Distributed Fair Scheduling for Many-Cores

[*]Anuj Pathania, [†]Vanchinathan Venkataramani, [*]Muhammad Shafique, [†]Tulika Mitra, [*]Jörg Henkel
[*]Chair of Embedded System (CES), Karlsruhe Institute of Technology, Germany
[†]School of Computing (SoC), National University of Singapore, Singapore
Corresponding Author: anuj.pathania@kit.edu

***Abstract*—Transition of embedded processors from multi-cores to many-cores continues unabated. Many-cores execute tens of tasks in parallel and in some contexts, it is crucial that the processing cores are distributed fairly amongst the tasks. Traditional queue-based centralized fair schedulers designed for multi-cores will have excessive overhead on many-cores due to the enlarged optimization search-space. Further, the processing requirements of executing tasks may vary under different phases of their execution necessitating lightweight dynamic fair schedulers to regularly perform partial reallocation of the cores. We introduce a distributed dynamic fair scheduler that can scale up with the increase in number of cores because it disburses the processing overhead of scheduling amongst all the cores. Based on observations made for task executions on many-cores, we propose an optimal solution under certain constraints for the fair scheduling problem, which in general is NP-Hard.**

I. Introduction

Number of computing cores on-chip is increasing rapidly with every technology generation. Multi-core processors with a dozen processing cores are now expected to be replaced by many-core processors with hundreds of processing cores [1], [2]. Many-cores execute tens or hundreds of tasks in parallel to fully exploit their compute potential. Achieving high performance is often the goal of the systems. However, in some systems fairness is more emphasized than performance. For example, in embedded systems wherein certain critical tasks should not experience substantial performance degradation to prevent system failure or in server systems wherein tasks from different users agnostically running together should not experience any discrimination. Such systems need to ensure that all executing tasks are given their fair share of cores based on their requirements, and performance gains in one task does not happen at the expense of performance drop in another.

Schedulers are low level Operating System (OS) routines, which allocate cores to the executing tasks [3]. Linux Completely Fair Scheduler (CFS) [4] is currently the most widely used default fair scheduler in multi-core OS. It allocates near-equal time slices to executing tasks, so that each task gets equal share of CPU time. Authors in [5] extend CFS to asymmetric multi-cores. In multi-cores, number of tasks dominates the number of cores permitting the applicability of concepts like time slicing or runtime queues. In many-cores, number of cores in general outnumbers the number of tasks thereby making notion of round robin executions redundant. Additionally, state-of-the-art fair schedulers for multi-cores proposed in research [6] are innately centralized and will not scale up as we transition from multi-cores to many-cores.

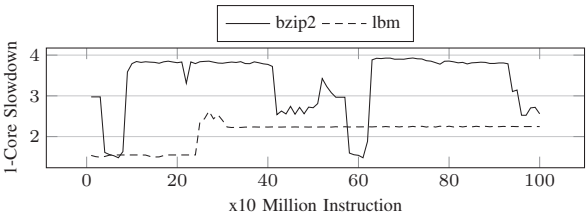


Fig. 1: Profiles of *bzip2* and *lbm* benchmarks showing changes in their single-core slowdowns compared to 8-core.

Distributed schedulers for many-cores [7] are largely performance-oriented disregarding fairness. Authors in [8] proposed a lightweight runtime fair scheduling heuristic based on the notion of efficiency that is scalable but results in suboptimal fair schedules. We on the other hand propose a scheduler, which is not only scalable but is also proven to be optimal in term of fairness under certain conditions.

Fair scheduling problem becomes further challenging as the processing requirements of executing tasks keep changing as the tasks go through different phases of their execution [9]. This results in a task experiencing variable slowdown during its execution. The **slowdown** of a task running on N-core is computed as the ratio of its maximum achievable Instructions per Cycle (IPC) on the system to the IPC it achieves on N-core in an isolated execution. *We use **adaptive many-cores** in this work, which are special types of many-cores [10], [11] that can speedup both single-threaded and multi-threaded tasks allowing us to explore diverse mix of workloads*. Figure 1 shows single-core IPC slowdown of two benchmarks (*bzip2* and *lbm*) averaged over every ten million of their instructions executed. To ensure optimal fairness at all times, a fair scheduler needs to redistribute cores from tasks entering low requirement phases to tasks entering high requirement phases. If cores are scarce, then the scheduler needs to ensure that all tasks experience similar resource crunch in the form of equal slowdown. This leads to the fairness problem.

We choose **variance in slowdowns** amongst the entire population of executing tasks as our fairness metric in this work [12]. Variance here quantifies the dispersion in the slowdowns of tasks being executed. Variance is **zero** when core allocation amongst them is completely fair and all tasks are experiencing exactly the same slowdown on the many-core. The metric can also detect task starvations. Starvation occurs when a task is denied execution by allocating zero cores. Slowdown of a task with no core assigned is infinite, making the variance infinite even if one task starves.

978-3-9815370-6-2/DATE16/

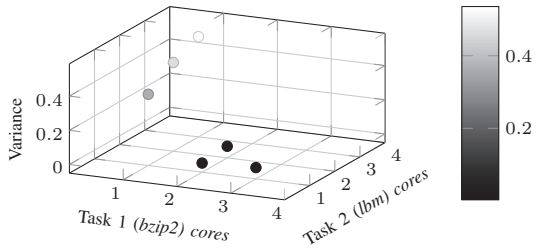
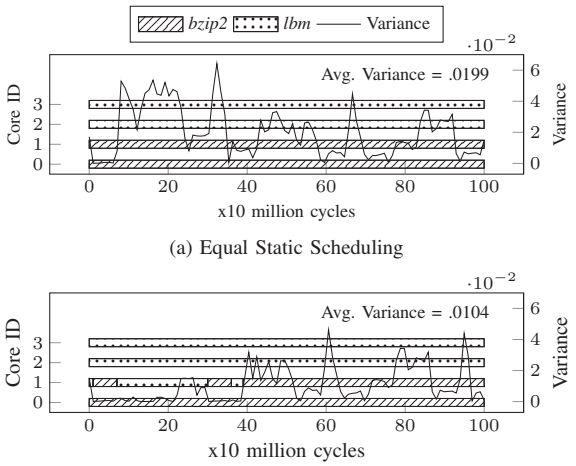


Fig. 2: Initial variance in 6 possible non-zero core allocations when 4 cores are distributed between 2 tasks (*bizp2* and *lbm*).



(a) Equal Static Scheduling

(b) Optimal Dynamic Fair Scheduling (Brute Force)

Fig. 3: Reduction in average variance by 47.69% achieved using dynamic fair scheduling in comparison to equal static scheduling on 4-core system with 2 tasks (*bzip2 and* lbm).

Pareto-Optimal Motivation: Fairness should not cause under-utilization of system resources. Figure 2 illustrates a simple example with the initial variance under six different possible allocation states when four cores need to be distributed amongst two tasks (*bizp2* and *lbm*) without starvation. Amongst all allocations, variance in slowdowns is minimum for allocation (2,1) at 0.002 but it does not allocate one of the available cores. This allocation is not Pareto-optimal, which in our context means that there is a core allocation possible in which slowdown of one task can be decreased without increase in slowdown of another task. A distributed fair scheduler needs to ensure that it does not converge to such points for the sake of fairness. In contrast, allocations (1,3), (2,2), (3,1) in which all four cores are allocated are not equally fair because of imbalance in slowdowns of the two tasks in those allocations. In our example amongst all Pareto-optimal allocations, allocation (3,1) has the lowest variance of 0.015. Note that it is not always possible to achieve the complete fairness (i.e. zero variance) because a task can be allocated only a discrete number of cores. *We define Pareto-optimal task-to-core allocations with minimum variance in task execution slowdowns as an optimally fair allocation for many-cores.*

Dynamic Scheduling Motivation: We show the decrease in variance on multi-/many-core systems can be provided by an optimal (brute force) phase-aware dynamic fair scheduling. It redistributes cores every scheduling epoch in comparison to a static scheduler that gives equal cores to all tasks. The brute force solution is computationally too expensive to run on systems with many-cores. We run two tasks (*bzip2* and *lbm*) on a 4-core system. Figures 3a and 3b show core allocations over time with corresponding instantaneous variance under the static- and dynamic scheduler, respectively. Figure 3 shows that the dynamic scheduler reduces average variance in slowdowns by 47.49% when compared to the static scheduler. The reduction in variance is obtained due to back and forth transfer of core "1" amongst the two tasks based on their relative instantaneous demands.

Our Novel Contribution: In this work, we present a scheduler called *DFMS* (Distributed Fair Many-Cores Scheduler). DFMS performs dynamic scheduling to ensure that the tasks executing in a many-core system experience near-equal slowdowns throughout their execution. It distributes fair scheduling processing overhead across all the cores enabling scaling up with increase in number of cores.

The problem of variance minimization is well-studied [13] and is known to be **NP-Hard** [14] in scheduling for a general case. *In this work we show that based on observations made on many-cores, its fair scheduling problem's structure can be exploited to obtain an optimal fair schedule in polynomial time under some conditions.*

II. Fair Scheduling under *DFMS*

System Overview: *DFMS* uses a Multi-Agent System (MAS) to perform distributed fair scheduling for an N-core system executing A tasks. There is one-to-many mapping between tasks and cores, where one task can be assigned multiple discrete numbers of cores but a core cannot be assigned to more than one task. In this work, we assume that number of tasks is always less than the number of cores. Therefore, context-switching is not required. This keeps our fair scheduling problem discrete allowing us to create less complex and more scalable solutions. We also assume tasks are malleable, which means that the number of cores allocated to tasks can change during their execution.

System Model: We assign a unique agent to every executing task. Let there be A agents managing A tasks indexed by x. Each agent x holds C_x number of cores at a given time to execute its assigned task, which can be traded with other agents. C represents the current system-wide task-to-core allocations (system state). $\gamma_x(C_x)$ represents the instantaneous slowdown of the task associated with agent x when assigned C_x cores. $\bar{\gamma}(C)$ is the average instantaneous slowdown of all tasks under allocation C. $SS(C)$ and $\sigma^2(C)$ denote the corresponding sum of squares and variance, respectively.

$$\bar{\gamma}(C) = \frac{1}{A}\sum_{x=1}^{A}\gamma_x(C_x) \quad (1)$$

$$SS(C) = \sum_{x=1}^{A}\gamma_x(C_x)^2 \quad (2)$$

$$\sigma^2(C) = \frac{1}{A}\sum_{x=1}^{A}(\gamma_x(C_x)-\bar{\gamma}(C_x))^2 \quad (3)$$

Utility Model: Agents transfer cores amongst each other based on a utility function. Let $u_{i\to j}(\delta)$ represent the utility

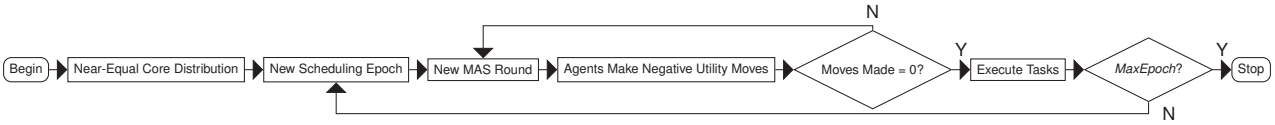


Fig. 4: Execution Flow for *DFMS*

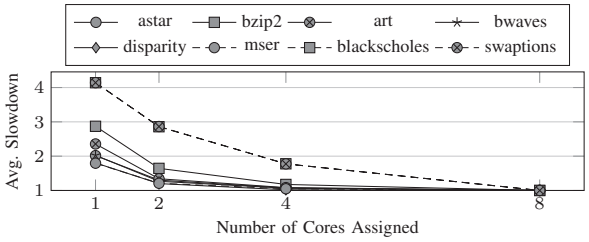


Fig. 5: Average slowdown observed in benchmarks of different types when assigned different numbers of cores.

of transferring δ cores from agent i to agent j assuming this is the only move possible. $\sigma^2(C)$ by definition is the mean of square of distance between slowdowns and mean slowdown. When δ cores are transferred from i to j, $\gamma_i(C)$ and $\gamma_j(C)$ change to $\gamma_i(C-\delta)$ and $\gamma_j(C+\delta)$, respectively. Their distances from the mean $\bar{\gamma}(C)$ change. But, $u_{i\to j}(\delta)$ also changes the mean itself from $\bar{\gamma}(C)$ to $\bar{\gamma}(C')$, where C' is the modified state after the move. This results in a change in distance from the mean for all elements. To make the problem tractable, effect of a move on other elements can be temporarily ignored by assuming $\bar{\gamma}(C) \approx \bar{\gamma}(C')$. Therefore, variance from a move will decrease if combined change in the square of distance of the new slowdowns from the mean is less than the original ones. This inspires our definition for $u_{i\to j}(\delta)$.

$$u_{i\to j}(\delta)=\gamma_i(C_i-\delta)^2+\gamma_j(C_j+\delta)^2-\gamma_i(C_i)^2-\gamma_j(C_j)^2 \quad (4)$$

Execution Flow: Figure 4 depicts the execution flow of *DFMS* graphically. Initially, all cores are distributed near-equally amongst all executing tasks. Every scheduling epoch, a series of MAS rounds take place to make core allocations fairer. Rounds stop when task-to-core allocations cannot be made any fairer. Scheduling epoch is the granularity at which scheduling is performed. *DFMS* operates at a granularity of 10 million cycles by default. In every MAS round, all agents communicate and exchange their instantaneous slowdowns. Agents then calculate the utilities of moves involving transferring cores they hold to every other agent. During a round, every agent randomly picks another agent with which negative utility moves are possible and exchanges cores with it. MAS rounds stop for the scheduling epoch when a round is completed without any negative utility move performed. Tasks are then executed with the converged allocation, following which the system state changes. MAS rounds commence again in the next scheduling epoch till scheduling ends at "*MaxEpoch*". Instantaneous slowdowns of the tasks can be predicted at runtime using performance prediction models for adaptive many-cores presented in [15] to avoid any profiling.

Slowdown Observations: Figure 5 shows the average slowdowns in embedded benchmarks of different types when assigned different non-zero number of cores. Slowdown in a task (benchmark) $\gamma_x(C_x)$ associated with an agent x **decreases monotonically** with increase in number of cores C_x because each core allocation brings with it a non-negative increase in task's IPC moving it closer to its maximum achievable IPC. Therefore, it is always beneficial to stay in Pareto-optimal allocation and keep cores assigned to any task rather than keeping them free. Slowdown is also **convex** with increase in number of cores because increase in IPC brought by each subsequent core allocation is less than the previous one until it saturates at the maximum possible IPC. This is because of the saturation of exploitable instruction level parallelism and thread level parallelism required by many-cores to speedup tasks. To the best of our knowledge, this is the first work, which exploits these properties in the context of the fair scheduling problem on many-cores.

Optimal Fairness: We now present proofs of theoretical guarantees for optimality and convergence provided by *DFMS*.

Lemma 1. DFMS *converges to Pareto-optimal allocation that minimizes the sum of squares of slowdowns SS on many-cores in O(A) rounds.*

Proof. After $u_{i\to j}(\delta)$, $SS(C)$ changes to $SS(C')$.

$$\begin{aligned} SS(C') &= \textstyle\sum_{x=1}^{A} \gamma_x(C_x)^2+\gamma_i(C_i-\delta)^2+\gamma_j(C_j+\delta)^2-\gamma_i(C_i)^2-\gamma_j(C_j)^2 \\ &= SS(C)+u_{i\to j}(\delta) \quad [\because \text{Eq. (2) and Eq. (4)}] \end{aligned}$$

So $SS(C') > SS(C)$ when $u_{i\to j} > 0$ and $SS(C') < SS(C)$ when $u_{i\to j} < 0$. Hence, only negative utility moves can reduce SS from any given state. When no negative utility move exists, SS cannot be reduced further and a minimum is reached. We now prove that this minimum minimizes SS.

Slowdowns are piecewise linear functions that are too computationally expensive to optimize [16]. Based on observations made in Figure 5, we assume that that slowdown $\forall_x\ \gamma_x(C_x)$ is convex-extensible to a non-negative discrete convex function of C_x on many-cores and error introduced by this convex relaxation [17] is minimal. Since square of a non-negative convex function remains convex, $\gamma_x(C_x)^2$ is a convex function of C_x. $SS(C)$ is a positive sum of convex functions, therefore is a discrete convex function of C. Every minima of a discrete convex function are its global minima [18].

Since cores-to-task allocations are discrete, there exist only a finite number of negative utility moves. Two agents i and j that exchange δ cores once will not exchange cores again unless disturbed by a third agent if δ minimizes $u_{i\to j}(\delta)$. $u_{i\to j}(\delta)$ is also a discrete convex function of δ that can be

minimized efficiently using gradient descent. We force agents in *DFMS* to always make δ minimizing moves. Thus, negative utility moves will exhaust in at worst $O(A)$ rounds, since an agent can interact with at most A agents (excluding repeated interactions). Under *DFMS*, by design only Pareto-optimal allocation are explored as cores are always transferred from one executing task to the other; hence proved. □

Theorem 1. DFMS *converges to an optimally fair allocation under a given performance constraint.*

Proof. Beginning with Equation (3),

$$\begin{aligned}
\sigma^2(C) &= \tfrac{1}{A}\textstyle\sum_{x=1}^{A}(\gamma_x(C_x)-\bar{\gamma}(C))^2 \\
&= \tfrac{1}{A}\textstyle\sum_{x=1}^{A}\big(\gamma_x(C_x)^2+\bar{\gamma}(C)^2-2\gamma_x(C_x)\bar{\gamma}(C)\big) \\
&= \tfrac{1}{A}\big(\textstyle\sum_{x=1}^{A}\gamma_x(C_x)^2+\sum_{x=1}^{A}\bar{\gamma}(C)^2-2\sum_{x=1}^{A}\gamma_x(C_x)\bar{\gamma}(C)\big) \\
&= \tfrac{1}{A}\big(\textstyle\sum_{x=1}^{A}\gamma_x(C_x)^2+A\bar{\gamma}(C)^2-2A\bar{\gamma}(C)^2\big) \quad [\because \text{Eq. (1)}] \\
&= \tfrac{1}{A}\big(\textstyle\sum_{x=1}^{A}\gamma_x(C_x)^2-A\bar{\gamma}(C)^2\big) \\
&= \tfrac{1}{A}\big(\textstyle\sum_{x=1}^{A}\gamma_x(C_x)^2\big)-\bar{\gamma}(C)^2 \\
&= \tfrac{1}{A}SS(C)-\bar{\gamma}(C)^2 \quad [\because \text{Eq. (2)}] \qquad (5)
\end{aligned}$$

We measure many-core performance as the sum of slowdowns. We operate *DFMS* with an extra condition that performance $\sum_{x=1}^{A}\gamma_x(C_x)$ should not change while performing $u_{i\to j}(\delta)$. Based on Equation (1), this makes $\bar{\gamma}(C)$ a constant independent of C since A is also a constant. Therefore, based on Equation (5) variance $\sigma^2(C)$ is minimized when $SS(C)$ is minimized; hence proved using Lemma 1. □

Polynomial time algorithm for maximizing performance on many-cores is known [19]. Since slowdowns on many-cores saturate after certain number of core allocations, they are convex but not strictly convex. Therefore, there exist multiple allocations with maximum performance but not all of these allocations are equally fair in slowdowns. Theorem 1 as a special case allows search of allocations with optimal fairness under optimal performance in polynomial time. To the best of our knowledge, ours is the first work that present this result.

Heuristic Fairness: When performance is not fixed, $\bar{\gamma}(C)$ remains a function of C. $-\bar{\gamma}^2(C)$ is a concave function of C. Based on Equation (5), $\sigma^2(C)$ is a sum of a concave function and convex function, which is neither concave nor convex. This makes variance hard to minimize optimally even while executing tasks with convex slowdowns. For a general case, we change *DFMS* to be a heuristic distributed local search.

When δ cores are transferred from agent i to agent j, variance changes from $\sigma^2(C)$ to $\sigma^2(C')$. Let Δ_S and Δ_{SS} represent the difference between new slowdowns and old slowdowns, and difference between square of new slowdowns and square of old slowdowns, respectively.

$$\begin{aligned}
\Delta_S &= \gamma_i(C_i-\delta)+\gamma_j(C_j+\delta)-\gamma_i(C_i)-\gamma_j(C_j) \\
\Delta_{SS} &= \gamma_i(C_i-\delta)^2+\gamma_j(C_j+\delta)^2-\gamma_i(C_i)^2-\gamma_j(C_j)^2 \\
\sigma^2(C') &= \tfrac{1}{A}(SS(C)+\Delta_{SS})-(\bar{\gamma}(C)+\tfrac{\Delta_S}{A})^2 \quad \text{[Similar to Eq. (5)]} \\
&= \tfrac{SS(C)}{A}+\tfrac{\Delta_{SS}}{A}-\bar{\gamma}(C)^2-\tfrac{\Delta_S^2}{A^2}-\tfrac{2\bar{\gamma}(C)\Delta_S}{A} \\
&= \sigma^2(C)+\tfrac{\Delta_{SS}}{A}-\tfrac{\Delta_S^2}{A^2}-\tfrac{2\bar{\gamma}(C)\Delta_S}{A} \quad [\because \text{Eq. (5)}]
\end{aligned}$$

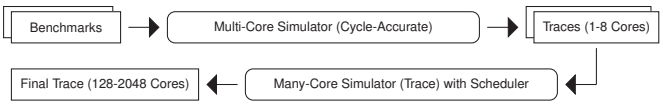


Fig. 6: Experimental Setup.

Now $\sigma^2(C') < \sigma^2(C)$ if,

$$\begin{aligned}
0 &> \tfrac{\Delta_{SS}}{A}-\tfrac{\Delta_S^2}{A^2}-\tfrac{2\bar{\gamma}(C)\Delta_S}{A} \\
&> \Delta_{SS}-\tfrac{\Delta_S^2}{A}-2\bar{\gamma}(C)\Delta_S \\
&> \Delta_{SS}-2\bar{\gamma}(C)\Delta_S \quad [\because A>>\Delta_S^2 \text{ on many-cores}] \qquad (6)
\end{aligned}$$

We redefine $u_{i\to j}(\delta) = \Delta_{SS} - 2\bar{\gamma}(C)\Delta_S$, for minimizing variance under no performance constraint. Note that this version of *DFMS* is not optimal because it does not guarantee that local minima reached are also the global minima. *Other heuristics can also be used here but the question of finding a polynomial time distributed algorithm for performance independent optimal fairness on many-cores still remains open.*

Complexity: Under *DFMS*, every agent does $O(A)$ utility calculations in every round. Each utility calculation $u_{i\to j}(\delta)$ is required to find the value of δ, requiring a worst-case of $O(N)$ calculations. Further, there can be at worst $O(A)$ rounds. Hence, worst-case complexity of total calculations is $O(A^2N)$ in a scheduling epoch. Calculations are randomly distributed over N cores and therefore per-core worst-case calculations are approximately $O(A^2)$. Since $O(A)$ messages need to be broadcasted every round, *DFMS* has a total communication complexity of $O(A^2)$ per scheduling epoch. *DFMS* has only $O(1)$ space complexity since no additional data structure needs to be maintained for performing scheduling.

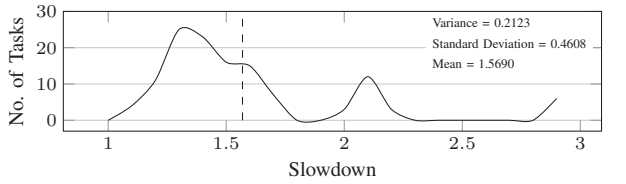
III. Experimental Evaluations

We evaluate *DFMS* using a two stage simulator as shown in Figure 6. The first stage of this simulator is derived from cycle-accurate *gem5* [20] simulator based on *Bahurupi* [11] adaptive many-core architecture. It is used to obtain isolated execution traces of tasks with up to eight cores allocated. These traces are then used in a second trace-driven simulator to produce approximate time-wise feasible many-core simulations with hundreds of processing cores and executing tasks. Each core in the many-core is a 2-way out-of-order core implementing ARMv7 ISA with a 4-way associative 64KB L1 instruction- and data caches. Cores share an 8-way associative 2MB unified L2 cache. All caches have a line size of 64 bytes. This experimental setup is unable to capture the performance impact of concurrent task execution due to different communications involved vis-a-vis shared-cache and NoC [21]. We plan to extend this work with a communication model in the future.

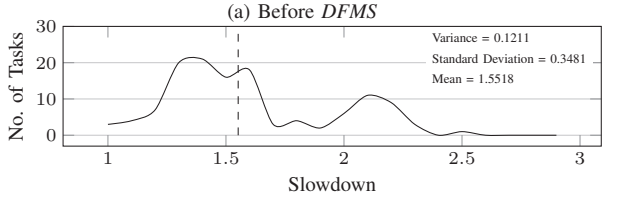
We use 36 benchmarks (26 sequential and 10 parallel) as listed in Table I from SPEC- 2000 and 2006 [22], SD-VBS [23], PARSEC [24] and SPLASH-2 [25] suites with "ref", "full-hd" and "sim-small" inputs, respectively to create workloads. All benchmarks are executed in syscall emulation mode. Workloads are created from random composition of all available benchmarks with uniform distribution.

TABLE I: Benchmarks used in the evaluations.

Type	*Benchmark Name*
Integer	astar, bzip2, gobmk, h264ref, hmmer, mcf, omnetpp, perlbench, sjeng, twolf, vortex
Float	art, bwaves, calculix, equake, gemsfdtd, lbm, namd, povray, tonto
Vision	disparity, mser, sift, svm, texture, tracking
Parallel	blackscholes, cholesky, fmm, fluidanimate, lu, radix, radiosity, swaptions, streamcluster, water-sp



(a) Before *DFMS*



(b) After *DFMS*

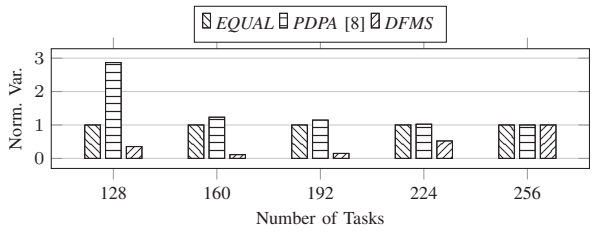
Fig. 7: Distribution of slowdowns of executing tasks around mean slowdown before and after *DFMS* is applied on 256-core system with 128 tasks with an initial equal core distribution under performance constraint.

Scheduling is performed at a granularity of 10 million cycles, which translates to a scheduling epoch of 10ms on system running at 1GHz. All simulations are executed for a minimum of two billion cycles. For our evaluation, we use closed systems where task instances restart execution from the beginning as soon as they finish. Closed systems facilitate scheduler evaluations, but *DFMS* is not limited to them.

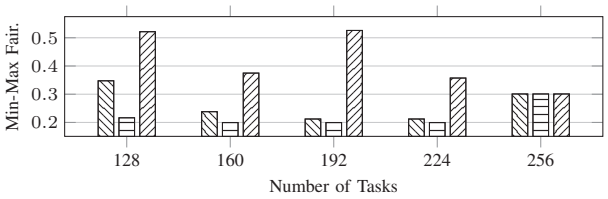
Comparative Baselines: We compare *DFMS* against two other schedulers to prove its efficacy. *EQUAL* is a static fair scheduler that distributes cores near-equally amongst all executing tasks. We choose to compare against this simple approach to show that *EQUAL* does not result in fair scheduling even though it is intuitive, scalable and easy to implement.

We also choose to compare against a heuristics based dynamic fair scheduler for many-cores called *PDPA* [8], which works on the notion of "*ExecutionEfficiency*" defined as speedup per unit core. Note that *ExecutionEfficiency* is neither a convex nor a concave metric on many-cores even though speedup can be assumed to be a concave function. It has two empirically determined thresholds *target_eff* and *high_eff*, whose values are based on the system load. Tasks are allocated cores in every scheduling epoch so that their execution efficiencies are between *target_eff* and *high_eff*. We choose to compare against this approach (with default threshold values) to show that threshold based heuristics though scalable and easy to implement, can be easily outperformed. *PDPA* is modified to enforce Pareto-optimality for fair comparison.

Optimal Fairness for Fixed Performance: Theoretically there exist an exponential number of allocations of equal



(a) Variance in Slowdown based Fairness (Lower Value Better)



(b) Min-Max Slowdown Ratio based Fairness (Higher Value Better)

Fig. 8: Fairness under different schedulers on a closed 256-core system with varisized workloads. *DFMS* provides superior fairness in comparison to other schedulers.

performance with different fairness. In practice, such allocations only exist when there exists a core transfer such that increase in slowdown of one task is exactly equal to decrease in slowdown in another task. This would be at best rare in real world under full precision. We define $\Delta_{\bar{\gamma}} = \Delta_S/\bar{\gamma}(C)$ as change in sum over mean. Equation (6) can be rewritten as $0 > \Delta_{SS} - 2\bar{\gamma}^2(C)\Delta_{\bar{\gamma}}$ to include $\Delta_{\bar{\gamma}}$. As long as $\Delta_{\bar{\gamma}} \approx 0$, variance can be minimized near-optimally using *DFMS*. We run *DFMS* under a constraint $|\Delta_{\bar{\gamma}}| \leq .01$, the closest to zero we can get in our experiments for any reconfiguration to happen. The bounds on $\Delta_{\bar{\gamma}}$ can be relaxed further but only with loss of optimality. On a 256-core system with 128 tasks this results in 42.95% reduction in variance with only 1.09% change in mean from an initial equal core distribution. Figure 7 shows how the physical distribution of slowdowns of the executing tasks shifts around the mean slowdown in the system before and after *DFMS* is applied. It can be seen that after *DFMS* the dispersion in the slowdowns reduces substantially resulting in more fairness.

Improved Fairness for Variable Performance: *DFMS* operates heuristically using Equation (6) as utility when performance is unspecified. Figure 8a shows average variance on a 256-core system with varisized workloads under different schedulers (normalized to *EQUAL* scheduler). With full load (256 tasks), the completely fair allocation is simply to assign one core to each task as under any other allocation a task will inevitably starve pushing system variance to infinite. All schedulers are able to discover the same optimal solution. As system load decreases further, number of surplus cores increase. This results in expansion of the optimization search-space, making it more challenging to maintain fair allocations. Figure 8a shows that *DFMS* results in better fair schedules in comparison to *EQUAL* and *PDPA* under all loads. EQUAL performs worse since tasks have inherently different execution patterns resulting in different slowdowns for the same number

TABLE II: Different overheads when scheduling under *DFMS* on varisized systems while executing half loads. Overheads are normalized to the first test case of 128-core system.

Cores	Tasks	Total Proc.	Per-Core Proc.	Comm.
128	64	1x	1x	1x
256	128	5.64x	2.74x	1.25x
512	1024	25.90x	6.16x	3.01x
1024	2048	103.82x	12.34x	9.06x
2048	1024	415.69x	24.41x	16.60x

of allocated cores as shown in Figure 5. *PDPA* does not work because there is no unique set of thresholds that can result in optimal fair schedule for all possible kind of workloads.

Another common metric used to measure fairness is the ratio of minimum and maximum slowdown amongst all executing task in the system. Value "1" indicates maximum fairness while value "0" indicates the minimum fairness. We observed in our experiments that this metric has high correlation to the variance metric that *DFMS* optimizes. Figure 8b shows average min-max fairness for different schedulers. Evaluations show that DFMS also performs better than the baselines when using the min-max fairness metric.

Scalability: True overhead numbers can only be shown in real hardware and also depends upon the quality of the implementation. In our simulations, we assume average number of message broadcasted per scheduling epoch as an approximate measure of communication overhead and average number of floating point operations per scheduling epoch for scheduling estimation as an approximate measure of processing overhead. Total processing overhead is measured as floating point operations performed across all cores, while per-core processing overhead is measured as maximum floating point operations performed by a core amongst all cores in a scheduling epoch.

Table II shows how different overheads increase as we move from 128-core system to 2048-core system, all with half loads. The overheads are normalized to 128-core system test case. The observations made are in sync with the theoretical complexity of *DFMS* stated in Section II.

IV. Conclusion

In this paper, we proposed a fair scheduler for many-cores called *DFMS*. *DFMS* is a lightweight dynamic scheduler that can be applied at runtime for constant partial reallocation of processing cores to maintain fair allocation at all times even under the continuously changing processing requirements of the executing tasks. *DFMS* is proven theoretically to converge to the optimal fair allocation state for a given performance. Further, by distributing its fair scheduling processing overhead across all cores in the system, it can scale up as the number of cores on many-core chips increase.

Acknowledgement

This work was supported in parts by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89) and in parts by Huawei International Pte. Ltd. research grant in Singapore.

References

[1] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe, "Invasive Manycore Architectures," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

[2] J.-A. Carballo, W.-T. J. Chan, P. Gargini, A. Kahng, and S. Nath, "ITRS 2.0: Toward a Re-Framing of the Semiconductor Technology Roadmap," in *International Conference on Computer Design (ICCD)*, 2014.

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

[4] C. S. Pabla, "Completely Fair Scheduler," *Linux Journal*, 2009.

[5] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, "ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems," in *Symposium On Applied Computing (SAC)*, 2015.

[6] C. Wu, J. Li, D. Xu, P.-C. Yew, J. Li, and Z. Wang, "FPS: A Fair-Progress Process Scheduling Policy on Shared-Memory Multiprocessors," *Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

[7] T. Ebi, M. Faruque, and J. Henkel, "TAPE: Thermal-Aware Agent-Based Power Economy Multi/Many-core Architectures," in *International Conference On Computer Aided Design (ICCAD)*, 2009.

[8] T. Sun, H. An, T. Wang, H. Zhang, and X. Sui, "CRQ-Based Fair Scheduling on Composable Multicore Architectures," in *International Conference on Supercomputing (ICS)*, 2012.

[9] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase Behavior in Serial and Parallel Applications," in *International Symposium on Workload Characterization (IISWC)*, 2012.

[10] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *SIGARCH Computer Architecture News*, 2007.

[11] M. Pricopi and T. Mitra, "Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture," *Transactions on Architecture and Code Optimization (TACO)*, 2012.

[12] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-Threaded Processors," *Computer Architecture Letters (CAL)*, 2011.

[13] W. Kubiak, "Completion Time Variance Minimization on a Single Machine is Difficult," *Operations Research Letters (ORL)*, 1993.

[14] T. Baker, J. Gill, and R. Solovay, "Relativizations of the P=NP Question," *SIAM Journal on Computing (SICOMP)*, 1975.

[15] V. Vanchinathan, "Performance Modeling of Adaptive Multi-core Architecture," Master's thesis, National University of Singapore, 2015.

[16] A. Toriello and J. P. Vielma, "Fitting Piecewise Linear Continuous Functions," *European Journal of Operational Research (EJOR)*, 2012.

[17] E. Chlamtac and M. Tulsiani, "Convex Relaxations and Integrality Gaps," in *Semidefinite, Conic and Polynomial Optimization*, 2012.

[18] K. Murota, "Submodular Function Minimization and Maximization in Discrete Convex Analysis," *RIMS Kokyuroku Bessatsu B*, 2010.

[19] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger, "Multitasking Workload Scheduling on Flexible-Core Chip Multiprocessors," in *Parallel Architectures and Compilation Techniques*, 2008.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The gem5 Simulator," in *SIGARCH Computer Architecture News*, 2011.

[21] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-Aware Heuristics for Run-Time Task Mapping on NoC-Based MPSoC Platforms," *Journal of Systems Architecture (JSA)*, 2010.

[22] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, 2006.

[23] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego Vision Benchmark Suite," in *International Symposium on Workload Characterization (IISWC)*, 2009.

[24] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *SIGARCH Computer Architecture News*, 1995.