Goal-Driven Autonomy for Efficient On-chip Resource Management: Transforming Objectives to Goals

Elham Shamsa[1], Anil Kanduri[1], Amir M. Rahmani[2,3], Pasi Liljeberg[1], Axel Jantsch[3], and Nikil Dutt[2]

[1]*Department of Future Technologies, University of Turku, Turku, Finland*
[2]*Department of Computer Science, University of California, Irvine, USA*
[3]*Institute of Computer Technology, TU Wien, Vienna, Austria*
{elsham, spakan, pakrli}@utu.fi, {a.rahmani, dutt}@uci.edu, axel.jantsch@tuwien.ac.at

***Abstract*—Run-time resource allocation of heterogeneous multi-core systems is challenging with varying workloads and limited power and energy budgets. User interaction within these systems changes the performance requirements, often conflicting with concurrent applications' objective and system constraints. Current resource allocation approaches focus on optimizing fixed objective, ignoring the variation in system and applications' objective at run-time. For an efficient resource allocation, the system has to operate autonomously by formulating a hierarchy of goals. We present goal-driven autonomy (GDA) for on-chip resource allocation decisions, which allows systems to generate and prioritize goals in response to the workload and system dynamic variation. We implemented a proof-of-concept resource management framework that integrates the proposed goal management control to meet power, performance and user requirements simultaneously. Experimental results on an Exynos platform containing ARM's big.LITTLE-based heterogeneous multi-processor (HMP) show the effectiveness of GDA in efficient resource allocation in comparison with existing fixed objective policies.**

***Index Terms*—Goal-Driven Autonomy, Autonomous and Self-Aware Systems, On-chip Resource allocation, Heterogeneous Multi-core Systems**

I. Introduction

Mobile heterogeneous multi-core processors (HMPs) require intelligent run-time strategies in order to match applications' requirements with hardware capabilities to maximize resource efficiency [1]. Adapting to varying concurrent workload characteristics and user requirements under stringent power, energy, and thermal budgets makes run-time resource management more challenging [2]. Considering *user*, *application*, and *processor* as three major levels of abstraction, each layer presents a diverse set of demands [3]. For example, these may include (but not limited to) quality of experience (QoE) at user-level, minimum performance guarantees at application-level, and honoring thermal safety at processor-level. Demands expressed at each layer can be transformed into quantifiable entities, which we refer to as *constraints*. Resource management policies form *objective* which could be expressed as a linear cost function that fits one or more constraints. Single-objective indicates having a cost function with one constraint, however, fixed objective refers to a cost function with one or more fixed constraints.

State-of-the-art run-time management approaches have used dynamic policies for application mapping [4], thread-to-core binding [5], task migration [2], power budget allocation [1], power density and thermal management. Each policy typically has fixed objective such as maximizing performance, minimizing power consumption, honoring power budgets, ensuring thermal safety etc. These policies largely focus on satisfying the fixed objective, leaving the possibility of being agnostic to other objectives. Some of the policies however have considered multiple objectives simultaneously, deliberately making known compromises. Unpredictability in user activity pattern and workload intensity affects the power, energy, and thermal budget available. This changes the constraints at each layer which could be formed into a new objective, while objectives from different layers could be overlapping, conflicting or orthogonal. Fixed objective resource management policies, both single and multi-objective, are oblivious to such scenarios. This restricts their resource efficiency to only a fixed set of workload conditions and system state.

A robust and efficient way for adapting to varying constraints and objectives is to abstract away such behavior into hierarchical entities that can embed multiple objectives and their significance. We refer to the abstracted information as *goals*, which would drive the resource allocation decisions. Goal can be formulated as a weighted combination of different objectives. Multiple conflicting objectives can be unified through goal formulation, forming a hierarchy of goals with different priorities [6]. Goal driven resource allocation thus identifies the importance of different objectives and makes suitable decisions. As workload characteristics and system dynamics vary, goals and their priorities also can vary [7]. Resource allocation decisions can be automatically adapted to fulfill the hierarchy of goals by arbitrating among varying priorities of goals.

With this motivation, we propose goal-driven autonomy (GDA) for on-chip resource allocation and management. Our approach is inspired by the rich literature on GDA in the fields of artificial intelligent and robotics [8], [9]. We design a hierarchical goal manager which considers user, application, and system's objectives, dynamically formulates goals by identifying the importance of each of the objectives. The goal manager arbitrates among different goals to classify them as per their urgency and enforce resource allocation policy that is suitable to meet the current goal hierarchy. Resource allocation decisions made are evaluated at run-time by assigning weighted rewards to quantify the efficiency of a specific policy, which will guide the subsequent decisions. Our contributions to this end are:

- A resource management framework with a hierarchical goal manager that abstracts distinct user, application, and system's demands into *goals*.
- A dynamic goal formulation in lieu of multiple conflicting

978-3-9819263-2-3/DATE19/©2019 EDAA

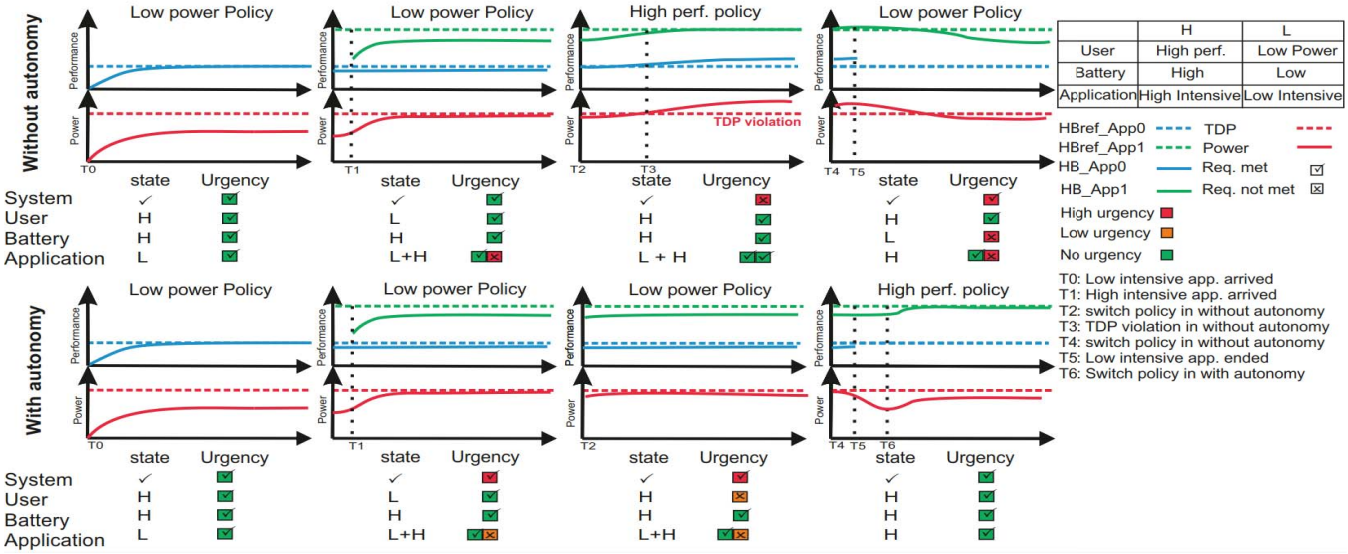

Fig. 1: Motivation example demonstrating goal driven autonomy

user, application, and system demands and objectives

- An on-line learning for evaluating and prioritizing among goals to provide goal driven autonomy for efficient resource allocation.
- A middleware to enforce resource allocation decisions guided by the goal manager and an evaluation on a real heterogeneous hardware testbed of Exynos-based Odroid XU3 over varying workload conditions.

We provide background and motivation for using GDA based resource allocation in Section II. Our proposed approach is described in Section III. We evaluate resource allocations strategies with and without GDA over micro-kernels on real hardware testbed, as presented in Section IV. Section V summarizes relevant related works and Section VI concludes with possible future works.

II. Background and Motivation

State-of-the-art run-time management policies focus on fixed objectives and there has been no prior work in applying goal-driven autonomy for computing systems resource management. A naive policy switching might satisfy more than one objective, where each policy is more suitable for specific objectives. However, the reactive nature of such decisions only leads to an oscillation among different policies for different target objectives [3]. The system must monitor its environment during run-time to formulate goals and autonomously prioritize among different goals that satisfies the most important objective at any given instance [7]. We demonstrate the effect of autonomous goal formulation in the multi-objective resource allocation through an example in Figure 1. The figure illustrates the power consumption and corresponding performance for two applications in two different scenarios - with and without autonomy. We consider four entities viz., system, user, battery, and application in decreasing order when evaluating their priorities during resource allocation. Each entity has a *state* i.e., current scenario in which they are operating, and *urgency* i.e., the extent of their priority. For the system, we use two states indicating operating under power budgets (shown in ✓) and violating power budget (shown in X). The user's state are confined to two commands - high performance (shown in H) and power saving (shown in L). The battery's states are high-level and low-level while the applications' states show high and low performance requirements. At *T0*, a low intensive application, *App0*, arrives while the user command is for high performance. In this scenario, a policy that suits less intensive workloads is selected. This policy is modeled similar to the approach presented in [2]. Since the application is less intensive, the user command for high performance is satisfied within the power budget, using the low-power policy itself. At *T1*, a high intensive application *App1* arrives, resulting in an increase in the power consumption. While the power consumption is below TDP, performance requirements of *App1* are not met with the low-power policy. The system without autonomy identifies the performance requirement (shown in red under urgency) and assigns the highest priority to application requirements. The resource allocation policy is thus switched to high performance at *T2* to satisfy *App1*'s performance requirements. The high performance policy is modeled based on the strategy presented in [5]. Although the application requirements are now met, a TDP violation occurs at *T3* with the higher performance, as shown under urgency. At *T4*, the system becomes the most prioritized entity and a decision to switch the policy back to low power mode is taken. Consequently, at *T5*, the power consumption is lowered below the TDP, although the performance of *App1* is not met, which is again prioritized. Such switching between low-power and high-performance policies is relatively adaptable to varying workload conditions, however with constant oscillations between the two policies. For the similar scenario, a more autonomous approach would be able to differentiate between priorities of different entities while making policy switching decisions. In the example presented with autonomy, the system is prioritized already at *T1*, since the power consumption is closer to the TDP with a likeliness of potential violation. This is shown in red tick under the urgency column, since power has not yet been violated. At the same time, the performance requirement

of *App1* is not met. In this case, the autonomous resource management approach still prioritizes the system (shown in red under urgency) over the application (shown in orange under urgency) i.e., a *potential* system violation has higher priority than an *instantaneous* performance violation. Based on the priorities identified, execution continues with the low-power policy which suits the required low-power operation. In the same scenario, conventional resource management approaches would not switch between any policies being oblivious to changing workload scenarios, while a naive goal management system would reactively switch between policies. At *T4*, *App0* leaves the system, leaving enough power headroom. The system's state and urgency are updated accordingly. With the system no longer being in urgency and the performance of *App1* not met, application becomes the priority. As a result, at *T6*, the manager switches the policy to high-performance in order to meet the performance requirements of *App1*. As such, goal driven autonomy for resource allocation can improve the overall efficiency, making justifiable compromises when needed.

III. GDA for On-chip Resource Management

In this section, we present our proposed approach and framework for GDA based on-chip resource allocation. Cognitively, humans pursue number of concurrent goals - spawning, terminating, (re-)prioritizing them dynamically by arranging them in hierarchical structures [10]. Researchers have studied the origin of goals and cognitive architectures to handle the goals of artificial agents. GDA is a goal reasoning model which allows agents to dynamically generate their goals in response to environmental changes [7].

Our GDA conceptual model has a controller which interacts with the execution environment. The controller is responsible for monitoring the state of each entity viz., system, user, and application, and make appropriate resource allocation decisions. The controller includes a state detector, priority re-assignor, and goal enforcer. The state detector monitors critical system parameters i.e., power consumption and performance metrics from the execution environment in order to determine the state. The priority re-assigner generates one or more goals in response to the new state. The goal enforcer enforces resource allocation decisions to meet the new goal. We designed the GDA model for HMP architectures due to the higher complexity of resources and knobs available in such platforms. Figure 2 illustrates a hierarchical view of the proposed GDA based resource management framework. The details of each component within this framework are described in the following.

A. State Detector

As mentioned in the previous sections, we consider system, user, and application as the major entities that affect resource management. To dynamically scale the resources proportional to each of these entities, we monitor their state at run-time. We model the state as a vector of *Power*, *User Cmd*, and *Performance*, representing the system, user and applications' requirements, respectively. Within the state vector, *Power* represents the instantaneous power consumption (P) of the chip in comparison with a fixed upper bound on power (TDP).

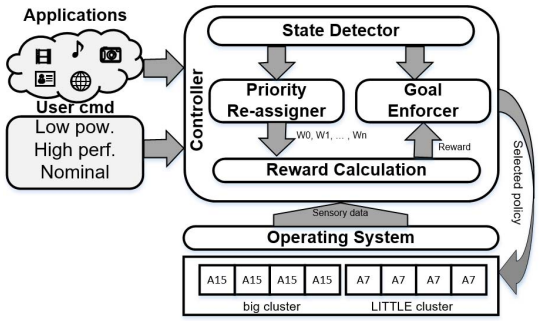

Fig. 2: Goal management system architecture

The three possible states are violation ($P > TDP$), potential violation ($TDP > P > 0.8 \times TDP$), and no violation ($P < 0.8 \times TDP$). The *User Cmd* represents interactive input request made by the user, which includes high-performance and power-saving. Such modes are often available in battery-powered systems such as smartphones. The *Performance* represents the extent of performance requirements being met for all the currently running applications. Each application is modeled to have a specific range of performance requirements. We monitor the performance of each application at run-time and compare it against the pre-defined requirements to determine performance violation. Thus, the state detector identifies the state of each of these entities, representing power, performance, and user requirements under the current workload scenario.

B. Priority Re-assigner

We define a hierarchy of primary, secondary, and tertiary goals with decreasing order of priorities, as shown in Figure 3. In our proposed approach, we map system, user, and application entities to primary, secondary, and tertiary goals, respectively. This emphasizes on the importance of thermal safety, followed by user satisfaction and applications performance. Our generic model for goal hierarchy can support several levels of goals and sub-goals. We define urgency as the extent of the violation of a measured parameter, i.e., power consumption and performance. The urgency is calculated as shown in Equation (1)

$$U_{Pow} = \frac{P_{curr}}{P_{ref}}, U_{Perf} = \frac{perf_{max} - perf_{curr}}{perf_{max} - perf_{ref}} \quad (1)$$

where U_{pow} and U_{Perf} are the urgency of power and performance, P_{curr} is the instantaneous power consumption, and P_{ref} is the fixed upper bound on power (TDP). When the power consumption exceeds TDP ($P_{curr} > P_{ref}$), the urgency $U_{pow} > 1$, indicating a high urgency. $perf_{curr}$ is the measured performance, $perf_{max}$ is the maximum required performance, and when *user Cmd* is *high-performance* $perf_{ref}$ is:

$$Perf_{ref} = \frac{perf_{max} + perf_{min}}{2} \quad (2)$$

and when *user command* is *power saving*:

$$Perf_{ref} = perf_{min} \quad (3)$$

We use Application heartbeats API [11] for measuring performance of each application. Applications periodically issue heartbeats where the number of heartbeats per epoch indicates the application performance. If the heartbeats are within the specified minimum and maximum requirement the urgency of

performance is low. Similar to the power urgency, $U_{Perf} > 1$ when performance requirements are not met. An urgency which is higher than 1 indicates a high priority objective, while lower than 1 lowers the priority. The priority reassignment receives the state vector from the state detector and calculates the corresponding urgencies. These urgencies will be potentially used to determine the priorities of the next set of goal(s). We arbitrate and re-assign priorities of multiple goals in a weighted manner considering the goal hierarchy. Within the same level of hierarchy, the goal with the highest urgency will be prioritized among multiple goals. Among goals belonging to different levels, the urgency of the goal is weighted with the level of hierarchy to determine the eventual priority. Finally, the re-assigned priorities are evaluated to determine the new goal(s).

C. Goal Enforcer

Once the new set of goals are determined at the priority re-assignment stage, we enforce the goals through choosing appropriate *action*. In our framework, an action corresponds to invocation of a specific resource allocation policy. From the available set of actions, the goal enforcer module selects the best action that is more likely to satisfy the highest priority goal. In the initial case at time *t* = 0s where the system starts execution for the first time, a random action is selected. After this initial selection, all the subsequent enforcement decisions are based on *reward function* that is calculated by learning from the system dynamics from the previous cycles. The enforcer selects the action with the highest reward. Details on reward function calculation are explained as follows.

D. Reward Calculator

The Reward Calculator module estimates the efficacy of a chosen action for a target goal using a reward function. The reward function is dynamically changed based on the priority that is assigned to each goal by the Priority Re-assigner. The goals with higher priority have higher weights in the reward function. The reward function is expressed in Equation (4).

$$Reward = W_0 \times R_0 + W_1 \times R_1 + W_2 \times R_2 + ... + W_n \times R_n \quad (4)$$

where W_i is the weight of i_{th} goal as determined by the Priority Re-assigner and R_i is the reward that calculated for each goal after choosing an action (policy execution). The rewards for power and performance objectives in our framework are expressed as:

$$R_{Power} = \frac{P_{ref} - P_{curr}}{P_{ref}} \quad (5)$$

where R_{Power} is the reward for power goal.

$$R_{Perf} = \frac{\sum_{i=1}^{n} \frac{Perf_i - Perf_{min}}{Perf_{max} - Perf_{min}}}{n} \quad (6)$$

where R_{Perf} is the reward for performance goal, $Perf_i$ is the measured performance of the i_{th} application and n is the total number of applications running. For instance, considering two different types of objectives which lead to two goals viz., power and performance, the reward function is:

$$Reward = W_{Power} \times R_{Power} + W_{Perf} \times R_{Perf} \quad (7)$$

In Equation (7), R_{Power} and R_{Perf} can be calculated by (5) and (6) and W_{Power} and W_{Perf} are determined by Priority Re-assigner module. A relatively higher reward indicates fulfilling the target goals, such that eventually multiple objectives are satisfied. Over the run-time, the outcome of the reward keeps varying with varying workload conditions and user requests. However, higher overall reward represents a higher percentage of prioritized objectives (goals) being met. The reward for each action under a specific state vector is saved into a look up table. After every set of goal enforcer's decisions, the reward estimates are updated.

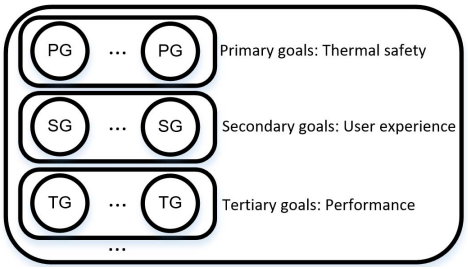

Fig. 3: Goal hierarchy.

IV. Experimental Evaluation

A. Experimental Setup

We perform our evaluations on the Hardkernel Odroid XU3 board [12], which contains an HMP with two clusters (4 big (A15) and 4 little (A7) CPU cores). The big cores provide high-performance, while the LITTLE cores are power efficient. Memory is shared across all cores. We implement the GDA as a part of Mars framework [13]. A Linux user-space daemon process invokes the GDA every parameterizable epoch. In our experiments, we set the epoch to 1s. For thermal safety, we set the power reference to 4W. We use the on-board power sensors of Odroid XU-3 to monitor per-cluster power consumption. We use the Application Heartbeats API [11] to monitor each application's performance. The applications periodically issue heartbeats and inform the system about their performance. To evaluate our proposed approach, we use a set of synthetic micro-benchmarks with attributes that reflect interactive and I/O dependent nature of applications [4] [14]. These micro-benchmarks have configurable active and idle periods to emulate a wide range of workload distribution patterns.

B. Experimental Results

For the evaluation purpose, we used 7 applications form the micro-kernel benchmarks, with a mix of high and low intensive workloads. We compare our proposed solution with two policies which are adapted to our framework based on state-of-the-art approaches [2], [5] as the *low-power* and *high-performance* policies. These approaches have fixed objectives of minimizing power consumption and maximizing performance within power caps, respectively. For power and performance decisions, we used per-cluster DVFS and task migration. The sequence of incoming applications follows the trend of high intensity at the beginning, followed by low intensity - which is kept constant for all the approaches under evaluation. Table I lists the objective, consideration of user commands, power and performance violation, and average power consumption (in W) with each approach. The low power policy (LP) is efficient in honoring power budgets at the expense of performance degradation, while the high performance policy (HP) meets

TABLE I: Comparison of proposed solution with existing approaches

Tech.	Obj	Cmd	Pwr viol.	Perf. viol.	Avg. pwr
LP policy	Power	X	3%	65%	2.99
HP policy	Perf.	X	67%	0%	3.8
GDA	Dynamic	✓	20%	34%	3.2

the performance requirements always, compromising thermal safety in a form of over-boosting. Each of these policies cater their fixed objectives better, yet they are oblivious to other objectives with changing workload scenarios. Further, they do not consider user commands, limiting their efficiency to fixed conditions. The colored cells show the efficiency of our GDA approach in managing conflicts when the goals are competing (e.g., HP policy results in a significant power violations when the system is highly loaded). The GDA approach balances both the power and performance objectives, satisfying a higher percentage of goals.

Figure 4 presents the power consumption of the system using two fixed objective policies and the proposed GDA approach. At the beginning, workload intensity, application arrival rate, and user's performance requests are high. The high performance policy aggressively scales the resources and violates power budget, while the low power policy operates within the power budgets. Under such intense workload scenarios, GDA approach violates power budget to an extent. This is due to the fact that both applications' and user command's urgency outweigh the urgency of the system within the goal hierarchy. At t = 20s, the user command is changed back to power saving, prompting a change in urgencies. The GDA approach identifies this change and responds by switching the policy to low power (LP). This is reflected in reduced power consumption at around t = 22s. High intensive workloads leave the system starting from t = 30s. A steep reduction in power consumption can be seen with each approach. The user command is updated back to high performance at t = 60s. However, the workload intensity at this phase is relatively low in a way that the high performance request can be already met with the low power policy itself. This shows that GDA can also intelligently detect such scenarios and avoid unnecessary energy dissipation. In this case, if we use naive policy switching, it identifies the user command change and responds by switching the policy to high performance which leads to more power consumption. However, the GDA controller monitors the state vector, evaluates the urgencies of different goals and aptly chooses to remain with the low power policy.

Performance of each individual application over the execution time using the three different strategies is shown in Figure 5. In all sub-graphs, the x-axis shows time (s) and the y-axis shows the measured heartbeats per 100 ms. The red dotted line is the performance requirement for each application, calculated as the average of maximum and minimum requirements specified. With *App1* and *App2* being high intensive, their performance requirements are thoroughly met with the high performance policy, whereas the low power policy violates these requirements for longer periods. GDA approach delivers the required performance during the phases where user command and applications' dominate the urgency.

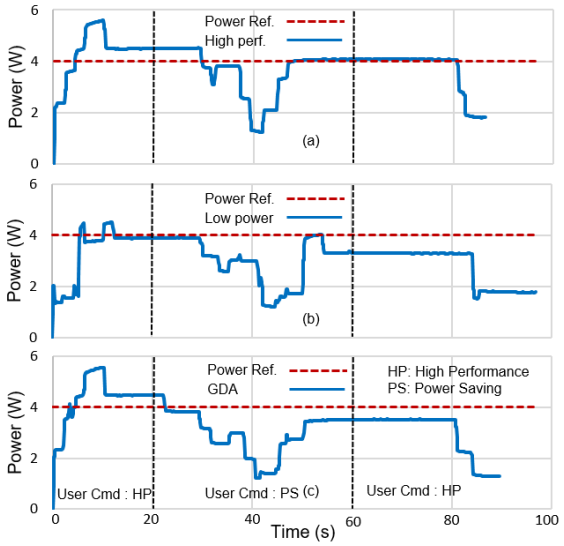

Fig. 4: Power consumption measures. (a) high performance policy, (b) low power policy, (c) GDA.

At t = 20s, as the user command is no longer requesting high performance, the applications' urgency is reduced. Consequently, the GDA prioritized system goal of low power operation, which is thus reflected in performance violation for a shorter duration, as shown in Figure 5 (c). The performance for *App3*, *App4*, *App5*, and *App7* are similar for major periods of execution with all three approaches. Since they are of lower intensity, each approach is able to provide enough resources within the power budgets to meet both power and performance criteria. Specifically, the low power policy is much more efficient than the high performance policy with its suitability to this workload scenario. Performance of *App6* is also met with each of these strategies - despite being high intensive, *App6* executes largely in combination with other low intensive applications. The GDA uses the benefits of these two policies opportunistically to satisfy both power and performance requirements during the run-time. The GDA i) generates new goals based on the requirements, ii) switches between different goals as per justifiable priorities, and iii) to satisfy the goal with the highest priority. The average power consumption with GDA is 3.2W, while the low power policy and high performance policies stay on either side of GDA at 2.9W and 3.8W, respectively. Similarly, the overall execution time of the set of applications with GDA is 89s, whereas low power policy and high performance policies finish execution within 96s and 86s respectively.

V. Related Work

Resource Management Several works on run-time resource management targeted meeting performance requirements while optimizing power consumption [1], [2], [4], [5], [14]. The HPM scheduler presented in [2] deployed several proportional-integral-derivative (PID) controllers for resource sharing, DVFS and task migration decisions. The primary objective of each of these controllers is to minimize power consumption with graceful degradation of performance. A resource management framework for many-core systems to maximize performance under power caps is proposed in [5].

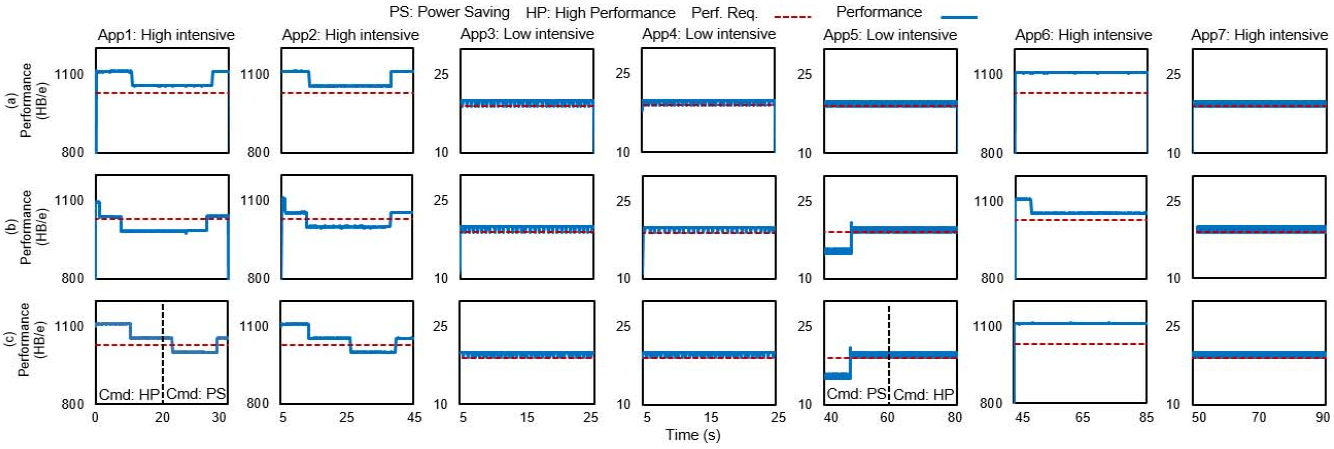

Fig. 5: Performance of running applications in order of arriving. (a) high performance policy, (b) low power policy, (c) GDA.

The authors use software approximation as another knob to address performance degradation in power actuation. A similar idea using approximation for run-time management of heterogeneous architectures is presented in [1]. The authors use a coordinated actuation of approximation, DVFS, CPU quota scaling, and task migration with the objective of maximizing performance within minimal power consumption. Machine learning methods have recently applied for resource management. A reinforcement learning-based approach is proposed in [15]. The authors using online reinforcement learning to achieve energy saving. Another learning-based approach in [16] is proposed a modular Q-learning based DVFS control to optimize the CPU frequency for multi-core processors.

To summarize, existing resource allocation approaches are confined to fixed objectives whereas QoS requirements and workload conditions in which they operate are variable.

Goal Driven Autonomy GDA is a conceptual model for online planning in autonomous systems. As presented in [7], GDA starts with an initial goal, generates necessary action to achieve the current goal by observing and interacting with the environment. New goals are generated when there is any discrepancy between the expected state and the current state using a goal formulator. While GDA has not previously used for on-chip resource management, there are several investigations on GDA and goal management [8], [9], [17]. The authors of [10] use a reactive goal management procedure to create and prioritize new goals in long-term goal memory. LGDA (Learning GDA) in [8] learns its goal selection function using Q-learning. Further, GDA has been used for controlling unexpected events in gaming scenarios [8]–[10].

VI. Conclusions

We proposed the idea of abstracting multiple conflicting objectives at run-time into *goals* that reflect overall system dynamics including power consumption, performance and user experience. We presented a resource management framework which embeds a hierarchical goal manager - to identify system dynamics, formulate goals and prioritize among different goals. We implemented a control strategy for resource allocation decisions that are driven by the autonomous goal manager. We evaluated the proposed GDA approach on a heterogeneous mutli-core platform against state-of-the-art resource management policies without autonomy. GDA provides a higher success rate in meeting different objectives by making reasonable trade-offs wherever necessary. Fine grained multiple sub-goal arbitration and reflective autonomy for resource management are planned for future works.

Acknowledgment

We acknowledge financial support by the Marie Curie Actions of the European Union's H2020 Program, NSF Information Processing Factory grant and Academy of Finland project ACTER.

References

[1] A. Kanduri *et al.*, "Approximation-aware Coordinated Power/performance Management for Heterogeneous Multi-cores," in *Proc. of DAC*, 2018.
[2] T. S. Muthukaruppan *et al.*, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. of DAC*, 2013.
[3] A. Rahmani *et al.*, "HDGM: Hierarchical dynamic goal management for many-core resource allocation," *IEEE Embedded Systems letters*, 2017.
[4] B. Donyanavard *et al.*, "Sparta: Runtime task allocation for energy efficient heterogeneous manycores," in *Proce. of CODES+ISSS*, 2016.
[5] A. Kanduri *et al.*, "Approximation knob: Power capping meets energy efficiency," in *Proc. of ICCAD*, 2016.
[6] E. Shamsa *et al.*, "Goal Formulation: Abstracting Dynamic Objectives for Efficient On-chip Resource Allocation," in *Proc. of NorCAS*, 2018.
[7] M. Klenk *et al.*, "Goal-Driven Autonomy For Responding To Unexpected Events In Strategy Simulations," *Computational Intelligence*, 2013.
[8] U. Jaidee *et al.*, "Integrated learning for goal-driven autonomy," in *Proc. Int. Joint Conference on Artificial Intelligence*, 2011.
[9] B. Weber *et al.*, "Learning from Demonstration for Goal-Driven Autonomy." in *AAAI*, 2012.
[10] D. Choi, "Reactive goal management in a cognitive architecture," *Cognitive Systems Research*, 2011.
[11] H. Hoffmann *et al.*, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," in *Proc. of conference on Autonomic computing*, 2010.
[12] Hardkernel. ODROID-XU. [Online]. Available: http://www.hardkernel.com/main/main.php
[13] T. Muck *et al.*, "Adaptive-Reflective Middleware for Power and Energy Management in Many-Core Heterogeneous Systems." in *Many Core Computing: Hardware and Software, IET*, 2019.
[14] A. Rahmani *et al.*, "SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management," in *Proc. of ASPLOS*, 2018.
[15] R. A. Shafik *et al.*, "Learning transfer-based adaptive energy minimization in embedded systems," 2016.
[16] Z. Wang *et al.*, "Modular reinforcement learning for self-adaptive energy efficiency optimization in multicore system," in *Proc. of ASP-DAC*, 2017.
[17] J. Powell *et al.*, "Active and interactive learning of goal selection knowledge," in *Proc. of AAAI*, 2011.