 tasks. For a 10 ms scheduling epoch at which a multicore OS operates, this will result in impractical overhead of 79.85%. The greedy scheduler in comparison will have an overhead of only 0.61% for the same size problem.

Furthermore, the overhead of dynamic programming-based scheduler grows exponentially with both increase in the number of cores

TABLE II
COMPARISON OF TIME TAKEN BY THE GREEDY SCHEDULER AND THE DYNAMIC PROGRAMMING-BASED SCHEDULER TO SOLVE RUN-TIME MANY-CORE SCHEDULING PROBLEM OF DIFFERENT SIZES

Cores	Tasks	Dynamic Programming (ms)	Greedy (ms)	Improvement
64	32	7.985	0.061	130.90x
	64	16.203	0.077	210.42x
	128	32.910	0.108	304.72x
128	64	71.005	0.113	628.36x
	128	144.214	0.144	1,001.48x
	256	285.617	0.223	1,280.79x
256	128	586.276	0.250	2,345.11x
	256	1,163.108	0.318	3,657.57x
	512	2,314.798	0.555	4,170.80x
512	256	4,553.491	0.635	7,170.85x
	512	9,056.819	0.866	10,458.22x
	1024	18,031.729	1.578	11,426.95x

and the tasks independently executing on them. On the other hand, overhead of greedy scheduler grows much more slowly. Even on a 512-core running 1024 tasks, overhead of greedy algorithm stands at an acceptable 15.78%. To reduce overhead further, it seems a many-core OS will either need to operate at much higher values of scheduling epoch or algorithms even faster than greedy need to be devised.

It is important to note that beside direct reduction in processing overhead, a reduction in space overhead also plays critical role in reducing the total problem solving time. When operating with large data-structures, a scheduler is forced to perform page-swaps with main memory when the last-level cache is saturated. Since main memory accesses have several times the latencies of cache accesses, the performance suffers enormously. Given the fact that space-overhead for our greedy scheduler is much smaller than dynamic programming-based scheduler, cache-saturation will happen in the former for problem of much larger size than the latter.

V. CONCLUSION

In this paper, we proposed a theoretically optimal greedy algorithm for the problem of run-time many-core scheduling. Given the large optimization search-space, the problem requires light-weight algorithms that can be applied online.

A dynamic programming-based scheduler can solve the problem optimally but has high scheduling overheads associated with it. As an alternative, we proposed a scheduler based on a greedy algorithm in this paper. The proposed greedy scheduler exploits the concavity in IPC extraction inherent in many-core workloads to maintain theoretical optimality.

In practice, it requires $10\,000\times$ less time than a dynamic programming-based scheduler to reach a solution, while providing near-optimal performance.

REFERENCES

[1] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, "Distributed run-time resource management for malleable applications on many-core platforms," in *Proc. ACM Design Autom. Conf. (DAC)*, Austin, TX, USA, 2013, pp. 1–6.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Parallel Architect. Compilation Tech. (PACT)*, Toronto, ON, Canada, 2008, pp. 72–81.

[3] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Architect. News*, vol. 39, no. 2, pp. 1–7, 2011.

[4] J. Błażewicz, M. Machowiak, J. Węglarz, M. Y. Kovalyov, and D. Trystram, "Scheduling malleable tasks on parallel processors to minimize the makespan," *Ann. Oper. Res.*, vol. 129, no. 1, pp. 65–80, 2004.

[5] S. Boyd-Wickizer *et al.*, "Corey: An operating system for many cores," in *Proc. Oper. Syst. Design Implement. (OSDI)*, San Diego, CA, USA, 2008, pp. 43–57.

[6] S. Buchwald, M. Mohr, and A. Zwinkau, "Malleable invasive applications," in *Proc. Softw. Eng.*, Dresden, Germany, 2015, pp. 123–126.

[7] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Proc. Job Sched. Strategies Parallel Process.*, Orlando, FL, USA, 1998, pp. 1–24.

[8] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger, "Multitasking workload scheduling on flexible-core chip multiprocessors," in *Proc. Int. Conf. Parallel Architect. Compilation Tech. (PACT)*, Toronto, ON, Canada, 2008, pp. 187–196.

[9] J. Henkel *et al.*, "Invasive manycore architectures," in *Proc. Asia South Pac. Design Autom. Conf. (ASP-DAC)*, Sydney, NSW, Australia, 2012, pp. 193–200.

[10] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, Jul. 2000.

[11] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[12] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proc. Linux Symp.*, vol. 2. Ottawa, ON, Canada, 2006, pp. 215–230.

[13] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Distributed fair scheduling for many-cores," in *Proc. Design Autom. Test Europe (DATE)*, Dresden, Germany, 2016, pp. 379–384.

[14] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Distributed scheduling for many-cores using cooperative game theory," in *Proc. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 133:1–133:6.

[15] M. Pricopi and T. Mitra, "Bahurupi: A polymorphic heterogeneous multi-core architecture," *ACM Trans. Architect. Code Optim.*, vol. 8, no. 4, 2012, Art. no. 22.

[16] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2013, pp. 1:1–1:10.

[17] V. Vanchinathan, "Performance modeling of adaptive multi-core architecture," M.S. thesis, School Comput., Nat. Univ. Singapore, Singapore, 2015.

[18] S. K. Venkata *et al.*, "SD-VBS: The San Diego vision benchmark suite," in *Proc. Int. Symp. Workload Characterization (IISWC)*, Austin, TX, USA, 2009, pp. 55–64.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. SIGARCH Comput. Architect. News*, 1995, pp. 24–36.

[20] S. M. Zahedi and B. C. Lee, "REF: Resource elasticity fairness with sharing incentives for multiprocessors," in *Proc. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 145–160.