# Scalable Dynamic Task Scheduling on Adaptive Many-Core

Vanchinathan Venkataramani[†], Anuj Pathania[†], Muhammad Shafique[+], Tulika Mitra[†], Jörg Henkel[*]
[†]School of Computing, National University of Singapore, Singapore
[*]Chair of Embedded System, Karlsruhe Institute of Technology, Germany
[+]Institute of Computer Engineering, Vienna University of Technology (TU Wien), Austria
Corresponding Author: vvanchi@comp.nus.edu.sg

*(Invited Paper)*

***Abstract*—Workloads from autonomous systems project an unprecedented processing demand onto their underlying embedded processors. Workload comprises of an ever-changing mix of multitudes of sequential and parallel tasks. Adaptive many-core processors with their immense yet flexible processing potential are up to the challenge. Adaptive many-core house together tens of base cores capable of forming more complex cores at run-time. Adaptive many-cores, therefore, can accelerate both sequential and parallel tasks whereas non-adaptive many-cores can only accelerate the latter. Adaptive many-cores can also reconfigure themselves to conform to the needs of any workload whereas non-adaptive many-cores - homogeneous or heterogeneous - are inherently limited given their immutable design. The accompanying qualitative schedule is the key to achieving the real potential of an adaptive many-core. The scheduler must move base cores between tasks on the fly to meet the goals of the overlying autonomous system. The scheduler also needs to scale up with the increase in the number of cores in adaptive many-cores without making compromises on the schedule quality. We present a near-optimal distributed scheduler for maximizing performance on adaptive many-cores. We also introduce an online performance prediction technique for adaptive many-cores that enable the proposed scheduler to operate without any task profiling.**

## I. Introduction

General-purpose processors can speed up the execution of tasks by exploiting two kinds of parallelism inherent in the code, namely Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). Current-generation many-core architectures with a number of simple cores on chip are perfectly attuned to accommodate TLP code where the threads are distributed across the cores. Each individual simple core, however, lacks the aggressive mechanisms — such as wide-issue superscalar out-of-order (ooo) execution — required to take advantage of ILP. Thus, the sequential code fragments with ILP (but no TLP) becomes the performance bottleneck of the entire task according to Amdahl's Law [1].

Adaptive many-cores are upcoming processors [2] containing a set of simple physical cores that can naturally exploit TLP; but subsets of these cores can be coalesced together at runtime to form varisized wide-issue ooo *virtual* cores capable of extracting ILP from the sequential code fragment and transparently accelerate task execution without programmer intervention. Thus, adaptive many-cores are equally adept in handling ILP and TLP code fragments.

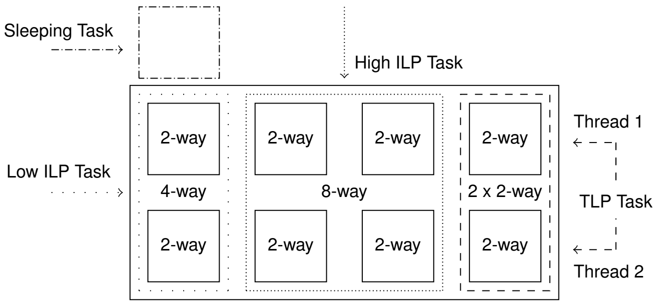

Fig. 1: A generic adaptive many-core architecture with eight 2-way base cores. The number of cores allocated to a task depends on the task's ILP/TLP potential.

*The key to reach the true potential of an adaptive many-core architecture is the quality of the scheduler.* The objective of this work is to design an effective dynamic scheduling strategy for ILP (and/or TLP) tasks on an adaptive many-core. This is challenging as the scheduler has to determine the granularity (size) of the virtual cores in addition to the spatio-temporal mapping of the tasks to the cores. Moreover, as the parallelism profile of a task varies across different phases of the task, a runtime scheduling policy with low overhead is imperative compared to a static policy. In this paper, we propose a distributed scheduler that satisfies all these considerations with the goal of maximizing the performance.

A generic adaptive many-core architecture considered in this paper comprises of $n$ simple base cores each with $z$-way issue ooo pipeline. We need to execute $m$ tasks on this architecture. Each of the $m$ tasks can be assigned multiple cores for execution. A sequential single-threaded task is assigned $x \leq n$ base cores forming a virtual $x \times z$-way issue ooo core extracting ILP. A parallel multi-threaded task is assigned $x \leq n$ base cores that exploit TLP by executing the threads of the task in parallel on $x$ cores. The number of cores assigned to a task depends on its ILP/TLP potential. A task that is not assigned any base core is put to sleep. The above formulation can be conceptually applied to several of the proposed adaptive multi/many-core architectures discussed in Section II with minimal modifications for architecture specific constraints. *We avoid adding constraints imposed by any specific architecture to keep this work generalized.* Homogeneous many-cores are a special case of the formulation where only TLP tasks can be allocated to more than one core.

Figure 1 illustrates a generic adaptive many-core where varisized virtual cores are executing single-threaded tasks with different levels of ILP, along with two threads of a multi-threaded task executing in parallel on different cores. There is also a sleeping task that is awaiting assignment of base cores.

Schedulers are Operating System (OS) sub-routines that optimize underlying processor's performance by determining task-to-core assignments [3]. Existing schedulers for adaptive multi-cores [4], [5] are inadequate as they require offline task profiling. These schedulers being centralized, will also not scale up with increasing number of base cores. Furthermore, the parallelism profile of a task varies over its lifetime [6] enforcing the need for lightweight dynamic schedulers that can re-organize the underlying core assignments at runtime according to the workload. Centralized schedulers also act as a central point of failure.

**Our Novel Contributions and Concept Overview:** Multi-Agent System (MAS) based schedulers are inherently distributive [7] and allow proper analysis when designed formally. We design a dynamic MAS scheduler called *DPMS* (Distributed Performance Many-Core Scheduler) for adaptive many-cores. *DPMS* avoids profiling by employing a new performance prediction technique for adaptive many-cores presented here. *DPMS can also be applied to existing homogeneous many-cores, latter being a special case of adaptive many-cores.* It guarantees convergence to a solution in given number of steps from any initial state. The proposed scheduler results in 200x reduction in the overheads compared to a centralized oracle scheduler on a 64-core many-core system.

Our contributions in this work are as follows.

- We present an optimal MAS scheduler called *DPMS*, which efficiently performs distributive dynamic scheduling on adaptive many-cores. The scheduler is derived from our work originally presented in [8].
- We develop a performance prediction technique for single-thread benchmarks on adaptive many-cores, which enables *DPMS* to operate efficiently without profiling.
- We quantize the superior scalability of our proposed distributed scheduler *DPMS* against state-of-the-art centralized scheduler in many-cores using real-world representative cycle-accurate simulations.

## II. Related Work

Several adaptive multi-/many-core architectures have been proposed in the literature. We briefly summarize few of them - *Core Fusion* [9] can fuse symmetric out-of-order cores using a complex reconfigurable hardware; *T-Flex* [10] can create larger (powerful) cores from smaller (weaker) cores, which use *EDGE* ISA; *Bahurupi* [11] coalesce simple symmetric out-of-order cores by a hardware-software co-design and use of sentinel instructions; *Federation* [12] combines simple scalar in-order cores to make a complex out-of-order core by introducing additional logic to internal pipeline stages; *WiDGET* [13] is a reconfigurable processor whose execution units can merge together dynamically to become more capable; *Voltron* [14] can couple symmetric cores together, which then execute multiple instruction streams in parallel, similar to a *VLIW* processor. Authors in [15] proposed a framework (by making modifications to existing OS) in which a set of functions are provided for implementing schedulers in adaptive multi-cores. However, they do not propose schedulers.

The proposed *DPMS* scheduler operates on a generalized abstraction of simple symmetric out-of-order base cores forming larger (complex and powerful) out-of-order cores through coalitions. The idea should be applicable to almost all the reviewed adaptive multi-core architectures though this is not empirically tested. Thus, *DPMS* can be extended to work on any of those proposed adaptive architectures.

Analytical models have been used for performance prediction of different types of processors. Analytical performance models are built either using mechanistic or empirical modeling. The mechanistic models [16] utilize processor architecture details to perform predictions, while empirical models [17] treat processors as a black box and use statistical techniques such as regression. Authors in [18] proposed a hybrid mechanistic-empirical model for CPI stack [19] for symmetric multi-cores. The work presented in [20] used mechanistic-empirical modeling to do performance prediction for asymmetric multi-cores. *DPMS* also needs to do performance prediction for tasks being executed on varisized core-coalitions, for which we employ mechanistic-empirical modeling. Authors in [21] proposed a specialized performance prediction technique for *TFlex* adaptive multi-core using specific depth estimators. In contrast, we propose a generic architecture-agnostic technique based on standard hardware counters.

Performance oriented scheduling for adaptive multi-cores has been previously researched. Threshold based scheduler (*THRESH*) for Bahurupi adaptive multi-core was presented in [5]. It allocates a task to base cores (if available) as long as the average speedup of every consequent base core allocation is more than an empirically determined threshold but does not provide any specific order in which the tasks must be evaluated. The scheduler is simple to implement but still requires profiled information before hand in the form of expected speedups of tasks on different sized coalitions. A fixed threshold impacts its efficiency as there is no single ideal threshold value that can work efficiently for all possible workloads. If the threshold value is too high, fewer core-coalitions would form that could cause system underutilization. If the value is too low, the tasks evaluated first will consume all the base cores, leaving none for the tasks evaluated later even if allocations to later tasks can result in higher performance. Furthermore, a similar threshold based scheduler called *PDPA* was also presented in [4] and suffers from similar drawbacks as the scheduler in [5].

Authors in [4] presented a set of schedulers for *T-Flex* adaptive multi-core, amongst which *PROFILE* performed the best. *PROFILE* also assumes that the average speedups for all tasks being executed in the system are known. It then maximizes the total average speedup using Dynamic Programming (DP). The advantage of *PROFILE* over threshold based schedulers is that it does not require any user-defined parameter; but DP schedulers have higher overheads if executed continuously

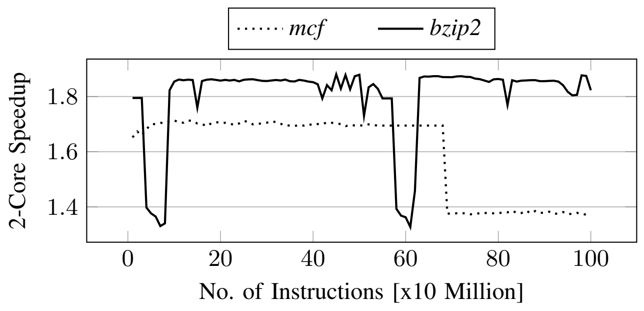

Fig. 2: Execution profiles of *mcf* and *bzip2* benchmarks showing entropy in their speedup during execution [8].

at run-time. Furthermore, both *THRESH* and *PROFILE* are schedulers operating on average values that miss out on performance enhancing opportunities available from exploitation of variations in instantaneous speedups of tasks. Another promising alternative is to use fast centralized algorithms that can operate efficiently under certain assumptions [22]. All centralized algorithms inherently suffer from the shortcoming of having a single point of failure.

MAS design provides an option for developing distributed schedulers [23], [24]. Authors in [25] introduce a heuristic MAS called *DistRM* that uses agents performing local communications to achieve a good enough solution with low overheads. MAS schedulers based on game theory [26]–[28], with foundations in Nash Equilibrium have been proposed before for many-cores. These schedulers will need modifications to work on adaptive many-cores as they inherently assume fixed number and types of cores. Furthermore, without the performance prediction models for adaptive many-cores we develop in this work, they will still have limitations of not being able to work without profiling even after the modifications. *DPMS* also provides stronger formal guarantees on convergence, optimality of convergence and time to convergence than these schedulers even for homogeneous many-cores.

Authors in [29] present a parallelizable optimal algorithm for makespan minimization problem of malleable tasks with concave speedups that requires complete task profiles. A task is defined as malleable if the number of cores allocated to it can change during its execution [30]. Authors in [31] present a distributed hybrid algorithm - a run-time heuristic to be used in conjunction with design space exploration (operating points) - for multi-objective optimization that also uses convex optimization. We study the problem of performance maximization at run-time in this work but similar to [29], we use inherent concavity in speedups of malleable tasks to reach the optimal solution. The proposed *DPMS* is a form of distributed consensus coordinate descent algorithm [32], which are a class of algorithms that can be used as parallel solvers for convex optimization problems.

## III. Motivational Case Study

We present a motivational example to stress the need for dynamic schedulers for adaptive many-cores. We assume a four-core processor executing two SPEC [33], [34] benchmarks (*mcf* and *bzip2*). Figure 2 shows the corresponding 2-core speedup of both benchmarks averaged over every ten million

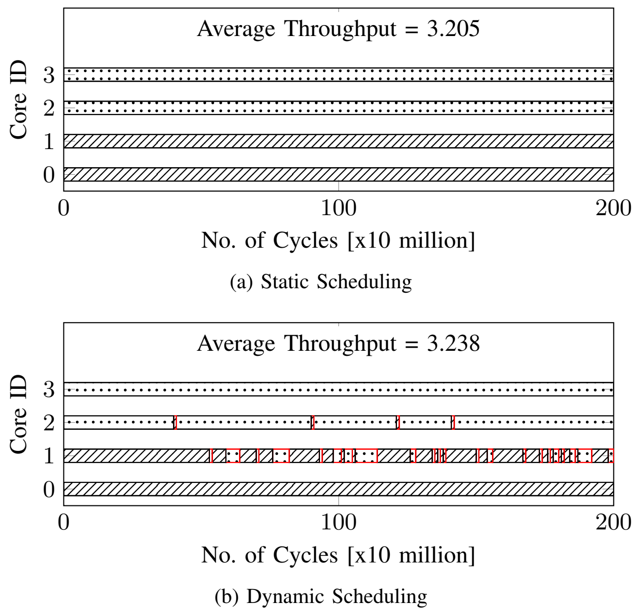

Fig. 3: Motivational example showing improved throughput under dynamic scheduling over static scheduling on an adaptive multi-core with four base cores executing *mcf* and *bzip2*.

committed instructions. $N$-core speedup for a benchmark is defined as ratio of its IPC when assigned $N$ cores versus ratio of its IPC when assigned only 1 core. The speedups vary over the benchmarks lifetime. **Throughput is defined as aggregate speedup experienced by all executing tasks.**

Figure 3 shows the base core allocations under static and dynamic scheduling. Figure 3b shows dynamic scheduling under *DPMS*, which changes core-coalitions at run-time to exploit speedup variability. Dynamic scheduling results in 1.02% higher throughput in comparison to static scheduling. We optimize speedup as it is observed to be a better metric than for example Instruction per Second (IPS) [35], [36].

## IV. Scheduling with *DPMS*

We now present details of a multi-agent scheduler called *DPMS*, which performs dynamic scheduling on adaptive many-cores distributively. We introduce various models that capture the scheduling dynamics.

**System Model:** We assign an agent to each task in our adaptive many-core. Let there be A agents representing A tasks, indexed using the symbol $x$. Each agent $x$ holds $C_x$ number of cores. Let $C$ represent the state of the many-core encompassing all task-to-core assignments. Let $\rho(C_x)$ represent the instantaneous speedup of task represented by agent $x$. Let $\rho(C)$ represent the throughput of many-core.

$$\rho(C) = \sum_{x=1}^{A} \rho(C_x) \tag{1}$$

**Utility Model:** Agents exchange cores based on the value of their utility functions. Let $u_{i \to j}(\Delta)$ represent the utility of transferring $\Delta$ cores from agent $i$ to $j$ measured in terms of resultant change in their combined speedup.

$$u_{i \to j}(\Delta) = \rho(C_i - \Delta) - \rho(C_j + \Delta) \tag{2}$$

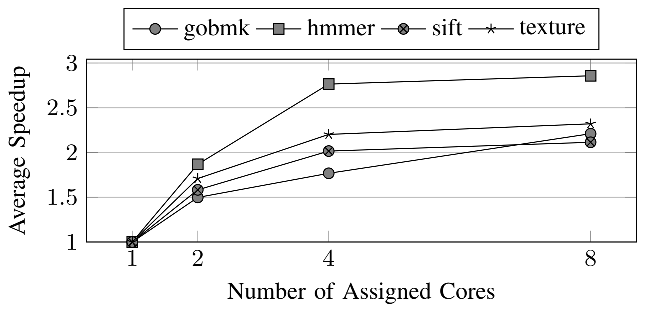

Fig. 4: Average speedup of different tasks when assigned different number of cores.

**Speedup Observation:** Figure 4 shows the average speedups for benchmarks (tasks) of different types when assigned different number of cores. It can be observed from the figure that the average speedup is both monotonically increasing and concave. Note that similar concave behavior is also observed in other benchmarks. This is because of the saturation of exploitable ILP or TLP in the benchmarks with increasing number of assigned cores. Therefore, the problem of throughput maximization of malleable tasks at run-time can be interpreted as a convex optimization problem. *DPMS* uses a coordinated descent algorithm to reach the solution but other convex optimization techniques can also be potentially applied here. Agents also internally smoothen speedup predictions for concavity if non-concave predictions are made.

**Optimal Throughput:** We now present proof of throughput optimality and convergence provided by *DPMS*. Proof is inspired by our work presented in [37] and is a simplification of the original proof presented in [8].

**Theorem 1.** *DPMS converges to task-to-core assignments that maximizes the throughput in O(A) rounds.*

*Proof.* After core exchange under utility $u_{i\to j}(\Delta)$, throughput $\rho(C)$ changes to $\rho(C')$.

$$\rho(C') = \rho(C) + u_{i\to j}(\Delta) \tag{3}$$

So $\rho(C') > \rho(C)$ when $u_{i\to j}(\Delta) > 0$ and $\rho(C') < \rho(C)$ when $u_{i\to j}(\Delta) < 0$. Hence, only core exchange between agents with positive utility can increase throughput from any given state. When no positive utility move exists, throughput cannot be increased further and a maximum is reached. We now prove that this maximum maximizes the throughput.

Speedups are piecewise linear functions that are too computationally expensive to optimize [38]. Based on observations made in Figure 4, we assume that speedup $\forall_x\ \rho(C_x)$ is concave-extensible to a non-negative discrete concave function of $C_x$ and error introduced by this concave relaxation [39] is minimal. $\rho(C)$ is a positive sum of concave functions, therefore is a discrete concave function of $C$ as described by Equation (1). Every maxima of a discrete concave function are its global maxima [40].

As cores-to-task assignments are discrete, there exist only a finite number of positive utility moves. Two agents $i$ and $j$ that exchange $\Delta$ cores once will not exchange cores again unless disturbed by a third agent if $\Delta$ maximizes $u_{i\to j}(\Delta)$. $u_{i\to j}(\Delta)$ is also a discrete concave function of $\Delta$ that can be maximized efficiently using gradient descent. We force agents in *DPMS* to always make $\Delta$ maximizing moves. Thus, positive utility moves will exhaust in at worst $O(A)$ rounds, since an agent can interact with at most $A$ agents (excluding myopic repeated interactions); hence proved. ☐

**Complexity:** Each round of *DPMS* requires $O(|A|)$ agents to perform $O(|C|)$ utility calculations described by Equation (2). In the worst case, equilibrium can take up to $O(|A|)$ rounds (refer Theorem 1). In totality, a maximum of $O(|A|^2|C|)$ calculations are required to ensure stability. However, the processing overhead is distributed across all the base cores in the system resulting in $O(|A|^2)$ worst-case calculations per core.

## V. Performance Prediction

To realize *DPMS* proposed in Section IV, we develop performance prediction models for adaptive many-cores. The models are developed only once at design-time. They are based on low-level hardware counters and hence are independent of the executing tasks and their phases. This design-time modeling has no overhead during run-time in *DPMS*. The models are used by agents in *DPMS* to estimate potential changes in their utilities given by Equation (2), based on which they make decisions to join or break-away from core-coalitions. Utility estimation by definition requires prediction of Instruction per cycle (IPC) of the tasks on core-coalitions of different sizes, which can then be used to perform speedup predictions. The performance prediction models we propose are based on building of Cycles per Instruction (CPI) stacks [18] of tasks being executed on adaptive many-cores. CPI is inverse of IPC. We first estimate the CPI and then convert it to IPC.

CPI stacks have been used in the past to predict performance for both symmetric [18] and asymmetric multi-cores [20], [41]. CPI stack based prediction models for non-adaptive architectures assume static cores and cannot accommodate the possibility of allocating multiple cores to an ILP task. Hence, we are required to develop new models. The hardware counters used in our models should be available on all kinds of base cores, hence making our performance prediction technique potentially portable across adaptive many-core architectures.

Let $P(C_x)$ be the instantaneous CPI of the task associated with agent $x$ with $C_x$ number of cores assigned. $P(C_x)$ is the sum of CPI due to steady-state instruction execution ($P_S(C_x)$) and CPI due to stalls during execution ($P_M(C_x)$).

$$P(C_x) = P_S(C_x) + P_M(C_x) \tag{4}$$

### A. Estimation of Steady-state CPI

Steady-state CPI $P_S(C_x)$ is dependent upon the structural hazards and inter-instruction dependency faced by task assigned to $C_x$. Given the high book-keeping overhead of counters maintaining structural hazards and inter-instruction dependency, it is unlikely for any real hardware to contain them. Authors in [18] modeled steady CPI as $D^{-1}$, where $D$ is dispatch width of the processor. This assumption is applicable only to completely balanced pipelines, wherein the number of functional units in a processor is equal to its dispatch width and structural hazards do not manifest. This assumption also ignores the data dependency among the instructions.

Instead, we employ linear regression to obtain $P_S(C_x)$. We model CPI in quanta of $I$ instructions and size of $I$ needs to be empirically determined. We set size of $I$ to 10 million instructions. Ideally $I$ should be as small as possible so that even minutest CPI variation can be modeled. As the value of $I$ reduces, noise in the observations increases. When $I$ is very small, CPI prediction using regression becomes too inaccurate.

Let $I$ be composed of $I_{int}$ integer and $I_{fp}$ float instructions. $I_{int}$ and $I_{fp}$ could be committed in $\theta_{int} * I_{int}$ and $\theta_{fp} * I_{fp}$ cycles, respectively assuming there are no structural hazards and inter-instruction dependencies. $\theta_{int}$ and $\theta_{fp}$ are cycles taken to commit a single integer and float instruction, respectively. When structural hazards and inter-instruction dependencies manifest, the functional units will be busy. Let $I_{busy}$ be the number of times functional unit reported busy in $I$ instructions and resulted in stalling for $\theta_{busy} * I_{busy}$ cycles. $\theta_{busy}$ is number of cycles wasted every time functional unit reported busy. Thus, $P_S(C_x)$ for $I$ instructions executed on $C_x$ can be modeled as Equation (5), where $\theta_{int}$, $\theta_{fp}$ and $\theta_{busy}$ are system specific constants that can either be provided by system designer or can be determined with the help of regression.

$$P_S(C_x) = (\theta_{int}\ I_{int} + \theta_{fp}\ I_{fp} + \theta_{busy}\ I_{busy})/I \qquad (5)$$

## B. Estimation of Miss CPI

Miss CPI $P_M(C_x)$ is determined by stalls that happen because of branch mis-predictions and cache misses during execution on $C_x$. We assume adaptive many-core has separate L1 instruction and data cache. We assume a unified L2 cache as L2 cache misses are disproportionately dominated by data misses. The design of cache-hierarchy is in sync with most of the commercially deployed micro-architectures. We also assume that our base cores are out-of-order, which given their complex design are more difficult to model than in-order cores.

In $I$ instructions, let $I_{L1iMiss}$, $I_{L1dMiss}$ be the number of L1 instruction cache and L1 data cache misses that leads to wastage of $\theta_{L1iMiss} * I_{L1iMiss}$ and $\theta_{L1dMiss} * I_{L1dMiss}$ cycles, respectively. $\theta_{L1iMiss}$ and $\theta_{L1dMiss}$ represent the cycles wasted in one single L1 instruction and L1 data cache miss, respectively. Let $I_{BrMiss}$ represent the number of branch mis-predictions that happen in $I$ instructions. $I_{BrMiss}$ leads to wastage of $(\theta_{FrLen} + \theta_{BrRes}) * I_{BrMiss}$ cycles. $\theta_{FrLen}$ is front end length of the pipeline measured in cycles and $\theta_{BrRes}$ represent the cycles needed to find the outcome of a single branch instruction (or branch resolution time). $\theta_{BrRes}$ varies from one branch instruction to another depending upon whether the resolution of the branch instruction is dependent on another instruction in the pipeline or not, but this variation is not substantial. Therefore, we can simplify the model by assuming it to be a constant. Let $I_{L2miss}$ be the number of L2 cache misses that happen in $I$ instructions. For in-order cores, number of cycles wasted in $I_{L2miss}$ would be $\theta_{L2miss} * I_{L2miss}$, where $\theta_{L2miss}$ is the cycles wasted in a single L2 cache miss. However, in our case, we divide this term by Memory Level Parallelism (MLP) factor to account for overlapping last level cache accesses in an out-of-order core. Thus, MLP is primarily dependent on the number of L2 misses, the last level cache in conventional adaptive many-core architectures. MLP is modeled as $[\gamma_1 * (I_{L2miss})^{\gamma_2}]$, where $\gamma_1$ and $\gamma_2$ are constants obtained using polynomial regression on performance counter values including CPI.

Therefore, CPI $P_M(C_x)$ for $I$ instructions on $C_x$ can be modeled as Equation (6). Similar to Equation 5, $\theta_{L1iMiss}, \theta_{L1dMiss}, \theta_{FrLen}, \theta_{BrRes}$ and $\theta_{L2Miss}$ are constants, which can either be provided by the designer or can be determined by performing micro-benchmarking.

$$\begin{aligned} P_M(C_x) = {} & (\theta_{L1iMiss} I_{L1iMiss} + \theta_{L1dMiss} I_{L1dMiss} \\ & + (\theta_{FrLen} + \theta_{BrRes}) I_{BrMiss} + \theta_{L2Miss} I_{L2Miss}/MLP)/I \end{aligned} \qquad (6)$$

## C. Varisized coalition CPI Estimation

Our MAS requires agents to use current CPI $P(C_x)$ on $C_x$ to estimate CPI $P(C_x')$ when $C_x$ changes in size to $C_x'$. Agents must estimate both the steady-state CPI $P_S(C_x')$ and miss CPI $P_M(C_x')$ from current $P_S(C_x)$ and $P_M(C_x)$, respectively and then use them in Equation (5) to obtain estimated $P(C_x)$.

**Varisized Coalition Steady-state CPI Estimation:** For $I$ instructions, the count of $I_{int}$ integer and $I_{fp}$ float instructions do not change, when underlying core assignment changes size from $C_x$ to $C_x'$. Based on Equation (4) to predict steady CPI $P_S(C_x')$ we only need to predict the functional unit busy $I'_{busy}$ on $C_x'$ from $I_{busy}$ on $C_x$. As all our base cores are identical, the number of functional units in a core-coalition will increase linearly with increase in the size of the core-coalition. $I'_{busy}$ besides $I_{busy}$, also depends upon instruction composition of $I$ that consists of $I_{int}$ integer, $I_{fp}$ float, $I_{br}$ branch and $I_{mem}$ memory (load or store) instructions. We obtain the relationship with help of linear regression.

$$I'_{busy} = \beta_1 I_{busy} + \beta_2 I_{int} + \beta_3 I_{fp} + \beta_4 I_{br} + \beta_5 I_{mem} + \beta_6$$

**Varisized Coalition Miss CPI Estimation:** Miss CPI $P_M(C_x')$ on $C_x'$ also needs to be estimated from miss CPI $P_M(C_x)$ on $C_x$. Based on Equation (6), we need to estimate number of L1 instruction cache misses $I'_{L1iMiss}$, L1 data cache misses $I'_{L1dMiss}$, branch misdirections $I'_{BrMiss}$ and L2 cache misses $I'_{L2miss}$ for $I$ instructions on $C_x'$ using current observation of $I_{L1iMiss}$, $I_{L1dMiss}$, $I_{BrMiss}$ and $I_{L2miss}$ on $C_x$, respectively.

All the base cores in our adaptive many-core have the same branch predictors and cache sizes. Thus, an assignment $C_x$ of size $|C_x|$ will have $|C_x|$ times the L1 instruction and data cache of a single base core. L2 cache being a unified cache shared by all the cores within a coalition does not change in size with change in the core-coalition size but the number of L2 cache miss for varisized core-coalitions will still be different because of change in the size of the corresponding L1 cache. L1 instruction and data cache have the same associativity across different core-coalition sizes. Hence, these misses are estimated using a linear regression model. Linear regression can also be used to estimate branch mis-predictions. We observe high variability in L2 cache-misses and use polynomial regression of degree two for them.

$$\begin{aligned} I'_{L1iMiss} &= \beta_7 I_{L1iMiss} + \beta_8 \\ I'_{L1dMiss} &= \beta_9 I_{L1dMiss} + \beta_{10} \end{aligned}$$

$$\begin{aligned} I'_{BrMiss} &= \beta_{11} I_{BrMiss} + \beta_{12} \\ I'_{L2miss} &= \beta_{13} I_{L2miss} + \beta_{14} I^2_{L2miss} + \beta_{15} \end{aligned}$$

Note that the prediction models introduced in this section are only valid for ILP based performance prediction in single-threaded benchmarks. For parallel multi-threaded tasks, we choose to continue using profiles instead of performance prediction models in this work. Existing CPI stack based performance prediction models proposed for non-adaptive multi-cores [18], [20] can only operate with single-threaded tasks. Performance prediction models based on speedup stacks introduced in [42] can potentially be adapted to perform TLP based performance prediction for malleable multi-threaded tasks. Development of complete performance prediction model for any generic multi-threaded task with consideration of inter-thread synchronizations is beyond the scope of this paper.

## VI. Experimental Evaluations

A two-stage simulation is used in this evaluation because unlike other many-cores [43] a real-world adaptive many-core is not yet available. In the first stage, a generic adaptive many-core is modeled using cycle-accurate *gem5* [44] simulator. This many-core consists of eight 2-way out-of-order *ARM* base cores utilizing *ARM*v7 ISA. Each core contains a dedicated 4-way associative 64 KB L1 instruction and data caches. Additionally, all cores share an 8-way associative 2 MB unified L2 cache. Thus, each task can employ up to 8-core coalition.

The cycle-accurate simulation time increases exponentially with an increase in the number of base cores. Therefore, modeling hundreds of cores with the cycle-accurate simulator is time consuming. To solve this shortcoming, the second stage utilizes execution traces from stage one to realize an adaptive many-core that contains up to a maximum of 256 cores. Note that malleable tasks can only be simulated using trace-based simulators. Trace-based simulators are widely used in many-core scheduler evaluations [45], [46].

Workloads are created from 24 sequential single-threaded (ILP) benchmarks comprising of integer, float and vision benchmarks from *SPEC 2000* [33], *SPEC 2006* [34] and *SD-VBS* suites [47]. Additionally, 10 multi-threaded TLP benchmarks from *PARSEC* [48] and *SPLASH-2* [49] suites are also used. Table I states the 34 benchmarks used in this evaluation. *ARM* cross compiler with "-O2" optimization flag is used to compile all the benchmarks and executed on *gem5* System Call Emulation (SE) mode. "Ref" input is used in *SPEC* benchmarks while "full-hd' input is used in *SD-VBS* benchmarks. "Sim-small" input is used in the multi-threaded benchmarks from *PARSEC* and *SPLASH-2* benchmark suites. Each benchmark executes for a maximum of 2 billion cycles. The performance counter values utilized in Section V are obtained per benchmark from *gem5*.

We invoke the scheduler every 10 million cycles. This corresponds to decision making every 10 ms, if the system operates at 1 GHz. Note that 10 ms corresponds to the default operating granularity of the Linux scheduler [50]. The overheads for breaking or making core-coalitions in adaptive multi-core architectures are very small. For example, it is 500 cycles in *T-Flex* [10] and 100 cycles in *Bahurupi* [11]. In comparison to the scheduling epoch these penalties are negligible.

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

TABLE I: List of benchmarks used in the evaluation.

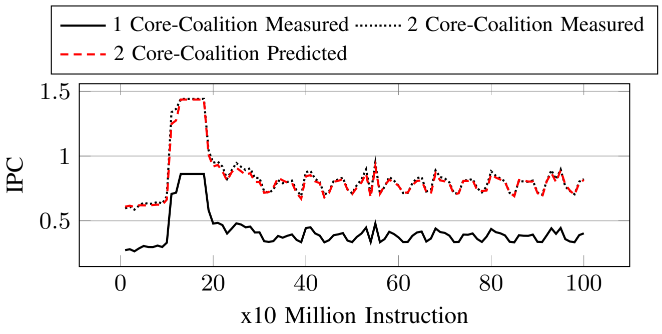

Fig. 5: *h264ref* IPC prediction on 2-core from observed 1-core IPC, compared against actual 2-core measured IPC.

| Name | 2-Core | 3-Core |
|---|---|---|
| *art* | 6.2 | 4.4 |
| *astar* | 1.0 | 4.6 |
| *bwaves* | 1.77 | 12.2 |
| *bzip2* | 8.0 | 14.3 |
| *disparity* | 4.5 | 12.9 |
| *equake* | 16.6 | 6.7 |
| *h264ref* | 3.5 | 5.7 |
| *mcf* | 2.5 | 4.6 |
| *omnetpp* | 10.3 | 7.9 |
| *perlbench* | 3.5 | 3.3 |
| *tracking* | 4.3 | 8.7 |
| *svm* | 6.5 | 13.4 |

(a) Training Set

| Name | 2-Core | 3-Core |
|---|---|---|
| *calculix* | 17.3 | 11.1 |
| *gemsfdtd* | 2.6 | 15.7 |
| *gobmk* | 7.2 | 10.1 |
| *hmmer* | 9.3 | 23.5 |
| *lbm* | 4.2 | 7.9 |
| *mser* | 3.9 | 5.9 |
| *namd* | 4.0 | 14.4 |
| *povray* | 8.4 | 2.1 |
| *sift* | 12.2 | 9.7 |
| *texture* | 4.4 | 2.0 |
| *tonto* | 4.6 | 10.7 |
| *twolf* | 8.2 | 5.0 |

(b) Testing Set

TABLE II: Average percentage errors in predicting 2- and 3-core IPC from 1-core measured IPC for ILP benchmarks.

### A. Performance Prediction Accuracy

We evaluate the accuracy of the regression based performance prediction models presented in Section V for ILP task executing on adaptive many-cores. We divide all our available ILP benchmarks equally among training and test sets. Both sets contain equal mix of compute intensive and memory intensive benchmarks, for holistic modeling and model evaluation. The data from benchmarks in the training set is used to obtain the regression constants of the proposed models. Data is generated using detailed cycle-accurate simulations. Models are then evaluated for accuracy on benchmarks in the test set again using cycle-accurate simulations.

Figure 5 shows 2-core IPC measurements against 2-core IPC predictions from 1-core IPC measurements for *h264ref* averaged over every 10 million instructions committed. *h264ref* has high entropy and the figure shows that we can predict its IPC with high accuracy throughout its lifetime. Table II shows average error in predicted IPC for 2- and 3-core from 1-core measured IPC for benchmarks from the testing and training

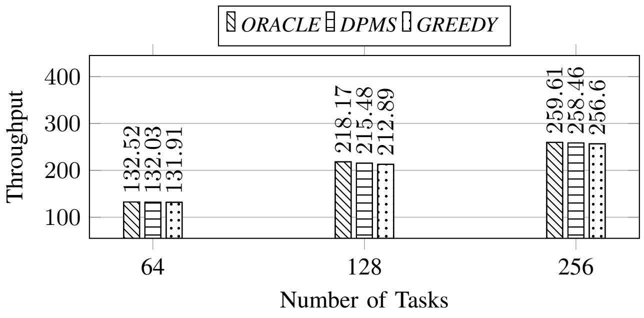

Fig. 6: Performance of different schedulers on a closed 256-core adaptive many-core with ILP workloads.

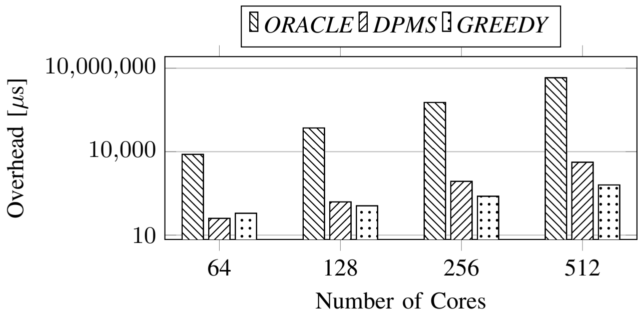

Fig. 7: Scheduling overhead for different schedulers on varisized many-cores under half load.

set. Our models have 6.97% and 8.52% error in IPC prediction on average for all IPC benchmarks across all core assignment sizes in the training and test inputs respectively.

## B. Scheduler Evaluation and Analysis

We now empirically evaluate the performance of our *DPMS* and begin by creating an oracular algorithm from *PROFILE* [4], which has been discussed in Section II. We extend *PROFILE* scheduler to *ORACLE* scheduler that is capable of exploiting dynamic scheduling. *ORACLE* uses dynamic programming to maximize instantaneous speedup of tasks every scheduling epoch without any prescient knowledge of the future in the form of recorded average speedup like *PROFILE*. It does assume perfect knowledge of instantaneous speedup for all tasks on all possible core assignment sizes. For *ORACLE*, solution in a $n$ core system is defined as

$$\text{Maximize} \sum_{x=1}^{|A|} \rho(C_x), \text{ given constraint } \sum_{x=1}^{|A|} |C_x| \leq n$$

We also compare against a fast centralized *GREEDY* algorithm of our own design [22] that also exploits concavity in speedup like *DPMS* to give near-optimal solutions.

**Performance:** We simulate a closed system on a 256-core adaptive many-core. In a closed system, the system begins with an immutable predefined set of tasks; instances of those tasks on completion immediately rejoin the system for re-execution. Throughput measured in terms of aggregate speedup is used as the performance metric where higher throughput is better. Figure 6 shows the throughput with different schedulers under different system loads. As we operate with one-thread per core model and each task produces at least one thread with minimal speedup of one, it is not meaningful to simulate systems with number of tasks more than number of cores.

For example, as seen in Figure 6, under half-load, i.e. 128 tasks, the performance of *DPMS* and *GREEDY* is 1.38% and 2.42% lower than optimal, respectively. Even though both *DPMS* and *GREEDY* are optimal solutions, the performance drop in these approaches are slightly more than *ORACLE* because of the speedup concavity assumption, which may not always hold for some tasks. ILP tasks can show non-concave behavior when some secondary bottleneck like memory opens up due to assignment of more cores.

**Scalability:** The biggest advantage of *DPMS* over *ORACLE* is its ability to perform distributed scheduling, which can scale up better with increase in the number of base cores. In a nutshell, *DPMS* reduces the per-core processing overhead by disbursing the processing across all the cores in the many-core but this also results in an increase in communication overhead when compared to *ORACLE*. Therefore, it is important to get a measure of the real-world benefits that can be obtained from *DPMS* over *ORACLE* by trading-off per-core processing overhead with communication overhead.

As running schedulers with real workloads cycle-accurately on *gem5* is time-wise infeasible for large size many-cores, we instead execute the logic of the schedulers cycle-accurately with representative input and report the corresponding problem solving time. While *ORACLE* is implemented as a centralized application written in *C*, *DPMS* is implemented as a distributed multi-threaded *C++* application. Each agent in *DPMS* is implemented as one thread using *POSIX-thread*.

Figure 7 shows the worst-case overhead observed on varisized many-cores under *DPMS* and *ORACLE* on a logarithmic scale. Note that overhead due to performance estimator is also included in *DPMS*. It can be seen from the figure that *DPMS* solves the many-core scheduling problem much faster in practice. For a 64-core many-core, *ORACLE* requires 7.985 ms to solve the problem whereas *DPMS* only requires 0.040 ms. Therefore, *DPMS* leads to 200x reduction in total overhead in comparison to *ORACLE* on a 64-core many-core system. Performance of *DPMS* on a 64-core many-core system is even faster than 0.061 ms taken by the *GREEDY* algorithm.

## VII. Conclusion

In this paper, we propose a multi-agent scheduler called *DPMS* for distributed scheduling on adaptive many-cores. We also introduce a performance prediction technique for adaptive many-cores that enables *DPMS* to operate without any a-priori profiling. *DPMS* guarantees convergence to an optimal throughput maximizing state under the predicted information in given number of steps from any initial state. It results in 200x reduction in scheduling overheads compared to a centralized oracle scheduler on a 64-core many-core system.

## 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 National Research Foundation, Prime Minister’s Office, Singapore under its Industry-IHL Partnership Grant and Huawei International Pte. Ltd. NRF2015-IIP003.

## References

[1] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Spring Joint Computer Conference (SJCC)*, 1967.

[2] T. Mitra, "Heterogeneous Multi-Core Architectures," *Information and Media Technologies (IMT)*, 2015.

[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] 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.

[5] M. Pricopi and T. Mitra, "Task Scheduling on Adaptive Multi-Core," *Transactions on Computers (TC)*, 2013.

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

[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] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Distributed Scheduling for Many-Cores Using Cooperative Game Theory," in *Design Automation Conference (DAC)*, 2016.

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

[10] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable Lightweight Processors," in *International Symposium on Microarchitecture (MICRO)*, 2007.

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

[12] D. Tarjan, M. Boyer, and K. Skadron, "Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue," in *Design Automation Conference (DAC)*, 2008.

[13] Y. Watanabe, J. D. Davis, and D. A. Wood, "WiDGET: Wisconsin Decoupled Grid Execution Tiles," in *Computer Architecture News*, 2010.

[14] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-Thread Applications," in *High Performance Computer Architecture (HPCA)*, 2007.

[15] T. Sun, H. An, Y. Ren, M. Mao, Y. Liu, M. Xu, and Q. Li, "FACRA: Flexible-core Architecture Chip Resource Abstractor," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2010.

[16] M. Breughe, S. Eyerman, and L. Eeckhout, "A Mechanistic Performance Model for Superscalar In-Order Processors," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[17] B. C. Lee and D. M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *SIGPLAN Notices*, 2006.

[18] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-Empirical Processor Performance Modeling for Constructing CPI Stacks on Real Hardware," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[19] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads," in *International Symposium on Workload Characterization (IISWC)*, 2011.

[20] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-Performance Modeling on Asymmetric Multi-Cores," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013.

[21] Y. Ren, H. An, T. Sun, M. Cong, and Y. Wang, "Dynamic Resource Tuning for Flexible Core Chip Multiprocessors," in *International Conference of Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2010.

[22] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Optimal Greedy Algorithm for Many-Core Scheduling," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[23] I. Galanis, D. Olsen, and I. Anagnostopoulos, "A Multi-Agent Based System for Run-Time Distributed Resource Management," in *International Symposium on Circuits and Systems (ISCAS)*, 2017.

[24] T. Melissaris, I. Anagnostopoulos, D. Soudris, and D. Reisis, "Agora: Agent and Market-Based Resource Management for Many-Core Systems," in *International Conference on Electronics, Circuits and Systems (ICECS)*, 2016.

[25] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel, "DistRM: Distributed Resource Management for On-Chip Many-Core Systems," in *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.

[26] I. Ahmad, S. Ranka, and S. U. Khan, "Using Game Theory for Scheduling Tasks on Multi-Core Processors for Simultaneous Optimization of Performance and Energy," in *International Symposium on Parallel and Distributed Processing (ISPDC)*, 2008.

[27] S. Wildermann, T. Ziermann, and J. Teich, "Game-Theoretic Analysis of Decentralized Core Allocation Schemes on Many-Core Systems," in *Design, Automation and Test in Europe (DATE)*, 2013.

[28] M. Shafique and J. Henkel, "Agent-Based Distributed Power Management for Kilo-Core Processors," in *International Conference on Computer-Aided Design (ICCAD)*, 2013.

[29] P. Sanders and J. Speck, "Efficient Parallel Scheduling of Malleable Tasks," in *Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[30] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, "Distributed Run-time Resource Management for Malleable Applications on Many-Core Platforms," in *Design Automation Conference (DAC)*, 2013.

[31] S. Wildermann, M. Glaß, and J. Teich, "Multi-Objective Distributed Run-Time Resource Management for Many-Cores," in *Design, Automation & Test in Europe (DATE)*, 2014.

[32] S. J. Wright, "Coordinate Descent Algorithms," *Mathematical Programming*, 2015.

[33] J. L. Henning, "SPEC CPU2000 : Measuring CPU Performance in the New Millennium," *Computer*, 2000.

[34] Henning, John L, "SPEC CPU2006 Benchmark Description," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[35] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Mutlithreading Processor," *SIGPLAN Notices*, 2000.

[36] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *Micro*, 2008.

[37] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Distributed Fair Scheduling for Many-Cores," in *Design, Automation & Test in Europe (DATE)*, 2016.

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

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

[40] K. Murota, "Submodular Function Minimization and Maximization in Discrete Convex Analysis," *Discrete Applied Mathematics*, 1984.

[41] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE)," in *Computer Architecture News (CAN)*, 2012.

[42] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications," in *International Symposium on Performance Analysis of Systems and Software*, 2012.

[43] V. Tsoutsouras, I. Anagnostopoulos, D. Masouros, and D. Soudris, "A Hierarchical Distributed Runtime Resource Management Scheme for NoC-Based Many-Cores," *Transactions on Embedded Computing Systems (TECS)*, 2018.

[44] 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 *Computer Architecture News (CAN)*, 2011.

[45] D. Olsen and I. Anagnostopoulos, "Performance-Aware Resource Management of Multi-Threaded Applications on Many-Core Systems," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2017.

[46] S. Pagani, H. Khdr, W. Munawar, J.-J. Chen, M. Shafique, M. Li, and J. Henkel, "TSP: Thermal Safe Power: Efficient Power Budgeting for Many-Core Systems in Dark Silicon," in *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014.

[47] 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.

[48] 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.

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

[50] V. Pallipadi and A. Starikovskiy, "The Ondemand Governor," in *The Linux Symposium*, 2006.