Scalable Power Management for On-Chip Systems with Malleable Applications

Muhammad Shafique, *Member, IEEE*, Anton Ivanov, Benjamin Vogel, and Jörg Henkel, *Fellow, IEEE*

Abstract—We present a scalable Dynamic Power Management (DPM) schem e where malleable applications may change their degree of parallelism at run time depending upon the workload and performance constraints. We employ a per-application predictive power manager that autonomously controls the power states of the cores with the goal of energy efficiency. Furthermore, our DPM allows the applications to lend their idle cores for a short time period to expedite other critical applications. In this way, it allows for application-level scalability, while aiming at the overall system energy optimization. Compared to state-of-the-art centralized and distributed power management approaches, we achieve up to 58 percent (average ≈15-20 percent) ED^2P reduction.

Index Terms—Power management, low power, manycore, self-adaptation, scalability, energy efficiency, malleable applications

1 Introduction

Due to increasing core integration, emerging on-chip systems are envisaged to contain 1,000s of cores (ITRS prediction: ≈1,500 cores by 2020, ≈5,900 cores in 2026) [1], [2], [3]. Designing efficient Dynamic Power Management (DPM, i.e., selecting an appropriate low-power state) policies for these on-chip systems exhibits various challenges, out of which, two important ones are: (1) scalability and complexity issues to determine appropriate power states of the cores; (2) power/energy efficiency issues related to applications with suddenly changing workloads.

DPM scalability: A large number of cores means a large power management design space, which depends upon the number of cores, number of supported power states, workload/power statistics, etc. [4], [5]. Therefore, the majority of state-of-the-art policies that rely on centralized decision making (like [6], [7], [8]) cannot be efficiently employed in future any more. In [8], [9] controller-based power management policies with chip-wide DVFS are applied. The MaxBIPS policy [6] employs a prediction-based algorithm with exhaustive search to find the optimal power level for all cores. These policies may be prohibitive though due to the exponential increase in the decision space and high overhead (in terms of computation/delay). While fast search and optimization heuristics may be applied to prune the decision space, recent trends for manycore systems have shown a paradigm shift towards scalable/distributed DPM techniques [4], [5], [14]. *However, to cope with the rapidly growing complexity of future manycore systems, scalable DPM policies need to enable autonomous management of power states of the cores to operate efficiently as a whole.*

Mixed malleable workload challenge: Consideration of (massively) multi-threaded applications with mixed-workload properties complicates the DPM problem. To exploit the plethora of cores, new application models, i.e., the so-called *malleable applications*, have evolved [10], [11], [12]. These applications can dynamically vary their degree of thread-level parallelism depending upon the number of free cores to handle changing/mixed workloads and performance requirements. To do so, the malleable applications perform *two key operations* at run time (see application model in Section 3).

1) *Expand:* In case of an increased workload, a malleable application requires more cores from the resource manager. New threads are instantiated on the newly-allocated cores. This process of demanding and obtaining more cores, and thread instantiating is defined as the *Expand* operation, as the application *owns* more cores (i.e., the cores are dedicated to a particular so-called "owner application"[1]); see Fig. 1.
2) *Shrink:* In case of a reduced workload, a malleable application may meet its performance requirements with a lesser number of cores compared to what it originally owns. Therefore, it may return the so-called *dispensable cores* to the resource manager and a de-allocation of threads is performed accordingly; see Fig. 1. The returned cores from a *shrinking* application can then be allocated to an *expanding* application.

An Example: Fig. 1 shows an abstract scenario where two applications (i.e., *A* and *B*) are *expanding* and *shrinking* at run time. Note, for an application, first an *Expand* will come and then a *Shrink*. Simultaneous *Expand* operations by multiple applications (e.g., *A* and *B* in Fig. 1) correspond to *resource competition*. In such situations, simultaneously executing applications compete for the same set of cores at the same time to improve their performance/energy efficiency (though only one may obtain a particular subset).

• *The authors are with the Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany. E-mail: {muhammad.shafique, anton.ivanov, benjamin.vogel, henkel}@kit.edu.*

Manuscript received 21 May 2015; revised 1 Mar. 2016; accepted 2 Mar. 2016. Date of publication 9 Mar. 2016; date of current version 14 Oct. 2016. Recommended for acceptance by W. Shi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2540631

1. Applications other than the "owner application" are not allowed to use these cores unless the owner application returns/trades these cores to/with others.

0018-9340 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

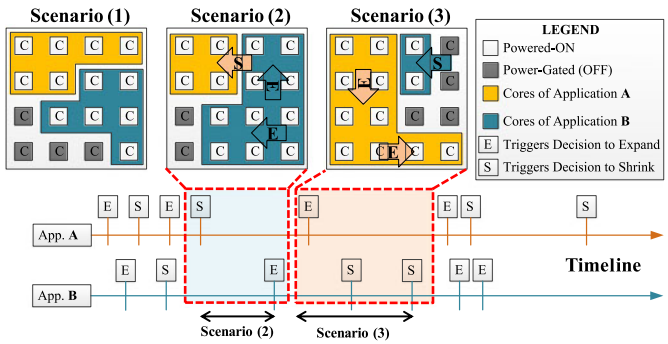

Fig. 1. Expanding and shrinking applications on a manycore system: different scenarios show varying *Expand-to-Shrink* and *Shrink-to-Expand* time periods.

In low workload scenarios, an application may perform a *Shrink* operation (e.g., *B* in Scenario (3)) to facilitate other competing applications.

Recent state-of-the-art in scalable power management either employs hierarchical power control [19], core-level power management [16], or distribute power budget among cores or groups of cores [4], [18]. They mostly consider independent workloads for each core and *do not account for the expanding and shrinking nature of malleable applications*. As a result, they degrade the system performance/power efficiency due to:

1) unnecessarily setting a set of cores in the low power mode for an application which is going to *Expand* soon; and/or
2) impairing *Expand*/*Shrink* operations and dependent workloads of malleable applications.

Furthermore, concurrent executions of malleable applications lead to unpredictable scenarios of varying resource demands and resource competition. The unawareness of the *Expand*/*Shrink* operations in the power management decision may exacerbate the load imbalance and may lead to power inefficiency; see motivational analysis and case studies in Section 4.

Required is a scalable DPM for on-chip systems executing several malleable applications with mixed (i.e., both dependent and independent) workloads. It needs to enable autonomous management of cores in an energy-efficient way while accounting for the *Expand*/*Shrink* knowledge.

1.1 Our Novel Contributions and Principal Concept

We present a scalable Dynamic Power Management scheme for on-chip manycore systems. To enable application-level scalability, it associates a *dedicated local power manager*[2] (see Fig. 2) with each application that autonomously manages the owned cores and their respective power states in order to improve the energy and performance efficiency. It thereby leverages the ability of malleable applications to adapt to different amount of resources in an energy-efficient way. The local power managers also account for the overlapping *expanding* and *shrinking* resource requirements of concurrently executing malleable applications.

2. A local power manager can be implemented as a spawned child thread at the application startup time; it can also be a part of a local resource manager in a distributed resource management paradigm.

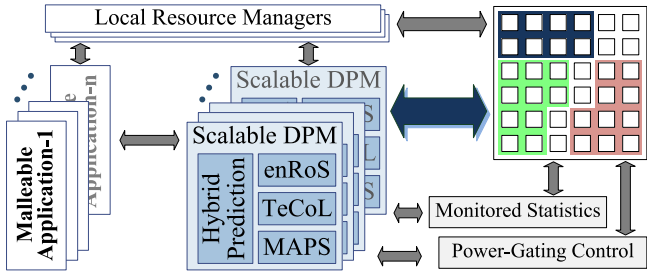

Fig. 2. Overview of our scalable DPM scheme.

A local power manager employs the following components.

- *An energy-aware resource shrinking (enRoS) Policy*: It executes an *energy-efficient Shrink operation* and selects a subset of potentially *dispensable cores* that are returned to the local resource manager. It performs a *proactive resource reservation* to improve the energy efficiency of the owner application in the upcoming *Expand* operations. Since a potential reservation may decrease the performance of competing applications, our DPM incorporates:
- *A temporary core lending (TeCoL) Policy*: It allows the owner applications to *temporarily* lend its reserved cores to the critical competing applications. It alleviates the effects of resource monopolization and lowers the performance degradation of the competing applications due to resource deficiency.
- *A malleability-aware power state selection (MAPS) Policy*: It selects an appropriate sleep state for each core considering the mixed-malleable workload properties.
- *A hybrid prediction technique*: Power-efficient resource reservation, resource lending, and power state selection require a robust prediction of the upcoming *Expand* and *Shrink* operations, i.e., number of required cores and time periods between *Expand* and *Shrink*. To facilitate this, a hybrid prediction technique is employed that *jointly* accounts for the application-level and hardware-level knowledge of *Expand* and *Shrink* behavior under fluctuations of the application workload.

Evaluation for energy efficiency and scalability: The scalable DPM is evaluated for energy efficiency (Energy-Delay^2-Product) and compared to different state-of-the-art centralized and distributed power management schemes. We also illustrate the scalability towards large-scale on-chip systems (up to 1,024 cores) executing numerous mixed-workload malleable applications (5–200 applications); see Section 6.

Paper organization: Section 2 discusses the related work. Section 3 presents the system models whereas Section 4 presents a motivational case study to highlight the potential issues w.r.t. performance and energy efficiency in the presence of *Expand* and *Shrink* operations. Section 5 provides in-depth details on our novel scalable DPM. Section 6 discusses the experimental setup and results followed by overhead discussion in Section 7. Section 8 briefly introduces the supplementary material, which can be found on the Computer Society Digital Library at http://www.ieeecomputersociety.org/10.1109/TC.2016.2540631, and Section 9 concludes the paper.

2 Related Work

In the following, we discuss prominent approaches in centralized and distributed power management.

2.1 Centralized Power Management

Intel *Foxton* technology [9] employs chip-level DVFS to control power and temperature, while the work of [8] proposes a power controller to improve the system performance. Though DVFS will prevail, a large body of research has explored parallel concepts of DPM that incorporate a multiple power state model [24], [29] for each core. This provides a tradeoff between leakage savings and wakeup overhead. A practical example of this is Intel's *PENRYN* technology [21]. For DPM, state-of-the-art either relies on *hardware monitoring* or exploits coarse-grained *application knowledge* to predict an appropriate sleep state [28], [30]. The work in [30] exploits application knowledge of a multimedia application for efficient DPM in a multi-ASIP processor. However, these centralized techniques target small-scale systems with fewer application threads and cores, thus they may not be efficient/scalable for massively multi-threaded applications and large-sized on-chip systems due to state space explosion issues in their decision algorithms.

2.2 Distributed and Scalable Power Management

To address the scalability issues, recently several distributed power management techniques have emerged. The work of [19] exploits the concepts from control theory and builds a system of coordinated controllers to achieve optimal power-performance efficiency in asymmetric processors. The framework in [19] does not scale to large-sized systems due to their integrated controller structure and closed-loop central power budgeting for the whole system. The *UtilPG* and *IdlePG* policies [16] power-gate the core in case the utilization and the core idle time are under the predefined thresholds, respectively. These policies may restrict the cores to the owner applications and may lead to severe performance degradation of competing applications, especially in scenarios of sudden *Expand*. The work of [18] exploits the absolute BIPS to allocate the chip-level power budget to different power islands. A three step approach is employed in [4]: (1) the chip-level power budget is adapted using a controller; (2) the cores of applications are grouped and the chip-level power budget is distributed among these groups; (3) the group-level power budget is distributed among different cores depending upon the estimated thread criticality [17]. However, these policies consider either *independent* or *partially-dependent* workloads, thus cannot be effectively applied in scenarios of mixed-malleable workloads. Moreover, they may be inefficient in scenarios of overlapping *Expand*/*Shrink* operations of resource-competing applications due to lack of application-level knowledge of malleability that may be leveraged to take proactive power management decisions.

In distinction to the above-discussed state-of-the-art, our DPM is scalable to large-scale on-chip systems, as it employs a *dedicated local power manager* for each malleable application. It enables autonomous energy management and control of the cores of its owner application. Furthermore, it exploits the knowledge of mixed-malleable workloads and overlapping *Expand*/*Shrink* operations while proactively avoiding resource monopolization through temporarily lending of reserved cores to competing applications.

Before proceeding to the details of our novel DPM, we will present the system model and an experimental case study to highlight the issues related to the *energy-aware shrinking*.

3 System Models and Preliminaries

Processor model: The manycore system is modeled as a set of cores $\boldsymbol{C} = \{C_1, C_2, \ldots, C_N\}$ connected via an on-chip network.

Malleable application model: The on-chip system concurrently executes several malleable applications[3] $\boldsymbol{A} = \{A_1, A_2, \ldots, A_M\}$. At time t, an application A_i owns N_i number of cores and may perform one of the following two operations.

1) *Expand:* When an application starts, it gets one core for initialization. To handle the *increased* workload, A_i requires more cores compared to what it previously owns. Therefore, at first, it triggers an *Expand* operation, and will start negotiating with other applications to acquire more cores in a distributed resource management paradigm [10]. These negotiations involve interaction and exchange of information between the local resource managers of different applications [10], [27]. The *Expand* operation is given as: $E_i(t) = N_i(t) - N_i(t-1)$. Once the cores are acquired, more threads of A_i are spawned on the newly allocated cores to execute its parallelizable sections.
2) *Shrink:* Once the execution of the parallelizable section finishes and a sequential execution section starts, A_i requires lesser cores compared to what it previously owned. In this case, it can potentially *return* or *reserve* the cores by triggering a *Shrink* operation. The *Shrink* operation is: $S_i(t) = N_i(t-1) - N_i(t)$. $S_i(t)$ is the number of returned cores, that the local manager trades with other competing applications. At a *Shrink* operation, the application A_i may reserve $R_i(t)$ number of cores to handle an upcoming *Expand* while using $C_i(t)$ for the current workload processing at time t. Therefore, the total number of owned cores at a *Shrink* operation is $N_i(t) = C_i(t) + R_i(t)$. At a *Shrink* operation, the application A_i deallocates its threads from the returned cores, while keeping the program in a retentive state for the reserved cores. Note, at the first *Expand* operation, $N_i(t) = C_i(t)$, because $R_i(t) = 0$ at that time.

Expand and *Shrink* operations allow a malleable application to adapt its degree of parallelism to handle changing workload scenarios and to exploit the available parallelism. A *mixed-malleable workload* scenario corresponds to the case where different concurrently executing malleable applications have distinct *Expand*/*Shrink* profiles and thereby providing varying degree of parallelism and different number of active threads. The malleable workload of an application

3. Examples: slice-parallelizable video encoders, scalable tracking algorithms, database applications with search functions for a large-size data set, etc.

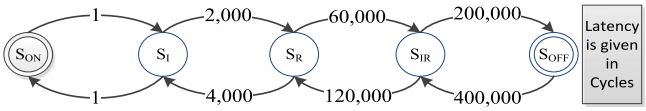

Fig. 3. Used power state machine with latencies of state transition.

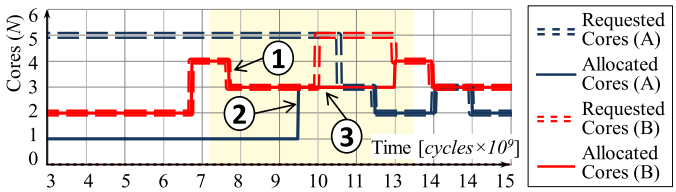

Fig. 4. Energy unaware *shrinking* is performed.

A_i at time t is given as $w_i(t)$, *s.t.* $N_i(t) = f(w_i(t))$. The transfer function f can be determined using *either* analytical models like in [12] *or* statistical analysis with extensive performance simulations as adopted in this paper. Each application A_i has a set of workload classes WC_i, obtained using offline statistical analysis of execution properties w.r.t. the input data characterization. The concept is to leverage the application-specific knowledge to obtain hints towards potential core requirements of an application under a given workload.

At time t, the associated time periods for an application A_i are:

- $TSE_i(t)$: time between current *Shrink* and upcoming *Expand*.
- $TES_i(t)$: time between current *Expand* and upcoming *Shrink*.
- $TSS_i(t)$: time between current *Shrink* and upcoming *Shrink*.
- $TEE_i(t)$: time between current *Expand* and upcoming *Expand*.

Programming Malleable Applications – Potential techniques to develop such malleable applications are Intel's threading building blocks (TBB) [13] and resource-aware programming paradigms [12], [14], [15]. The latter provides special syntax and compiler support for *Expand* and *Shrink* operations such that, application developers specify hints for resource requirements in the program (see details in [14], [15]).

Energy efficiency function: To account for both the power and performance effects, we employ the *Energy-Delay*^{2}*-Product* – ED^2P as the energy efficiency function. This is important because, if one application's energy benefits hurt others' performance efficiency then it may not be acceptable in our scenario as allowing such a condition will lead to complete monopolization by one application. For energy, our DPM accounts for the dynamic energy due to switching activity, leakage energy, and resource allocation/de-allocation energy as a result of an *Expand* or *Shrink* operation.

Power state model: Based on the multi-state power model of advanced processors like in [21], each core C_j may be put into one of the following power states: $P_j = \{ON, I, R, NR, OFF\}$; see Fig. 3. This provides a tradeoff between leakage savings and wakeup overhead. The power states are:

- *ON*: the core is powered-ON at the nominal voltage-frequency (V-f) level and consumes full power;
- *I*: the core is in the *idle* state and clock gating is active;
- *R*: the core is in the *retentive* state, PLL is active, and L1 cache is flushed;
- *NR*: the core is in the *non-retentive* state, PLL is inactive, and L2 cache is flushed;
- *OFF*: the core is in the *switched-OFF* state. The complete state of the core is backed up in an off-chip or a small on-chip memory powered by the I/O supply to allow faster wakeup times during the power up phase.

Note, a core can operate at different V-f levels in the ON state depending upon the workload. To stay orthogonal to DVFS, and to purely demonstrate the benefits of our scalable DPM policy, we assign nominal V-f level in the ON state. Therefore, employing DVFS is out of the scope of this paper.

4 Motivational Case Study

In this section, we analyze and discuss potential issues w.r.t. performance and energy efficiency related to *Expand* and *Shrink* operations for two malleable video coding applications (denoted as *A* and *B*) from the *Parsec* benchmark suite [20]. During a *Shrink*, a malleable application may have two options that are evaluated using the following case studies that represent resource-competing scenarios for an eight core system.

Case Study 1 – Shrink without reservation: In this case, all *dispensable cores* are returned to the local resource manager. Fig. 4 shows *Expand/Shrink* (in terms of core requirements) and the resource allocation of two applications *A* and *B*. Label ① shows the scenario where the application *B* performs a *Shrink* operation (4→3 cores) and returns 1 *dispensable core* back to its local resource manager. This core is later on allocated to the application *A* (see label ②). A problem occurs when *B* performs an *Expand* operation right afterwards but does not receive back its returned core (see label ③). This deficiency of one core results in performance loss and energy inefficiency of *B*.

From the power management perspective: In case a time-based DPM (like [16]) is employed, it would power-gate the idle core returned by the application *B*. However, this core is immediately allocated and used by *A*, thus resulting in frequent OFF-ON that may not be energy-wise beneficial considering the short *Shrink-to-Expand* time periods. Since in a distributed paradigm one application may not have the full knowledge of the other applications, such a time-based scheme may not provide effective power management.

A Potential Solution and Related Challenges: To address the above-stated issues, an application may temporarily reserve the *dispensable cores* instead of returning them back to the local resource manager. However, this requires:

1) A *scalable per-application resource reservation policy* that accounts for the knowledge of the *Expand/Shrink* operations of other simultaneously executing applications; and
2) Accurately predicting the *Expand-to-Shrink* and *Shrink-to-Expand* time periods.

Frequent Expand/Shrink operations may lead to an increased frequency of local resource management decisions, and thus results in excessive latency and energy overhead due to:

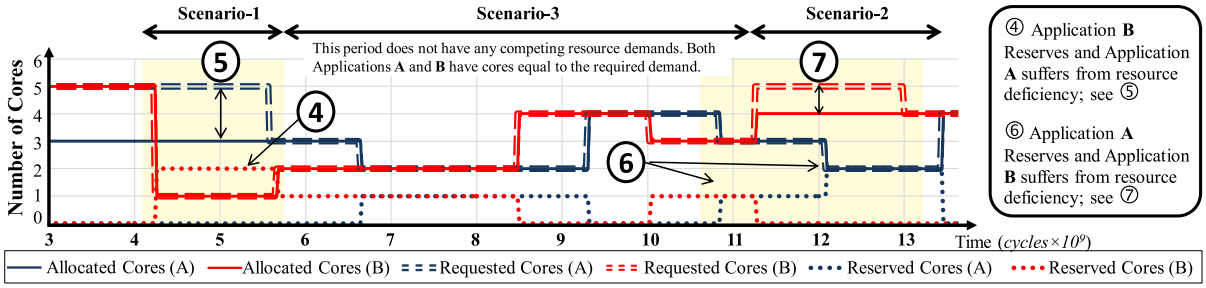

Fig. 5. Motivating the need of energy-awareness in *shrinking*: highlighting the conflicting scenarios of *Expanding* and *Shrinking* for two concurrently executing malleable video coding applications.

- Frequent resource negotiations among local resource managers of different malleable applications, decision logic, search for free cores, continuous adaptation, etc.
- Resource allocation and de-allocation: spawning a new thread of the *expanding* application on the newly-allocated core and stopping a thread of the *shrinking* application on this core, task migration, etc.

The latency overhead of applications' local resource management ranges from 0.1 ms to 10 ms at 2 GHz with an energy cost of 20 μJ per activation of the local resource management [11]. Consequently, a long latency also results in degraded energy efficiency of the application. Therefore, if *Expand-to-Shrink* and *Shrink-to-Expand* time periods of different applications are correctly predicted and jointly considered, significant energy savings can be obtained by avoiding frequent local resource management decisions and sudden variations in the power states of dispensed cores.

Case Study 2 – Shrink with reservation: If not carefully determined, the reservation decisions may result in severe resource monopolization and serious degradation in the performance/energy of other competing malleable applications. We illustrate the issues of *direct* reservation at every *Shrink* operation with the help of the following scenarios in Fig. 5.

Scenario 1: Application *B* reserved two cores (see label ④) and application *A* suffers from resource deficiency that also leads to energy inefficiency of *A* (see label ⑤).

Scenario 2: Application *A* reserved one core (see label ⑥) and application *B* suffers from resource deficiency that also leads to energy inefficiency of *B* (see label ⑦).

Note that the reserved core is idle or power-gated during the reservation period.

A Potential Solution and Related Challenges: An option could be to temporarily lend the reserved core for a short period to improve the energy efficiency of *A* (e.g., *B* lends one core to *A* in scenario 1). However, *B* needs to get its core back before its next *Expand*. This decision of lending has to be taken at a *Shrink* while considering its impact on the overall energy consumption. On one hand, lending will dismiss the option of power-gating. On the other hand, lending will improve the energy efficiency of the borrowing application. Hence, total energy benefit needs to be computed at the *Shrink* operation, as it depends upon the impact of *shrinking*, reservation, and lending on the speedup and energy efficiency of the owner and competing applications. *Therefore, an application-level energy-aware shrinking policy is required* that integrates a core lending policy to realize scalable power management.

5 Scalable Dynamic Power Management

5.1 System Overview

Fig. 6 illustrates the flow of our DPM implemented as an application's local power manager. It consists of four components.

1) *A hybrid prediction technique* **[Section 5.2]** that predicts the time period and number of required cores for *Expand-to-Shrink* and *Shrink-to-Expand* operations (i.e., $TSE_i(t)$, $TES_i(t)$, $TSS_i(t)$, $TEE_i(t)$). It *jointly* accounts for (1) the application-level knowledge of *Expand*/*Shrink* profiles and workload characterization; along with (2) the hardware-level knowledge (i.e., run-time monitored statistics) in order to provide an accurate prediction of *Expand* and *Shrink* profiles under uncertainties (like sudden application workload fluctuations). A combination of both application-level and hardware-level information defines the *hybrid* nature of the prediction.
2) *An energy-aware resource shrinking (enRoS) policy* **[Section 5.3]** that accounts for the mixed malleable workloads to perform an *energy-efficient resource shrinking of the owner application*. It performs an online analysis of *Expand* and *Shrink* profiles of the owner and other resource-competing applications.

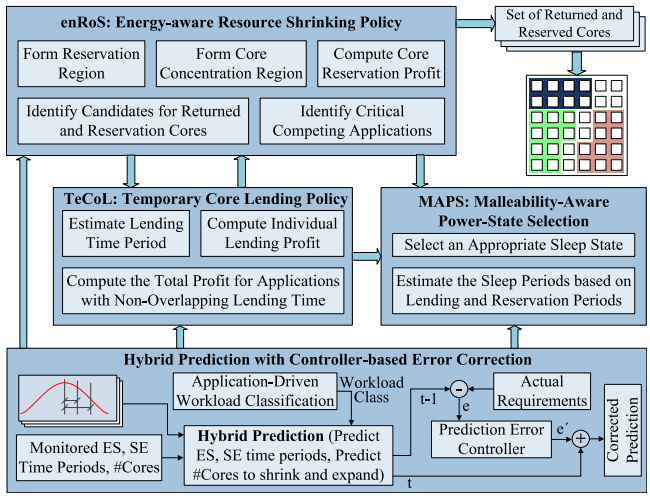

Fig. 6. Overview of our scalable DPM in the local power manager with energy-aware *shrinking* and core lending policies.

1. **HybridPrediction** (*Input*: application-based PDF and monitored history (H) of *Expand* and *Shrink* operations for application A_i; workload classes WC_i; *Output:* predicted values for *parameters*)
2. **For each** $v \in V = \{TSE_i, TES_i, TSS_i, TEE_i, N_i, S_i, C_i\}$ **do**
3. $wc \leftarrow$ matchWorkloadClass(WC_i, v.PDF);
4. $v.P_A \leftarrow F(wc, \mu+2\sigma)$;
5. $v.P_H \leftarrow$ mean(v.H);
6. $v.P \leftarrow (v.P_A + v.P_H) / 2$;
7. $v.P \leftarrow$ applyCorrection(v.p,$v.p_{t-1}$);
8. **endFor**
9. **return** $V = \{TSE_i, TES_i, TSS_i, TEE_i, N_i, S_i, C_i\}$;

Fig. 7. Pseudo-code of the hybrid prediction technique.

The enRoS identifies the *shrinking* candidate cores depending upon the current workload properties and location of allocated cores. Afterwards, it selects a set of competing applications. The enRoS evaluates the energy benefit of the *shrinking* candidates for returning them to the competing applications or *proactively reserves* them until the upcoming *Expand* of the owner application. These reserved cores are *unavailable* from the local resource manager's perspective and are *not* returned during the *Shrink* operation, while at the same time *available* from the owner application's perspective for its execution.

3) *A temporary core lending (TeCoL) policy* [Section 5.3.6] that evaluates the energy profit of *temporarily lending the reserved cores to the critical competing applications* within the reservation time period (i.e., before the next usage by the owner application). It thereby contributes towards the overall energy efficiency of the whole system while lowering the performance degradation of the competing applications.
4) *A malleability-aware power state selection (MAPS) policy* [Section 5.4] that selects an appropriate sleep state depending upon the predicted upcoming *Expand* of the owner application. To obtain a robust prediction for the power-gating of cores, the information available to the enRoS policy (like prediction of *Expand*/*Shrink* time periods and resources, and reservation time periods) can be exploited.

We explain these four components of our scalable DPM in the following. For discussion on the implementation issues, refer to Section SVII in the Supplementary Material, available online.

5.2 Hybrid Prediction of Expand and Shrink Operations

Once an application decides to *shrink*, the upcoming *Shrink-to-Expand* (TSE_i) and *Expand-to-Shrink* (TES_i) time periods and number of cores required in each time period are predicted. History-based prediction techniques like [16] may lead to significant mispredictions in case of suddenly changing workloads [26]. Application-based prediction techniques (like [26], [28], [30]) may handle such workload fluctuations, but do not account for the hardware-level information (like monitored statistics and idle period fluctuations that depend upon various hardware effects like cache misses, interference on shared resources, etc.), thus may lead to mispredictions. To overcome these limitations, we employ a hybrid prediction technique as formulated in Fig. 7 (see a high-level overview in Fig. 6).

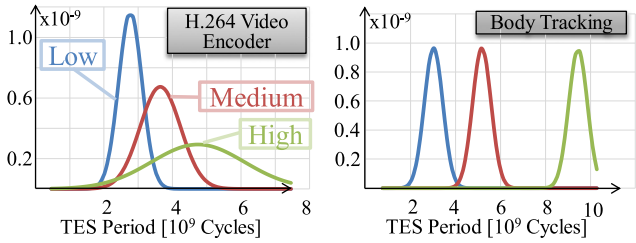

Fig. 8. Probability density functions (PDFs) of *Expand-to-Shrink* time periods for different workload classes of two test applications.

The inputs for prediction are the application-dependent workload classes WC_i, probability-density functions (PDF) of different parameters associated with *Expand*/*Shrink*, and the monitored history data H of these parameters. First the workload class is matched based on the current data set at the *Expand*/*Shrink* point and an appropriate PDF is selected based on the workload class. For instance, Fig. 8 shows the PDFs for two applications with three different workload classes, i.e., *low*, *medium*, or *high*. These workload classes depend upon the application type and are obtained using an offline statistical analysis of the application workload. The offline profiling is done for various test inputs. For each profile run, different values of the parameters (like TSE_i and N_i) are recorded to generate a data set and distribution is modeled to obtain different PDFs (see Fig. 8) for different workload classes. Each class represents a set of inputs with distinct execution properties and parameter PDFs. As shown in Fig. 8 for the H.264 video encoder, different input sets are different videos partitioned into (1) high motion/textured videos lead to the *High* workload class; (2) medium motion/textured videos lead to the *Medium* workload class; and (3) videos with low motion/homogeneous objects lead to the *Low* workload class. Note: a change from *low* to *high* workload class denotes an abrupt change in the workload and the core requirement of an application varies significantly when moving from one workload class to another. Afterwards, the *application-based prediction* of different parameters is computed as a highly-probable value obtained using the PDF with "mean of distribution (μ) plus two standard deviations (σ)" providing an estimate with a probability of 0.94 (following a Gaussian distribution, line 4). See more discussion in Section SVII in Supplementary Material, available online.

The *history-based prediction* is obtained as the average of the history window (line 5). It compensates for various hardware-level uncertainties like cache misses. The *hybrid prediction* is obtained as the average of application-based and history-based predictions. Since the prediction may deviate from the actual requirements, our DPM employs a prediction correction using a simple PID-based controller. The controller parameters (K_P, K_I, K_D) are computed using the Ziegler-Nichols Method [25] with the settings of Eq. (1)

$$K_P = 0.6 \times K_{PC}; K_I = K_P/(0.5 \times T_C); \\ K_D = K_P \times (0.125 \times T_C); K_{PC} = 0.8; T_C = 2; \quad (1)$$

```
//Step-1: Compute the Number of Returned Cores and Reservation Candidates
1.  Ci(t) ← Ni(t) − Si(t);
2.  if(Ci(t) ≥ Ei(t+1))   then Ri(t) ← 0;
3.  else Ri(t) ← min(Si(t), Ei(t+1) − Ci(t));   endif
4.  Si(t) ← Si(t) − Ri(t);
//Step-2: Identify Cores for Reservation
5.  D ← ∅;  minSum ← maxINT;  CC ← ∅;
6.  For each Core c ∈ {Cross, Median1, Median2, Random1, Random2} do
7.    For each Core x ∈ Ai.C do
8.      D(x) ← computeManhattanDistance(c, x);
9.    endFor
10.   Sort(D, ascendingOrder);
11.   Sumc ← Σ_{k=0...Ci(t)} D(k);
12.   if(Sumc < minSum)  then  minSum ← Sumc;  CC ← c;  endif
13. endFor
14. RC ← getCores(CC, Ai.C, Ri(t));   R ← ∅;   NRes ← 0;
15. S ← getCores(CC, Ai.C, Ri(t)+Si(t));
//Step-3: Identify Competing Applications in the Reservation Region
16. For each Core r ∈ RC do
17.   nn ← findNearestNeighbor(CC, D);
18.   RR ← getRectangularRegion(r, nn); // Reservation Region
19.   AR ← getApps(RR);  // get all applications in the reservation region
20.   CA ← ∅;
21.   For each Application a ∈ AR do  // find the competing applications
22.     if(TSEa(t+1) is within the Reservation Period) then
23.       if(Ea(t+1) > NIdleCores in RR)  then  CA.add(a);  endif
24.     endif
25.   endFor
//Step-4: Compute the Energy Profit of the Owner Applications
26.   profitRes ← (ΔEdyn + ΔEleak − (Pleak(r) × TSEi) + Eshrink) × ΔDi²;
//Step-5 and 6: TeCoL: Temporary Core Lending to the Competing
27.   ProfitLend ← 0;
28.   For each Application a ∈ CA do
29.     a.Loss ← (ΔEdyn + ΔEleak + ΔERM) × ΔDa²
30.     if(TSEa(t+1) is within the Reservation Period) then
31.       a.ProfitLend ← GetLendingProfit(r,a);   endif
32.     if(a.ProfitLend ≠ overlapping and within Reservation Period) then
33.       ProfitLend ← ProfitLend + a.ProfitLend;   endif
34.   endFor
//Step-7: Compute the Effective Loss and Effective Profit
35.   eLoss ← max_{∀a∈CA} (a.Loss);
36.   eProfit ← profitRes + ProfitLend;
//Step-8: Reservation Decision
37.   if(eProfit > eLoss) then
38.     R.add(r);   NRes ++;
39.     if(NRes ≥ Ri(t)) then   break;   endif
40.     S.remove(r);
41.   endif
42. endFor
43. Si(t) ← (Ri(t) − NRes) + Si(t);   Ni(t) ← Ri(t) + Ci(t);
```

Fig. 9. Pseudo-code of energy-aware resource *shrinking* and lending.

The prediction error is added to the new prediction as a correction factor (line 7). This controller loop only works on the prediction error and it is not dependent upon the periodic or non-periodic nature of *Expand*/*Shrink* operations. The set of parameters is forwarded to enRoS, TeCoL, and MAPS policies.

5.3 enRoS: Energy-Aware Resource Shrinking

Fig. 9 shows the detailed pseudo-code of our DPM with enRoS and TeCoL. The algorithm is processed for each *Shrink* operation at time t for a given application $A_i \in \boldsymbol{A}$. The problem is to determine sets of returned cores $\boldsymbol{S}$ and reserved cores $\boldsymbol{R}$, such that the overall system's energy and performance efficiency (in terms of ED^2P) is improved. The algorithm operates in the following eight key steps that are explained below.

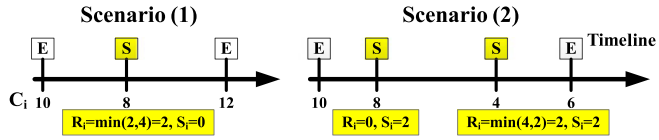

Fig. 10. An example showing the computation of number of returned and reservation candidates at a *Shrink* operation.

5.3.1 Step 1: Compute the Number of Returned Cores and Reservation Candidates [Lines 1–4]

First, the number of returned cores $S_i(t)$ and the number of potential reservation candidates $R_i(t)$ are computed. If the number of cores required to execute the current workload $C_i(t)$ is greater than or equal to the core requirements at the upcoming *Expand* $E_i(t+1)$ then all the extra cores are returned back to the local resource manager, i.e., a full *Shrink* is performed and there is no need to reserve any core. Otherwise, a partial *Shrink* is performed and a subset of cores is reserved for the upcoming *Expand*. For this, $R_i(t)$ and $S_i(t)$ are computed; see an example computation in Fig. 10 for two scenarios. The design of the formula $R_i(t) = min(S_i(t), E_i(t+1) - C_i(t))$ is based on three points:

- $S_i(t)$, because reservation can only be made for the *dispensable* cores that are the *shrinking* candidates.
- $E_i(t+1) - C_i(t)$, because those will be the number of cores that will be required by the owner application in the next *Expand* operation. Therefore, in the best case, those many cores should be available through reservation because performing another resource management round to get more cores will incur overhead and delay, as discussed in Section 4.
- *Minimum of above two:* because, an application can only reserve cores that are candidate for returning to the local resource manager at the *Shrink* operation, and does not need more cores than what it is going to ask in the next *Expand* operation, considering the decision window of E→S→E.

Note, when expanding to 12 cores from a *Shrink* state of eight cores (see Fig. 10, scenario 1), only at most two cores can potentially come from the reservations. Still two more cores (or even four if no reservation was performed at the previous *Shrink*) will be required. These additional cores will be obtained through the resource management layer, i.e., by negotiating with other applications and considering the communication cost, such that total system performance is maximized (as shown by the flow in Fig. 2). The reservation candidates are evaluated for their reservation benefit in the next steps.

5.3.2 Step 2: Identify Reservation Candidates [Lines 5–15]

In the second step, the returned cores S and the reservation candidate cores RC are determined. The core selection for *shrinking* is based on the following. Let us assume a scenario where an application A_i has a mixture of concentrated and slightly dispersed distribution of owned cores. The scatter in the owned cores may be due to two reasons: (1) the fragmentation issues that result from continuous trading over a longer period, and unaligned and varying rates of *Expand*/

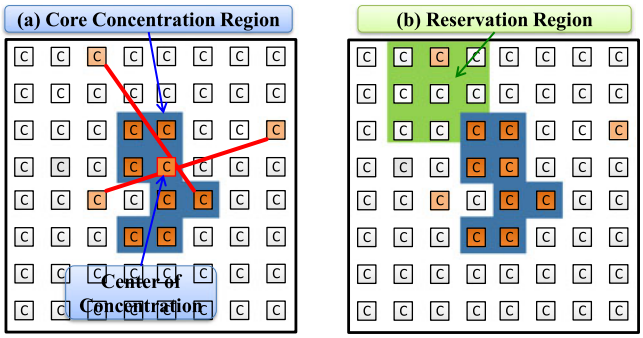

Fig. 11. An example showing (a) formation of core concentration region; (b) formation of reservation region.

Shrink operations of concurrently executing applications, and/or (2) the successful trading with a slightly farther application that *may afford* giving it. In such a case, a farther core may be beneficial for returning/reservation because it has a higher communication overhead and is less important for current speedup requirements to process the $w_i(t)$. In case of reservation, such a farther core is also hard to be reserved by the owner application A_i because other applications in the close vicinity of this reservation candidate may have a relatively higher energy benefit compared to the owner application. It thereby reduces the risk of monopolization. For this, *enRoS* forms a *core concentration region*, such that the number of cores outside the region is greater than or equal to $R_i(t) + S_i(t)$. Note, most of the cores owned by an application are in the *core concentration region* (see an example in Fig. 11) and only a few cores are outside it, which are candidates for returning/reservation. This will alleviate the fragmentation issue over the period of time, especially, in case of returning the cores at a *Shrink* operation as an owner application tends to keep cores in its *core concentration region* and trades farther cores first.

To form a *core concentration region*, we first compute the sum of *Manhattan Distance D* from a given core to all other cores owned by the application A_i. The one with the minimum sum value *minSum* will be given as the *center of concentration*. However, its complexity is $N_i(t) \times (N_i(t) - 1)$. Therefore, enRoS employs the following set of initial points as the potential candidates for the *center* of "core concentration region" (line 6).

- $Median_1$: median of line joining top-left and bottom-right points.
- $Median_2$: median of line joining top-right and bottom-left points.
- *Cross*: crossing of two lines from top-bottom and left-right points.
- $Random_1$ and $Random_2$: two random starting points to avoid trapping in the locally optimal solutions.

Note, a designer may specify any set of candidate center points; it does not affect the generality of our DPM. For each center candidate c, following steps are performed (lines 7-12). The Manhattan distances D from the core c to all other cores owned by A_i are computed. The array D is sorted in an ascending order. The cores required for the current execution phase, i.e., $C_i(t)$ are selected, and the sum of their Manhattan distances Sum_c is computed. The center candidate with the minimum sum value *minSum* is selected as the *center of concentration* CC and cores outside this region are extracted as the set of returned cores S and reservation candidate cores RC. Note, a reservation candidate can still be close to the core concentration region, the communication factor is incorporated in estimating the performance and energy benefits of the owner application. However, other parameters also matter for estimating the profit of reservation, see Steps 4-6.

An Example: Fig. 11a illustrates the process of formation of the "core concentration region". The cross candidate point is shown here as the *center of concentration*. There are three cores outside the concentration region. They are potential candidates for *shrinking* and *reservation*.

5.3.3 Step 3: Identify Competing Applications in the Reservation Region [Lines 16–25]

It may happen that all the reservation candidates may not be reserved due to the energy inefficiency of competing applications and some of them are also returned along with the original candidates for returned cores. Therefore, for each reservation candidate, first *a set of competing applications* is identified and evaluated for the reservation energy profit. Identifying the set of competing cores operates in the following two steps:

a) Form the Reservation Regions: since there may be several applications executing concurrently, it may be time-wise very complex to trade each and every reservation candidate with all the other malleable applications in case of large on-chip systems. Therefore, a so-called *reservation region* is formed. It is a rectangular region around each reservation candidate r, such that r is the center of rectangle and the diagonal length is equal to the distance between the reservation candidate r and the nearest core in the concentration region *nn* (lines 17-18). Fig. 11b shows an example scenario with the rectangular reservation region for the top-left core outside the concentration region. Note, the nearest core in the concentration region already shows that beyond this point, most of the cores are actually owned by the application which is going to *reserve* or *return* the candidate r. Therefore, the probability of having a more competing application in the core concentration region is low. Moreover, forming a rectangular region (1) avoids potential overlap with other reservation regions so that the same competing application is not evaluated multiple times, and (2) restricts the number of competing applications so that the trading overhead is kept low (see overhead discussion in *Supplementary Material, available online*). Note, the temporary lending will ameliorate a competing application.

b) Identify Applications within the Reservation Region: afterwards, competing applications are selected within the reservation region RR. A set of applications A_R executing within RR is obtained (line 19). An application $a \in A_R$ is denoted as *competing* if it fulfills two properties.

- The *Expand* operation of a occurs within the reservation period (line 22). If not, then this application is not competing and its local resource manager[4] will

4. Note that the resource allocation is out of the scope of this paper. We assume a distributed resource management policy like [11].

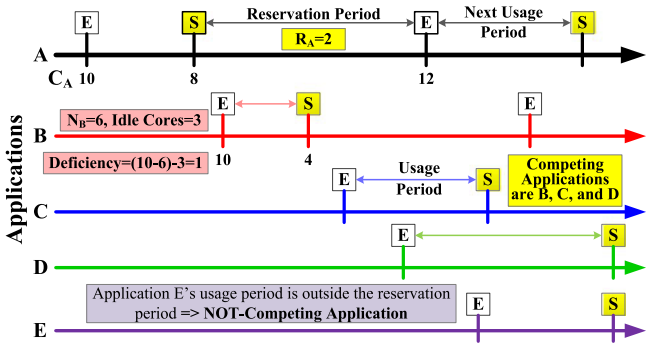

Fig. 12. An example showing different competing applications.

obtain more cores through negotiation with other applications. If the *Expand* occurs within the reservation period, then the energy and performance efficiency of the application a may suffer due to the reservation of the candidate r.

- The number of idle cores in the reservation region is checked. If this fulfills the requirements of the competing applications then it may be safe to reserve the core r. Otherwise, this application is added to the set of competing applications CA (line 23).

An Example: Fig. 12 illustrates an example scenario of competing applications. Note, the *Expand* operation of application E is outside the reservation region. Therefore, application B, C, and D are identified as the competing applications.

Now, for each reservation candidate r, the reservation profit is computed considering the ED^{2}P profit of the owner applications for the upcoming *Expand* operation and the ED^{2}P loss of the competing applications.

5.3.4 Step 4: Compute the Energy Profit of the Owner Application [Line 26]

The energy profit of the owner applications comprises of three key energy components as described below:

i) Dynamic Energy Benefit of the Owner Application (ΔE_{dyn}): If the core r would have been returned, there would have been a potential risk of not getting it back that would have resulted in performance loss and energy inefficiency. In case of proactive reservation of core r, this risk is avoided. Moreover, the next *Expand* operation of the owner application will be immediate as the thread (code and data) may be available for execution depending upon the core's power state. Therefore, the application will have immediate speedup/energy improvement.

ii) Leakage Energy Benefit of the Owner Application ($\Delta\ E_{leak} + P_{leak}(r) \times TSE_i$): If the core is not returned, it may be put into a low power state that may provide a higher potential of leakage energy savings. It may also affect the leakage energy due to a faster execution in the next usage period $TES_i(t+1)$.

iii) Dynamic Energy Benefit for avoiding the Shrink Operation (E_{shrink}): Since the thread code of the owner application may be available on the reserved core, the energy overhead of thread allocation and de-allocation can be saved. Moreover, it will also reduce the energy/performance overhead of the next resource management decision.

5.3.5 Step 5: Compute the Energy Loss of the Competing Applications

As discussed earlier, core reservation may affect the dynamic and leakage energy efficiency of the competing applications. Therefore for each competing application a, the potential energy and performance losses due to the deficiency of r are computed. The energy loss is determined using the dynamic and leakage energy differences with and without the candidate r. Since deficiency of the core r may lead to extended resource search, another parameter is the difference for the recourse manager energy overhead ($\Delta\ E_{RM}$).

5.3.6 Step 6: TeCoL: Temporary Core Lending to the Competing Applications [Lines 27–34]

Our DPM employs a temporary core lending (TeCoL) policy that allows an owner application a_i to temporarily lend its reserved core to the most critical competing application. It evaluates the ED^{2}P profit of temporarily lending a reserved core r to all the competing applications in $a \in CA$ within the reservation period (lines 30-31, Eq. (1)). The lending profit consists of the dynamic and leakage energy benefit due to faster execution, leakage energy loss due to *not* power-gating the core during reservation, and the energy loss due to thread (de)-allocation for the lending and borrowing applications. The competing application a only borrows the reservation candidate r when the lending benefit is positive. Since multiple applications may benefit from temporary lending depending upon the time period between their respective *Expand* and *Shrink*, the goal of TeCoL is to compute the total ED^{2}P profit of lending core r to all non-overlapping applications from CA (lines 32-33)

$$a.Profit_{Lend} = \begin{pmatrix} \Delta E_{dyn} + \Delta E_{Leak} - P_{leak}(r) \times T_{Lend(a)} \\ -2 \times (E_{Alloc} + E_{DeAlloc}) \end{pmatrix} \times \Delta D_a^2. \quad (2)$$

An Example: An example scenario is shown in Fig. 13, where potential borrowing applications are B and C. The lending period is determined by subtracting the thread allocation and de-allocation time from the reservation period. The application D is not considered for lending because its potential lending period is less than the thread allocation and de-allocation time. This is necessary to avoid the performance penalty to the owner application. Note that the lending periods of applications B and C are non-overlapping. Therefore, the same reserved core may be lent to both applications in case the lending profit is positive.

Discussion on the Borrowing Applications: *Only* the *expanding applications* can be in the borrowing lists of the shrinking applications at a given point in time. This already restricts the number of potential borrowing scenarios. The maximum number of cores that an application can borrow is restricted by its additional core requirements during the current *Expand* operation, i.e., $C_i(t+1) - N_i(t)$. First, the local resource manager will try to find more cores through inter-application negotiations in the resource allocation step. If this does not fulfill the requirements, *only* then the temporary borrowing will be required and this application is registered in CA of the relevant *reservation regions*, i.e., where

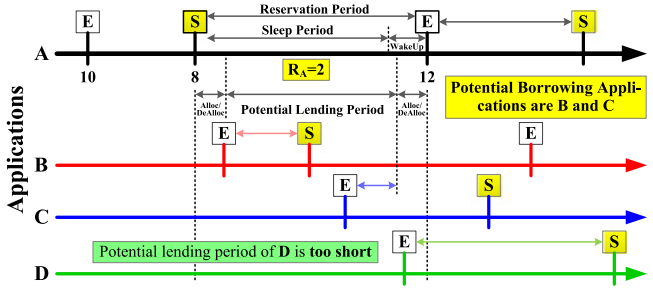

Fig. 13. An example showing the lending periods and borrowing applications.

its owned cores exist. After the borrowing requirements of an application are fulfilled, it is unregistered from the candidate list *CA*.

5.3.7 Step 7: Compute the Effective ED^{2}P Loss and Profit [lines 35–36]

The effective ED^{2}P loss *eLoss* is computed considering the maximum ED^{2}P loss from the set of competing applications. Effective profit is given as the *sum* of the reservation profit of the owner application *and* the total lending profit.

5.3.8 Step 8: Selective Resource Reservation and Shrink Decision [lines 37–42]

If the effective ED^{2}P profit is greater than the effective ED^{2}P loss, the core r is reserved. Otherwise it is returned as part of the *Shrink* operation. Due to an overall positive ED^{2}P benefit, the enRoS policy realizes an energy-aware *Shrink*.

5.4 MAPS: Malleability-Aware Power-State Selection

Fig. 14 presents the pseudo-code of our MAPS policy. It selects an appropriate sleep state $p \in P_r$ for every reserved core $r \in R$ depending upon the application-specific properties of the malleable workloads (i.e., reservation and lending time periods). Fig. 13 shows the pseudo-code of our MAPS policy. The algorithm starts with a loop over all the reserved cores. First, the sleep duration T_{Sleep} is computed considering the predicted time between the *Shrink* and the upcoming *Expand* (i.e., $TSE_i(t)$ for the owner application a_i) and the total lending period to the competing applications. Afterwards, for each power state p the leakage energy benefit E_{Leak} for setting the core r into p state is computed and

```
1.  State ← ON;        max ELeak ← 0;
2.  For each Core r ∈ R do
3.    TSleep ← predict(r, TSEi(t)) − Σ∀a∈CA TLend(a) ;
4.    For each PowerState p ∈ Pr do
5.      ELeak ← PLeak(r, p) × TSleep;
6.      if ((ELeak > EWakeup(r, p)) and (ELeak > max ELeak)) then
7.        State ← p;        max ELeak ← ELeak ;
8.      endif
9.    endFor
10.   r.setPowerState(State);
11. endFor
```

Fig. 14. Pseudo-code of the MAPS policy.

TABLE 1
Experimental Setup with Tools Used

Performance Simulator	Gem5 (hg dev, April 2012) [22] gives cycle-accurate traces considering the shared memory/cache resources (core-private L1 data cache and shared L2 data cache with MOESI snooping based cache coherency protocol)
Core Configuration	Alpha 21264, 2 GHz [22]
Power Simulator	McPAT 0.8 r274 [23]
Technology Node	45 nm HP (ITRS) [23], [1]
Power State Machine	45 nm PENRYN [21] Table 3
Operating System	Linux 2.6.27.2
Application Set	Parsec [20]; applications were extended manually with *Expand* and *Shrink* operations
Manycore Simulator	In-house C++ high-level manycore simulator with integrated scalable DPM. It reads 100s-1,000s of real traces generated from Gem5 and McPAT and simulates a system of many cores with a NoC infrastructure.

checked for whether it amortizes the wakeup overhead from state p to the ON state. The power state that provides the highest leakage energy saving is selected for the core r.

6 Results and Discussion

6.1 Experimental Setup

Table 1 shows our experimental setup with different tools used along with their sources. Evaluations are performed for different applications from the *Parsec* benchmark suite with different input combinations/mixtures to realize mixed-malleable workloads; see details in Table 2. To realize *Expand* and *Shrink* operations, we have manually extended these selected applications. The resource management is based on the work of [11]. For each local resource management, the latency overhead ranges from 0.1 to 10 ms at 2 GHz, while the energy overhead is 20 μJ per activation. The NoC is based on x-y routing with a 4-stage pipelined router implementation. The *flit router energy* $= 1.56 \times 10^{-10}$ J and the *flit wire energy* $= 3.88 \times 10^{-11}$ J. Note, the latency and energy overhead of the proposed technique is included in the results. We have used a 45 nm technology. The details of different power states and state transition overheads are given in Table 3. The sub-threshold leakage is 1.77 W and the gate leakage is 0.12 W.

For fairness of comparison, we provide the same prediction and power state machine to all the comparison partners. Moreover, we also make all comparison partners aware of the workload variations through *Expand* and *Shrink* commands.

Comparison partners: we compare our DPM with advanced state-of-the-art DPM policies of [6] (MaxBIPS)

TABLE 2
Applications and Input Combinations Used in Experiments

Application Type 1		**bodytrack (bt)**
Input Sets	*A*	1,000, 2,000, 4,000 particles
	B	1,000, 2,000, 4,000 particles
Application Type 2		**x264 git dev branch**
Input Sets	*mix1*	bluesky, station, eledream
	mix2	in_to_tree, old_town_cross, eledream
	mix3	sunflower, station, crowd_run

TABLE 3
Power State Table [21]

Power States and Transitions	Delay [Cycles]	Energy Overhead [J]	Power Savings Factor
Running –> Idle	1	9.44×10^{-10}	-
Idle –> Running			
Idle –> State Retentive	2,000	1.32×10^{-6}	**1**
State Retentive –> Idle	4,000	2.64×10^{-6}	
State Retentive –> State Irretentive	60,000	1.47×10^{-5}	**0.4**
State Irretentive –> State Retentive	120,000	2.95×10^{-5}	
State Irretentive –> Off	200,000	1.28×10^{-5}	**0.12**
Off –> State Irretentive	400,000	2.57×10^{-5}	
Off	-	-	**0.016**

and [16] (IdlePG, UtilPG), and our SEAD policy [5] (i.e., the DPM policy of the base conference version of this transaction submission). The evaluation is performed for varying number of cores (36–1,024), applications threads (5–200), and input mixtures. We target our evaluation considering distributed computing scenarios (e.g., in data centers) where a system is typically hosting many instances of the same application (each instance can be multithreaded) processing heterogeneous workloads. For instance, a cloud service is hosting web-based video encoding/decoding services to different users. Our system is based on *partitioned global address space*, where different applications have their dedicated address space in the main memory, while only different threads of an application are allowed to share the data. This will alleviate *inter-application interference*, while *intra-application interference* is considered in the experiments as the cycle accurate traces for each multithreaded scenario are obtained from cycle-accurate Gem5 simulations with shared memory settings. *Inter-application interference* from NoC usage/contention can still happen.

6.2 Comparison to State-of-the-Art

Fig. 15 provides an overview of ED^{2}P improvements of our DPM compared to state-of-the-art by means of box plots and average value ("diamond" symbol). Each box plot illustrates the summary of 405 experiments, such that every experiment has a different number of cores (ranging from 36 to 1,024), number of application threads (ranging from 5 to 200), and five different application and workload mixtures (as shown in Table 2). To further analyze the distribution of ED^{2}P savings compared to state-of-the-art for different applications and mixtures, Fig. 16 illustrates the box plots per application per comparison partner. Each box plot corresponds to the summary of 81 experiments.

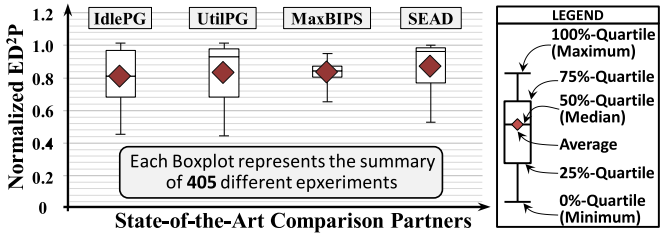

Fig. 15. Boxplot summary of ED^{2}P improvements of our DPM compared to state-of-the-art power management policies.

Summarizing the results of all 405 comparisons, our DPM provides up to 58 percent (average ≈15-20 percent) ED^{2}P reduction compared to state-of-the-art. Note: even 15-20 percent ED^{2}P savings are significant as the comparison is purely based on the power management policy and all other system conditions are set to be same for all the comparison partners. The fact that 75 percent -Quartile is below 1 illustrates that our DPM is beneficial in most of the cases.

Fig. 16 illustrates that the box plots for our savings are concentrated for the "bodytrack" application. The spread is wider for the "x264" application due to high workload variations. Overall, our DPM policy outperforms state-of-the-art due to efficient reservation and lending between different *Shrink-to-Expand* time periods.

Comparing to the IdlePG policy [16] (Figs. 15 and 16a): IdlePG is a timeout based power management policy. It monitors all cores in the system for idleness and power gates them if their idle time exceeds a predefined threshold. Therefore, applications do not return their *dispensable cores*. IdlePG does neither account for other simultaneously executing applications nor recognizes which set of cores belongs to which application, rather independently monitors each core. Therefore, IdlePG suffers from energy inefficiency along with significant performance degradation; especially in case of sudden *Expand* operations. In contrast, our DPM leverages the application specific knowledge and monitored statistics to enhance the prediction of time period and core requirements during the *Expand*/*Shrink* operations. Moreover, our hybrid prediction allows for accurately predicting the overlapping *Expand* and *Shrink* time periods of different applications. Our DPM thereby achieves up to 54 percent (average 19 percent) ED^{2}P reduction compared to the IdlePG policy.

Comparing to the UtilPG policy [16] (Figs. 15 and 16b): UtilPG samples the system utilization after every predefined time window and makes a decision to power-gate a core in case its utilization is under a pre-defined threshold. These power-gated cores are unavailable for execution. Though taking utilization into account provides some

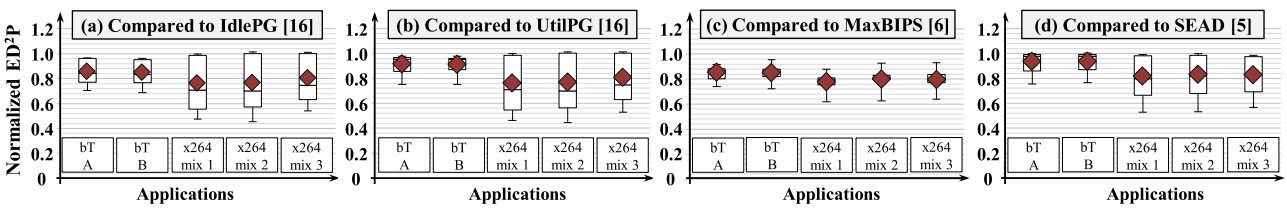

Fig. 16. Boxplot summary of ED^{2}P improvements for different applications: comparing the ED^{2}P improvements of our DPM to different state-of-the-art power management policies (each box plot denotes the summary of 81 experiments).

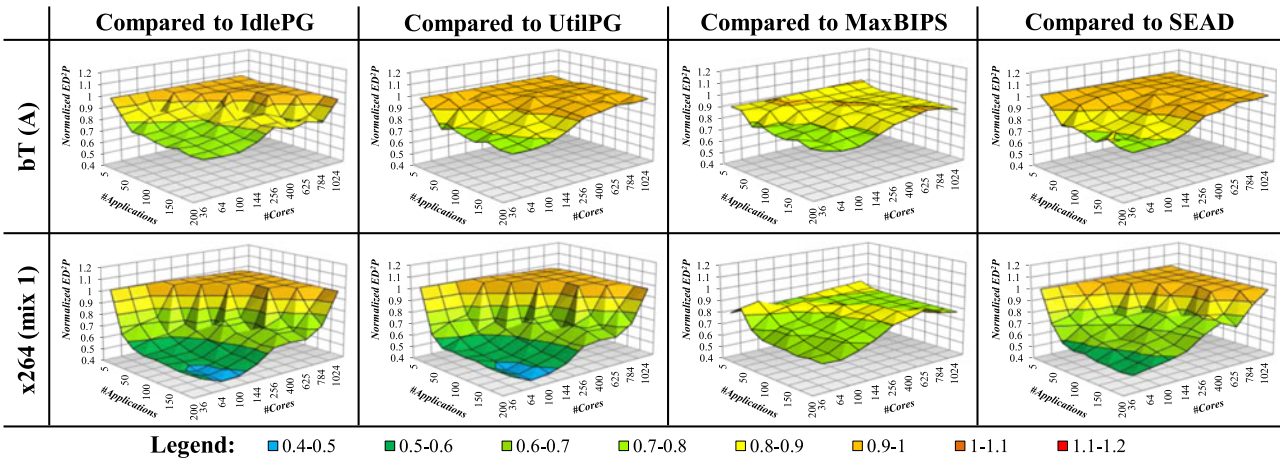

Fig. 17. Scalability results for different number of cores and applications: Showing ED^2P reduction of our DPM achieved compared to different state-of-the-art DPMs.

benefit compared to IdlePG, UtilPG still suffers from energy inefficiency in case of multiple competing applications due to the unavailability of the knowledge of other applications. In contrast, our DPM accounts for other applications in the reservation benefit function, thus allows *shrinking* in case other applications are in urgent need of the resource. Moreover, the TeCoL policy allows other competing applications to temporarily borrow cores and thereby further improves their energy efficiency. As a result, compared to the UtilPG [16], our DPM provides up to 56 percent (average 17 percent) ED^2P reduction.

Comparing to the MaxBIPS policy [6] (Figs. 15 and 16c): MaxBIPS is a throughput optimizing power management policy, where execution phases are predicted based on the application's prior knowledge and the workload demand of the future is unknown. The benefit of our DPM arises from core reservation and temporary lending, as MaxBIPS will return the cores under low-workload scenarios and/or put them in low-power state. Furthermore, the hybrid prediction provides an edge through accurate estimation of the upcoming workload variations at different *Expand*/*Shrink* operations. Note, in cases when there is no sudden *Expand* operation after a *Shrink*, MaxBIPS may be beneficial, as it improves performance of other applications executing in the system. However, in scenarios with abrupt and frequent workload variations reservation is advantageous and our DPM outperforms MaxBIPS. Overall, we achieve up to 35 percent (average 17 percent) ED^2P reduction compared to the MaxBIPS. The concentrated box plots illustrate that our DPM provides relatively stable ED^2P improvements for a wide range of comparison scenarios.

Comparing to the SEAD policy [5] (Figs. 15 and 16d): The SEAD policy represents the contribution of the conference paper (i.e., energy-aware core reservation) on which this transaction is partly based on. Therefore, comparison with the SEAD policy illustrates the improvements of extensions/enhancements of this transaction paper, i.e., the enhanced reservation and novel core lending policies. In particular, our enRoS policy is a more sophisticated reservation policy that identifies reservation candidates considering reservation regions and core concentration regions, and it thereby provides improved reservation decisions. Furthermore, our TeCoL policy allows for temporary lending to avoid aggressive performance degradation of other competing applications. SEAD's reservation decisions often lead to resource monopolization by one application, while our lending is a useful and scalable way to temporarily share reserved resources to improve the energy efficiency of neighboring/competing applications. Our DPM thereby achieves up to 47 percent (average 13 percent) ED^2P savings compared to the SEAD policy. Note, all lending decisions are made locally to the reserving applications. The lending benefits will be shown using a detailed run-time scenario in *Section SII of Supplementary Material, available online*.

6.3 Scalability Results

Fig. 17 illustrates ED^2P savings normalized to IdleGP [16], UtilPG [16], MaxBIPS [6] and SEAD [5] policies for two mixes for an increasing number of cores and applications running in the system. Each 3D plot shows results of 81 experiments. See results and discussion for more mixes in Section SI of the *Supplementary Material, available online*.

Based on the number of cores and the number of applications three regions in these plots (Fig. 17) can be distinguished: (1) the cases where applications receive enough number of cores, (2) the cases where applications suffer from resource deficiency and need to compete for resources, and (3) the cases in between regions (1) and (2) where applications get the number of cores close to their demands but still need to compete for resources to fully fulfill their performance constraints. These plots illustrate that our DPM policy achieves improved energy efficiency in most of the cases. However, there are a few cases where our DPM slightly suffers from energy inefficiency. This is because of the expansion of an application by a significant amount, while another application already reserved many cores. This sudden expansion was not sufficient to be compensated by the temporary lending. Therefore, though one application exhibits better energy efficiency, another application suffered. However, in case of longer executions, due to the system feedback, our DPM moves towards overall energy efficient operating points and the overall savings improve. Moreover, our policy still improves over the SEAD approach as the temporary lending eliminates scenarios

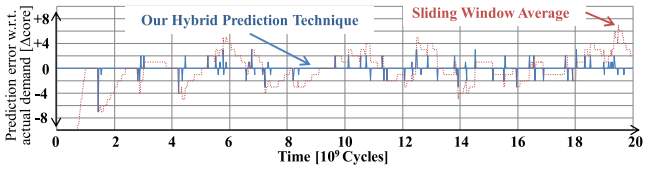

Fig. 18. Prediction accuracy.

of severe resource monopolization. Note, there are certain sets of spikes in the ED^2P savings. This is because of the fact that an application ED^2P is improved in steps, i.e., when moving to the next ED^2P point an application requires more than 1 core. This short-time energy penalty would also be compensated in long execution cases.

The points in Fig. 17 corresponding to normalized ED^2P value close to 1 denote the scenarios where savings diminish due to a high number of concurrently executing applications that will tend to keep all the cores busy and lead to the saturation point in terms of power-gating potential. Note, this does not undermine the scalability, rather shows the limited available potential for power-gating. Whenever there is a power saving potential, our approach always provides better ED^2P savings. The key benefit for a large number of cores that our approach provides is faster power-management decisions due to local trading and reduced overhead of the distributed decision logic.

In *Section SI of the supplementary material*, available online, we provide plots for additional three mixes along with detailed discussion of important observations in different comparison plots.

6.4 Results for Prediction Accuracy

Fig. 18 shows the prediction accuracy for our hybrid prediction technique and a sliding average history based predictor compared to the actual core requirements of applications at run time. On average, our technique accurately predicts the core requirements (avg. error = 0.00 cores, standard deviation = 0.57 cores). However, the sliding window average based predictor underestimates the core requirements by on average 0.94 cores, standard deviation = 5.39 cores. The improved prediction accuracy is primarily due to the joint consideration of application-specific knowledge and hardware-level monitored statistics.

7 Overhead Analysis

Most of the application-related requirement information is gathered offline, for instance, the speedup-resource curves (average speedup for a given number of cores) and workload classes with PDFs of different parameters as discussed in Section 5.2. At run-time, only the monitored information per application is tracked by the local power manager of each application. This information provides the history-related prediction part being used by the hybrid prediction of Fig. 7, which is maintained in a moving average fashion. As shown in Fig. 7, total of seven different parameters are tracked per Expand or Shrink operation. Typically, an application has less than 10 Expand/Shrink operations, which are aligned to the parallel and sequential section boundaries. The memory overhead ranges from $5 * 7 = 35$ to $10 * 7$

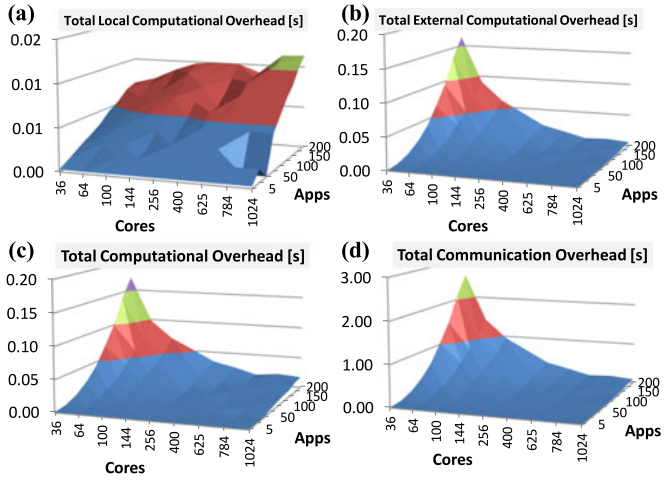

Fig. 19. (a) Local computation overhead, (b) External computation overhead, (c) Total computation overhead, (d) Total communication overhead.

= 70 integers, i.e., (140 – 280 bytes per application only) and computation overhead is only a couple of tens of arithmetic operations per Expand/Shrink operation. Therefore, it incurs a negligible energy overhead (approximately 220 nJ per Expand/Shrink operation) compared to the overall energy consumption and energy savings. Since the Expand/Shrink operations occur at long time intervals (in terms of tens of milliseconds), the performance overhead of this prediction part is also negligible compared to the performance savings and these time intervals. The actual information "how many threads/cores each application is using" is basically the result of a decision of local resource managers and does not need to be transferred; rather, it is stored locally per manager. This requires 16-bits per application.

Every application has its own local power manager, and makes its independent decisions at the Shrink operations. Therefore, the frequency of these power management operations is low, for instance, 2-4 times per complete iteration of an application (e.g., in one frame encoding). A local power manager only maintains the state information per application, and not for the complete system. Only the ED^2P benefit/loss information of different negotiating applications is shared among each other. The results include the overhead in terms of energy and delay in case of SEAD and Our New Policy. However, for the state-of-the-art, we purposely do not include their overhead because those centralized schemes take significantly more time in decision making and gathering system-wide information (e.g., MaxBIPS is exhaustive), and the performance numbers for a large number of cores will primarily be dominated by their overhead as they are limited to small-sized systems. Studies in [10] show that the communication overhead of the distributed decision making policy is typically in the range of 12 percent of that of a centralized decision making policy for a 1,024 core system with a total workload of 32 multi-threaded applications.

Detailed overhead discussion: The pseudo-code of Fig. 9 consists of computation logic (steps: 1, 2, 4, 7, 8 in the figure) and the logic which involves communication (steps: 3, 5 and 6). Fig. 19 shows the computation overhead that consists of two components: (1) issuing requests and making

decisions (i.e., the active part); and (2) replying to the incoming requests (i.e., the passive part). The active part involves relatively more processing, but is executed rarely; see Fig. 19(a). The passive part is simple, but is executed more often; see Fig. 19(b). The communication overhead, shown in Fig. 19(d), is added to the active part as it influences the speed of making the decision. The communication overhead is estimated using number of requests * number of packets * overhead per packet. Fig. 19 (c, d) shows that the *total accumulated worst-case overhead* (i.e., the case when cores are too few and number of application is too high, considering some applications will get cores once they are free from other applications) is less than 3.2 seconds for the complete application execution run. As the energy and delay results in the Table SIV.1 in *Supplementary Material*, available online, an application execution run is about 70 seconds (for 36 cores, including the overhead of 3.2 seconds) for our approach, while baseline is 80 seconds. This represents that the total accumulated worst-case overhead is about 4.5 percent of the total execution time. Even including the overhead, we have 12.5 percent performance improvement over the baseline case, for the worst-case scenario. In most of the cases, the accumulated overhead is below 0.5 seconds which is only 0.7 percent of the total execution time, and it represents only 5 percent of the total savings.

Regarding the energy consumption, we have implemented a simple NoC based on x-y routing with a 4-stage pipelined router implementation. The flit router energy $= 1.56 \times 10^{-10}$ J and the flit wire energy $= 3.88 \times 10^{-11}$ J. The total accumulated worst-case communication energy overhead is insignificant (below 0.1 percent of the total energy savings compared to baseline, see Table SIV.1 in *Supplementary Material, available online*). The power state transition overhead is shown in Table 3, which is the range of 1.32 to 25.7 μJ depending upon the state transition. However, these state transitions are only at the Shrink operations, which occur at intervals given in multiple of milliseconds. In summary, the overhead of the proposed approach is negligible compared to the savings.

8 Supplementary Material

The supplementary material, which can be found on the Computer Society Digital Library at http://www.ieeecomputersociety.org/10.1109/TC.2016.2540631, provides the following technical details and results: (SI) Detailed results on scalability and optimization space; (SII) A detailed run-time scenario for an experiment; (SIII) Discussion on conflicts and stability; (SIV) Detailed energy-delay results; (SV) ED^{2}P results for application mixes; (SVI) Detailed discussion on offline training, and (SVII) Discussion on implementation issues.

9 Conclusion

We presented a scalable dynamic power management scheme that introduces local power managers for each malleable application, that manage their cores and the corresponding power states autonomously. Our DPM allows for continuous "trading" between different local power managers for proactive resource reservation and temporary core lending such that each application optimizes for its energy efficiency while tending towards maximizing the overall system efficiency. Application-level knowledge for improved prediction of upcoming resource requirements and corresponding time periods is exploited. The energy efficiency is up to 58 percent reduced ED^{2}P and there is scalability w.r.t. number of cores and application threads compared to three other state-of-the-art DPM policies. As a result, the autonomous per-application power management provides a basis for scalable power management of future manycore on-chip systems.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89); http://invasic.de [31]. We would also like to thank Prof. Siddharth Garg from NYU, USA for his valuable discussions.

References

[1] ITRS-International Technology Roadmap for Semiconductors. (2011). System drivers [Online]. Available : http://www.itrs.net
[2] Tilera Corporation. (2013). Tile-GX processor family [Online]. Available: www.tilera.com
[3] Nvidia Tesla. (2013). Tesla processor family [Online]. Available: www.nvidia.com
[4] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *Proc. IEEE 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 449–460.
[5] M. Shafique, B. Vogel, and J. Henkel, "Self-adaptive hybrid dynamic power management scheme for many-core systems," in *Proc. IEEE Des., Autom. Test Eur. Conf. Exhib.*, 2013, pp. 51–56.
[6] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 347–358.
[7] K. Meng, R Joseph, R. P. Dick, and L. Shang, "Multi-optimization power management for chip multiprocessors," in *Proc. IEEE 17th Int. Conf. Parallel Archit. Compilation Tech.*, 2008, pp. 177–186.
[8] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *Proc. IEEE 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 314–324.
[9] R. McGowen, C. Poirier, C. Bostak, and S. Naffziger, "Power and temperature control on a 90-nm Itanium family processor," *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 229–237, Jan. 2006.
[10] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel, "DistRM: Distributed resource management for on-chip many-core systems," in *Proc. CODES+ISSS*, 2011, pp. 119–128.
[11] H. Shojaei, A.-H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes, "A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management," in *Proc. 46th ACM/IEEE Des. Autom. Conf.*, 2009, pp. 917–922.
[12] G. Sabin, M. Lang, and P. Sadayappan, "Moldable parallel job scheduling using job efficiency: An iterative approach," in *Proc. 12th Int. Workshop Job Scheduling Strategies Parallel Process.*, 2007, vol. 4376, pp. 94–114.
[13] J. Reinders, *Intel Threading Building Blocks*. Sebastopol, CA, USA: O'Reilly & Associates, 2007.
[14] J. Henkel, et al., "Invasive Manycore Architectures," in *Proc. 17th Asia South Pacific Des. Autom. Conf.*, 2012, pp. 193–200.
[15] L. Mureau, and C. Queinnec, "Resource aware programming," *ACM Trans. Program. Languages Syst.*, vol. 27, no. 3, pp. 441–476, May 2005.
[16] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram, "A case for guarded power gating for multi-core processors," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 291–300.
[17] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proc. IEEE 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 290–301.

[18] A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das, "CPM for CMPs: Coordinated power management for chip-multiprocessors," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2010, pp. 1–12.
[19] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. IEEE 50th ACM/EDAC/IEEE Des. Autom. Conf.*, 2013, pp. 1–9.
[20] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton Univ., Princeton, NJ, 2011.
[21] George, et al., "PENRYN: 45-nm next generation Intel Core 2 Processor," in *Proc. IEEE Asian Solid-State Circuits Conf.*, Nov. 2007, pp. 14–17.
[22] 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. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 469–480.
[24] H. Singh, K. Agarwal, D. Sylvester, and K. J. Nowka, "Enhanced leakage reduction techniques using intermediate strength power gating," *IEEE Trans. VLSI*, vol. 15, no. 11, pp. 1215–1224, Nov. 2007.
[25] M. A. Johnson, and M. H. Moradi, *PID Control: New Identification and Design Methods*. Berlin, Germany: Springer, 2005.
[26] M. Shafique, B. Zatt, F. L. Walter, S. Bampi, and J. Henkel, "Adaptive power management of on-chip video memory for multiview video coding," in *Proc. 49th ACM/EDAC/IEEE Des. Autom. Conf.*, 2012, pp. 866–875.
[27] B. Yang, L. Guang, T.C. Xu, A.W. Yin, T. Säntti, and J. Plosila, "Multi-application multistep mapping method for many-core network-on-chips," *NORCHIP*, 2010, pp. 1–6.
[28] X. Liu, P. J. Shenoy, and M. D. Corner, "Chameleon: Application-level power management," *IEEE Trans. Mobile Comput.*, vol. 7, no. 8, pp. 995–1010, Aug. 2008.
[29] S. Roy, N. Ranganathan, and S. Katkoori, "State-retentive power gating of register files in multi-core processors featuring multi-threaded in-order cores," *IEEE Trans. Comput.*, vol. 60, no. 11, pp. 1547–1560, Nov. 2010.
[30] H. Javaid, M. Shafique, J. Henkel, and S. Parameswaren, "System-level application-aware dynamic power management in adaptive pipelined MPSoCs for multimedia," in *Proc. IEEE Int. Conf. Computer-Aided Design* , 2011, pp. 616–623.
[31] 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," *17th Asia and South Pacific Design Autom. Conf. (ASP-DAC)*, pp. 193–200, 2012.

Muhammad Shafique received the PhD degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in 2011. He is currently a research group leader at the Chair for Embedded Systems, KIT. He has over ten years of research and development experience in power-/performance-efficient embedded systems in leading industrial and research organizations. He holds one U.S. patent. His current research interests include design and architectures for embedded systems with focus on low power and reliability. He received 2015 ACM/SIGDA Outstanding New Faculty Award, six gold medals, the CODES+ISSS 2015, 2014 and 2011 Best Paper Awards, AHS 2011 Best Paper Award, DATE 2008 Best Paper Award, DAC 2014 Designer Track Poster Award, ICCAD 2010 Best Paper Nomination, several HiPEAC Paper Awards, Best Master Thesis Award, and SS'14 Best Lecturer Award. He is the TPC co-Chair of ESTIMedia 2015 and 2016, and has served on the TPC of several IEEE/ACM conferences like ICCAD, DATE, CASES, and ASPDAC.

Anton Ivanov received his Dipl-Ing degree in computer science from Chita State University in 2011. He was a researcher at CES, KIT, Germany from 2011-2014. His main research interests are in many-core systems.

Benjamin Vogel received his Diplom in computer science from KIT, Germany. He was a researcher at CES, KIT, Germany from 2010-2013. His main research interests are in many-core systems and low-power design.

Jörg Henkel received the PhD degree from Braunschweig University with "Summa cum Laude". He is with the Karlsruhe Institute of Technology (KIT), Germany, where he is heading the Chair for Embedded Systems CES. Before, he was a senior research staff member at NEC Laboratories in Princeton, NJ for seven years. He has chaired major conferences in design automation for embedded systems as general chair and program chair and is member of various steering committees. He is an editorial board member of various journals like the *IEEE TVLSI*, *IEEE TCAD*, *IEEE TCPS*, *IEEE TMSCS*, *JOLPE*, etc. He received the 2008 DATE Best Paper Award, the 2009 IEEE/ACM William J. McCalla ICCAD Best Paper Award, the Codes+ISSS 2015, 2014 and 2011 Best Paper Awards, and the MaXentric Technologies AHS 2011 Best Paper Award. He is the chairman of the IEEE Computer Society, Germany Section, and has been the Editor-in-Chief of the ACM Transactions on Embedded Computing Systems (ACM TECS) for two periods. He is an initiator and the coordinator of the German Research Foundation's (DFG) program on 'Dependable Embedded Systems' (SPP 1500). He is the site coordinator (Karlsruhe site) of the Three-University Collaborative Research Center on "Invasive Computing" (DFG TR89). Since 2012, he is an elected board member of the German Research Foundation's (DFG) board on "Computer Architecture and Embedded Systems". He holds ten US patents and is a Fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.