Machine Learning Techniques to Support Many-Core Resource Management: Challenges and Opportunities

Martin Rapp[1], Hussam Amrouch[1], Marilyn Wolf[2], Jörg Henkel[1]
[1]*Karlsruhe Institute of Technology,* [2]*University of Nebraska*
{martin.rapp, amrouch, henkel}@kit.edu, mwolf@unl.edu

***Abstract*—Resource management in many-core processors, housing tens to hundreds of cores on a single die, becomes more and more challenging due to the ever-increasing number of possible management decisions (e.g., task mapping). Machine learning (ML) techniques emerge as promising solutions to support resource management algorithms in taking the best decisions due to their adaptability. However, there are several challenges with ML-based solutions. We discuss two key challenges in detail. Firstly, ML-based techniques often suffer from high computational complexity for the inference at run-time – which is especially critical when it comes to the embedded system domain. Secondly, employing ML techniques as a "black box" may result in deriving models that fail in reflecting the reality.**

We take a task migration technique that maximizes the performance of a thermally-constrained many-core as a case study. This technique selects the migration to execute next with the support of a neural network (NN) that predicts the performance impact of a migration. We demonstrate the above-mentioned challenges in this case study and discuss potential remedies. To lower the run-time overhead, we discuss overhead-aware design of the NN and using already existing accelerators in smartphone SoCs. Finally we also demonstrate how existing domain knowledge can be introduced into the models to ensure that models are consistent with the reality and experimentally show the potential.

***Index Terms*—Resource Management, Machine Learning, Gray Box Modeling, Neural Network Design, Accelerator**

I. Introduction

The achievable performance and power-efficiency of a many-core processor strongly depend on the employed resource management technique [1]. This technique must make proper decisions for a multitude of functions like scheduling, mapping, dynamic voltage and frequency scaling (DVFS), etc. In the past years, the number of cores in modern processors has continuously increased and has reached hundreds of cores in a single many-core processor [2]. Consequently, the number of possible actions experiences a combinatorial explosion. For example, there are 10^{28} possible mappings of 16 threads on a 64-core many-core with a one-thread-per-core model. Furthermore, resource management (e.g., task mapping) is NP-complete in the general case [3]. A huge number of possible actions together with NP-completeness renders it impractical to always select the optimal actions at run-time leading to a plethora of heuristic resource management techniques [4].

978-1-7281-5758-0/19/$31.00 ©2019 IEEE

These heuristics mostly rely on design-time models, e.g., a model of the cooling system [5]. Such design-time models come with several limitations. First, these models are built at design-time and therefore, cannot react to run-time changes or operational variations. Hence, run-time adaptiveness is very difficult to achieve. For example, if an RC-thermal model [6] predicts too high temperatures, this can be due to several parameters being off: the floorplan, thermal conductivity of modeled materials, or even the ambient temperature. Even if it is noticed that the model is inaccurate, still the model cannot easily be corrected. A second difficulty with analytical design-time models is that they require lots of insights into the system to create them. E.g., an RC-thermal model requires the floorplan, which is usually kept confidential by manufacturers.

The success of machine learning (ML)-based techniques originated in perceptional tasks like image classification [7]. ML-based techniques have since been successfully employed in many areas of computer engineering, and recently in run-time resource management [8]. They allow to overcome the drawbacks of heuristic approaches. Most ML-based models come with algorithms to adapt them at run-time. If model predictions are inaccurate, the parameters can easily be adjusted using these algorithms. This introduces adaptiveness with respect to run-time changes and operational variations. Furthermore, such models can be built without detailed insights into the modeled system, which may be very difficult to obtain. However, using ML-based techniques for run-time resource management creates unique challenges that need to be carefully considered.

Our contributions within this work are as follows:

- We highlight key challenges when employing machine learning to support run-time resource management.
- We show potential remedies for these challenges.
- We demonstrate in a proof-of-concept how introducing domain knowledge into ML model creation improves consistency of model predictions with the reality.

II. Machine Learning for Resource Management

Fig. 1 presents the two fundamental approaches how ML can be used to support resource management: learning actions or learning properties of the system and environment.

Resource management can abstractly be modeled as taking decisions based on the observed state of the system and

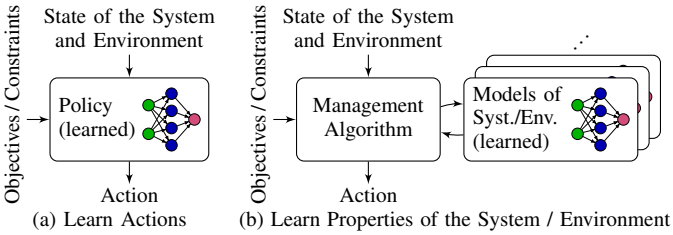

Fig. 1. Comparison of two alternative approaches how to support resource management by ML. (a) directly learn actions. (b) learn (physical) properties of the system or environment (e.g., power, performance) and use these models for predictions.

environment in order to achieve a given objective while satisfying given constraints. Therefore, it appears straightforward to directly learn the best action for a certain state (Fig. 1a). There are two main ways to achieve this: imitation learning (IL) and reinforcement learning (RL).

In IL, a model is trained based on expert knowledge [9]. The expert knowledge consist of a set of states and corresponding optimal actions. If the model generalizes well, it learns optimal or close-to-optimal actions also in unseen states. The main difficulty is that obtaining the optimal action in a certain state may not be feasible, making it difficult to create training data.

RL does not rely on known optimal actions but instead finds good actions by trial-and-error [10]. The information how well the policy performs is encoded in a reward function that combines objectives and constraints into a single scalar value. This reward function does not correspond to any physical value. *Formulating such a reward function is difficult.* If the function is not defined properly, the learned policy might find a solution that results in a high reward but does not reflect the goal that the designer intended. This effect is commonly known as *reward hacking* [11]. Both RL and IL share the drawback that changing the objective or constraints at run-time requires re-training of the policy – harming adaptability. Furthermore, it is very difficult to reverse-engineer why a certain action is taken – resulting in poor interpretability.

To solve these issues, ML can be used to learn physical properties of the system or environment instead [12] (Fig. 1b). These could be properties like power consumption, performance, or cooling. One or several such models are then used in combination with a management algorithm. The management algorithm decides the next action by using the models to predict the impact of candidate actions before executing them. Such an approach solves several problems with RL or IL-based approaches. Firstly, individual models can be trained independently, making it easier to create training data. Secondly, interpretability is better because the outputs of the models are physical properties like power or temperature – metrics that a human designer can easily understand. Finally, changing the objective or constraints at run-time can be done without affecting the ML-based models.

Therefore, we consider in this work approaches where instead of learning actions directly, properties of the system and environment are learned. Due to their potential, we also focus on neural networks (NNs) to learn these properties.

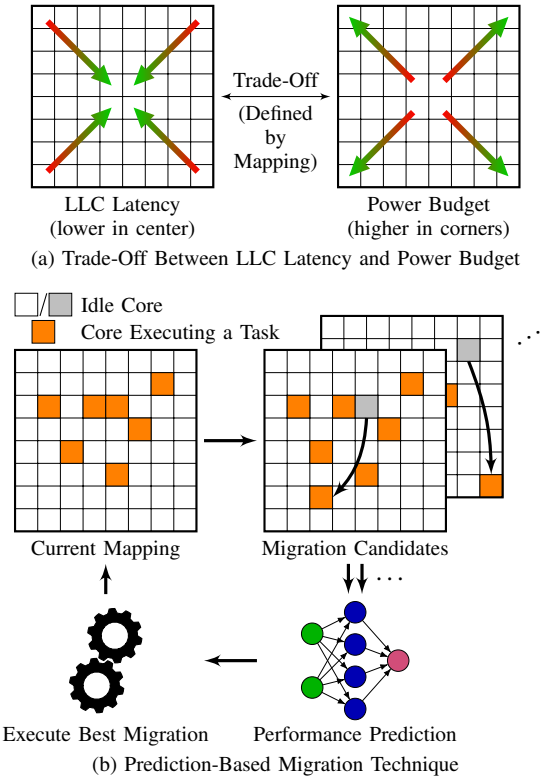

Fig. 2. Overview of a task migration approach to maximize the performance of many-cores with S-NUCA caches [12]. (a) Mapping on S-NUCA many-cores has to make a trade-off between minimizing the average LLC latency by mapping threads to the center and maximizing the power budget by mapping threads far from each other (i.e., to the corners) [14]. The performance-maximizing trade-off depends on the thread and may change over time. (b) Based on the current mapping, migration candidates are created. The impact of each migration on the overall performance is predicted without executing the migration. Only the most promising migration candidate is executed.

III. Task Migration on S-NUCA Many-Cores

This section presents a case study where ML has been used to guide task migration on a thermally-constrained many-core with static non-uniform cache access (S-NUCA) caches [12]. S-NUCA is a cache architecture where the last-level cache (LLC) is physically-distributed among all tiles in the many-core, but logically-shared [13]. Fig. 2a shows the trade-off that task mapping has to make on such a system. The closer a thread is mapped to the center of the many-core, the lower gets the average LLC latency that it experiences. However, mapping all threads to the center to minimize the LLC latency creates a thermal hotspot. This lowers the thermally safe power budgets [5] of threads, which lowers their sustainable voltage/frequency-levels. Maximizing the power budget is achieved by mapping threads far from each other. This results in heavy usage of cores close to the corners of the many-core, which results in a high average LLC latency. In summary, task mapping has to make a trade-off between minimizing the average LLC latency and maximizing the thermally-safe power budget to maximize the performance [14].

The performance of different threads depends differently on these two factors – leading to a different performance-maximizing trade-off. Moreover, execution phases within a

thread may lead to a dynamically changing trade-off. Together with changing workloads (arriving and leaving tasks) this makes task migration indispensable for maximizing the performance [12]. *This case study tackles the problem how to determine the best task migrations at run-time that maximize the global performance by considering the dynamically changing task characteristics and workload.*

Fig. 2b gives an overview how this task migration technique works [12]. Based on the current mapping, many migration candidates are created. Each of these migration candidates changes the mapping of threads to cores. This affects the average LLC latency observed by the migrated threads because their distances to the center of the many-core is changed. Additionally, changing the mapping changes the thermally-safe power budget of all threads, potentially affecting the performance of all threads.

In order to select the best migration candidate, the impact of the migration on the overall performance needs to be known. One way to achieve this is a sampling-based approach, where many migrations are tested, each for only a short time. Such an approach is not feasible here due to the sheer number of migration candidates. Therefore, it is required to estimate the performance impact of a migration *without executing it*.

To achieve this, a NN is trained that estimates the performance (IPS, instructions per second) of a thread after migration. This NN requires features about the thread itself (e.g., cache access statistics) and about the operating points (position and power budget) before and after the migration. Such an approach allows to capture the impact of task migration on the performance of different threads with taking their characteristics into account. Training data is created by comparing traces of benchmark applications at different operating points. Each training example describes the impact of a potential migration of a single thread of a benchmark at a certain execution phase from and to a certain operating point.

However, there are several challenges that occur while implementing such an approach. These, together with potential remedies, will be discussed in the next section.

IV. Challenges and Opportunities

This section describes challenges that arise when employing ML in run-time resource management. These challenges are described based on the case study presented in Section III. We discuss potential remedies for each challenge and demonstrate most of them using proof-of-concept implementations.

A. Run-Time Overhead

The computational effort to perform inference of a NN can be high, depending on the NN topology. Large NNs that are used in image classification can require millions of MAC operations for a single inference. Clearly, such large networks cannot be employed for run-time resource management – limiting the complexity of functions that can be learned. In the case study presented in the previous section, a very small NN with three fully-connected layers with 24, 16, and 1 neuron(s), respectively, was used. The total run-time overhead

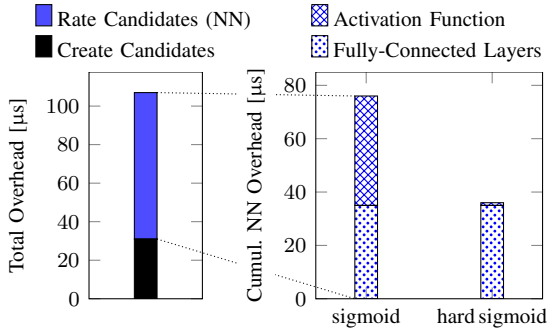

Fig. 3. The software run-time overhead of deciding the next migration (left) comprises two main parts: creating migration candidates and rating each of them. Creating migration candidates requires recalculating thermally-safe power budgets, rating requires inference of a NN. The majority of the overhead is required for NN inference. Further breaking down the NN inference overhead (right) reveals that calculating the activation function takes more than half of the overhead if the commonly used activation function *sigmoid* is employed. Replacing that function with an approximate version (*hard sigmoid*) reduces the overhead drastically.

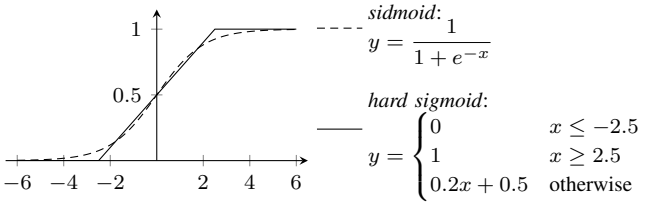

Fig. 4. Comparison of the two activation functions *sigmoid* and an approximate version *fast sigmoid* that does not require calculating an expensive exponential term.

of the migration technique was 107 µs for a system utilization of 50 %, including generation and rating of all migration candidates. This number is determined using the simulation setup described in [12]. The migration technique is executed in parallel on all 64 cores. We demonstrate here where the overhead comes from and how to further reduce it.

Fig. 3 shows the overhead for deciding the next migration when the system utilization is 50 %. First, around 1,500 migration candidates need to be created. This takes around 31 µs. The computational effort mainly is required for calculating the updated thermally-safe power budgets for every migration. Then, as depicted in Fig. 2b, every migration candidate is rated by predicting the impact of the migration on the system performance. Rating a single migration candidate requires a single inference of the NN that is done in software. Rating all 1,500 migration candidates takes about 76 µs and therefore makes up for the majority of the run-time overhead. We now demonstrate two possible remedies to reduce the overhead: firstly, we revisit the NN architecture and secondly, we investigate the usage of existing accelerators in smartphone SoCs.

Breaking down the NN overhead reveals that less than half of the overhead is required for calculating the matrix-vector multiplication in the fully-connected layers. Calculating the activation function is the major source of overhead. The activation function that has been employed is *sigmoid*, which is a commonly used activation function. However, evaluating the *sigmoid* function requires calculating an exponential term

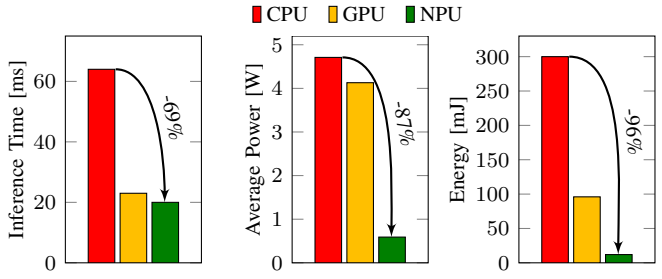

Fig. 5. Comparison of CPU (big cores), GPU, and NPU performance, power and energy for a single inference of *SqueezeNet* on the Kirin 970 SoC [15].

which is computationally expensive. Fig. 4 shows the *sigmoid* function, as well as an approximate version called *hard sigmoid*. *Hard sigmoid* is piecewise linear and even constant for large input values. It has properties similar to the *ReLU* activation function. Replacing the *sigmoid* activation function by *hard sigmoid* requires re-training of the NN. The accuracy if the NN decreases by 2 % (root mean squared error increases by 2 %). However, Fig. 3 shows that the run-time overhead decreases by 31 % just by replacing the activation function. *When designing a NN for run-time inference, it is not only important to achieve high accuracy, but also important to maintain low overhead. This forms a Pareto-optimization.*

Another way how to reduce the run-time overhead is to use hardware acceleration instead of inference in software. Modern smartphone SoCs commonly come with special hardware for inference of NNs [16]. These accelerators are meant to speed up mobile apps that use NNs for user tasks such as image processing or speech recognition. However, these accelerators will be mostly idle and can be used by the operating system instead to support run-time resource management. These accelerators provide high performance paired with low power, and consequently, low energy consumption compared to CPU or even GPU. Fig. 5 shows these metrics for the neural processing unit (NPU) of the Kirin 970 SoC for inference of *SqueezeNet* [15]. *SqueezeNet* is a NN that is built for the ImageNet challenge and classifies images into 1,000 classes. The NPU is capable of reducing inference time, power and energy by 69 %, 87 % and 96 %, respectively, compared to the CPU. *Using the already present neural network accelerators in modern smartphone SoCs forms a huge potential to reduce the overhead of ML-based resource management.*

These accelerators are designed for inference of NNs and support features like quantization to even further reduce the overhead. Because run-time training of NNs is not a typical requirement for mobile apps, the accelerators and the required libraries do not support it. *It remains an open challenge how to use the existing accelerators not only for inference, but also for training of NNs.*

B. Learned Models May Be Unrealistic

Another challenge with ML-based resource management is that the learned models do not necessarily correctly reflect trends that are present in reality. Models are ultimately used to compare the effects of different potential management

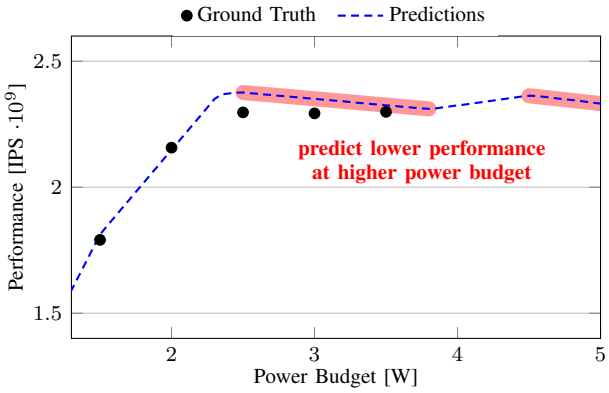

Fig. 6. Example of unrealistic model predictions. The performance of a thread never decreases if the power budget is increased. However, the model has been trained without this domain knowledge – treating the performance as a black box. This results in trends in the prediction that do not reflect reality and ultimately may result in suboptimal management decisions.

actions in order to decide the next action. Therefore, relative accuracy is more important than absolute accuracy. However, it is very difficult to measure relative accuracy in a loss function. Commonly used loss functions for regression aim at minimizing the absolute error (e.g., mean squared error). As a consequence, trends that are present in reality may not be reflected in the model, leading to an unrealistic model.

1) Example of Unrealistic Predictions: One example observed in the case study presented in Section III is monotonicity of performance and power budget. If a larger power budget is granted to a thread, its performance can never decrease. It is important to notice the difference between power budget and power consumption. The power budget sets an upper limit on the power consumption. Increasing the limit always infers fewer constraints on the thread – potentially increasing its performance, but never decreasing it. Fig. 6 shows unrealistic predictions of the baseline model. The model uses standard fully-connected layers with *hard sigmoid* as activation function. No additional constraints are used.

The model is used to predict the performance (IPS) of a slave thread of *PARSEC blackscholes* [17] for a changing power budget. The other input parameters of the NN like source of the migration and task characteristics are fixed. *Blackscholes* has not been used during training of the NN. The black dots in Fig. 6 show the ground truth, i.e. values that are measured during execution of *blackscholes* under different power budgets. The blue dashed line shows the predictions using the NN. Overall, the NN captures the dependency of performance and power budget, showing good generalization. However, even though the absolute error might be low, still model predictions contain trends that contradict the reality. Between 2.5 W and 3.8 W, as well as between 4.5 W and 5.0 W, the model predicts lower performance at increasing power budget. The predictions are used to compare the impact of different management actions. Different migration candidates result in different thermally-safe power budgets for all threads due to a changed mapping. Consequently, the model outputs

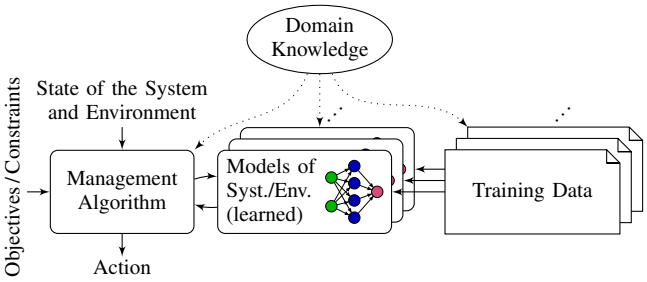

Fig. 7. Existing domain knowledge can be utilized at different places. i) for creating training data, ii) for designing the model architecture, or iii) for implementing the management algorithm.

at different power budgets are compared. Because *relative accuracy* is not given, ultimately, the model would result in a suboptimal migration being selected.

The reason for these unrealistic predictions is that the underlying physical system is treated as a black box. The only knowledge that is extracted are the distinct training examples. These training examples do not carry additional information that a designer has, such as monotonicity of performance w.r.t. power budget. Instead of discarding the existing domain knowledge, it must be used for designing the resource management system. This leads to an approach called *gray-box modeling*. Using domain knowledge has recently been identified as one of the key open challenges [18].

2) How to Use Domain Knowledge: Fig. 7 demonstrates where the domain knowledge can be used. In fact, domain knowledge can be integrated in any stage, starting from creating training data, model design and training itself, up to when the models are used by the management algorithm. The following gives a brief overview which type of domain knowledge can be leveraged in which stage.

Transformational invariance, where a transformation of the input features does not change the output, can be used while creating the training data. A classical example from image classification is rotational invariance, where a tilted image still contains the same object. Additional training examples are created from existing training examples using the transformation. This method is known as *training data augmentation* [7].

Symmetry is a special case of transformational that can be also addressed during training of the model. An example is a model that compares two inputs vectors. The effect of switching these input vectors on the output values can be well-defined and can therefore be captured through training data augmentation. Another way how to address this to constrain weights of the network to be symmetric [19].

Saturation effects, where the output does not change if a certain input feature is increased or decreased beyond a threshold can be either addressed through training data augmentation or within the management algorithm. An example is performance w.r.t. power budget. If the power budget allows for using the peak voltage and frequency levels, further increasing it does not increase the performance. Section IV-B3 shows experimentally how this increases the model quality.

Relative trends, such as monotonicity, can be introduced into the design and training of the model. Examples are

TABLE I
OVERVIEW WHERE TO USE EXISTING DOMAIN KNOWLEDGE

Knowledge	Where to use	How to use
Transformational invariance	Training data	Augmentation
Symmetry	Training data Model	Augmentation Symmetric weights
Saturation	Training data Mgmt. algorithm	Augmentation Saturate inputs
Relative trends	Model Model	Architecture Positive weights
Absolute values	Model	Output act. func.

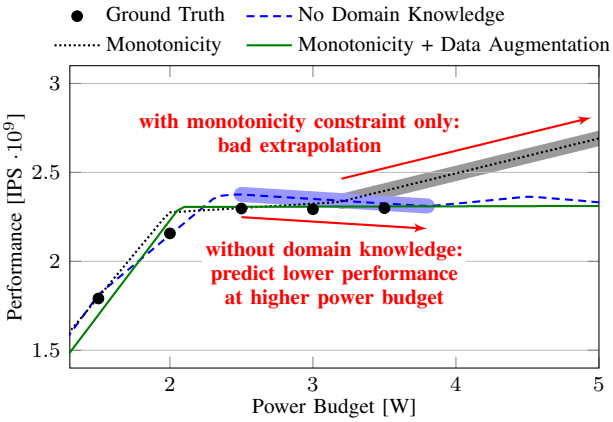

Fig. 8. Proof-of-concept how introducing domain knowledge improves predictions on unseen data. The designer has domain knowledge that performance must increase monotonically with the power budget. The model that was trained without domain knowledge predicts lower performance at higher power budget, which may result in suboptimal management decisions. Introducing a monotonicity constraint fixes this issue, but results in bad extrapolation at higher power budgets. This can be fixed by artificially augmenting the training data. Data is augmented not by collecting new application traces at higher power budgets, but by artificially generating more training data from already existing traces by exploiting domain knowledge.

performance increasing with power budget, power increasing with the supply voltage or temperature increasing with power consumption. Several NN architectures have been proposed that enforce monotonicity [20]. Usually, the model output then needs to be monotonic w.r.t. all input features. However, this is not the case in the studied case study. In the case of mixing monotonic and non-monotonic features, monotonicity can be enforced only for some features by using monotonic activation functions and restrict all weights on all paths from these features through the NN to the output to be positive. We demonstrate this technique in Section IV-B3.

As a last example, domain knowledge about absolute values can be introduced into the design of the model. E.g., an always positive output, such as power or performance, can be enforced by adding a ReLU activation at the model output. A summary of all techniques is presented in Table I.

3) Proof-of-Concept Case Study: The remainder of this section presents a solution for the unrealistic model predictions presented in Fig. 6. The model shows acceptable absolute error, but wrongly predicts that performance decreases with increasing power budget. However, since the model is used

to compare different potential management actions, relative accuracy is more important. The erroneous prediction is also shown in Fig. 8 to allow for an easy comparison.

To address the domain knowledge w.r.t. monotonicity, we restrict all weights in the NN that lie on a path between the input feature that contains the power budget and the output to be positive. Together with the monotonic activation function (*hard sigmoid*), the NN is restricted to learn monotonic functions w.r.t. power budget, but can still learn arbitrary functions w.r.t. all other input features. The resulting predictions are shown in Fig. 8. As expected, the prediction is monotonic.

However, a second issue has emerged now. The model predicts that the performance increases even for power budgets beyond 3 W, resulting in poor extrapolation. The cause behind this is that the training data only spans the range up to 3.5 W. There are several ways how to solve this. As an straightforward solution, more training data can be collected. This is a time-consuming process requiring to run many different tasks at different mappings at higher power budgets to fully cover higher power budgets. Collecting this training data takes days.

A more efficient solution is to use domain knowledge. Inspecting the already existing training data unveils that the peak voltage and frequency levels have been reached already. Therefore, we know that further increasing the power budget cannot increase the performance. We can augment the training data by taking existing training examples, increasing the input feature that contains the power budget and storing them as new training examples without changing the output (performance). This process barely induces additional overhead and can even be done on-the-fly while parsing the training data. Fig. 8 also shows the resulting predictions. Now, predictions are very accurate and most importantly, capture trends correctly. *This case study demonstrates how leveraging existing domain knowledge increases the model quality significantly.*

V. Conclusion

In this work, we presented our vision of ML-supported run-time resource management. We characterized the two main approaches how this can be achieved: directly learn decisions and learn physical properties of the system and environment. Based on a case study where the performance impact of task migration is predicted using a NN, we highlighted two key challenges with ML-supported resource management. The first key challenge is run-time overhead. We demonstrated how through overhead-aware design of the NN or through reusing already existing accelerators the overhead can be reduced. The second challenge is model predictions that do not reflect trends in reality, such as monotonicity. The root-cause for such unrealistic models is that existing domain knowledge is not used. We gave an overview how and where different types of domain knowledge can be used and demonstrated in a proof-of-concept case study how performance-prediction w.r.t. power budget can be improved.

Employing ML for run-time resource management induces specific challenges that must be overcome to fully unlock its potential. We demonstrated how this can be achieved.

Acknowledgment

This work is in parts funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 – TRR 89 Invasive Computing.

References

[1] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends," in *Design Automation Conference (DAC)*. IEEE, 2013, pp. 1–10.

[2] C. Ramey, "Tile-Gx100 Manycore Processor: Acceleration Interfaces and Architecture," in *Hot Chips Symposium (HCS)*. IEEE, 2011.

[3] M. R. Garey and D. S. Johnson, "Complexity Results for Multiprocessor Scheduling under Resource Constraints," *SIAM Journal on Computing*, 1975.

[4] E. L. de Souza Carvalho, N. L. V. Calazans, and F. G. Moraes, "Dynamic Task Mapping for MPSoCs," *IEEE Design & Test of Computers*, 2010.

[5] S. Pagani, H. Khdr, J.-J. Chen, M. Shafique, M. Li, and J. Henkel, "Thermal Safe Power (TSP): Efficient Power Budgeting for Heterogeneous Manycore Systems in Dark Silicon," *IEEE Transactions on Computers (TC)*, vol. 66, no. 1, pp. 147–162, 2017.

[6] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan, "HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 5, pp. 501–513, 2006.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[8] S. Pagani, S. M. PD, A. Jantsch, and J. Henkel, "Machine learning for power, energy, and thermal management on multi-core processors: A survey," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[9] S. K. Mandal, G. Bhat, C. A. Patil, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning," *Transactions on Very Large Scale Integration Systems (TVLSI)*, 2019.

[10] Z. Chen and D. Marculescu, "Distributed Reinforcement Learning for Power Limited Many-Core System Performance Optimization," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. EDA Consortium, 2015, pp. 1521–1526.

[11] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete Problems in AI Safety," *arXiv preprint arXiv:1606.06565*, 2016.

[12] M. Rapp, A. Pathania, T. Mitra, and J. Henkel, "Prediction-Based Task Migration on S-NUCA Many-Cores," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019.

[13] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2002, pp. 211–222.

[14] M. Rapp, M. Sagi, A. Pathania, A. Herkersdorf, and J. Henkel, "Power- and Cache-Aware Task Mapping with Dynamic Power Budgeting for Many-Cores," *Transactions on Computers (TC)*, 2019.

[15] S. Wang, A. Pathania, and T. Mitra, "Neural Network Inference on Mobile SoCs," *arXiv preprint arXiv:1908.11450*, 2019.

[16] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "AI Benchmark: Running Deep Neural Networks on Android Smartphones," in *European Conference on Computer Vision (ECCV)*, 2018.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008, pp. 72–81.

[18] J. R. Doppa, J. Rosca, and P. Bogdan, "Autonomous Design Space Exploration of Computing Systems for Sustainability: Opportunities and Challenges," *IEEE Design & Test*, vol. 36, no. 5, pp. 35–43, 2019.

[19] K. Hakhamaneshi, N. Werblun, P. Abbeel, and V. Stojanović, "BagNet: Berkeley Analog Generator with Layout Optimizer Boosted with Deep Neural Networks," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019.

[20] H. Zhang and Z. Zhang, "Feedforward Networks With Monotone Constraints," in *International Joint Conference on Neural Networks (IJCNN)*, vol. 3. IEEE, 1999, pp. 1820–1823.